

Intensity Frontier

Common Offline Documentation: *art* Workbook and Users Guide

Alpha Release 0.51

December 10, 2013

This version of the documentation is written for version v0.00.15 of the
art-workbook code.

Scientific Computing Division
Future Programs and Experiments Department
Scientific Software Infrastructure Group

Principal Author: Rob Kutschke
Editor: Anne Heavey *art* Developers: L. Garren, C. Green,
J. Kowalkowski, M. Paterno and P. Russo

Contents

Contents	3
art Glossary	13
List of Figures	23
List of Tables	25
Listings	27
I Introduction	3
1 How to Read this Documentation	1
2 Conventions Used in this Documentation	1
3 Introduction to the art Event Processing Framework	1
3.1 What is <i>art</i> and Who Uses it?	1
3.2 Why <i>art</i> ?	2
3.3 C++ and C++11	2
3.4 Getting Help	3
3.5 Overview of the Documentation Suite	3
3.5.1 The Introduction	3
3.5.2 The Workbook	4
3.5.3 Users Guide	4
3.5.4 Reference Manual	4
3.5.5 Technical Reference	5
3.5.6 Glossary	5
3.6 Some Background Material	5
3.6.1 Events and Event IDs	5
3.6.2 <i>art</i> Modules and the Event Loop	6
3.6.3 Module Types	9
3.6.4 <i>art</i> Data Products	10
3.6.5 <i>art</i> Services	10

3.6.6	Shareable Libraries and <i>art</i>	12
3.6.7	Build Systems and <i>art</i>	12
3.6.8	External Products	12
3.6.9	The Event-Data Model and Persistency	14
3.6.10	Event-Data Files	15
3.6.11	Files on Tape	15
3.7	The Toy Experiment	16
3.7.1	Toy Detector Description	16
3.7.2	Workflow for Running the Toy Experiment Code	17
3.8	Rules, Best Practices, Conventions and Style	21
4	Unix Prerequisites	1
4.1	Introduction	1
4.2	Commands	1
4.3	Shells	2
4.4	Scripts: Part 1	3
4.5	Unix Environments	3
4.5.1	Layering Environments	3
4.5.2	Examining and Using Environment Variables	4
4.6	Paths and <code>\$PATH</code>	6
4.7	Scripts: Part 2	7
4.8	bash Functions and Aliases	8
4.9	Login Scripts	8
4.10	Suggested Unix and bash References	9
5	Site-Specific Setup Procedure	1
6	Get your C++ up to Speed	1
6.1	Introduction	1
6.2	Establishing the Environment	2
6.2.1	Initial Setup	2
6.2.2	Subsequent Logins	3
6.3	C++ Exercise 1: The Basics	3
6.3.1	Concepts to Understand	3
6.3.2	How to Compile, Link and Run	4
6.3.3	Suggested Homework	5
6.3.4	Discussion	5
6.3.5	How was this Exercise Built?	6
6.4	C++ Exercise 2: About Compiling and Linking	6
6.4.1	What You Will Learn	6
6.4.2	The Source Code for this Exercise	7
6.4.3	Compile, Link and Run the Exercise	8
6.4.4	Alternate Script <code>build2</code>	11
6.4.5	Suggested Homework	12
6.5	C++ Exercise 3: Libraries	13
6.5.1	What You Will Learn	14

6.5.2	Building and Running the Exercise	14
6.6	Classes	17
6.6.1	Introduction	17
6.6.2	C++ Exercise 4 v1: The Most Basic Version	18
6.6.3	C++ Exercise 4 v2: The Default Constructor	22
6.6.4	C++ Exercise 4 v3: Constructors with Arguments	23
6.6.5	C++ Exercise 4 v4: Colon Initializer Syntax	25
6.6.6	C++ Exercise 4 v5: Member functions	27
6.6.7	C++ Exercise 4 v6: Private Data and Accessor Methods	30
6.6.7.1	Setters and Getters	30
6.6.7.2	What's the deal with the underscore?	34
6.6.7.3	An example to motivate private data	35
6.6.8	C++ Exercise 4 v7: The inline keyword	35
6.6.9	C++ Exercise 4 v8: Defining Member Functions within the Class Declaration	37
6.6.10	C++ Exercise 4 v9: The stream insertion operator	38
6.6.11	Review	40
6.7	C++ References	41
7	Using External Products in UPS	1
7.1	The UPS Database List: PRODUCTS	1
7.2	UPS Handling of Variants of a Product	3
7.3	The <code>setup</code> Command: Syntax and Function	3
7.4	Current Versions of Products	4
7.5	Environment Variables Defined by UPS	5
7.6	Finding Header Files	6
7.6.1	Introduction	6
7.6.2	Finding <i>art</i> Header Files	6
7.6.3	Finding Headers from Other UPS Products	8
7.6.4	Exceptions: The Workbook, ROOT and Geant4	8
II	Workbook	11
8	Preparation for Running the Workbook Exercises	1
8.1	Introduction	1
8.2	Getting Computer Accounts on Workbook-enabled Machines	1
8.3	Choosing a Machine and Logging In	2
8.4	Launching new Windows: Verify X Connectivity	3
8.5	Choose an Editor	3
9	Exercise 1: Run Pre-built art Modules	1
9.1	Introduction	1
9.2	Prerequisites	1
9.3	What You Will Learn	1
9.4	Running the Exercise	2

9.4.1	The Pieces	2
9.4.2	Log In, Set Up and Execute <i>art</i>	2
9.4.2.1	Standard Procedure	3
9.4.2.2	Procedure allowing Self-managed Working Directory	3
9.5	Logging In Again	4
9.6	Examine Output	4
9.7	Understanding the Configuration File <i>hello.fcl</i>	5
9.7.1	Some Bookkeeping Syntax	5
9.7.2	Some Physics Processing Syntax	7
9.7.3	Command line Options	8
9.7.4	Maximum Number of Events to Process	8
9.7.5	Changing the Input Files	9
9.7.6	Skipping Events	11
9.7.7	Identifying the User Code to Execute	12
9.7.8	Paths	13
9.7.9	Writing an Output File	15
9.8	Understanding the Process for Exercise 1	16
9.8.1	Follow the Site-Specific Setup Procedure (Details)	16
9.8.2	Make a Working Directory (Details)	17
9.8.3	Setup the toyExperiment UPS Product (Details)	17
9.8.4	Copy Files to your Current Working Directory (Details)	18
9.8.5	Source <i>makeLinks.sh</i> (Details)	18
9.8.6	Run <i>art</i> (Details)	19
9.9	How does <i>art</i> find Modules?	19
9.10	The <i>art</i> Run-time Environment	20
9.11	Finding FHiCL files: <i>FHICL_FILE_PATH</i>	21
9.11.1	The <i>-c</i> command line argument	22
9.11.2	<i>#include</i> Files	22
10	Exercise 2: Build and Run Your First Module	1
10.1	Introduction	1
10.2	Prerequisites	2
10.3	What You Will Learn	3
10.4	Setting up to Run Exercises: Standard Procedure	3
10.4.1	“Source Window” Setup	3
10.4.2	Examine Source Window Setup	4
10.4.2.1	About <i>git</i> and What it Did	4
10.4.2.2	Contents of the Source Directory	5
10.4.3	“Build Window” Setup	5
10.4.4	Examine Build Window Setup	6
10.5	Setting up to Run Exercises: Self-managed Working Directory	8
10.6	Logging In Again	9
10.7	The <i>art</i> Development Environment	10
10.8	Running the Exercise	11
10.8.1	Run <i>art</i> on <i>first.fcl</i>	11
10.8.2	The FHiCL File <i>first.fcl</i>	12

10.8.3	The Source Code File <code>First_module.cc</code>	12
10.8.3.1	The <code>#include</code> Files	14
10.8.3.2	The Declaration of the Class <code>First</code>	14
10.8.3.3	The Constructor for the Class <code>First</code>	16
10.8.3.4	Aside: Unused Formal Parameters	17
10.8.3.5	The Member Function <code>analyze</code> and <code>art::Event</code>	18
10.8.3.6	<code>art::EventID</code>	19
10.8.3.7	<code>DEFINE_ART_MACRO</code> : The Module Maker Macros	21
10.8.3.8	Some Alternate Styles	22
10.9	What does the Build System Do?	24
10.9.1	The Basic Operation	24
10.9.2	Incremental Builds and Complete Rebuilds	26
10.9.3	Finding Header Files at Compile-time	27
10.9.4	Finding Shared Library Files at Link-time	28
10.9.5	Build System Details	30
10.10	Suggested Activities	31
10.10.1	Create Your Second Module	31
10.10.2	Use <code>artmod</code> to Create Your Third Module	32
10.10.3	Running Many Modules at Once	34
10.10.4	Access Parts of the <code>EventID</code>	35
10.11	Final Remarks	36
10.11.1	Why is there no <code>First_module.h</code> File?	36
10.11.2	The Three File Module Style	37
10.12	Review	39
10.12.1	What Makes a class an Analyzer Module	39
10.12.2	Flow from source to <code>.fcl</code>	39
11	Exercise 3: The Optional Member Functions of <code>art</code> Modules	1
11.1	Introduction	1
11.2	Prerequisites	1
11.3	What You Will Learn	1
11.4	Setting up to Run this Exercise	2
11.5	The Source File <code>Optional_module.cc</code>	2
11.6	The classes <code>art::Run</code> , <code>art::RunID</code> , <code>art::SubRun</code> and <code>art::SubRunID</code>	3
11.7	Running this Exercise	5
11.8	The Member Function <code>beginJob</code>	6
11.9	Suggested Activities	7
11.9.1	Add the Matching <code>end</code> Member functions	7
11.9.2	Run on Multiple Input Files	8
12	Exercise 4: A First Look at Parameter Sets	1
12.1	Introduction	1
12.2	Prerequisites	2
12.3	What You Will Learn	2
12.4	Setting up to Run this Exercise	2

12.5	The File <code>pset01.fcl</code>	3
12.6	Running the Exercise	4
12.7	The Source code file <code>PSet01_module.cc</code>	4
12.8	A Comment About Templates	9
12.9	Exceptions	9
12.9.1	Suggested Exercises	11
12.10	Parameters and Data Members	11
12.11	Optional Parameters with Default Values	12
12.11.1	Suggested Exercises	13
12.11.2	Policies About Optional Parameters	13
12.12	Numerical Types, Precision and Canonical Forms	14
12.12.1	Suggested Exercises	15
12.13	Review	16
13	Exercise 5: Making Multiple Instances of a Module	1
13.1	Introduction	1
13.2	Prerequisites	1
13.3	What You Will Learn	1
13.4	Setting up to Run this Exercise	2
13.5	The Source File <code>Magic_module.cc</code>	2
13.6	The FHiCL File <code>magic.fcl</code>	3
13.7	Running the Exercise	3
13.8	Discussion	3
13.8.1	Order of Analyzer Modules is not Important	3
13.8.2	Two Meanings of <i>Module Label</i>	4
13.9	Suggested Exercise	5
13.10	Review	5
14	Accessing Data Products	1
14.1	Prerequisites	1
14.2	What You Will Learn	1
14.3	Running the Exercise	1
14.4	Discussion	1
14.5	Suggested Activities	1
15	Making Histograms and TFileService	1
15.1	Prerequisites	1
15.2	What You Will Learn	1
15.3	Running the Exercise	1
15.4	Discussion	1
15.5	Suggested Activities	1
16	Looping Over Collections	1
16.1	Prerequisites	1
16.2	What You Will Learn	1
16.3	Running the Exercise	1

16.4 Discussion	1
16.5 Suggested Activities	1
17 The Geometry Service	1
17.1 Prerequisites	1
17.2 What You Will Learn	1
17.3 Running the Exercise	1
17.4 Discussion	1
17.5 Suggested Activities	1
18 The Particle Data Table	1
18.1 Prerequisites	1
18.2 What You Will Learn	1
18.3 Running the Exercise	1
18.4 Discussion	1
18.5 Suggested Activities	1
19 GenParticle: Properties of Generated Particles	1
19.1 Prerequisites	1
19.2 What You Will Learn	1
19.3 Running the Exercise	1
19.4 Discussion	1
19.5 Suggested Activities	1
III Users Guide	3
20 Obtaining Credentials to Access Fermilab Computing Resources	1
20.1 Kerberos Authentication	1
20.2 Fermilab Services Account	2
21 Using git	1
22 art Run-time and Development Environments	1
22.1 The <i>art</i> Run-time Environment	1
22.2 The <i>art</i> Development Environment	4
23 art Framework Parameters	1
23.1 Parameter Types	1
23.2 Structure of <i>art</i> Configuration Files	2
23.3 Services	4
23.3.1 System Services	4
23.3.2 FloatingPointControl	4
23.3.3 Message Parameters	5
23.3.4 Optional Services	5
23.3.5 Sources	5
23.3.6 Modules	5

24 Job Configuration in art: FHiCL	1
24.1 Basics of FHiCL Syntax	1
24.1.1 Specifying Names and Values	1
24.1.2 FHiCL-reserved Characters and Keywords	3
24.2 FHiCL Keywords Reserved to <i>art</i>	4
24.3 Structure of a FHiCL Run-time Configuration File for <i>art</i>	5
24.4 Order of Elements in a FHiCL Run-time Configuration File for <i>art</i>	8
24.5 The <i>physics</i> Portion of the FHiCL Configuration	10
24.6 Choosing and Using Module Labels and Path Names	11
24.7 Scheduling Strategy in <i>art</i>	12
24.8 Scheduled Reconstruction using Trigger Paths	14
24.9 Reconstruction On-Demand	16
24.10 Bits and Pieces	16
25 Data Products	1
25.1 Overview	1
25.2 The Full Name of a Data Product	1
26 Producer Modules	1
27 Analyzer Modules	1
28 Filter Modules	1
29 art Services	1
30 art Input and Output	1
30.1 Input Modules	1
30.1.1 Configuring Input Modules to Read from Files	1
30.2 Output Filtering	3
30.3 Configuring Output Modules	5
31 art Misc Topics that Will Find Home	1
31.0.1 The Bookkeeping Structure and Event Sequencing Imposed by <i>art</i>	1
31.1 Rules for Module Names	2
31.2 Data Products and the Event Data Model	4
31.3 Basic <i>art</i> Rules	4
31.4 Compiling, Linking, Loading and Executing C++ Classes and <i>art</i> Modules	5
31.5 Shareable Libraries and <i>art</i>	5
31.6 Namespaces, <i>art</i> and the Workbook	7
31.7 Orphans	8
31.8 Code Guards	9
31.9 Inheritance	10
31.9.1 Introduction	10
31.9.2 Homework	11

31.9.3 Discussion	12
31.10Inheritance Relic	12
31.11Pointers	13
31.12RootOutput and table of event IDs	13
31.13Troubleshooting	13

IV Index	15
-----------------	-----------

art Glossary

abstraction	the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer (adapted from Wikipedia's entry for "Abstraction (computer science)").
analyzer module	an <i>art</i> module that may read information from the current event but that may not add information to it; e.g., a module to fill histograms or make printed output
API	Application Programming Interface
art	The <i>art</i> framework (<i>art</i> is not an acronym) is the software framework developed for common use by the Intensity Frontier experiments to develop their offline code and non-real-time online code
art module	see <i>module</i>
art path	a FHiCL sequence of <i>art</i> moduleLabels that specifies the work the job will do
artdaq	a toolkit that lives on top of <i>art</i> for building high-performance event-building and event-filtering systems; this toolkit is designed to support efficient use of multi-core computers and GPUs. A technical paper on <i>artdaq</i> can be found at .
bash	a UNIX shell scripting language that is used by some of the support scripts in the workbook exercises
boost	a class library with new functionality that is being prototyped for inclusion in future C++ standards
build system	turns source code into object files, puts them into a shared library, links them with other libraries, and may also run tests, deploy code to production systems and create some documentation.

buildtool	a Fermilab-developed tool (part of cetbuildtools) to compile, link and run tests on the source code of the Workbook
catch	See <i>exception</i> in a C++ reference
cetbuildtools	a build system developed at Fermilab
CETLIB	a utility library used by <i>art</i> (developed and maintained by the <i>art</i> team) to hold information that does not fit naturally into other libraries
class	The C++ programming language allows programmers to define program-specific data types through the use of <i>classes</i> . Classes define types of data structures and the functions that operate on those data structures. Instances of these data types are known as <i>objects</i> . Other object oriented languages have similar concepts.
CLHEP	a set of utility classes; the name is an acronym for a Class Library for HEP
collection	
configuration	see <i>run-time configuration</i>
const member function	a member function of a class that does not change the value of non-mutable data members; see <i>mutable data member</i>
constructor	a function that (a) shares an identifier with its associated class, and (b) initializes the members of an object instantiated from this class
DAQ	data acquisition system
data handling	
Data Model	see <i>Event Data Model</i>
data product	Experiment-defined class that can represent detector signals, reconstructed data, simulated events, etc. In <i>art</i> , a <i>data product</i> is the smallest unit of information that can be added to or retrieved from an event.
data type	See <i>type</i>
declaration (of a class)	the portion of a class that specifies its type, its name, and any data members and/or member functions it has
destructor	a function that (a) has the same identifier as its associated class but prefaced with a tilde ($\sim$), and (b) is used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed

Doxygen	a system of producing reference documentation based on comments in source code
ED	a prefix used in <i>art</i> (e.g., for module types) meaning <i>event-data</i>
EDAnalyzer	see <i>analyzer module</i>
EDFilter	see <i>filter module</i>
EDOutput	see <i>output module</i>
EDProducer	see <i>producer module</i>
EDSource	see <i>source module</i>
Event	In HEP there are two notions of the word <i>event</i> that are in common use; see <i>event (unit of information)</i> or <i>event (interaction)</i> . In this documentation suite, unless otherwise indicated, we mean the former.
Event (interaction)	An <i>event (unit of data)</i> may contain more than one fundamental interaction; the science goal is always to identify individual fundamental interactions and determine their properties. It is common to use the word <i>event</i> to refer to one of the individual fundamental interactions. In the near detector of a high-intensity neutrino experiment, for example, there may be multiple neutrino interactions within the unit of time that defines a single <i>event (unit of information)</i> . Similarly, in a colliding-beam experiment, an <i>event (unit of information)</i> corresponds to the information from one beam crossing, during which time there may be multiple collisions between beam particles.
Event (unit of information)	In the general HEP sense, an <i>event</i> is a set of raw data associated in time, plus any information computed from the raw data; <i>event</i> may also refer to a simulated version of same. Within <i>art</i> , the representation of an <i>event (unit of information)</i> is the class <code>art::Event</code> , which is the smallest unit of information that <i>art</i> can process. An <code>art::Event</code> contains an event identifier plus an arbitrary number of <i>data-products</i> ; the information within the <i>data-products</i> is intrinsically experiment dependent and is defined by each experiment. For bookkeeping convenience, <i>art</i> groups events into a hierarchy: a <i>run</i> contains zero or more <i>subRuns</i> and a <i>subRun</i> contains zero or more events.
Event Data Model (EDM)	Representation of the data that an experiment collects, all the derived information, and historical records necessary for reproduction of result

event loop	within an <i>art</i> job, the set of steps to perform in order to execute the per-event functions for each event that is read in, including steps for begin/end-job, begin/end-run and begin/end-subRun
event-data	all of the data products in an experiment's files; plus the meta-data that accompanies them. The HEP software community has adopted the word <i>event-data</i> to refer to the software details of dealing with the information found in <i>events</i> , whether the events come from experimental data or simulations.
event-data file	a collective noun to describe both data files and files of simulated events
exception, to throw	a mechanism in C++ (and other programming languages) to stop the current execution of a program and transfer control up the call chain; also called <i>catch</i>
experiment code	see <i>user code</i>
external product	for a given experiment, this is a software product that the experiment's software (within the <i>art</i> framework) does not build, but that it uses; e.g., ROOT, Geant4, etc. At Fermilab external products are managed by the in-house UPS/UPD system, and are often called <i>UPS products</i> or simply <i>products</i> .
FermiGrid	a batch system for submitting jobs that require large amounts of CPU time
FHiCL	Fermilab Hierarchical Configuration Language (pronounced "fickle"), a language developed and maintained by the <i>art</i> team at Fermilab to support run-time configuration for several projects, including <i>art</i>
FHiCL-CPP	the C++ toolkit used to read FHiCL documents within <i>art</i>
filter module	an <i>art</i> module that may alter the flow of processing modules within an event; it may add information to the event
framework (<i>art</i>)	The <i>art</i> framework is an application used to build physics programs by loading physics algorithms, provided as plug-in modules; each experiment or user group may write and manage its own modules. <i>art</i> also provides infrastructure for common tasks, such as reading input, writing output, provenance tracking, database access and run-time configuration.
framework (generic)	an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software (significantly abbreviated from Wikipedia's entry for "software framework"); note that the actual functionality provided by any given framework, e.g., <i>art</i> , will be tailored to the given needs.

free function	a function without data members; it knows only about arguments passed to it at run time; see <i>function</i> and <i>member function</i>
Geant4	a toolkit for the simulation of the passage of particles through matter, developed at CERN. http://geant4.cern.ch/
git	a source code management system used to manage files in the <i>art</i> Workbook; similar in concept to the older CVS and SVN, but with enhanced functionality
handle	a type of smart pointer that permits the viewing of information inside a data product but does not allow modification of that information; see <i>pointer</i> , <i>data product</i>
IF	Intensity Frontier
ifdh_sam	a UPS product that allows <i>art</i> to use <i>SAM</i> as an external run-time agent that can deliver remote files to local disk space and can copy output files to tape. The first part of the name is an acronym for Intensity Frontier Data Handling.
implementation	the portion of C++ code that specifies the functionality of a declared data type; where as a struct or class declaration (of a data type) usually resides in a <i>header</i> file (.h or .hh), the implementation usually resides in a separate source code file (.cc) that “#includes” the header file
instance	see <i>instantiation</i>
instantiation	the creation of an object instance of a class in an OOP language; an instantiated object is given a name and created in memory or on disk using the structure described within its class declaration.
jobsub-tools	a UPS product that supplies tools for submitting jobs to the Fermigrid batch system and monitoring them.
Kerberos	a single sign-on, strong authentication system required by Fermilab for access to its computing resources
kinit	a command for obtaining Kerberos credentials that allow access to Fermilab computing resources; see <i>Kerberos</i>
member function	(also called <i>method</i>) a function that is defined within (is a <i>member</i> of) a class; they define the behavior to be exhibited by instances of the associated class at program run time. At run time, member functions have access to data stored in the instance of the class with they are associated, and are thereby able to control or provide access to the state of the instance.

message facility	a UPS product used by <i>art</i> and experiments' code that provides facilities for merging messages with a variety of severity levels, e.g., informational, error, and so on; see also <i>mf</i>
message service	
method	see <i>member function</i>
mf	a namespace that holds classes and functions that make up the message facility used by <i>art</i> and by experiments that use <i>art</i> ; see <i>message facility</i>
module	a C++ class that obeys certain rules established by <i>art</i> and whose source code file gets compiled into a shared object library that can be dynamically loaded by <i>art</i> . An <i>art</i> module “plugs into” a processing stream and performs a specific task on units of data obtained using the Event Data Model, independent of other running modules. See also <i>moduleLabel</i>
module_type	a keyword known to <i>art</i> in the parameter set describing an <i>art</i> module; it specifies the name of a shared library to be loaded
moduleLabel	a user-defined identifier whose value is a parameter set that <i>art</i> will use to configure a module; see <i>module</i> and <i>parameter set</i>
Monte Carlo method	a class of computational algorithms that rely on repeated random sampling to obtain numerical results; i.e., by running simulations many times over in order to calculate those same probabilities heuristically just like actually playing and recording your results in a real casino situation: hence the name (Wikipedia)
mutable data member	The keyword “mutable” is used to allow a particular data member of const object to be modified. This is particularly useful if most of the members should be constant but a few need to be updateable (from highprogrammer.com).
namespace	a container within a file system for a set of identifiers (names); usually grouped by functionality, they are used to keep different subsets of code distinguishable from one another; identical names defined within different namespaces are disambiguated via their namespace prefix
ntuple	an ordered list of n elements used to describe objects such as vectors or tables
object	an instantiation of any data type, built-in types (e.g., int, double, float) or class types; i.e., a location range in memory containing an instantiation
object-oriented language	a programming language that supports OOP; this usually means support for classes, including public and private

data and functions

- object-oriented programming (OOP) a programming language model organized around *objects* rather than procedures, where *objects* are quantities of interest that can be manipulated. (In contrast, programs have been viewed historically as logical procedures that read in data, process the data and produce output.) Objects are defined by *classes* that contain attributes (data fields that describe the objects) and associated procedures. See *C++ class*; *object*.
- OOP see *object oriented programming*
- output module an *art* module that writes data products to output file(s); it may select a subset of data products in a subset of events; an *art* module contains zero or more output modules
- parameter set a C++ class, defined by FHiCL-CPP, that is used to hold run-time configuration for *art* itself or for modules and services instantiated by *art*. In a FHiCL file, a parameter set is represented by a FHiCL *table*; see *table*
- path a generic word based on the UNIX concept of PATH that refers to a colon-separated list of directories used by *art* when searching for various files (e.g., data input, configuration, and so on)
- physics in *art*, *physics* is the label for a portion of the run-time configuration of a job; this portion contains up to five sections, each labeled with a reserved keyword (that together form a parameter set within the FHiCL language); the parameters are *analyzers*, *producers*, *filters*, *trigger_paths* and *end_paths*.
- pointer a variable whose value is the address of (i.e., that points to) a piece of information in memory. A native C++ pointer is often referred to as a *bare pointer*. *art* defines different sorts of *smart pointers* (or *safe pointers*) for use in different circumstances. One commonly used type of smart pointer is called a *handle*.
- process_name a parameter to which the user assigns a mnemonic value identifying the physics content of the associated FHiCL parameter set (i.e., the parameters used in the same FHiCL file). The *process_name* value is embedded into every data product created via the FHiCL file.
- producer module an *art* module that may read information from the current event and may add information to it
- product See either *external* product or *data* product
- redmine an open source, web-based project management and bug-tracking

	tool used as a repository for <i>art</i> code and related code and documentation
ROOT	an HEP data management and data presentation package used by <i>art</i> and supported by CERN; <i>art</i> is designed to allow output of event-data to files in ROOT format, in fact currently it is the only output format that <i>art</i> implements
ROOT files	There are two types of ROOT files managed by <i>art</i> : (1) event-data output files, and (2) the file managed by TFileService that holds user-defined histograms, ntuples, trees, etc.
run	a period of data collection, defined by the experiment (usually delineates a period of time during which certain running conditions remain unchanged); a run contains zero or more subRuns
run-time configuration	(processing-related) structured documents describing all processing aspects of a single job including the specification of parameters and workflow; in <i>art</i> it is supplied by a FHiCL file; see <i>FHiCL</i>
safe pointer	see <i>pointer</i>
SAM	(Sequential data Access via Metadata) a Fermilab-supplied product that provides the functions of a file catalog, a replica manager and some functions of a batch-oriented workflow manager
scope	
sequence (in FHiCL)	one or more comma-separated FHiCL values delimited by square brackets (
	...
	) in a FHiCL file is called a <i>sequence</i> (as distinct from a <i>table</i>)
service	in <i>art</i> , a singleton-like object (type) whose lifetime and configuration are managed by <i>art</i> , and which can be accessed by module code and by other services by requesting a <i>service handle</i> to that particular service. The service <i>type</i> is used to provide geometrical information, conditions and management of the random number state; it is also used to implement some internal functionality. See also <i>T File Service</i>
shared library	
signature (of a function)	the unique identifier of a C++ function, which includes: (a) its name, including any class name or namespace components, (b) the number and type of its arguments, (c) whether it is a member function, (d) whether it is a const function (Note that the signature of a function does not include its return type.)

site	As used in the <i>art</i> documentation, a <i>site</i> is a unique combination of experiment and institution; used to refer to a set of computing resources configured for use by a particular experiment at a particular institution. This means that, for example, the Workbook environment on a Mu2e-owned computer at Fermilab will be different than that on an Mu2e-owned computer at LBL. Also, the Workbook environment on a Mu2e-owned computer at Fermilab will be different from that on an LBNE-owned computer at Fermilab.
smart pointer	see <i>pointer</i>
source	(refers to a <i>data</i> source) the name of the parameter set inside an FHiCL file describing the first step in the workflow for processing an event; it reads in each event sequentially from a data file or creates an empty event; see also <i>source code</i> ; see also <i>EDsource</i>
source code	code written in C++ (the programming language used with <i>art</i>) that requires compilation and linking to create an executable program
source module	an <i>art</i> module that can initiate an <i>art</i> path by reading in event(s) from a data file or by creating an empty event; it is the first step of the processing chain
standard library, C++	the C++ standard library of routines
std	identifier for the namespace used by the C++ standard library
struct	identical to a C++ class except all members are <i>public</i> (instead of <i>private</i>) by default
subRun	a period of data collection within a run, defined by the experiment (it may delineate a period of time during which certain run parameters remain unchanged); a SubRun is contained within a <i>run</i> ; a subRun contains zero or more <i>events</i>
table (in FHiCL)	a group of FHiCL definitions delimited by braces (<code>{ ... }</code>) is called a <i>table</i> ; within <i>art</i> , a FHiCL table gets turned into an object called a <i>parameter set</i> . Consequently, a FHiCL table is typically called a <i>parameter set</i> . See <i>parameter set</i> .
template (C++)	Templates are a feature of C++ that allows for meta-programming. The practical meaning of this is that allows one to write an algorithm that will work on any type that provides the features required by the algorithm; the classic example is that the standard library provides a sort algorithm that will work for any type that provides a way to ask if one object of the type is less than another object of the type. <i>art</i> uses templates in

	several prominent places. You do not need to fully understand templates — just how to use them when you encounter them.
TFileService	an <i>art</i> service used by all experiments to give each module a ROOT subdirectory in which to place its own histograms, TTrees, and so on; see <i>TTrees</i> and <i>ROOT</i>
truth information	One use of simulated events is to develop, debug and characterize the algorithms used in reconstruction and analysis. To assist in these tasks, the simulation code often creates data products that contain detailed information about the right answers at intermediate stages of reconstruction and analysis; they also write data products that allow the physicist to ask “is this a case in which there is an irreducible background or should I be able to do better?” This information is called the <i>truth information</i> , the <i>Monte Carlo truth</i> or the <i>God’s block</i> .
TTrees	a ROOT implementation of a tree; see <i>tree</i> and <i>ROOT</i>
type	variables and objects in C++ must be classified into <i>types</i> , e.g., built-in types (integer, boolean, float, character, etc.), more complex user-defined classes/structures and typedefs; see <i>class</i> , <i>struct</i> , and <i>typedef</i> . The word <i>type</i> in the context of C++ and <i>art</i> is the same as <i>data type</i> unless otherwise stated.
UPS/UPD	a Fermilab-developed system for distributing software products
user code	experiment-specific and/or analysis-specific C++ code that uses the <i>art</i> framework; this includes any personal code you write that uses <i>art</i> .
variable	a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents.

List of Figures

3.1	The principal components of the <i>art</i> documentation suite	3
3.2	The geometry of the toy detector; the figures are described in the text. A uniform magnetic field of strength 1.5 T is oriented in the $+z$ direction.	17
3.3	Event display of a typical simulated event in the toy detector. . . .	19
3.4	Event display of another simulated event in the toy detector; a K^- (blue) is produced with a very shallow trajectory and it does not intersect any detector shells while the K^+ (red) makes five hits in the inner detector and seven in the outer detector	19
3.5	The final plot showing 870 reconstructed events out of 1000 generated events	20
4.1	Layers in the <i>art</i> Workbook (left) and experiment-specific (right) computing environments	5
6.1	Memory diagram at the end of a run of <code>Classes/v1/ptest.cc</code>	21
6.2	Memory diagram at the end of a run of <code>Classes/v6/ptest.cc</code>	33
9.1	Elements of the <i>art</i> run-time environment for the first Workbook exercise	21
10.1	Elements of the <i>art</i> development environment as used in most of the Workbook exercises; the arrows denote information flow, as described in the text.	10
22.1	Elements of the <i>art</i> run-time environment, just for running the Toy Experiment code for the Workbook exercises	2
22.2	Elements of the <i>art</i> run-time environment for running an experiment's code (everything pre-built)	2
22.3	Elements of the <i>art</i> run-time environment for a production job with officially tracked inputs	3
22.4	Elements of the <i>art</i> development environment as used in most of the Workbook exercises	4

22.5	Elements of the <i>art</i> development environment for building the full code base of an experiment	5
22.6	Elements of the <i>art</i> development environment for an analysis project that builds against prebuilt release	5
31.1	Illustration of compiled, linked “regular” C++ classes (not <i>art</i> modules) that can be used within the <i>art</i> framework. Many classes can be linked into a single shared library.	6
31.2	Illustration of compiled, linked <i>art</i> modules; each module is built into a single shared library for use by <i>art</i>	7

List of Tables

3.1	Compiler flags for the optimization levels defined by cetbuildtools ; compiler options not related to optimization or debugging are not included in this table.	13
3.2	Units used in the Workbook	17
5.1	Site-specific setup procedures for $IF(\gamma)$ Experiments at Fermilab; for the equivalent procedure at a non-Fermi site, consult an expert from that site. For the NO ν A experiment, the procedure is a sequence of three commands. For all others it is a single command, but in a few cases the command is so long that the table shows it split over two lines: you must type it as a single command on one line; the experiments whose command is so split are ArgoNeut, LBNE and MicroBoone.	2
7.1	For selected UPS Products, this table gives the names of the associated namespaces. The UPS products that do not use namespaces are discussed in Section 7.6.4. [‡] The namespace <code>tex</code> is also used by the <i>art</i> Workbook, which is not a UPS product.	9
8.1	Experiment-specific Information for New Users	2
8.2	Login machines for running the Workbook exercises	2
9.1	The input files provided by for the Workbook exercises	2
10.1	Compiler and Linker Flags for a Profile Build	31
12.1	caption	15
23.1	<i>art</i> Floating Point Parameters	4
23.2	<i>art</i> Message Parameters	5

Listings

6.1	The form of a class declaration	17
6.2	The contents of <code>v1/Point.h</code>	19
6.3	The contents of <code>v1/ptest.cc</code>	19
9.1	Sample output from running <code>hello.fcl</code>	4
9.2	The source parameter set from <code>hello.fcl</code>	6
9.3	The physics parameter set from <code>hello.fcl</code>	7
9.4	The remainder of <code>hello.fcl</code>	8
9.5	A FHiCL fragment illustrating module labels	13
9.6	Example of the value of <code>LD_LIBRARY_PATH</code>	19
10.1	Example of output created by <code>setup_for_development</code>	6
10.2	The contents of <code>First_module.cc</code>	13
10.3	An alternate layout for <code>First_module.cc</code>	23
10.4	The file <code>art-workbook/FirstModule/CMakeLists.txt</code>	30
10.5	The physics parameter set for <code>all.fcl</code>	34
10.6	The contents of <code>First.h</code> in the 3 file model	38
10.7	The contents of <code>First.cc</code> in the 3 file model	38
10.8	The contents of <code>First_module.cc</code> in the 3 file model	38
11.1	The output produced by <code>Optional_module.cc</code> when run using <code>optional.fcl</code>	6
12.1	The definition of <code>psetTester</code> from <code>pset01.fcl</code>	3
12.2	The output from <code>PSet01</code> when run using <code>pset01.fcl</code>	5
12.3	The Constructor of <code>PSet01_module.cc</code>	6
12.4	The expected output when the parameter <code>b</code> is removed from <code>pset01.fcl</code>	11
12.5	The expected output when the parameter <code>c</code> is misdefined in <code>pset01.fcl</code>	11
12.6	The output from <code>PSet02</code> when run using <code>pset02.fcl</code>	12
12.7	The output from <code>PSet03</code> when run using <code>pset03.fcl</code>	13
12.8	The output from <code>PSet04</code> when run using <code>pset04.fcl</code>	16
12.9	The output from <code>PSet04</code> when run using the modified version of <code>pset04.fcl</code> , which has an intentional error	16
13.1	The output from running <code>magic.fcl</code>	4
30.1	Reading in a ROOT data file	1
30.2	Reading in a ROOT data file	2
30.3	Reading in a ROOT data file	2
30.4	Reading in a ROOT data file	2

30.5 Reading in a ROOT data file	3
30.6 Reading in a ROOT data file	3
31.1 Module source sample	3

1

Part I

2

Introduction

1 How to Read this Documentation

The *art* document suite (in its current “alpha release” form) consists of an introductory section and the first few exercises of the Workbook¹, plus a glossary and an index. There are also some preliminary (incomplete and unreviewed) portions of the Users Guide included in the compilation.

If you are new to HEP data analysis...

Read Parts I and II (the introductory material and the Workbook) from start to finish. The Workbook is aimed at an audience who is familiar with (although not necessarily expert in) Unix, C++ and Fermilab’s UPS product management system, and who understands the basic *art* framework concepts. The introductory chapters prepare the “just starting out” reader in all these areas.

If you are an HEP data analysis expert...

Skim Chapter 3: this is where key terms and concepts used throughout the *art* document suite get defined. Skip the rest of the introductory material and jump straight into running Exercise 1 in Section 9.4 of the Workbook. Take the approach of: Don’t need it? Don’t read it.

If you are somewhere in between beginner and expert...

Skim the introductory material in Part I to glean what you need, but definitely read Chapter 3. Along with the experts, you can take the approach of: Don’t need it? Don’t read it.



¹The Workbook is expected to contain roughly 25 exercises when complete.

2 Conventions Used in this Documentation

Most of the material in this introduction and in the Workbook is written so that it can be understood by beginners in HEP computing; if it is not, please let us know (see Section 3.4)!

In some places, however, it will be necessary for a paragraph or two to be written for experts. Such paragraphs will be marked with a “dangerous bends” symbol in the margin, as shown at right. Beginners can skip these sections on first reading and come back to them at a later time.



The first instance of each term that is defined in the glossary is written in *italics* followed by a γ (Greek letter gamma), e.g., *framework*(γ).

Occasionally, text will be called out to make sure that you don’t miss it. Important or tricky terms and concepts will be marked with an “pointing finger” symbol in the margin, as shown at right.



Items that are even trickier will be marked with a “bomb” symbol in the margin, as shown at right. You really want to avoid the problems they describe.



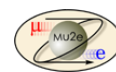
Text that refers in particular to Fermilab-specific information is marked with a Fermilab picture, as shown at right.



Text that refers in particular to information about using *art* at sites other than Fermilab is marked with a “generic site” picture, as shown at right. A *site* is defined as a unique combination of experiment and institution, and is used to refer to a set of computing resources configured for use by a particular experiment at a particular institution.



Experiment-specific information will be kept to an absolute minimum; where these items appear, they will be marked with an experiment-specific icon, e.g., the Mu2e icon at right.



Unix commands that you must type are shown preceded by a dollar sign prompt (\$) in typewriter font. Do not type the \$! Portions of the command for which you must substitute actual values are surrounded by angle brackets (< . . . >).

1 When an example Unix command line would overflow the page width, this
2 documentation will use a trailing backslash to indicate that the command is
3 continued on the next line. For example:

```
4 $ art -c hi.fcl inputFiles/input02_data.root \  
5   inputFiles/input03_data.root
```

6 This means that you should type the entire command on a single line, without
7 typing the dollar sign or the backslash. Computer output from a command is
8 shown in typewriter font.

3 Introduction to the *art* Event Processing Framework

3.1 What is *art* and Who Uses it?

art(γ) is an event-processing *framework*(γ) developed and supported by the Fermilab Scientific Computing Division (SCD). The *art* framework is used to build physics programs by loading physics algorithms, provided as plug-in modules. Each experiment or user group may write and manage its own modules. *art* also provides infrastructure for common tasks, such as reading input, writing output, provenance tracking, database access and run-time configuration.

The initial clients of *art* are the Fermilab Intensity Frontier experiments but nothing prevents other experiments from using it, as well. The name *art* is always written in *italic lower case*; it is not an acronym.

art is written in C++ and is intended to be used with user code written in C++. (*User code* includes experiment-specific code and any other user-written, non-*art*, non-*external-product*(γ) code.)

art has been designed for use in all places that an experiment might require a software framework, including:

- high-level software triggers
- online data monitoring
- calibration
- reconstruction
- analysis
- simulation

art is not designed for use in real-time environments, such as the direct interface with data-collection hardware.

The Fermilab SCD has also developed a related product named *artdaq*(γ), a layer that lives on top of *art* and provides features to support the construction of data-acquisition (*DAQ*(γ)) systems based on commodity servers. Further discussion of *artdaq* is outside the scope of this documentation.

The design of *art* has been informed by the lessons learned by the many High Energy Physics (HEP) experiments that have developed C++ based frameworks over the past 20 years. In particular, it was originally forked from the framework for the CMS experiment, *cmsrun*.

Experiments using *art* are listed at artdoc.fnal.gov under “Intensity Frontier Links.”

3.2 Why *art*?

In all previous experiments at Fermilab, and in most previous experiments elsewhere, infrastructure software (i.e., the framework, broadly construed – mostly forms of bookkeeping) has been written in-house by each experiment, and each implementation has been tightly coupled to that experiment’s code. This tight coupling has made it difficult to share the framework among experiments, resulting in both great duplication of effort and mixed quality.

art was created as a way to share a single framework across many experiments. In particular, the design of *art* draws a clear boundary between the framework and the user code; the *art* framework (and other aspects of the infrastructure) is developed and maintained by software engineers who are specialists in the field, not by physicists who are primarily interested in the science. Experiments use *art* as an *external package*. Despite some constraints that this separation imposes, it has improved the overall quality of the framework and reduced the duplicated effort. Therefore each experiment can build their physics software on top of a more complete and more robust foundation. Our goal is that this will make it easier to develop and maintain physics software, thereby improving the overall quality of the physics results.

3.3 C++ and C++11

In 2011, the International Standards Committee voted to approve a new standard for C++, called C++ 11.

Much of the existing user code was written prior to the adoption of the C++ 11 standard and has not yet been updated. As you work on your experiment, you are likely to encounter both code written the new way and code written the old way. Therefore, the Workbook will often illustrate both practices.



A very useful compilation of what is new in C++ 11 can be found at

1 <https://cdcv.s.fnal.gov/redmine/projects/gm2public/wiki/CPP2011>

2 This reference material is written for advanced C++ users.



3 3.4 Getting Help

4 Please send your questions and comments to art-users@fnal.gov. More support
5 information is listed at <http://artdoc.fnal.gov/artsupport.shtml>.

6 3.5 Overview of the Documentation Suite

7 When complete, this documentation suite will contain several principal compo-
8 nents, or *volumes*: the introduction that you are reading now, a Workbook, a
9 Users Guide, a Reference Manual, a Technical Reference and a Glossary. At the
10 time of writing, drafts exist for the Workbook, the Users Guide and the Glossary.
11 The components in the documentation suite are illustrated in Figure 3.1.

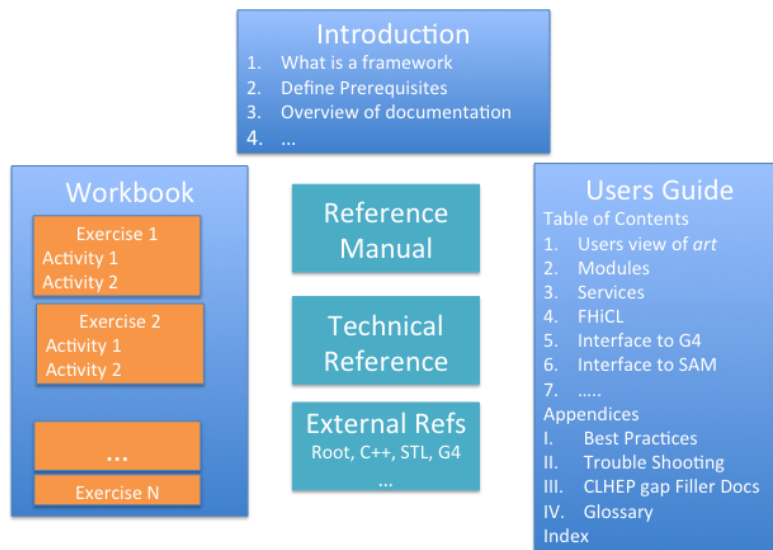


Figure 3.1: The principal components of the *art* documentation suite

12 3.5.1 The Introduction

13 This introductory volume is intended to set the stage for using *art*. It intro-
14 duces *art*, provides background material, describes some of the software tools on

- 1 which *art* depends, describes its interaction with related software and identifies
2 prerequisites for successfully completing the Workbook exercises.

3 3.5.2 The Workbook

4 The Workbook is a series of standalone, self-paced exercises that will introduce
5 the building blocks of the *art* framework and the concepts around which it is
6 built, show practical applications of this framework, and provide references to
7 other portions of the documentation suite as needed. It is targeted towards
8 physicists who are new users of *art*, with the understanding that such users will
9 frequently be new to the field of computing for HEP and to C++.

10 One of the Workbook's primary functions is training readers how and where
11 to find more extensive documentation on both *art* and external software tools;
12 they will need this information as they move on to develop and use the scientific
13 software for their experiment.

14 The Workbook assumes some basic computing skills and some basic familiarity
15 with the C++ computing language; Chapter 6 provides a tutorial/refresher for
16 readers whose C++ skills aren't quite up-to-speed.

17 The Workbook is written using recommended best practices that have become
18 current since the adoption of C++ 11.

19 Because *art* is being used by many experiments, the Workbook exercises are
20 designed around a *toy* experiment that is greatly simplified compared to any
21 actual experimental detector, but it incorporates enough richness to illustrate
22 most of the features of *art*. The goal is to enable the physicists who work through
23 the exercises to translate the lessons learned there into the environment of their
24 own experiments.

25 3.5.3 Users Guide

26 The Users Guide is targeted at physicists who have reached an intermediate level
27 of competence with *art* and its underlying tools. It contains detailed descriptions
28 of the features of *art*, as seen by the physicists. The Users Guide will provide
29 references to the *external products*(γ) on which *art* depends, information on how
30 *art* uses these products, and as needed, documentation that is missing from the
31 external products' proper documentation.

32 3.5.4 Reference Manual

33 The Reference Manual will be targeted at physicists who already understand
34 the major ideas underlying *art* and who need a compact reference to the Appli-

1 cation Programmer Interface (*API*(γ)). The Reference Manual will likely be
2 generated from annotated source files, possibly using *Doxygen*(γ).

3 3.5.5 Technical Reference

4 The Technical Reference will be targeted at the experts who develop and main-
5 tain *art*; few physicists will ever want or need to consult it. It will document the
6 internals of *art* so that a broader group of people can participate in development
7 and maintenance.

8 3.5.6 Glossary

9 The glossary will evolve as the documentation set grows. At the time of writing,
10 it includes definitions of *art*-specific terms as well as some HEP, Fermilab, C++
11 and other relevant computing-related terms used in the Workbook and the Users
12 Guide.

13 3.6 Some Background Material

14 This section defines some language and some background material about the
15 *art* framework that you will need to understand before starting the Work-
16 book.

17 3.6.1 Events and Event IDs

18 In almost all HEP experiments, the core idea underlying all bookkeeping is the
19 *event*(γ). In a triggered experiment, an event is defined as all of the information
20 associated with a single trigger; in an untriggered, spill-oriented experiment, an
21 event is defined as all of the information associated with a single spill of the beam
22 from the accelerator. Another way of saying this is that an event contains all of
23 the information associated with some time interval, but the precise definition of
24 the time interval changes from one experiment to another¹. Typically these time
25 intervals are a few nanoseconds to a few tens of mircoseconds. The information
26 within an event includes both the raw data read from the Data Acquisition
27 System (DAQ) and all information that is derived from that raw data by the
28 reconstruction and analysis algorithms. An event is the smallest unit of data
29 that *art* can process at one time.

¹There is a second, distinct, sense in which the word *event* is sometimes used; it is used as a synonym for a *fundamental interaction*; see the glossary entry for *event (fundamental interaction)*(γ). Within this documentation suite, unless otherwise indicated, the word *event* refers to the definition given in the main body of the text.

1 In a typical HEP experiment, the trigger or DAQ system assigns an event identifier (event ID) to each event; this ID uniquely identifies each event, satisfying
2 a critical requirement imposed by *art* that each event be uniquely identifiable
3 by its event ID. This requirement also applies to simulated events.
4

5 The simplest event ID is a monotonically increasing integer. A more common
6 practice is to define a multi-part ID and *art* has chosen to use a three-part ID,
7 including:

- 8 • *run*(γ) number
- 9 • *subRun*(γ) number
- 10 • *event*(γ) number

11 There are two common methods of using this event ID scheme and *art* allows
12 experiments to choose either:

- 13 1. When an experiment takes data, the event number is incremented every
14 event. When some condition occurs, the event number is reset to 1 and
15 the subRun number is incremented, keeping the run number unchanged.
16 This cycle repeats until some other condition occurs, at which time the
17 event number is reset to 1, the subRun number is reset to 0 (0 not 1 for
18 historical reasons) and the run number is incremented.
- 19 2. The second scheme is the same as the first except that the event number
20 monotonically increases throughout a run and does not reset to 1 on
21 subRun boundaries. The event number does reset to 1 at the start of each
22 run.

23 *art* does not define what conditions cause these transitions; those decisions are
24 left to each experiment. Typically experiments will choose to start new runs or
25 new subRuns when one of the following happens: a preset number of events is
26 acquired; a preset time interval expires; a disk file holding the output reaches a
27 preset size; or certain running conditions change.

28 *art* requires only that a subRun contain zero or more events and that a run
29 contain zero or more subRuns.

30 When an experiment takes data, events read from the DAQ are typically written
31 to disk files, with copies made on tape. The events in a single subRun may be
32 spread over several files; conversely a single file may contain many runs, each of
33 which contains many subRuns.

34 3.6.2 *art* Modules and the Event Loop

35 Users provide executable code to *art* in chunks called *art modules*(γ) that “plug
36 into” a processing stream and operate on event data. An *art* module (also called
37 simply a *module*) is an *art-ified* C++ class – more on this below.

1 The concept of reading events and, in response to each new event, calling the
 2 appropriate methods of each module, is referred to as the *event loop*(γ).

3 The concepts of the *art module* and the *event loop* will be illustrated via the
 4 following discussion of how *art* processes a job.

5 The simplest command to run *art* looks like:

```
6 $ art -c run-time-configuration-file.fcl
```

7 The *run-time configuration file*(γ) is a text file that tells one run of *art* what
 8 it should do. Run-time configuration files for *art* are written in the Fermilab
 9 Hierarchical Configuration Language *FHiCL*(γ), pronounced “fickle”) and the
 10 filenames end in *.fcl*. As you progress through the Workbook, this language
 11 and the conventions used in the run-time configuration file will be explained;
 12 the full details are available in Chapter 24 of the Users Guide. (The run-time
 13 configuration file is often referred to as simply the *configuration file* or even
 14 more simply as just the *configuration*(γ).)

15 When *art* starts up, it reads the configuration file to learn what input files
 16 it should read, what user code it should run and what output files it should
 17 write. As mentioned above, an experiment’s code (including any code written
 18 by individual experimenters) is provided in units called *art modules*. A mod-
 19 ule is simply a C++ class, provided by the experiment or user, that obeys a
 20 set of rules defined by *art* and whose *source code*(γ) file gets compiled into a
 21 shared *object*(γ) library that can be dynamically loaded by *art*. These rules will
 22 be explained as you work through the Workbook and they are summarized in
 23 Section 31.3.²



24 The code base of a typical experiment will contain many C++ classes. Only a
 25 small fraction of these will be modules; most of the rest will be ordinary C++
 26 classes that are used within modules³.

27 In some circumstances the configuration file tells *art* the order in which to run
 28 the modules, but other times, *art* is left to determine, on its own, the correct
 29 order of execution (*reconstruction on demand*). In either case, each module in
 30 the processing stream must run independently of the others.

31 *art* requires that each module provide some code that will be called once for
 32 every event. Imagine each event as a widget on an assembly line, and each
 33 module as a worker that needs to perform a set task on each widget. Further,
 34 workers must find out if they need to do some start-up or close-down jobs.
 35 Following this metaphor, any module may provide code to be called at the
 36 following times:

- 37 • at the start of the *art* job

²Many programming languages have an idea named *module*; the use of the term *module* by *art* and in this documentation set is an *art*-specific idea.

³*art* defines a few other specialized roles for C++ classes; you will encounter these in Sections 3.6.4 and 3.6.5.

- 1 • at the end of the *art* job
- 2 • at the start of each run
- 3 • at the end of each run
- 4 • at the start of each SubRun
- 5 • at the end of each SubRun



6 For those of you who are familiar with *inheritance* in C++, a module class
 7 (i.e., a “module”) must inherit from one of a few different module *base classes*.
 8 Each module class must override one pure-virtual member function from the
 9 base class and it may override other virtual member functions from the base
 10 class.

11 After *art* completes its initialization phase (intentionally not detailed here), it
 12 performs the following steps:

- 13 1. calls the *constructor*(γ) of every module in the configuration
- 14 2. calls the beginJob *member function*(γ) of every module that provides one
- 15 3. reads one event from the input source, and for that event
 - 16 (a) determines if it is from a run different from that of the previous event
 17 (true for first event in loop)
 - 18 (b) if so, calls the beginRun member function of each module that pro-
 19 vides one
 - 20 (c) determines if the event is from a subRun different from that of the
 21 previous event (true for first event in loop)
 - 22 (d) if so, calls the beginSubRun member function of each module that
 23 provides one
 - 24 (e) calls each module’s (required) per-event member function
- 25 4. moves to the next event and repeats the above per-event steps until it
 26 encounters a new subRun
- 27 5. closes out the current subRun by calling the endSubRun method of each
 28 module that provides one
- 29 6. repeats steps 4 and 5 until it encounters a new run
- 30 7. closes out the current run by calling the endRun method of each module
 31 that provides one
- 32 8. repeats steps 3 through 7 until it reaches the end of the source
- 33 9. calls the endJob method of each module that provides one
- 34 10. calls the *destructor*(γ) of each module

1 This entire set of steps comprises the event loop. Note that any given source
 2 file may contain runs, subRuns and/or events that are not contiguous; “next”
 3 in the above means “next in the file,” not necessarily the next numerically. And
 4 when one file is closed and a new one opened, the “next” event can be anything.
 5 One of *art*’s most visible jobs is controlling the event loop.



6 3.6.3 Module Types

7 Every *art* module must be one of the following five types, which are defined
 8 by the ways in which they interact with each event and with the event loop:

10 ***analyzer module*(γ)** May inspect information found in the event but may
 11 not add new information to the event; described in Chapter 27

12 ***producer module*(γ)** May inspect information found in the event and may
 13 add new information to the event; described in Chapter 26

14 ***filter module*(γ)** Same functions as a Producer module but may also tell
 15 *art* to skip the processing of some, or all, modules for the current event;
 16 may also control which events are written to which output; described in
 17 Chapter 28.

18 ***source module*(γ)** Reads events, one at a time, from some source; *art* re-
 19 quires that every *art* job contain exactly one source module. A source
 20 is often a disk file but other options exist and will be described in the
 21 Workbook and Users Guide.

22 ***output module*(γ)** Reads an event from memory and writes it to an output;
 23 an *art* job may contain zero or more output modules. An output is often
 24 a disk file but other options exist and will be described in the Workbook
 25 and in

26 Note that no module may change information that is already present in an event.



28 What does an analyzer do if it may neither alter information in an event nor
 29 add to it? Typically it creates printout and it creates ROOT files containing
 30 histograms, *trees*(γ) and *nuples*(γ) that can be used for downstream analysis.
 31 (If you have not yet encountered these terms, the Workbook will provide expla-
 32 nations as they are introduced.)

33 Most beginners will only write analyzer modules and filter modules; readers
 34 with a little more experience may also write producer modules. The Workbook
 35 will provide examples of all three. Few people other than *art* experts and each
 36 experiment’s software experts will write source or output modules, however, the
 37 Workbook will teach you what you need to know about configuring source and
 38 output modules.

3.6.4 *art* Data Products

This section introduces more ideas and terms dealing with event information that you will need as you progress through the Workbook.

The word *data product*(γ) is used in *art* to mean the unit of information that user code may add to an event or retrieve from an event. A typical experiment will have the following sorts of data products:

1. The DAQ system will package the raw data into data products, perhaps one or two data products for each major subsystem.
2. Each module in the reconstruction chain will create one or more data products.
3. Some modules in the analysis chain will produce data products; others may just make histograms and write information in non-*art* formats for analysis outside of *art*; they may, for example, write user defined ROOT TTrees.
4. The simulation chain will usually create many data products that describe properties of the simulated event; these data products can be used to develop, debug and characterize the reconstruction algorithms.

Because these data products are intrinsically experiment dependent, each experiment defines its own data products. In the Workbook, you will learn about a set of data products designed for use with the toy experiment. There are a small number of data products that are defined by *art* and that hold bookkeeping information; these will be described as you encounter them in the Workbook.



A data product is just a C++ *type*(γ) (a class, *struct*(γ) or typedef) that obeys a set of rules defined by *art*; these rules are very different than the rules that must be followed for a class to be a module. A data product can be a single integer, an large complex class hierarchy, or anything in between.

Very often, a data product is a *collection*(γ) of some experiment-defined type. The C++ standard libraries define many sorts of collection types; *art* supports many of these and also provides a custom collection type named `cet::map_vector`. Workbook exercises will clarify the *data product* and *collection type* concepts.

3.6.5 *art* Services

Previous sections of this Introduction have introduced the concept of C++ classes that have to obey a certain set of rules defined by *art*, in particular, modules in Section 3.6.2 and data products in Section 3.6.4. *art services*(γ) are yet another example of this.

1 In a typical *art* job, two sorts of information need to be shared among the
 2 modules. The first sort is stored in the data products themselves and is passed
 3 from module to module via the event. The second sort is not associated with
 4 each event, but rather is valid for some aggregation of events, subRuns or runs,
 5 or over some other time interval. Three examples of this second sort include
 6 the geometry specification, the conditions information⁴ and, for simulations, the
 7 table of particle properties.

8 To provide managed access to the second sort of information, *art* supports an
 9 idea named *art services* (again, shortened to *services*). Services may also be
 10 used to provide certain types of utility functions. Again, a service in *art* is just
 11 a C++ class that obeys a set of rules defined by *art*. The rules for services are
 12 different than those for modules or data products.

13 *art* implements a number of services that it uses for internal functions, a few
 14 of which you will encounter in the first couple of Workbook exercises. The
 15 *message service*(γ) is used by both *art* and experiment-specific code to limit
 16 printout of messages with a low severity level and to route messages to different
 17 destinations. It can be configured to provide summary information at the end of
 18 the *art* job. The *TFileService*(γ) and the *RandomNumberGenerator* service
 19 are not used internally by *art*, but are used by most experiments. Experiments
 20 may also create and implement their own services.

21 After *art* completes its initialization phase and before it constructs any modules
 22 (see Section 3.6.2), it

- 23 1. reads the configuration to learn what services are requested
- 24 2. calls the constructor of each requested service

25 Once a service has been constructed, any code in any module can ask *art* for
 26 a *smart pointer*(γ) to that service and use the features provided by that ser-
 27 vice. Similarly, services are available to a module as soon as the module is
 28 constructed.

29 It is also legal for one service to request information from another service as
 30 long as the dependency chain does not have any loops. That is, if Service
 31 A uses Service B, then Service B may not use Service A, either directly or
 32 indirectly.

33 For those of you familiar with the C++ Singleton Design Pattern, an *art* service
 34 has some differences and some similarities to a Singleton. The most important
 35 difference is that the lifetime of a service is managed by *art*, which calls the con-
 36 structors of all services at a well-defined time in a well-defined order. Contrast
 37 this with the behavior of Singletons, for which the order of initialization is un-
 38 defined by the C++ standard and which is an accident of the implementation



⁴The phrase “conditions information” is the currently fashionable name for what was once called “calibration constants;” the name change came about because most calibration information is intrinsically time-dependent, which makes “constants” a poor choice of name.

1 details of the loader. *art* also includes services under the umbrella of its power-
 2 ful run-time configuration system; in the Singleton Design pattern this issue is
 3 simply not addressed.

4 3.6.6 Shareable Libraries and *art*

5 When code is executed within the *art* framework, *art*, not the experiment,
 6 provides the main executable. The experiment provides its code to the *art*
 7 executable in the form of shareable object libraries that *art* loads dynamically
 8 at run time; these libraries are also called *dynamic load libraries* or *plugins*
 9 and their filenames are required to end in *.so*. For more information about
 10 shareable libraries, see Section 31.5.

11 3.6.7 Build Systems and *art*

12 To make an experiment's code available to *art*, the source code must be compiled
 13 and linked (i.e., *built*) to produce shareable object libraries (Section 3.6.6).
 14 The tool that creates the *.so* files from the C++ source files is called a *build*
 15 *system*(γ).

16 Experiments that use *art* are free to choose their own build systems, as long as
 17 the system follows the conventions that allow *art* to find the name of the *.so*
 18 file given the name of the module class. The Workbook will use a build system
 19 named *cetbuildtools*, which is a layer on top of *cmake*⁵.

20 The ***cetbuildtools*** system defines three standard compiler optimization levels,
 21 called “debug”, “profile” and “optimized”; the last two are often abbreviated
 22 “prof” and “opt”. When code is compiled with the “opt” option, it runs as
 23 quickly as possible but is difficult to debug. When code is compiled with the
 24 “debug” option, it is much easier to debug but it runs more slowly. When code
 25 is compiled with the “prof” option the speed is almost and fast as for an “opt”
 26 build and the most useful subset of the debugging information is retained. The
 27 “prof” build retains enough debugging information that one may use a profiling
 28 tool to identify in which functions the program spends most of its time; hence
 29 its name “profile”.



30 The compiler options corresponding to the three levels are listed in Table 3.1.
 31

32 3.6.8 External Products

33 As you progress through the Workbook, you will see that the exercises use some
 34 software packages that are part of neither *art* nor the toy experiment's code.

⁵***cetbuildtools*** is also used to build *art* itself.

Table 3.1: Compiler flags for the optimization levels defined by **cetbuildtools**; compiler options not related to optimization or debugging are not included in this table.

Name	flags
debug	-O0 -g
prof	-O3 -g -fno-omit-frame-pointer -DNDEBUG
opt	-O3 -DNDEBUG

1 The Workbook code, *art* and the software for your experiment all rely heavily
 2 on some external tools and, in order to be an effective user of *art*-based HEP
 3 software, you will need at least some familiarity with them; you may in fact
 4 need to become expert in some.

5 These packages and tools are referred to as *external products*(γ) (sometimes
 6 called simply *products*).

7 An initial list of the products you will need to become familiar with includes:

8 ***art*** the event processing framework

9 **FHiCL** the run-time configuration language used by *art*

10 **CETLIB** a utility library used by *art*

11 ***MF***(γ) a message facility that is used by *art* and by (some) experiments that
 12 use *art*

13 **ROOT** an analysis, data presentation and data storage tool widely used in
 14 HEP

15 ***CLHEP***(γ) a set of utility classes; the name is an acronym for *Class Library*
 16 *for HEP*

17 ***boost***(γ) a class library with new functionality that is being prototyped for
 18 inclusion in future C++ standards

19 **gcc** the GNU C++ compiler and run-time libraries; both the core language and
 20 the standard library are used by *art* and by your experiment's code.

21 ***git***(γ) a source code management system that is used for the Workbook and
 22 by some experiments; similar in concept to the older CVS and SVN, but
 23 with enhanced functionality

24 ***cetbuildtools***(γ) a Fermilab-developed external product that contains build-
 25 tool and related tools

26 ***UPS***(γ) a Fermilab-developed system for accessing software products; it is an
 27 acronym for *Unix Product Support*.

28 ***UPD***(γ) a Fermilab-developed system for distributing software products; it
 29 is an acronym for *Unix Product Distribution*.

1 *jobsub_tools*(γ) tools for submitting jobs to the Fermigrid batch system and
 2 monitoring them.

3 *ifdh_sam*(γ) allows *art* to use SAM as an external run-time agent that can
 4 deliver remote files to local disk space and can copy output files to tape.
 5 SAM is a Fermilab-supplied resource that provides the functions of a file
 6 catalog, a replica manager and some functions of a batch-oriented workflow
 7 manager x

8 Any particular line of code in a Workbook exercise may use elements from, say,
 9 four or five of these packages. Knowing how to parse a line and identify which
 10 feature comes from which package is a critical skill. The Workbook will provide
 11 a tour of the above packages so that you will recognize elements when they are
 12 used and you will learn where to find the necessary documentation.

13 The external products are made available to your code via a mechanism called
 14 UPS, which will be described in Section 7. UPS is, itself, just another external
 15 product. From the point of view of your experiment, *art* is an external product.
 16 From the point of view of the Workbook code, both *art* and the code for the
 17 toy experiment are external products.



18 Finally, it is important to recognize an overloaded word, *products*. When a
 19 line of documentation simply says *products*, it may be referring either to data
 20 products or to external products. If it is not clear from the context which is
 21 meant, please let us know (see Section 3.4).

22 3.6.9 The Event-Data Model and Persistency

23 Section 3.6.4 introduced the idea of *art* data products. In a small experiment,
 24 a fully reconstructed event may contain on the order of ten data products; in a
 25 large experiment there may be hundreds.

26 While each experiment will define its own data product classes, there are many
 27 ideas that are common to all data products in all experiments:

- 28 1. How does my module access data products that are already in the event?
- 29 2. How does my module publish a data product so that other modules can
 30 see it?
- 31 3. How is a data product represented in the memory of a running program?
- 32 4. How does an object in one data product refer to an object in another data
 33 product?
- 34 5. What metadata is there to describe each data product?
- 35 Such metadata might include: which module created it; what was the
 36 run-time configuration of that module; what data products were read by
 37 that module; what was the code version of the module that created it?

1 6. How does my module access the metadata associated with a particular
2 data product?

3 The answers to these questions form what is called the *Event-Data Model*(γ)
4 (EDM) that is supported by the framework.

5 A question that is closely related to the EDM is: what technologies are sup-
6 ported to write data products from memory to a disk file and to read them
7 from the disk file back into memory in a separate *art* job? A framework may
8 support several such technologies. *art* currently supports only one disk file for-
9 mat, a ROOT-based format, but the *art* EDM has been designed so that it will
10 be straightforward to support other disk file formats as it becomes useful to do
11 so.

12 A few other related terms that you will encounter include:

- 13 1. *transient representation*: the in-memory representation of a data product
- 14 2. *persistent representation*: the on-disk representation of a data product
- 15 3. *persistency*: the technology to convert data products back and forth be-
16 tween their persistent and transient representations

17 3.6.10 Event-Data Files

18 When you read data from an experiment and write the data to a disk file, that
19 disk file is usually called a *data file*.

20 When you simulate an experiment and write a disk file that holds the infor-
21 mation produced by the simulation, what should you call the file? The Par-
22 ticle Data Group has recommended that this not be called a “data file” or a
23 “simulated data file;” they prefer that the word “data” be strictly reserved for
24 information that comes from an actual experiment. They recommend that we
25 refer to these files as “files of simulated events” or “files of Monte Carlo events”
26 ⁶. Note the use of “events”, not “data.”

27 This leaves us with a need for a collective noun to describe both data files and
28 files of simulated events. The name in current use is *event-data files*(γ); yes
29 this does contain the word “data” but the hyphenated word, “event-data”, is
30 unambiguous and this has become the standard name.

31 3.6.11 Files on Tape

32 Many experiments do not have access to enough disk space to hold all of their
33 event-data files, ROOT files and log files. The solution is to copy a subset of

⁶ In HEP almost all simulations codes use *Monte Carlo*(γ) methods; therefore simulated events are often referred to as *Monte Carlo events* and the simulation process is referred to as *running the Monte Carlo*.

1 the disk files to tape and to read them back from tape as necessary.

2 At any given time, a snapshot of an experiment's files will show some on tape
3 only, some on tape with copies on disk, and some on disk only. For any given
4 file, there may be multiple copies on disk and those copies may be distributed
5 across many *sites*(γ), some at Fermilab and others at collaborating laboratories
6 or universities.

7 Conceptually, two pieces of software are used to keep track of which files are
8 where, a *File Catalog* and a *Replica Manager*. At Fermilab, the software that fills
9 both of these roles is called *SAM*(γ), which is an acronym for "Sequential data
10 Access via Metadata." SAM also provides some tools for Workflow management.
11 You can learn more about SAM at: <https://cdcvns.fnal.gov/redmine/projects>

12 The UPS product **ifdh_sam** provides the glue that allows an *art* job to interact
13 with SAM.

14 3.7 The Toy Experiment

15 The Workbook exercises are based around a made-up (*toy*) experiment. The
16 code for the toy experiment is deployed as a UPS product named *toyExperiment*.
17 The rest of this section will describe the physics content of *toyExperiment*; the
18 discussion of the software this product uses will unfold in the Workbook, in
19 parallel to the exposition of *art*.

20 The software for the toy experiment is designed around a toy detector, which is
21 shown in Figure 3.2. The *toyExperiment* code contains many C++ classes: some
22 modules, some data products, some services and some plain old C++ classes.
23 About half of the modules are producers that individually perform either one
24 step of the simulation process or one step of the reconstruction/analysis pro-
25 cess. The other modules are analyzers that make histograms and ntuples of the
26 information produced by the producers.

27 3.7.1 Toy Detector Description

28 The toy detector is a central detector made up of 15 concentric shells, with their
29 axes centered on the z axis; the left hand part of Figure 3.2 shows an xy view of
30 these shells and the right shows the radius vs z view. The inner five shells are
31 closely spaced radially and are short in z ; the ten outer shells are more widely
32 spaced radially and are longer in z . The detector sits in a uniform magnetic
33 field of 1.5 T oriented in the $+z$ direction. The origin of the coordinate system
34 is at the center of the detector. The detector is placed in a vacuum.

35 Each shell is a detector that measures (φ, z) , where φ is the azimuthal angle of a
36 line from the origin to the measurement point. Each measurement has perfectly

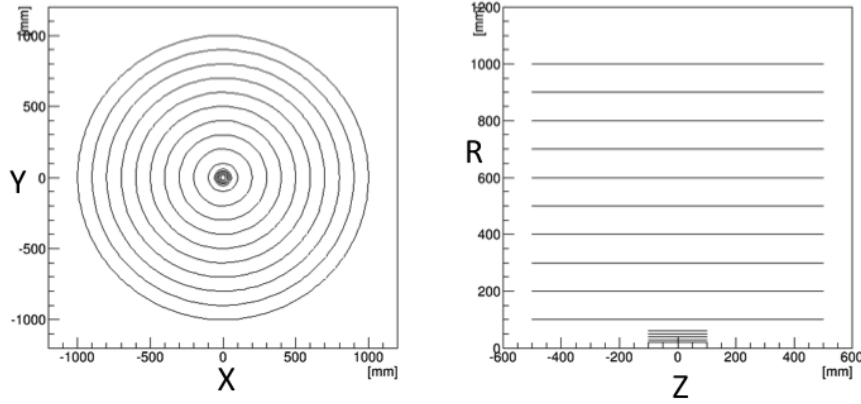


Figure 3.2: The geometry of the toy detector; the figures are described in the text. A uniform magnetic field of strength 1.5 T is oriented in the $+z$ direction.

Table 3.2: Units used in the Workbook

Quantity	Unit
Length	mm
Energy	MeV
Time	ns
Plane Angle	Radian
Solid Angle	Steradian
Electric Charge	Charge of the proton = +1
Magnetic Field	Tesla

1 gaussian measurement errors and the detector always has perfect separation of
 2 hits that are near to each other. The geometry of each shell, its efficiency and
 3 resolution are all configurable at run-time.

4 All of the code in the toyExperiment product works in the set of units described
 5 in Table 3.2. Because the code in the Workbook is built on toyExperiment, it
 6 uses the same units. *art* itself is not unit aware and places no constraints on
 7 which units your experiment may use.

8 The first six units listed in Table 3.2 are the base units defined by the CLHEP
 9 SystemOfUnits package. These are also the units used by Geant4.



10 3.7.2 Workflow for Running the Toy Experiment Code

11 The workflow of the toy experiment code includes five steps: three simulation
 12 steps, a reconstruction step and an analysis step:

- 1 1. event generation
- 2 2. detector simulation
- 3 3. hit-making
- 4 4. track reconstruction
- 5 5. analysis of the mass resolution

6 For each event, the event generator creates one particle with the following prop-
7 erties:

- 8 • Its mass is the rest mass of the ϕ meson; the event generator does not
9 simulate a natural width for this particle.
- 10 • It is produced at the origin.
- 11 • It has a momentum that is chosen randomly from a distribution that is
12 uniform between 0 and 2000 MeV/ c .
- 13 • Its direction is chosen randomly on the unit sphere.

14 The event generator then decays this particle to K^+K^- ; the center-of-mass
15 decay angles are chosen randomly on the unit sphere.

16 In the detector simulation step, particles neither scatter nor lose energy when
17 they pass through the detector cylinders; nor do they decay. Therefore, the
18 charged kaons follow a perfectly helical trajectory. The simulation follows each
19 charged kaon until it either exits the detector or until it completes the outward-
20 going arc of the helix. When the simulated trajectory crosses one of the detector
21 shells, the simulation records the true point of intersection. All intersections
22 are recorded; at this stage in the simulation, there is no notion of inefficiency
23 or resolution. The simulation does not follow the trajectory of the ϕ meson
24 because it was decayed in the generator.

25 Figure 3.3 shows an event display of a typical simulated event. In this event
26 the ϕ meson was travelling almost at 90° to the z axis and it decayed nearly
27 symmetrically; both tracks intersect all 15 detector cylinders. The left-hand
28 figure shows an xy view of the event; the solid lines show the trajectory of the
29 kaons, red for K^+ and blue for K^- ; the solid dots mark the intersections of
30 the trajectories with the detector shells. The right-hand figure shows the same
31 event but in an rz view.

32 Figure 3.4 shows an event display of another simulated event. In this event the
33 K^- is produced with a very shallow trajectory and it does not intersect any
34 detector shells while the K^+ makes five hits in the inner detector and seven in
35 the outer detector. Why does the trajectory of the K^+ end where it does? In
36 order to keep the exercises focused on *art* details, not geometric corner cases,
37 the simulation stops a particle when it completes its outward-going arc and
38 starts to curl back towards the z axis; it does this even if the the particle is still
39 inside the detector.

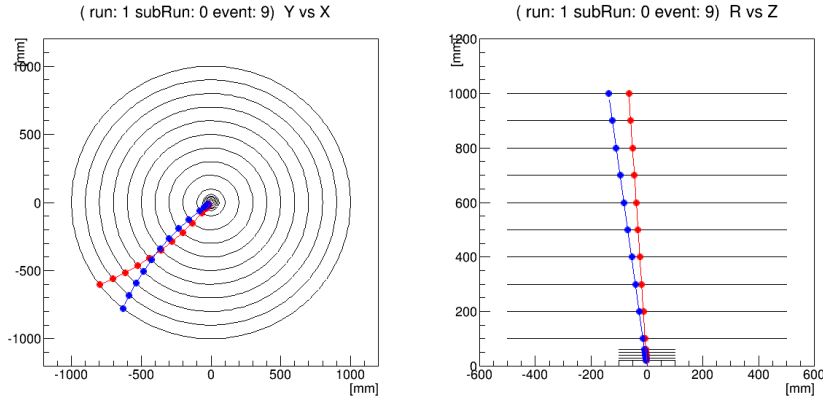
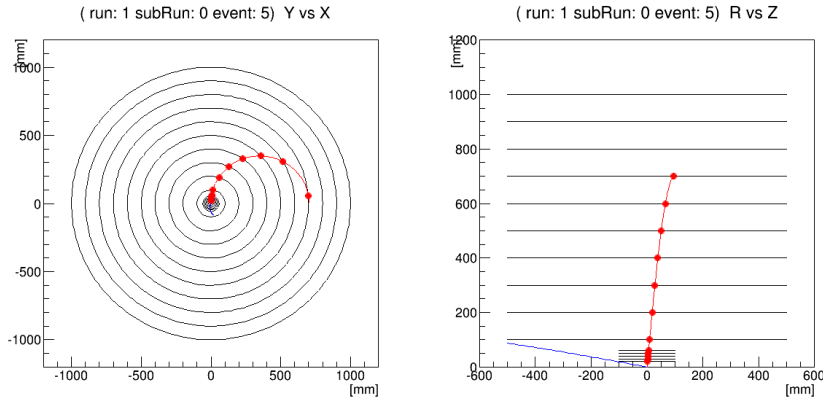


Figure 3.3: Event display of a typical simulated event in the toy detector.

Figure 3.4: Event display of another simulated event in the toy detector; a K^- (blue) is produced with a very shallow trajectory and it does not intersect any detector shells while the K^+ (red) makes five hits in the inner detector and seven in the outer detector

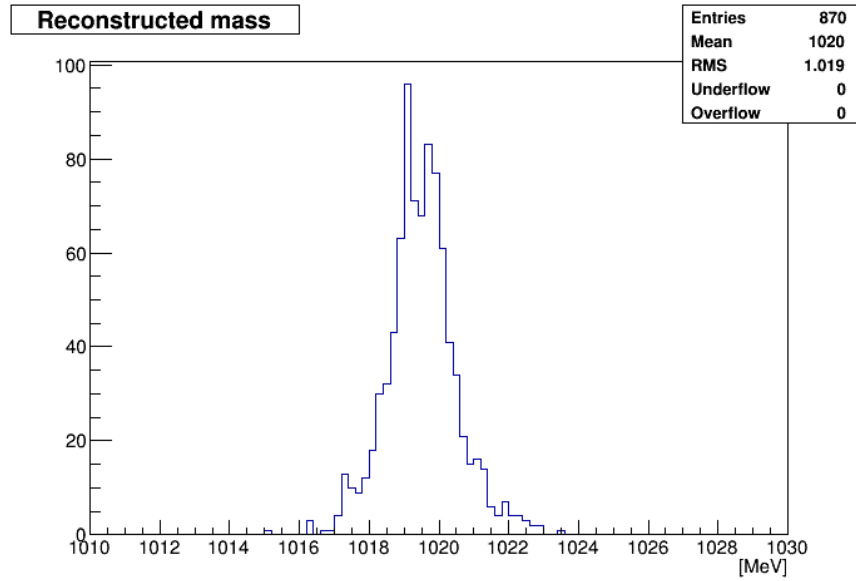


Figure 3.5: The final plot showing 870 reconstructed events out of 1000 generated events

1 The third step in the simulation chain (hit-making) is to inspect the intersections
 2 produced by the detector simulation and turn them into data-like hits. In this
 3 step, a simple model of inefficiency is applied and some intersections will not
 4 produce hits. Each hit represents a 2D measurement (φ, z) ; each component is
 5 smeared with a gaussian distribution.

6 The three simulation steps use tools provided by *art* to record the *truth infor-*
 7 *mation*(γ) about each hit. Therefore it is possible to navigate from any hit back
 8 to the intersection from which it is derived, and from there back to the particle
 9 that made the intersection.

10 The fourth step is the reconstruction step. The *toyExperiment* does not yet
 11 have properly working reconstruction code; instead it mocks up credible look-
 12 ing results. The output of this code is a data product that represents a fitted
 13 helix; it contains the fitted track parameters of the helix, their covariance matrix
 14 and collection of smart pointers that point to the hits that are on the recon-
 15 structed track. When we write proper tracking finding and track fitting code
 16 for the *toyExperiment*, the classes that describe the fitted helix will not change.
 17 Because the main point of the Workbook exercises is to illustrate the bookkeep-
 18 ing features in *art*, this is good enough for the task at hand. The output data
 19 product will contain 0, 1 or 2 fitted helices, depending on how many generated
 20 tracks passed the minimum hits cut.

Part

1 The fifth step in the workflow does a simulated analysis using the fitted helices
2 from the reconstruction step. It forms all distinct pairs of tracks and requires
3 that they be oppositely charged. It then computes the invariant mass of the
4 pair, under the assumption that both fitted helices are kaons. This module
5 is an analyzer module and does not make any output data product. But it
6 does make some histograms, one of which is a histogram of the reconstructed
7 invariant mass of all pairs of oppositely charged tracks; this histogram is shown
8 in Figure 3.5. When you run the Workbook exercises, you will make this plot
9 and can compare it to Figure 3.5.

10 3.8 Rules, Best Practices, Conventions and Style

11 In many places, the Workbook will recommend that you write fragments of code
12 in a particular way, to help you establish coding habits that will make your life
13 easier as you progress in your use of C++ and *art*. The reason for any particular
14 recommendation may be one of the following:

- 15 • It is a hard rule enforced by the C++ language or by one of the external
16 products.
- 17 • It is a recommended best practice that might not save you time or effort
18 now but will in the long run.
- 19 • It is a convention that is widely adopted; C++ is a rich enough language
20 that it will let you do some things in many different ways. Code is much
21 easier to understand and debug if an experiment chooses to always write
22 code fragments with similar intent using a common set of conventions.
- 23 • It is simply a question of style.

24 It is important to be able to distinguish between rules, best practices, conven-
25 tions and styles; this documentation will distinguish among these options when
26 discussing recommendations that it makes.

4 Unix Prerequisites

4.1 Introduction

You will work through the Workbook exercises on a computer that is running some version of the Unix operating system. This chapter describes where to find information about Unix and gives a list of Unix commands that you should understand before starting the Workbook exercises. This chapter also describes a few ideas that you will need immediately but which are usually not covered in the early chapters of standard Unix references.

If you are already familiar with Unix and the *bash*(γ) shell, you can safely skip this chapter.

4.2 Commands

In the Workbook exercises, most of the commands you will enter at the Unix prompt will be standard Unix commands, but some will be defined by the software tools that are used to support the Workbook. The non-standard commands will be explained as they are encountered. To understand the standard Unix commands, any standard Linux or Unix reference will do. Section 4.10 provides links to Unix references.

Most Unix commands are documented via the *man page* system (short for “manual”). To get help on a particular command, type the following at the command prompt, replacing `<command-name>` with the actual name of the command: ¹

```
$ man <command-name>
```

In Unix, everything is case sensitive; so the command `man` must be typed in lower case. You can also try the following; it works on some commands and not

¹Remember that a convention used in this document, is that a command you should type at the command prompt is indicated by a leading dollar sign; but you should not type the leading dollar sign. This was described in Section 2.

1 others:

2 \$ <command-name> --help

3 or

4 \$ <command-name> -?

5 Before starting the Workbook, make sure that you understand the basic usage
6 of the following Unix commands:

7 cat, cd, cp, echo, export, gzip, head,
8 less, ln -s, ls, mkdir, more, mv,
9 printenv, pwd, rm, rmdir, tail, tar

10 You also need to be familiar with the following Unix concepts:

- 11 • filename vs pathname
- 12 • absolute path vs relative path
- 13 • directories and subdirectories (equivalent to folders in the Windows and
14 Mac worlds)
- 15 • current working directory
- 16 • home directory (aka login directory)
- 17 • `../` notation for viewing the directory above your current working direc-
18 tory
- 19 • environment variables (discussed briefly in Section 4.5)
- 20 • *paths*(γ) (in multiple senses; see Section 4.6)
- 21 • file protections (read-write-execute, owner-group-other)
- 22 • symbolic links
- 23 • stdin, stdout and stderr
- 24 • redirecting stdin, stdout and stderr
- 25 • putting a command *in the background* via the `&` character
- 26 • pipes

27 4.3 Shells

28 When you type a command at the prompt, a Unix agent called a *Unix shell*,
29 or simply a *shell*, reads your command and figures out what to do. Some com-
30 mands are executed internally by the shell but other commands are dispatched
31 to an appropriate program or script. A shell lives between you and the under-
32 lying operating system; most versions of Unix support several shells. The *art*

1 Workbook code expects to be run in the *bash shell*. You can see which shell
2 you're running by entering:

3 `$ echo $SHELL`

4 For those of you with accounts on a Fermilab machine, your login shell was
5 initially set to the *bash shell*².

6 If you are working on a non-Fermilab machine and *bash* is not your default shell,
7 consult a local expert to learn how to change your login shell to *bash*.



8 4.4 Scripts: Part 1

9 In order to automate repeated operations, you may write multiple Unix com-
10 mands into a file and tell *bash* to run all of the commands in the file as if you
11 had typed them sequentially. Such a file is an example of a *shell script* or a
12 *bash script*. The *bash* scripting language is a powerful language that supports
13 looping, conditional execution, tests to learn about properties of files and many
14 other features.

15 Throughout the Workbook exercises you will run many scripts. You should
16 understand the big picture of what they do, but you don't need to understand
17 the details of how they work.



18 If you would like to learn more about *bash*, some references are listed in Sec-
19 tion 4.10.

20 4.5 Unix Environments

21 4.5.1 Layering Environments

22 Very generally, a Unix *environment* is a set of information that is made available
23 to programs so that they can find everything they need in order to run properly.
24 The Unix operating system itself defines a generic environment, but often this
25 is insufficient for everyday use. However, an environment sufficient to run a
26 particular set of applications doesn't just pop out of the ether, it must be
27 *established* or *set up*, either manually or via a script. Typically, on institutional
28 machines at least, system administrators provide a set of login scripts that
29 run automatically and enhance the generic Unix environment. This gives users
30 access to a variety of system resources, including, for example:

- 31 • disk space to which you have read access

² If you have had a Fermilab account for many years, your default shell might be something else. If your default shell is not *bash*, open a Service Desk ticket to request that your default shell be changed to *bash*.

- 1 • disk space to which you have write access
- 2 • commands, scripts and programs that you are authorized to run
- 3 • proxies and tickets that authorize you to use resources available over the
- 4 network
- 5 • the actual network resources that you are authorized to use, e.g., tape
- 6 drives and DVD drives

7 This constitutes a basic *working environment* or *computing environment*. En-
 8 vironment information is largely conveyed by means of *environment variables*
 9 that point to various program executable locations, data files, and so on. A
 10 simple example of an environment variable is HOME, the variable whose value is
 11 the absolute path to your home directory.

12 Particular programs (e.g., *art*) usually require extra information (i.e., another
 13 environment *layer*) on top of a standard working environment, e.g., paths to
 14 the program's executable(s) and to its dependent programs, paths indicating
 15 where it can find input files and where to direct its output, and so on. In addi-
 16 tion to environment variables, the *art*-enabled computing environment includes
 17 some aliases and *bash* functions that have been defined; these are discussed in
 18 Section 4.8.

19 In turn, the Workbook code, which must work for all experiments and at Fer-
 20 milab as well as at collaborating institutions, requires yet another environment
 21 layer – a *site-specific* layer.



22 Given the different experiments using *art* and the variety of laboratories and
 23 universities at which the users work, a *site*(γ) in *art* is a unique combination
 24 of *experiment* and *institution*. It is used to refer to a set of computing resources
 25 configured for use by a particular experiment at a particular institution. Setting
 26 up your site-specific environment will be discussed in Section 4.7.

27 When you finish the Workbook and start to run real code, you will set up your
 28 experiment-specific environment on top of the more generic *art*-enabled environ-
 29 ment, in place of the Workbook's. To switch between these two environments,
 30 you will log out and log back in, then run the script appropriate for the environ-
 31 ment you want. Because of potential naming “collisions,” it is not guaranteed
 32 that these two environments can be overlain and always work properly.

33 This concept of environment layering is illustrated in Figure 4.1.

34 4.5.2 Examining and Using Environment Variables

35 One way to see the value of an environment variable is to use the `printenv`
 36 command:

```
37 $ printenv HOME
```

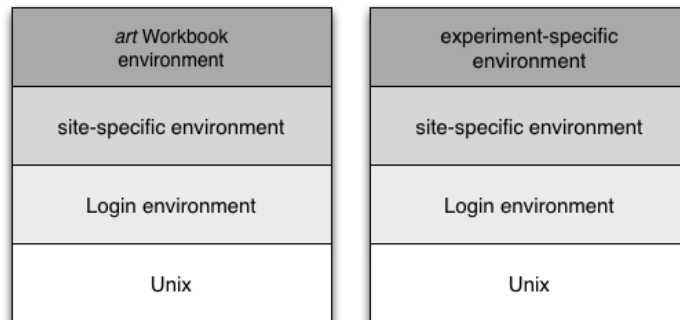


Figure 4.1: Layers in the *art* Workbook (left) and experiment-specific (right) computing environments

1 At any point in an interactive command or in a shell script, you can tell the
 2 shell that you want the value of the environment variable by prefixing its name
 3 with the `$` character:

4 `$ echo $HOME`

5 Here, `echo` is a standard Unix command that copies its arguments to its output,
 6 in this case the screen.

7 By convention, environment variables are virtually always written in all capital
 8 letters³.

9 There may be times when the Workbook instructions tell you to set an envi-
 10 ronment variable to some value. To do so, type the following at the command
 11 prompt:

12 `$ export <ENVNAME>=<value>`

13 If you read *bash* scripts written by others, you may see the following variant,
 14 which accomplishes the same thing:

15 `$ <ENVNAME>=<value>`

16 `$ export <ENVNAME>`

³Another type of variable, *shell variables*, are local to the currently-invoked shell and go away when the shell exits. By convention, these are written in lower or mixed case. These conventions provide a clue to the programmer as to whether changing a variable's value might have consequences outside the current shell.

4.6 Paths and \$PATH

Path (and *PATH*) is an overloaded word in computing. Here are the ways in which it is used:

path can refer to the location of a file or a directory; a path may be absolute or relative, e.g.
 /absolute/path/to/mydir/myfile or
 relative/path/on/same/branch/to/mydir/myfile or
 ../relative/path/on/different/branch/to/herdir/herfile

PATH refers to the standard Unix environment variable set by your login scripts and updated by other scripts that extend your environment; it is a colon-separated list of directory names, e.g.,
 /usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin.
 It contains the list of directories that the shell searches to find programs/files required by Unix shell commands (i.e., **PATH** is used by the shell to “resolve” commands).

path generically, any environment variable whose value is a colon-separated list of directory names e.g.,
 /abs/path/a:/abs/path/b:rel/path/c

In addition, *art* defines a fourth idea, also called a path, that is unrelated to any of the above; it will be described as you encounter it in the Workbook.

All of these *path* concepts are important to users of *art*. In addition to **PATH** itself, there are three **PATH**-like environment variables (colon-separated list of directory names) that are particularly important:

LD_LIBRARY_PATH used by *art* to resolve shareable libraries

PRODUCTS used by *UPS* to resolve external products

FHICL_FILE_PATH use by *FHiCL* to resolve `#include` directives.

When you source the scripts that setup your environment for *art*, these will be defined and additional colon-separated elements will be added to your **PATH**. You can look at the value of **PATH** (or the others):

```
$ printenv PATH
```

You can make the output easier to read by replacing all of the colons with newline characters:

```
$ printenv PATH | tr : \\n
```

In the above line, the vertical bar is referred to as a *pipe* and `tr` is a standard Unix command. A pipe takes the output of the command to its left and makes that the input of the command to its right. The `tr` command replaces patterns of characters with other patterns of characters; in this case it replaces every

1 occurrence of the colon character with the newline character. To learn why a
2 double back slash is needed, read bash documentation to learn about escaping
3 special characters.

4 4.7 Scripts: Part 2

5 There are two ways to run a bash script (actually three, but two of them are
6 the same). Suppose that you are given a bash script named `file.sh`. You can
7 do any of:

```
8 $ file.sh  
9 $ source file.sh  
10 $ . file.sh
```

11 The first version, `file.sh`, starts a new bash shell, called a subshell, and it
12 executes the commands from `file.sh` in that subshell; upon completion of
13 the script, control returns to the parent shell. At the startup of a subshell, the
14 environment of that subshell is initialized to be a copy of the environment of
15 its parent shell. If `file.sh` modifies its environment, then it will modify only
16 the environment of the subshell, leaving the environment of the parent shell
17 unchanged. This version is called *executing* the script.

18 The second and third versions are equivalent. They do not start a subshell;
19 they execute the commands from `file.sh` in your current shell. If `file.sh`
20 modifies any environment variables, then those modifications remain in effect
21 when the script completes and control returns to the command prompt. This
22 is called *sourcing* the script.

23 Some shell scripts are designed so that they must be sourced and others are
24 designed so that they must be executed. Many shell scripts will work either
25 way.

26 If the purpose of a shell script is to modify your working environment then it
27 must be sourced, not executed. As you work through the Workbook exercises,
28 pay careful attention to which scripts it tells you to source and which to execute.
29 In particular, the scripts to setup your environment (the first scripts you will
30 run) are bash scripts that must be sourced because their purpose is to configure
31 your environment so that it is ready to run the Workbook exercises.



32 Some people adopt the convention that all bash scripts end in `.sh`; others adopt
33 the convention that only scripts designed to be sourced end in `.sh` while scripts
34 that must be executed have no file-type ending (no “`.something`” at the end).
35 Neither convention is uniformly applied either in the Workbook or in HEP in
36 general.

37 If you would like to learn more about bash, some references are listed in Sec-
38 tion 4.10.

4.8 bash Functions and Aliases

The bash shell also has the notion of a *bash function*. Typically bash functions are defined by sourcing a bash script; once defined, they become part of your environment and they can be invoked as if they were regular commands. The `setup <product> “command”` that you will sometimes need to issue, described in Chapter 7, is an example. A bash function is similar to a bash script in that it is just a collection of bash commands that are accessible via a name; the difference is that bash holds the definition of a function as part of the environment while it must open a file every time that a bash script is invoked.

You can see the names of all defined functions with the bash command

```
$ declare -F
```

The bash shell also supports the idea of *aliases*; this allows you to define a new command in terms of other commands. You can see the definition of all aliases with the bash command

```
$ alias
```

You can read more about bash functions and aliases in any standard bash reference.

When you type a command at the command prompt, bash will resolve the command using the following order:

1. Is the command a known alias?
2. Is the command a bash keyword, such as `if` or `declare`?
3. Is the command a shell function?
4. Is the command a shell built-in command?
5. Is the command found in `$PATH`?

To learn how bash will resolve a particular command, give the bash command:

```
$ type <command-name>
```

4.9 Login Scripts

When you first login to a computer running the Unix operating system, the system will look for specially named files in your home directory that are scripts to set up your working environment; if it finds these files it will source them before you first get a shell prompt. As mentioned in Section 4.5, these scripts

- 1 modify your `PATH` and define `bash` functions, aliases and environment variables.
 2 All of these become part of your environment.
- 3 When your account on a Fermilab computer was first created, you were given
 4 standard versions of the files `.profile` and `.bashrc`; these files are used by
 5 `bash`⁴. You can read about login scripts in any standard `bash` reference. You
 6 may add to these files but you should not remove anything that is present.
- 7 If you are working on a non-Fermilab computer, inspect the login scripts to
 8 understand what they do.
- 9 It can be useful to inspect the login scripts of your colleagues to find useful
 10 customizations.
- 11 If you read generic Unix documentation, you will see that there are other login
 12 scripts with names like, `.login`, `.cshrc` and `.tcshrc`. These are used by
 13 the `csh` family of shells and are not relevant for the Workbook exercises, which
 14 require the `bash` shell.



15 4.10 Suggested Unix and `bash` References

16 The following cheat sheet provides some of the basics:

- 17 • <http://mu2e.fnal.gov/atwork/computing/UnixHints.shtml>

18 A more comprehensive summary is available from:

- 19 • <http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/GNU-Linux-Tools-Summary.html>

21 Information about writing `bash` scripts and using `bash` interactive features can
 22 be found in:

- 23 • BASH Programming - Introduction HOW-TO
 24 <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- 25 • Bash Guide for Beginners
 26 <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html>
- 27 • Advanced Bash Scripting Guide
 28 <http://www.tldp.org/LDP/abs/html/abs-guide.html>

29 The first of these is a compact introduction and the second is a more compre-
 30 hensive introduction.

31 The above guides were all found at the Linux Documentation Project: Work-
 32 book:

- 33 • <http://www.tldp.org/guides.html>

⁴These files are used by the `sh` family of shells, which includes `bash`.

5 Site-Specific Setup Procedure

Section 4.5 discussed the notion of a working environment on a computer. This chapter answers the question: How do I make sure that my environment variables are set correctly to run the Workbook exercises or my experiment's code using *art*?

Very simply, on every computer that hosts the Workbook, a procedure must be established that every user is expected to follow once per login session. In most cases (NO ν A being a notable exception), the procedure involves only sourcing a shell script (recall the discussion in Section 4.7). In this documentation, we refer to this procedure as the “site-specific setup procedure.” It is the responsibility of the people who maintain the Workbook software for each *site*(γ) to ensure that this procedure does the right thing on all the site's machines.

As a user of the Workbook, you will need to know what the procedure is and you must remember to follow it each time that you log in.



For all of the Intensity Frontier experiments at Fermilab, the site-specific setup procedure defines all of the environment variables that are necessary to create the working environment for either the Workbook exercises or for the experiment's own code.



Table 5.1 lists the site-specific setup procedure for each experiment. You will follow the procedure when you get to Section 9.4.

Table 5.1: Site-specific setup procedures for $IF(\gamma)$ Experiments at Fermilab; for the equivalent procedure at a non-Fermi site, consult an expert from that site. For the NO ν A experiment, the procedure is a sequence of three commands. For all others it is a single command, but in a few cases the command is so long that the table shows it split over two lines: you must type it as a single command on one line; the experiments whose command is so split are ArgoNeut, LBNE and MicroBoone.

Experiment	Site-Specific Setup Command
ArgoNeut	<code>source /grid/fermiapp/argoneut/code/t962soft/setup/ setup_t962_fnal.sh -r art_workbook -b prof</code>
Darkside	<code>source /ds50/app/ds50/ds50.sh</code>
LBNE	<code>source /grid/fermiapp/lbne/lar/code/larsoft/setup/ setup_larsoft_fnal.sh -r art_workbook -b prof</code>
MicroBoone	<code>source /grid/fermiapp/lbne/lar/code/larsoft/setup/ setup_larsoft_fnal.sh -r art_workbook -b prof</code>
Muon g-2	<code>source /gm2/app/software/prod/g-2/setup</code>
Mu2e	<code>setup mu2e</code>
NO ν A	<code>source /grid/fermiapp/nova/novaart/novasvn/srt/srt.sh export EXTERNALS=/nusoft/app/externals source \$SRT_DIST/setup/setup_novasoft.sh -b maxopt</code>

6 Get your C++ up to Speed

6.1 Introduction

3 ,

4 There are two goals for this chapter. The first is to illustrate the features of
5 C++ that will be important for users of the Workbook, especially those features
6 that will be used in the first few Workbook exercises. It does not attempt to
7 cover C++ comprehensively and it delegates as much as possible to the standard
8 documentation.

9 The second goal is to explain the process of turning source code files into an
10 *executable program*. The two steps in this process are *compiling* and *linking*. In
11 informal writing, the word *build* is sometimes used to mean just compiling or
12 just linking, but usually it refers to the two together.

13 A typical program consists of many source code files, each of which contains a
14 human-readable description of one component of the program. In the Workbook,
15 you will see source code files written in the C++ computer language; these
16 files have names that end in `.cc`. In C++, there is a second sort of source code
17 file, called a *header file* that ends in `.h`; in most, but not all, cases for every
18 file ending in `.cc` there is another file with the same name but ending in `.h`.
19 Header files can be thought of as the “parts list” for the corresponding `.cc` file;
20 you will see how these are used in Section 6.4.

21 In the compilation step each `.cc` file is translated into *machine code*, also called
22 *binary code* or *object code*, which is a set of instructions, in the computer’s
23 native language, to do the tasks described by the source code. The output of
24 the compilation step is called an *object file*; in the examples you will see in the
25 Workbook, object files always end in `.o`. But an object file, by itself, is not
26 an *executable program*. It is not executable because each `.o` file was created in
27 isolation and does not know about the other `.o` files.

28 It is often convenient to collect related groups of `.o` files and put them into
29 *libraries*. There are two kinds of library files, static libraries, whose names
30 end in `.a` and shared libraries whose names end in `.so`. Putting many `.o` files

1 into a single library allows you to use them as a single coherent entity. We will
2 defer further discussion of libraries until more background information has been
3 provided.

4 The job of the *linking* step is to read the information found in the various
5 libraries and `.o` files and form them into an *executable program*. When you run
6 the linker, you tell it the name of the file into which it will write the executable
7 program. It is a common, but not universal, practice that the filename of an
8 executable program has no extension (i.e. no `.something` at the end).

9 After the linker has finished, you can run your executable program typing the
10 filename of the program at the bash command prompt.

11 A typical program links both to libraries that were built from the program's
12 source code and to libraries from other sources. Some of these other libraries
13 might have been developed by the same programmer as general purpose tools
14 to be used by his or her future programs; other libraries are provided by third
15 parties, such as *art* or your experiment. Many C++ language features are
16 made available to your program by telling the linker to use libraries provided
17 by the C++ compiler vendor. Other libraries are provided by the operating
18 system.

19 Now that you know about libraries,, we can give a second reason why an object
20 file, by itself, is not an executable program: until it is linked, it does not have
21 access to the functions provided by any of the external libraries. Even the
22 simplest program will need to be linked against some of the libraries supplied
23 by the compiler vendor and by the operating system.

24 The names of all of the libraries and object files that you give to the linker is
25 called the *link list*.

26 This chapter is designed around a handful of exercises, each of which you will
27 first build and run, then “pick apart” to understand how the results were ob-
28 tained.

29 6.2 Establishing the Environment

30 6.2.1 Initial Setup

31 To start these exercises for the first time, do the following:

- 32 1. Log into the node that you will use for Workbook exercises.
- 33 2. Follow the site-specific setup procedure from Table 5.1.
- 34 3. Create an empty working directory and `cd` to it.
- 35 4. Run these commands to copy a gzipped tar file from the web, unpack it,
36 and get a directory listing:

```
1      $ wget http://artdoc.fnal.gov/C++UpToSpeed.tar.gz
2      $ tar xzf C++UpToSpeed.tar.gz
3      $ rm C++UpToSpeed.tar.gz
4      $ ls
5      BasicSyntax Build Classes Libraries setup.sh
6
7      5. Source the setup.sh script to select the correct compiler version and
8         define a few environment variables that will be used later in these exercises:
9         $ source setup.sh
```

9 After these steps, you are ready to begin the exercise in Section 6.3.

10 6.2.2 Subsequent Logins

11 If you log out and log back in again, reestablish your environment by following
12 these steps:

- 13 1. Log into the node that you will normally use.
- 14 2. Follow the site-specific setup procedure.
- 15 3. cd to the working directory you created in Section 6.2.1.
- 16 4. \$ source setup.sh
- 17 5. cd to the directory that contains the exercise you want to work on.

18 6.3 C++ Exercise 1: The Basics

19 6.3.1 Concepts to Understand

20 This section provides a program that illustrates the parts of C++ that are
21 assumed knowledge for the Workbook material. If you do not understand some
22 of the code in this example program, consult any standard C++ reference;
23 several are listed in Section 6.7.

24 Once you have understood this example program, you should understand the
25 following concepts:

- 26 1. how comments are indicated
- 27 2. what is a main program
- 28 3. how to write a C++ main program
- 29 4. how to compile, link and run the main program
- 30 5. how to distinguish between source, object and executable files
- 31 6. how to print to standard output, `std::cout`

- 1 7. how to declare and define *variables*(γ) of the some of the frequently used
- 2 built-in types: `int`, `float`, `double`, `bool`
- 3 8. the `{}` initializer syntax
- 4 9. assignment to variables
- 5 10. C++ arrays
- 6 11. several different forms of looping
- 7 12. comparisons: `==`, `!=`, `<`, `>`, `>=`, `<=`
- 8 13. `if-then-else`, `if-then-else if-else`
- 9 14. pointers
- 10 15. references
- 11 16. `std::string` (a type from the C++ Standard Library (*std*(γ))
- 12 17. the class template from the standard library, `std::vector<T>`

13 Regarding the last item, `std::vector<T>`, you need to know how to use it,
 14 but you do not need to understand how it works or how to write your own
 15 templates.

16 The above list explicitly does not include classes, objects and inheritance, which
 17 will be discussed in Sections 6.6 and 31.9.

18 6.3.2 How to Compile, Link and Run

19 In this section you will learn how to compile, link and run the small C++
 20 program that illustrates the features of C++ that are considered prerequisites.
 21 The main discussion of the details of compiling and linking will be deferred until
 22 Section 6.4.

23 We don't offer a lot of details up front; more will follow in Sections 6.3.5
 24 and 6.3.4. The idea here is to get used to the steps and see what results you
 25 get.

26 To compile, link and run the sample C++ program, called `t1`:

- 27 1. If not yet done, log in and establish the working environment (Section 6.2).
- 28 2. List the starting set of files:
- 29 `$ cd BasicSyntax/v1/`
- 30 `$ ls`
- 31 `build t1.cc t1.example.log`

32 The file `t1.cc` contains the source code of the *main program*, which is a
 33 function called `main()` { ...}. The file `build` is a script that will
 34

```
1      compile and link the code. The file t1.example.log is an example of
2      the output expected when you run t1.

3      3. Compile and link the code; then look at a directory listing:
4      $ build
5      t1.cc: In function int main():
6      t1.cc:43:26: warning: k may be used uninitialized in
7      this function [-Wuninitialized]
8      $ ls
9      build t1 t1.cc t1.example.log
10
11     The script named build compiles and links the code, and produces the
12     executable file t1. The warning message, issued by the compiler, also
13     comes during this step.

14     4. Run the executable file sending output to a log file:
15     $ ./t1 > t1.log
```

6.3.3 Suggested Homework

```
17     1. Compare your output with the standard example:
18     $ diff t1.log t1.example.log
19
20     There will almost certainly be a handful of differences.

21     2. Look at the file t1.cc and understand what it does, in particular the
22     relationship between the lines in the program and the lines in the output.

23     If you don't understand something, consult a standard C++ reference; see Sec-
24     tion 6.7. A few of your questions might also be answered in Section 6.3.4.
```

6.3.4 Discussion

```
26     Why do we expect several of the lines of the output to be different from those
27     in t1.example.log? There are two classes of answers: (1) an uninitialized
28     variable and (2) variation in variable addresses from run to run.

29     In t1.cc, the line

30     int k;

31     declares that k is a variable whose type is int but it does not initialize the
32     variable. Therefore the value of the variable k is whatever value happened to
33     be sitting in the memory location that the program assigned to k. Each time
34     that the program runs, the operating system will put the program into whatever
35     region of memory makes sense to the operating system; therefore the address of
```

1 any variable, and thus the value returned, may change unpredictably from run
2 to run.

3 This line is also the source of the warning message produced by the `build` script.
4 This line was included to make it clear what we mean by *initialized* variables and
5 *uninitialized* variables. Uninitialized variables are frequent sources of errors in
6 code and therefore you should *always* initialize your variables. In order to help
7 you establish this good coding habit, the remaining exercises in this series and
8 in the Workbook include the compiler option `-Werror`. This tells the compiler
9 to promote warning messages to error level and to stop compilation without
10 producing an output file.

11 The second line that may differ from one run to the next is:

```
12 float *pa=&a;
```

13 This line declares a variable `pa`, which is of type *pointer*(γ) to float, and it
14 initializes this variable to be the memory address of the variable `a` (`a` must
15 be of type float). Since the address may change from run to run, so may the
16 printout that starts `pa =`.

17 For similar reasons, the lines in the printout that start `&a =` and `&ra =` may
18 also change from run to run.

19 6.3.5 How was this Exercise Built?

20 Just to see how the exercise was built, look at the script `BasicSyntax/v1/build`
21 that you ran to compile and link `t1.cc`; the following command was issued:

```
22 c++ -Wall -Wextra -pedantic -std=c++11 -o t1 t1.cc
```

23 This turned the source file `t1.cc` into an executable program, named `t1` (the
24 argument to the `-o` (for “output”) option). We will discuss compiling and
25 linking in Section 6.4.

26 6.4 C++ Exercise 2: About Compiling and Link- 27 ing

28 6.4.1 What You Will Learn

29 In the previous exercise, the entire program was found in a single file and the
30 build script performed compiling and linking in a single step. For all but the
31 smallest programs, this is not practical. It would mean, for example, that
32 you would need to recompile and relink everything when you made even the
33 smallest change anywhere in the code; generally this would take much too long.
34 To address this, some computer languages, including C++, allow you to break

1 up a large program into many smaller files and rebuild only a small subset of
 2 files when you make changes in one.

3 There are two exercises in this section. In the first one the source code consists
 4 of three files. This example has enough richness to discuss the details of what
 5 happens during compiling and linking, without being overwhelming. The second
 6 exercise introduces the ideas of libraries and external packages.

7 6.4.2 The Source Code for this Exercise

8 The source code for this exercise is found in `Build/v1`, relative to your working
 9 directory. The relevant files are

10 `function.cc` `function.h` `t1.cc`

11 The file `t1.cc` is the file that contains the source code for the function `main()`
 12 `{ ... }` for this exercise. Every C++ program must have one and only one
 13 function named `main`, which is where the program actually starts execution.
 14 Note that the term *main program* sometimes refers to this function, but other
 15 times refers to the `.cc` file that contains it. In either case, *main program* refers
 16 to this function, either directly or indirectly. For more information, consult any
 17 standard C++ reference. The file `function.h` is a header file that declares a
 18 function named `function`. The file `function.cc` is another source code file;
 19 it provides the definition of that function.



20 Look at `t1.cc`: it both declares and defines the program's function `main()` `{`
 21 `... }` that takes no arguments. A function with this *signature*(γ) has special
 22 meaning to the compiler and the linker: they recognize it as a C++ *main*
 23 *program*. There are other signatures that the compiler and linker will recognize
 24 as a C++ main program; consult the standard C++ documentation.



25 To be recognized as a main program, there is one more requirement: `main()`
 26 `{ ... }` must be declared in the global namespace.

27 The body of the main program (between the braces), declares and defines a
 28 variable `a` and initializes it to the value of 3; it prints out the value of `a`. Then
 29 it calls a function that takes `a` as an argument and prints out the value returned
 30 by that function.

31 You, as the programmer using that function, need to know what the function
 32 does but the C++ compiler doesn't. It only needs to know the name, argument
 33 list and return type of the function — information that is provided in the header
 34 file, `function.h`. This file contains the line

35 `float function( float );`

36 This line is called the *declaration*(γ) of the function. It says (1) that the identifier
 37 `function` is the name of a function that (2) takes an argument of type `float`
 38 (the “float” inside the parentheses) and (3) returns a value of type `float`

1 (the “float” at the start of the line). The file `t1.cc` includes this header file,
 2 thereby giving the compiler these three pieces of information it needs to know
 3 about `function`.

4 The other three lines in `function.h` are *code guards*, described in Section 31.8.
 5 In brief, they deal with the following scenario: suppose that we have two header
 6 files, `A.h` and `B.h`, and that `A.h` includes `B.h`; there are many scenarios in
 7 which it makes good sense for a third file, either `.h` or `.cc`, to include both
 8 `A.h` and `B.h`. The code guards ensure that, when all of the includes have been
 9 expanded, the compiler sees exactly one copy of `B.h`.

10 Finally, the file `function.cc` contains the source code for the function named
 11 `function`:

```
12     float function ( float i ){
13         return 2.*i;
14     }
```

15 It names its argument `i`, multiplies this argument by two and returns that
 16 value. This code fragment is called the *definition* of the function or the *imple-*
 17 *mentation*(γ) of the function. (The C++ standard uses the word *definition* but
 18 *implementation* is in common use.)

19 We now have a rich enough example to discuss another case in which the same
 20 word is frequently used to mean two different things. Sometimes people use the
 21 phrase “the source code of the function named `function`” to refer collectively
 22 to both `function.h` and `function.cc`; sometimes they use it to refer ex-
 23 clusively to `function.cc`. Unfortunately the only way to distinguish the two
 24 uses is from context.

25 The word *header file* always refers unambiguously to the `.h` file. The term
 26 *implementation file* is used to refer unambiguously to the `.cc` file. This name
 27 follows from the its contents: it describes how to implement the items declared
 28 in the header file.

29 Based on the above description, when this exercise is run, we expect it to print
 30 out:

```
31     a =          3
32     function(a) 6
```

33 6.4.3 Compile, Link and Run the Exercise

34 To perform this exercise, first log in and `cd` to your working directory if you
 35 haven’t already, then

```
36     1. cd to the directory for this exercise and get a directory listing:
37         $ cd Build/v1
38         $ ls
```

Part

```
1      build build2 function.cc function.h t1.cc
```

```
2
```

```
3      The two files, build and build2 are scripts that show two different
4      ways to build the code.
```

```
5      2. Compile and link this exercise, then get an updated directory listing:
```

```
6      $ build
```

```
7      $ ls
```

```
8      build build2 function.cc function.h function.o t1 t1.cc
```

```
9      t1.o
```

```
10
```

```
11      Notice the new files function.o, t1 and t1.o.
```

```
12      3. Run the exercise:
```

```
13      $ ./t1
```

```
14      a = 3
```

```
15      function(a) 6
```

```
16
```

```
17      This matches the expected printout.
```

```
18      Look at the file build that you just ran. It has three steps; the first two
19      commands have the -c command line option while the last one does not:
```

```
20      1. It compiles the main program, t1.cc, into the object file (with the default
21         name) t1.o (which will now be the thing that the term main program
22         refers to):
```

```
23      c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c t1.cc
```

```
24      2. It (separately) compiles function.cc into the object file function.o:
```

```
25      c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c function.cc
```

```
26      3. It links t1.o and function.o to form the executable program t1 (the
27         name of the main program is the argument of the -o option):
```

```
28      c++ -std=c++11 -o t1 t1.o function.o
```

```
29      You should have noticed that the same command, c++, is used both for compil-
30      ing and linking. The full story is that when you run the command c++, you are
31      actually running a program that parses its command line to determine which,
32      if any, files need to be compiled and which, if any, files need to be linked. It
33      also determines which of its command line arguments should be forwarded to
34      the compiler and which to the linker. It then runs the compiler and linker as
35      many times as required.
```

```
36      If the -c option is present, it tells c++ to compile only, and not to link. If -c is
37      specified, the .cc file(s) to compile must also be specified. Each of the files will
38      be compiled to create its corresponding object file and then processing stops.
39      In our example, the first two commands each compile a single source file. Note
40      that if any .o files are given on the command line, c++ will issue a warning and
41      ignore them.
```

1 The third command (with no `-c` option) is the linking step. Even if the `-c`
2 option is missing, `c++` will first look for source files on the command line; if
3 it finds any, it will compile them and put the output into temporary object
4 files. In our example, there are none, so it goes straight to linking. The two
5 just-created object files are specified (at the end, here, but the order is not
6 important); the `-o t1` portion of the command tells the linker to write its
7 output (the executable) to the file `t1`.

8 As it is compiling the main program, `t1.cc`, the compiler recognizes every
9 function that is defined within the file and every function that is called by the
10 code in the file. It recognizes that `t1.cc` defines a function `main()` and that
11 `main()` calls a function named `function`, whose definition is not found inside
12 `t1.cc`. At the point that `t1.cc` calls `function`, the compiler will write
13 to `function` all of the machine code needed to prepare for the call; it will
14 also write all of the machine code needed to use the result of the function. In
15 between these two pieces, the compiler will write machine code that says “call
16 the function whose memory address is” but it must leave an empty placeholder
17 for the address. The placeholder is empty because the compiler does not know
18 the memory address of that function.

19 The compiler also makes a table that lists all functions defined by the file and
20 all functions that are called by code within the file. The name of each entry
21 in the table is called a *linker symbol* and the table is called a *symbol table*.
22 When the compiler was compiling `t1.cc` and it found the definition of the
23 main program, it created a linker symbol for the main program and added a
24 notation to say the this file contains the definition of that symbol. When the
25 compiler was compiling `t1.cc` and it encountered the call to `function`, it
26 created a linker symbol for this function; it marked this symbol as an *undefined*
27 *reference* (because it could not find the definition of `function` within `t1.cc`).
28 The symbol table also lists all of the places in the machine code of `t1.o` that
29 are placeholders that must be updated once the memory address of `function`
30 is known. In this example there is only one such place.

31 When the compiler writes an object file, it writes out both the compiled code
32 and the table of linker symbols.

33 In `t1.cc`, the compiled code for the line that begins `std::cout` will do its
34 work by calling a few functions that are found in the compiler-supplied libraries.
35 The linker symbols for these functions will also be listed as undefined references
36 in the symbol table of `t1.o`; the symbol table also lists the places within the
37 machine code of `t1.o` that need to be updated once the addresses of these
38 symbols are known.

39 The symbol table in the file `function.o` is simple; it says that this file defines
40 a function named `function` that takes a single argument of type `float` and that
41 returns a `float`.

42 The job of the linker (also invoked by the command `c++`) is to play match-
43 maker. First it inspects the symbol tables inside all of the object files listed on

1 the command line and looks for a linker symbol that defines the location of the
 2 main program. If it cannot find one, or if it finds more than one, it will issue
 3 an error message and stop. In this example

- 4 1. The linker will find the definition of a main program in `t1.o`.
- 5 2. It will start to build the executable (output) file by copying the machine
 6 code from `t1.o` to the output file.
- 7 3. Then it will try to resolve the unresolved references listed in the symbol
 8 table of `t1.o`; it does this by looking at the symbol tables of the other
 9 object files on the command line. It also knows to look at the symbol tables
 10 from a standard set of compiler-supplied and system-supplied libraries.
- 11 4. It will discover that `function.o` resolves one of the external references
 12 from `t1.o`. So it will copy the machine code from `function.o` to the
 13 executable file.
- 14 5. It will discover that the the other unresolved references in `t1.o` are found
 15 in the compiler-supplied libraries and will copy code from these libraries
 16 into the executable.
- 17 6. Once all of the machine code has been copied into the executable, the
 18 compiler knows the memory address of every function. The compiler can
 19 then go into the machine code, find all of the placeholders and update
 20 them with the correct memory addresses.

21 Sometimes resolving one unresolved reference will generate new ones. The linker
 22 iterates until (a) all references are resolved and no new unresolved references
 23 appear (success) or (b) the same unresolved references continue to appear (er-
 24 ror). In the former case, the linker writes the output to the file specified by the
 25 `-o` option; if no `-o` option is specified the linker will write its output to a file
 26 named `a.out`. In the latter case, the linker issues an error message and does
 27 not write the output file.

28 After the link completes, the files `t1.o` and `function.o` are no longer needed
 29 because everything that was useful from them was copied into the executable
 30 `t1`. You may delete the `.o` files, and the executable will still run.

31 6.4.4 Alternate Script `build2`

32 The script `build2` shows an equivalent way of building `t1` that is commonly
 33 used for small programs; it does it all on one line. To exercise this script:

- 34 1. Stay in the same directory as before, `Build/v1`.
- 35 2. Clean up from the previous build and look at the directory contents:
 36 `$ rm function.o t1 t1.o`
 37 `$ ls`
 38 `build build2 function.cc function.h t1.cc`

1 3. Run the build2 script, and again look at directory contents:

```
2       $ build2
3       $ ls
4       build build2 function.cc function.h t1 t1.cc
```

5
6 Note that t1 was created but there are no .o files.

7 4. Execute the program that you just built

```
8       $ ./t1
9       a = 3
10       function(a) 6
```

11

12 Look at the script build2; it contains only one line (shown as two here):

```
13    c++ -Wall -Wextra -pedantic -Werror -std=c++11 -o t1 \
14    t1.cc function.cc
```

15 This script automatically does the same operations as build but it knows that
16 the .o files are temporaries. Perhaps the command c++ kept the contents of
17 the two .o files in memory and never actually wrote them out as disk files. Or,
18 perhaps, the command c++ did explicitly create disk files and deleted them when
19 it was finished. In either case you don't see them when you use build2.

20 6.4.5 Suggested Homework

21 It takes a bit of experience to decipher the error messages issued by a C++
22 compiler. The three exercises in this section are intended to introduce you to
23 them so that you (a) get used to looking at them and (b) understand these
24 particular errors if/when you encounter them later.

25 Each of the following three exercises is independent of the others. Therefore,
26 when you finish with each exercise, you will need to undo the changes you made
27 in the source file(s) before beginning the next exercise.

28 1. In Build/v1/t1.cc, comment out the include directive for function.h;
29 rebuild and observe the error message.

30 2. In Build/v1/function.cc, change the return type to double; rebuild
31 and observe the error message.

32 3. In Build/v1/t1.cc, change float a=3. to double a=3.; rebuild
33 and run. This will work without error and will produce the same output
34 as before.

1 The first homework exercise will issue the diagnostic:

```
2 t1.cc: In function int main():
3 t1.cc:10:44: error: function was not declared in this scope
```

4 When you see a message like this one, you can guess that either you have
5 not included a required header file or you have misspelled the name of the
6 function.

7 The second homework exercise will issue the diagnostic:

```
8 function.cc: In function double function(float):
9 function.cc:3:27: error: new declaration double function(float)
10 In file included from function.cc:1:0:
11 function.h:4:7: error: ambiguates old declaration float function(float)
```

12 This error message says that the compiler has found two functions that have the
13 same signature but different return types. The compiler does not know which
14 of the two functions you want it to use.

15 The bottom line here is that you must ensure that the definition of a function is
16 consistent with its declaration; and you must ensure that the use of a function
17 is consistent with its declaration.

18 The third homework exercise illustrates the C++ idea of *automatic type conver-*
19 *sion*; in this case the compiler will make a temporary variable of type `float`
20 and set its value to that of `a`:

```
21 float tmp = a;
```

22 The compiler will then use this temporary variable as the argument of the func-
23 tion. Consult the standard C++ documentation to understand when automatic
24 type conversions may occur; see Section 6.7.

25 6.5 C++ Exercise 3: Libraries

26 Multiple compiled object code files can be grouped into a single file known as
27 a *library*, obviating the need to specify each and every object file when linking;
28 you can reference the libraries instead. This simplifies the multiple use and
29 sharing of software components.

30 Two Linux C/C++ library types can be created:

- 31 • static libraries of object code (filenames for which end in `.a`) that are
32 linked with, and become part of, the application (*art* does not use static
33 libraries)
- 34 • dynamically linked, shared object libraries (filenames end in `.so`); multi-
35 ple *art* jobs running simultaneously can dynamically load and unload the

1 same copy of a library of this kind instead of making an exclusive copy of
2 it; this substantially reduces the amount of memory needed by each job.

3 6.5.1 What You Will Learn

4 In this section you will repeat the example of Section 6.4 with a variation. You
5 will create an object library, insert `function.o` into that library and use that
6 library in the link step. This pattern generalizes easily to the case that you
7 will encounter in your experiment software, where object libraries will typically
8 contain many object files.

9 6.5.2 Building and Running the Exercise

10 To perform this exercise, do the following:

11 1. Log in and establish your working environment (Section 6.2).

12 2. `cd` to your working directory.

13 3. `cd` to the directory for this exercise and get a directory listing:

14 \$ `cd Libraries/v1`

15 \$ `ls`

16 `build build2 build3 function.cc function.h t1.cc`

17

18 The three files, `function.cc`, `function.h` and `t1.cc` are identical
19 to those from the previous exercise. The three files, `build`, `build2`
20 and `build3` are scripts that show three different ways to build the main
21 program in this exercise.

22 4. Compile and link this exercise using `build`, then compare the directory
23 listing to that taken pre-build:

24 \$ `build`

25 \$ `ls`

26 `build build3 function.h libpackage1.a t1.cc`

27 `build2 function.cc function.o t1 t1.o`

28

29 5. Execute the main program:

30 \$ `./t1`

31 `a = 3`

32 `function(a) 6`

33 This matches the expected printout. Now let's look at the script `build`. It has
34 four parts:

35 1. Compile `function.cc`; the same as the previous exercise:

36 \$ `c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c function.cc`

- 1 2. Create the library named `libpackage1.a` and add `function.o` to it:
 2 `$ ar rc libpackage1.a function.o`
 3 The name of the library must come before the name of the object file.
- 4 3. Compile `t1.cc`; the same as the previous exercise:
 5 `$ c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c t1.cc`
- 6 4. Link the main program against `libpackage1.a` and the system libraries:
 7 `$ c++ -o t1 t1.o libpackage1.a`

8 The two new features are in step 2, which creates the object library, and step
 9 4, in which `function.o` is replaced in the link list with `libpackage1.a`. If
 10 you have many `.o` files to add to the library, you may add them one at a time
 11 by repeating step 2 or you may add them all in one command. When you do the
 12 latter you may name each object file separately or may use a wildcard:

13 `$ ar rc libpackage1.a *.o`

14 In `libpackage1.a` the string `package1` has no special meaning; it was an
 15 arbitrary name chosen for this exercise. *Actually it was chosen in anticipation*
 16 *of a future exercise that is not yet written up.*

17 The other parts of the name, the prefix `lib` and the suffix `.a`, are part of
 18 a long-standing Unix convention and some Unix tools presume that object li-
 19 braries are named following this convention. You should always follow this
 20 convention. The use of this convention is illustrated by the scripts `build2` and
 21 `build3`.



22 To perform the exercise using `build2`, stay in the same directory and cleanup
 23 then rebuild as follows:

- 24 1. remove files built by `build1`
 25 `$ rm function.o t1.o libpackage1.a t1`
- 26 2. build the code with `build2` and look at the directory contents
 27 `$ build2`
 28 `$ ls`
 29 `build build3 function.h libpackage1.a t1.cc`
 30 `build2 function.cc function.o t1 t1.o`
- 31 3. run `t1` as before

32 The only difference between `build` and `build2` is the link line. The version
 33 from `build` is:

34 `c++ -o t1 t1.o libpackage1.a`

35 while that from `build2` is:

36 `c++ -o t1 t1.o -L. -lpackage1`

37 In the script `build`, the path to the library, relative or absolute, is written
 38 explicitly on the command line. In the script `build2`, two new elements are

1 introduced. The command line may contain any number of `-L` options; the
 2 argument of each option is the name of a directory. The ensemble of all of
 3 the `-L` options forms a search path to look for named libraries; the path is
 4 searched in the order in which the `-L` options appear on the line. The names of
 5 libraries are specified with the `-l` options (this is a lower case letter `L`, not the
 6 numeral one); if a `-l` option has an argument of `XXX` (or `package1`), then the
 7 linker will search the path defined by the `-L` options for a file with the name
 8 `libXXX.a` (or `libpackage1.a`).

9 In the above, the dot in `-L.` is the usual Unix pathname that denotes the
 10 current working directory. And it is important that there be no whitespace
 11 after a `-L` or a `-l` option and its value.

12 This syntax generalizes to multiple libraries in multiple directories as follows.
 13 Suppose that the libraries `libaaa.a`, `libbbb.a` and `libccc.a` are in the
 14 directory `L1` and that the libraries `libddd.a`, `libeee.a` and `libfff.a` are
 15 in the directory `L2`. In this case, the link list would look like (split here into
 16 two lines):

```
17 -L<path-to-L1> -laaa -lbbb -lccc
18 -L<path-to-L2> -lddd -leee -lfff
```

19 The `-L -l` syntax is in common use throughout many Unix build systems: if
 20 your link list contains many object libraries from a single directory then it is
 21 not necessary to repeatedly specify the path to the directory; once is enough.
 22 If you are writing link lists by hand, this is very convenient. In a script, if the
 23 path name of the directory is very long, this convention makes a much more
 24 readable link list.

25 To perform the exercise using `build3`, stay in the same directory and cleanup
 26 then rebuild as follows:

```
27 1. remove files built by build2
28    $ rm function.o t1.o libpackage1.a t1

29 2. build the code with build2 and look at the directory contents
30    $ build3
31    $ ls
32    build build3 function.h libpackage1.a t1.cc
33    build2 function.cc function.o t1

34 3. run t1 as before
```

35 The difference between `build2` and `build3` is that `build3` compiles the main
 36 program and links it, all one one line. `build2`, on the other hand did the two
 37 steps separately.

Part

1 6.6 Classes

2 6.6.1 Introduction

3 The comments in the sample program used in Section 6.3 emphasized that every variable has a type: `int`, `float`, `std::string`,
4 `std::vector<std::string>`, and so on. One of the basic building blocks
5 of C++ is that users may define their own types; user-defined types may be
6 built-up from all types, including other user-defined types.

8 The most common user-defined type is called a *class*(γ). As you work through
9 the Workbook exercises, you will see classes that are defined by the Workbook
10 itself; you will also see classes defined by the toyExperiment UPS product; you
11 will see classes defined by *art* and you will see classes defined by the many
12 UPS products that support *art*. You will also write some classes of your own.
13 When you work with the software for your experiment you will work with classes
14 defined within your experiment's software.

15 In general, a class contains both a *declaration* (what it consists of) and an *in-*
16 *stantiation*(γ) (what to do with the parts). The declaration contains some data
17 (called *data members* or *member datum*) plus some functions (called *member*
18 *functions*) that will (when instantiated) operate on that data, but it is legal for
19 a class declaration (and therefore, a class) to contain only data or only functions.
20 A class *declaration* has the form shown in Listing 6.1.

Listing 6.1: The form of a class declaration

```
21 class MyClassName{
22
23     // required: declarations of all members of the class
24     // optional: definitions of some members of the class
25
26 };
```

27 The string *class* is a keyword that is reserved to C++ and may not be used
28 for any user-defined *identifiers*.¹ This construct tells the C++ compiler that
29 MyClassName is the name of a class; everything that is between the braces
30 is part of the class declaration. The remainder of Section 6.6 will give many
31 examples of *members* of a class.

32 In a class declaration, the semi-colon after the closing brace is important.



33 The upcoming sections will illustrate some features of classes, with an emphasis
34 on features that will be important in the earlier Workbook exercises. This is
35 not intended to be a comprehensive description of classes. To illustrate, we
36 will show nine versions of a class named `Point` that represents a point in a

¹ An identifier is a user defined name; this includes, for example, the names of classes, the names of members of classes, the names of functions, the names of objects and the names of variables and so on.

1 plane. The first version will be simple and each subsequent version will add
2 features.

3 This documentation will use technically correct language so that you will find
4 it easier to read the standard reference materials.

5 6.6.2 C++ Exercise 4 v1: The Most Basic Version

6 Here you will see a very basic version of the class `Point` and an illustration of
7 how `Point` can be used. The ideas of *data members*, *objects* and *instantiation*
8 will be defined.

9 To build and run this example:

10 1. Log in and follow the follow the steps in Section 6.2.

11 2. `cd` to the directory for this exercise and examine it

12 \$ `cd Classes/v1/`

13 \$ `ls`

14 `Point.h ptest.cc`

15 Within the subdirectory `v1` the main program for this exercise is the
16 file `ptest.cc`. The file `Point.h` contains the first version of the class
17 `Point`; shown in Listing 6.2.

18 3. Build the exercise.

19 \$ `../build`

20 \$ `ls`

21 `Point.h ptest ptest.cc`

22 The file named `ptest` is the executable program.

23 4. Run the exercise.

24 \$ `./ptest`

25 `p0: (2.31827e-317, 0)`

26 `p0: (1, 2)`

27 `p1: (3, 4)`

28 `p2: (1, 2)`

29 `Address of p0: 0x7fff883fe680`

30 `Address of p1: 0x7fff883fe670`

31 `Address of p2: 0x7fff883fe660`

32 The values printed out in the first line of the output may be different when you
33 run the program (remember initializaion?). When you look at the code you will
34 see that `p0` is not properly initialized and therefore contains stale data. The
35 last three lines of output should also differ when you run the program; they are
36 memory addresses.

37 Look at the header file `Point.h` which shows the basic version of the class
38 `Point`. The three lines starting with `#` make up a code guard, described in
39 Section 31.8.

Listing 6.2: The contents of v1/Point.h

```

1  #ifndef Point_h
2  #define Point_h
3
4  class Point {
5  public:
6      double x;
7      double y;
8  };
9
10 #endif /* Point_h */

```

11 The class declaration says that the name of the class is `Point`; the body of the
 12 class declaration (the lines between the braces `{...}`) declares two data members
 13 of the class, named `x` and `y`, both of which are of type `double`. (The plural
 14 of *data member* is sometimes written *data members* and sometimes as *member*
 15 *data*.) The line `public:` says that the member data `x` and `y` are accessi-
 16 ble by any code. Instead of `public`, members may be declared `private` or
 17 `protected`; these ideas will be discussed later.

18 In this exercise there is no file `Point.cc` because the class `Point` consists
 19 only of a declaration; there is no implementation to put in a corresponding `.cc`
 20 file.

21 Look at the function `main()` (the *main program*) in `pctest.cc`, which illus-
 22 trates the use of the class `Point`; see Listing 6.3. This file includes `Point.h`
 23 so that the compiler will know about the class `Point` when it begins execu-
 24 tion. It also includes the C++ header `<iostream>` which enables printing
 25 with `std::cout`.

Listing 6.3: The contents of v1/pctest.cc

```

26 #include "Point.h"
27
28 #include <iostream>
29
30 int main() {
31
32     Point p0;
33     std::cout << "p0:_" << p0.x << ",_" << p0.y << ")" << std::endl;
34
35     p0.x = 1.0;
36     p0.y = 2.0;
37     std::cout << "p0:_" << p0.x << ",_" << p0.y << ")" << std::endl;
38
39     Point p1;
40     p1.x = 3.0;
41     p1.y = 4.0;
42     std::cout << "p1:_" << p1.x << ",_" << p1.y << ")" << std::endl;
43
44     Point p2 = p0;
45     std::cout << "p2:_" << p2.x << ",_" << p2.y << ")" << std::endl;
46
47     std::cout << "Address_of_p0:_" << &p0 << std::endl;

```



```

23     std::cout << "Address_of_p1:_" << &p1 << std::endl;
24     std::cout << "Address_of_p2:_" << &p2 << std::endl;
25
26     return 0;
27 }

```

6 Line 7, the first line in the `main()` program is:

```
7 Point p0;
```

8 This declares that `p0` is the name of a variable whose type is (the class) `Point`.
9 When this line of code is executed, the program will ensure that memory has
10 been allocated² to hold the data members of `p0`. If the class `Point` contained
11 code to initialize data members then the program would also run that, but
12 `Point` does not have any such code. Therefore the data members take on
13 whatever values happened to preexist in the memory that was allocated for
14 them.

15 Some other standard pieces of C++ nomenclature can now be defined:

- 16 1. The identifier `p0` refers to a variable in the source code whose type is
17 `Point`.
- 18 2. When the running program executes this line of code, it *instantiates*(γ)
19 the *object*(γ) with the identifier `p0`.
- 20 3. The object with the identifier `p0` is an *instance*(γ) of the class `Point`.
- 21 4. The identifier `p0` now also refers to a region of memory containing the
22 bytes belonging to an object of type `Point`.

23 An important take-away from the above is that a *variable* is an identifier in a
24 source code file while an *object* is something that exists in the computer memory.
25 Most of the time a one-to-one correspondence exists between variables in the
26 source code and objects in memory. There are exceptions, however, for example,
27 sometimes a compiler needs to make anonymous temporary objects that do not
28 correspond to any variable in the source code, and sometimes two or more
29 variables in the source code can refer to the same object in memory.

30 Line 8 (shown split here):

```

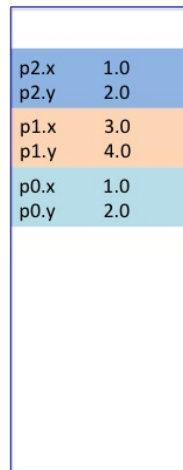
31 std::cout << "p0: (" << p0.x << ", "
32 << p0.y << ")" << std::endl;

```

33 prints out the values of the two data members. In C++, the dot (period)
34 character, when used this way, is called the *member selection operator*.

35 Lines 10 and 11 show how to modify the values of the data members of the
36 object `p0`. Line 12 makes a printout to verify that the values have indeed
37 changed.

² This is deliberately vague — there are many ways to allocate memory, and sometimes the memory allocation is actually done much earlier on, perhaps at link time or at load time.

Figure 6.1: Memory diagram at the end of a run of `Classes/v1/ptest.cc`

1 Lines 14-16 declare another object, named `p1`, of type `Point` and *assign* values
 2 to its data members. These are followed by a print statement.

3 Line 19, (`Point p2 = p0;`) declares that the object named `p2` is of type
 4 `Point` and it assigns the value of `p2` to be a copy of the value of `p0`. When the
 5 compiler sees this line, it knows to copy all of the data members of the class;
 6 this is a tremendous convenience for classes with many data members. Again,
 7 a print statement follows (line 20).

8 The last section of the main program (and of `ptest.cc` itself), lines 22-24,
 9 prints the address of each of the three objects, `p0`, `p1` and `p2`. The addresses
 10 are represented in hexadecimal (base 16) format. On almost all computers, the
 11 length of a `double` is eight bytes. Therefore an object of type `Point` will have
 12 a length of 16 bytes. If you look at the printout made by `ptest` you will see
 13 that the addresses of `p0`, `p1` and `p2` are separated by 16 bytes; therefore the
 14 three objects are contiguous in memory.

15 Figure 6.1 shows a diagram of the computer memory at the end of running
 16 `ptest`; the outer box (blue outline) represents the memory of the computer;
 17 each filled colored box represents one of the three objects in this program. The
 18 diagram shows them in contiguous memory locations, which is not necessary;
 19 there could have been gaps between the memory locations in Figure 6.1.

20 Now, for a bit more terminology: each of the objects `p0`, `p1` and `p2` has the
 21 three attributes required of an *object*:

- 22 1. a *state*, given by the values of its data members
- 23 2. the ability to have operations performed on it: e.g., setting/reading in
 24 value of a data member, assigning value of object of a given type to another

- 1 of the same type
- 2 3. an *identity*: a unique address in memory

3 6.6.3 C++ Exercise 4 v2: The Default Constructor

4 This exercise expands the class `Point` by adding a default *constructor*(γ).

5 To build and run this example:

- 6 1. Log in and follow the follow the steps in Section 6.2.

- 7 2. Go to the directory for this exercise:

8 \$ `cd Classes/v2`

9 \$ `ls`

10 `Point.cc Point.h ptest.cc`

11 In this example, `Point.cc` is a new file.

- 12 3. Build the exercise:

13 \$ `../build`

14 \$ `ls`

15 `Point.cc Point.h ptest ptest.cc`

16

- 17 4. Run the exercise:

18 \$ `ptest`

19 `p0: (0, 0)`

20 `p0: (3.1, 2.7)`

21 When you run the code, all of the printout should match the above printout
22 exactly.

23 Look at `Point.h`. There is one new line in the body of the class declara-
24 tion:

25 `Point();`

26 The parentheses tell you that this new member is some sort of function. A C++
27 class may have several different kinds of functions.



28 A function that has the same name as the class itself has a special role and is
29 called a *constructor*; if a constructor takes no arguments it is called a *default*
30 *constructor*. In informal written material, the word constructor is sometimes
31 written as *c'tor*.

32 `Point.h` declares that the class `Point` has a default constructor, but does not
33 define it (i.e., provide an implementation). The definition/implementation of
34 the constructor is found in the file `Point.cc`.

35 Look at the file `Point.cc`. It “includes” the header file `Point.h` because the
36 compiler needs to know all about this class before it can compile the code that it

1 finds in `Point.cc`. The rest of the file contains a *definition* of the constructor.
 2 The syntax `Point::` says that the function to the right of the `::` is part of (a
 3 member of) the class `Point`. The body of the constructor gives initial values
 4 to the two data members, `x` and `y`.

5 Look at the program `pctest.cc`. The first line of the main program is again

6 `Point p0;`

7 When the program executes this line, the first step is the same as before: it
 8 ensures that memory has been allocated for the data members of `p0`. This
 9 time, however, it also calls the default constructor of the class `Point` (declared
 10 in `Point.h`), which initializes the two data members (per `Point.cc`) such
 11 that they have well defined initial values. This is reflected in the printout made
 12 by the next line.

13 The next block of the program assigns new values to the data members of `p0`
 14 and prints them out.

15 In the previous example, `Classes/v1/pctest.cc`, a few things happened be-
 16 hind the scenes that will make more sense now that you know what a constructor
 17 is.

- 18 1. Since the class `Point` did not contain a default constructor, the compiler
 19 (implicitly) wrote a default constructor for you; this default constructor
 20 simply “default constructed” each of the data members.
- 21 2. The (implicit) constructor of the built-in type `double` did nothing, leav-
 22 ing the data members `x` and `y` uninitialized.

23 6.6.4 C++ Exercise 4 v3: Constructors with Arguments

24 This exercise introduces four new ideas:

- 25 1. constructors with arguments
- 26 2. the copy constructor
- 27 3. implicitly generated constructor
- 28 4. single-phase construction vs. two-phase construction

29 To build and run this exercise, `cd` to the directory `Classes/v3` and follow the
 30 same instructions as in Section 6.6.3. When you run the `pctest` program, you
 31 should see the following output:

```
32 $ ptest
33 p0: (1, 2)
34 p1: (1, 2)
```

35 Look at the file `Point.h`. This contains one new line:

```
1 Point( double ax, double ay);
```

2 This line declares a second constructor; we know it is a constructor because
 3 it is a function whose name is the same as the name of the class. It is distin-
 4 guishable from the default constructor because its argument list is different than
 5 that of the default constructor. As before, the file `Point.h` contains only the
 6 declaration of this constructor, not its *definition* (aka *implementation*).

7 Look at the file `Point.cc`. The new content in this file is the implementation of
 8 the new constructor; it assigns the values of its arguments to the data members.
 9 The names of the arguments, `ax` and `ay`, have no meaning to the compiler; they
 10 are just identifiers. It is good practice to choose names that bear an obvious
 11 relationship to those of the data members. One convention that is sometimes
 12 used is to make the name of the argument be the same as that of the data
 13 member, but with a prefix letter `a`, for argument. Whatever convention you
 14 (or your experiment) choose(s), use it consistently. When you update code that
 15 was initially written by someone else, follow whatever convention they adopted.
 16 Choices of style should be made to reinforce the information present in the code,
 17 not to fight it.

18 Look at the file `pctest.cc`. The first line of the main program is now:

```
19 Point p0(1.,2.);
```

20 This line declares the variable `p0` and initializes it by calling the new con-
 21 structor defined in this section. The next line prints the value of the data
 22 members.

23 The next line of code

```
24 Point p1(p0);
```

25 introduces the *copy constructor*. A copy constructor is indicated by code (like
 26 the above) that wants to create an exact copy (e.g., `p1`, data-member-for-data-
 27 member, of an existing type/class (e.g., `tt p0`). This exercise does not provide
 28 a copy constructor so the compiler implicitly declares a copy constructor, with
 29 public access, for that class. (The compiler puts the constructor code directly
 30 into the object file; it does not affect the source file.)

31 In general³ if no user-defined constructor exists for a class `A`, the compiler im-
 32 plicitly declares a default, parameterless constructor `A::A()` when it needs to
 33 create an object of type `A`. The constructor will have no constructor initializer
 34 and a null body.

35 We recommend that for any class whose data members are either built-in types
 36 or simple aggregates of built-in types, of which `Point` is an example, you let
 37 the compiler write the copy constructor for you.

³Some text in this section is adapted from material in
publib.boulder.ibm.com/infocenter.

1 If your class has data members that are pointers, or data members that manage
2 some external resource, such as a file that you are writing to, then you will very
3 likely need to write your own copy constructor. There are some other cases in
4 which you should write your own copy constructor, but discussing them here
5 is beyond the scope of this document. When you need to write your own copy
6 constructor, you can learn how to do so from any standard C++ reference; see
7 Section 6.7.

8 The next line in the file prints the values of the data members of `p1` and you
9 can see that the copy constructor worked as expected.

10 Notice that in the previous version of `pctest.cc`, the variable `p0` was initialized
11 in three lines:

```
12 Point p0;  
13 p0.x = 3.1;  
14 p0.y = 2.7;
```

15 This is called *two-phase construction*. In contrast, the present version uses
16 *single-phase construction* in which the variable `p0` is initialized in one line:

```
17 Point p0(1.,2.);
```

18 We strongly recommend using single-phase construction whenever possible. Ob-
19 viously it takes less real estate, but more importantly:



- 20 1. Single-phase construction more clearly conveys the intent of the program-
21 mer: the intent is to initialize the object `p0`. The second version says
22 this directly. In the first version you needed to do some extra work to
23 recognize that the three lines quoted above formed a logical unit distinct
24 from the remainder of the program. This is not difficult for this simple
25 class, but it can become so with even a little additional complexity.
- 26 2. Two-phase construction is less robust. It leaves open the possibility that
27 a future maintainer of the code might not recognize all of the follow-on
28 steps that are part of construction and will use the object before it is fully
29 constructed. This can lead to difficult-to-diagnose run-time errors.

30 6.6.5 C++ Exercise 4 v4: Colon Initializer Syntax

31 This version of the class `Point` introduces *colon initializer syntax* for construc-
32 tors.

33 To build and run this exercise, `cd` to the directory `Classes/v4` and follow the
34 same instructions as in the previous two sections. When you run the `pctest`
35 program you should see the following output:

```
36 $ pctest  
37 p0: (1, 2)  
38 p1: (1, 2)
```

1 The file `Point.h` is unchanged between this version and the previous one.

2 Now look at the file `Point.cc`, which contains the *definitions* of both constructors. The first thing to look at is the default constructor, which has been
3 rewritten using colon initializer syntax. The rules for the colon-initializer syntax
4 are:
5

- 6 1. A colon must immediately follow the closing parenthesis of the argument
7 list.
- 8 2. There must be a comma-separated list of data members, each one initial-
9 ized by calling one of its constructors.
- 10 3. In the initializer list, the data members must be listed in the order in
11 which they appear in the class declaration.
- 12 4. The body of the constructor, enclosed in braces, must follow the initializer
13 list.
- 14 5. If a data member is missing from the initializer list, its default constructor
15 will be called (constructors for the missing data members will be called in
16 the order in which data members were specified in the class declaration).
- 17 6. If no initializer list is present, the compiler will call the default constructor
18 of every data member and it will do so in the order in which data members
19 were specified in the class declaration.

20 If you think about these rules carefully, you will see that in `Classes/v3/Point.cc`:

- 21 1. the compiler did not find an initializer list, so it wrote one that default-
22 constructed `x` and `y`
- 23 2. it then wrote the code to make the assignments `x=0` and `y=0`

24 On the other hand, when the compiler compiled the code for the default constructor in `Classes/v4/Point.cc`, it did the following

- 25 1. it wrote the code to construct `x` and `y`, both set to zero.

26

27 Therefore, the machine code for the `v3` version does more work than that for
28 the `v4` version. In practice `Point` is a sufficiently simple class that the compiler
29 likely recognized and elided all of the unnecessary steps in `v3`; it is likely that
30 the compiler actually produced identical code for the two versions of the class.
31 For a more complex class, however, the compiler may not be able to recognize
32 meaningless extra work and it will write the machine code to do that extra
33 work.

34 In many cases it does not matter which of these two ways you use to write
35 a constructor; but on those occasions that it does matter, the right answer is
36 always the colon-initializer syntax. So we strongly recommend that you always
37 use the colon initializer syntax. In the Workbook, all classes are written with
38 colon-initializer syntax.

1 Now look at the second constructor in `Point.cc`; it also uses colon-initializer
 2 syntax but it is laid out differently. The difference in layout has no meaning to
 3 the compiler — whitespace is whitespace. Choose which ever seems natural to
 4 you.

5 Look at `pctest.cc`. It is the same as the version `v3` and it makes the same
 6 printout.

7 6.6.6 C++ Exercise 4 v5: Member functions

8 This section will introduce *member functions*(γ), both *const member func-*
 9 *tions*(γ) and non-const member functions. It will also introduce the header
 10 `<cmath>`.

11 To build and run this exercise, cd to the directory `Classes/v5` and follow the
 12 same instructions as in Section 6.6.3. When you run the `pctest` program you
 13 should see the following output:

```
14 $ pctest
15 Before p0: (1, 2)  Magnitude: 2.23607  Phi: 1.10715
16 After  p0: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
```

17 Look at the file `Point.h`. Compared to version `v4`, this version contains three
 18 additional lines:

```
19 double mag() const;
20 double phi() const;
21 void scale( double factor );
```

22 All three lines declare *member functions*. As the name suggests, a *member*
 23 *function* is a function that can be called and it is a member of the class. Contrast
 24 this with a *data member*, such as `x` or `y`, which are not functions. A member
 25 function may access any or all of the member data of the class.

26 The member function named `mag` does not take any arguments and it returns
 27 a double; you will see that the value of the double is the magnitude of the 2-
 28 vector from the origin to (x, y) . The keyword `const` represents a contract
 29 between the definition/implementation of `mag` and any code that uses `mag`; it
 30 “promises” that the implementation of `mag` will not modify the value of any
 31 data members. The consequences of breaking the contract are illustrated in the
 32 homework at the end of this subsection.

33 Similarly, the member function named `phi` takes no arguments, returns a double
 34 and has the `const` keyword. You will see that the value of the double is the
 35 azimuthal angle of the vector from the origin to the point (x, y) .

36 The third member function, `scale`, takes one argument, `factor`. Its return
 37 type is `void`, which means that it returns nothing. You will see that this mem-
 38 ber function multiplies both `x` and `y` by `factor` (i.e., changing their values).

1 This function declaration does not have the `const` keyword because it actually
 2 does modify member data.



3 If a member function does not modify any data members, you should always
 4 declare it `const` simply as a matter of course. Any negative consequences of
 5 not doing so might only become apparent later, at which point a lot of tedious
 6 editing will be required to make everything right.

7 Look at `Point.cc`. Near the top of the file an additional include directive has
 8 been added; `<cmath>` is a header from the C++ standard library that declares
 9 a set of functions for computing common mathematical operations and trans-
 10 formations. Functions from this library are in the *namespace*(γ) `std`.

11 Later on in `Point.cc` you will find the definition of `mag`, which computes
 12 the magnitude of the 2-vector from the origin to (x, y) . To do so, it uses
 13 `std::sqrt`, a function declared in the `<cmath>` header that takes the square
 14 root of its argument. The keyword `const` that was present in the declaration
 15 of `mag` must also be present in its definition.

16 The next part of `Point.cc` contains the definition of the member function
 17 `phi`. To do its work, this member function uses the `atan2` function from the
 18 standard library.

19 The next part of `Point.cc` contains the definition of the member function
 20 `scale`. You can see that this member function simply multiplies the two data
 21 members by the value of the argument.

22 The file `pctest.cc` contains a `main()` program that illustrates these new
 23 features. The first line of this function declares and initializes an object, `p0`, of
 24 type `Point`. It then prints out the value of its data members, the value returned
 25 from calling the function `mag` and the value returned from calling `phi`. This
 26 shows how to access a member function: you write the name of the variable,
 27 followed by a dot (the *member selection operator*), followed by the name of the
 28 member function and its argument list.

29 The next line calls the member function `scale` with the argument 3. The
 30 printout verifies that the call to `scale` had the intended effect.

31 One final comment is in order. Many other modern computer languages have
 32 ideas very similar to C++ classes and C++ member functions; in some of those
 33 languages, the name *method* is the technical term corresponding to *member*
 34 *function* in C++. The name *method* is not part of the formal definition of
 35 C++, but is commonly used nonetheless. In this documentation, the two terms
 36 can be considered synonymous.

37 Here we suggest four activities as homework to help illustrate the meaning of
 38 `const` and to familiarize you with the error messages produced by the C++
 39 compiler. Before moving to a subsequent activity, undo the changes that you
 40 made in the current activity.

1. In the definition of the member function `Point::mag()`, found in `Point.cc`, before taking the square root, multiply the member datum `x` by 2.

```

3      double Point::mag() const{
4          x *= 2.;
5          return std::sqrt( x*x + y*y );
6      }

```

Then build the code again; you should see the following diagnostic message:

```

9      Point.cc: In member function double Point::mag() const:
10     Point.cc:13:8: error: assignment of member Point::x in read-only object

```

2. In `pctest.cc`, change the first line to

```

12     Point const p0(1,2);

```

Then build the code again; you should see the following diagnostic message:

```

15     pctest.cc: In function int main():
16     pctest.cc:13:14: error: no matching function for call to
17     Point::scale(double) const
18     pctest.cc:13:14: note: candidate is:
19     In file included from pctest.cc:1:0:
20     Point.h:13:8: note: void Point::scale(double) <near match>
21     Point.h:13:8: note:   no known conversion for implicit this
22     parameter from const Point* to Point*

```

3. In `Point.h`, remove the `const` keyword from the declaration of the member function `Point::mag()`:

```

25     double mag();

```

Then build the code again; you should see the following diagnostic message:

```

28     Point.cc:12:8: error: prototype for double Point::mag() const
29     does not match any in class Point
30     In file included from Point.cc:1:0:
31     Point.h:11:10: error: candidate is: double Point::mag()

```

4. In `Point.cc`, remove the `const` keyword in definition of the member function `mag`. Then build the code again; you should see the following diagnostic message:

```

35     Point.cc:12:8: error: prototype for double Point::mag()
36     does not match any in class Point
37     In file included from Point.cc:1:0:
38     Point.h:11:10: error: candidate is: double Point::mag() const

```

The first two homework exercises illustrate how the compiler enforces the contract defined by the keyword `const` that is present at the end of the declaration

1 of `Point::mag()` and that is absent in the definition of the member function
 2 `Point::scale()`. The contract says that the definition of `Point::mag()`
 3 may not modify the values of any data members of the class `Point`; users of
 4 the class `Point` may count on this behaviour. The contract also says that
 5 the definition of the member function `Point::scale()` may modify the val-
 6 ues of data members of the class `Point`; users of the class `Point` must as-
 7 sume that `Point::scale()` will indeed modify member data and act accord-
 8 ingly.⁴

9 In the first homework exercise, the value of a member datum is modified, thereby
 10 breaking the contract. The compiler detects it and issues a diagnostic mes-
 11 sage.

12 In the second homework exercise, the variable `p0` is declared `const`; therefore
 13 the code may not call non-`const` member functions of `p0`, only `const` member
 14 functions. When the compiler sees the call to `p0.mag()` it recognizes that this
 15 is a call to `const` member function and compiles the call; when it sees the call
 16 to `p0.scale(3.)` it recognizes that this is a call to a non-`const` member
 17 function and issues a diagnostic message.

18 The third and fourth homework exercises illustrate that the compiler considers
 19 two member functions that are identical except for the presence of the `const`
 20 keyword to be different functions⁵. In homework exercise 3, when the com-
 21 piler tried to compile `Point::mag() const` in `Point.cc`, it looked at the
 22 class declaration in `Point.h` and could not find a matching member function
 23 declaration; there was a close, but not exact match. Therefore it issued a diag-
 24 nostic message, telling us about the close match, and then stopped. Similarly,
 25 in homework exercise 4, it also could not find a match.

26 6.6.7 C++ Exercise 4 v6: Private Data and Accessor 27 Methods

28 6.6.7.1 Setters and Getters

29 This version of the class `Point` is used to illustrate the following ideas:

- 30 1. The class `Point` has been redesigned to have private data members with
 31 access to them provided by *accessor functions* and *setter functions*.
- 32 2. the *this pointer*
- 33 3. Even if there are many objects of type `Point` in memory, there is only
 34 one copy of the code.

⁴ C++ has another keyword, *mutable*, that one can use to exempt individual data members from this contract. Its use is beyond the scope of this introduction and it will be described when it is encountered.

⁵ Another way of saying the same thing is that the `const` keyword is part of the *signature*(γ) of a function.

1 A 2D point class, with member data in Cartesian coordinates, is not a good
 2 example of *why* it is often a good idea to have private data. But it does have
 3 enough richness to illustrate the mechanics, which is the purpose of this section.
 4 Section 6.6.7.3 discusses an example in which having private data makes obvious
 5 sense.

6 To build and run this exercise, `cd` to the directory `Classes/v6` and follow the
 7 same instructions as in Section 6.6.3. When you run the `pctest` program you
 8 should see the following output:

```
9 $ pctest
10 Before p0: (1, 2) Magnitude: 2.23607 Phi: 1.10715
11 After p0: (3, 6) Magnitude: 6.7082 Phi: 1.10715
12 p1: (0, 1) Magnitude: 1 Phi: 1.5708
13 p1: (1, 0) Magnitude: 1 Phi: 0
14 p1: (3, 6) Magnitude: 6.7082 Phi: 1.10715
```

15 Look at `Point.h`. Compare it to the version in `v5`:

```
16 $ diff -wb Point.h ../v5/
```

17 Relative to version `v5` the following changes were made:

18 1. four new member functions have been declared,

19 (a) `double x() const;`

20 (b) `double y() const;`

21 (c) `void set( double ax, double ay);`

22 (d) `void set( Point const& p);`

23 2. the data members have been declared private

24 3. the data members have been renamed from `x` and `y` to `x_` and `y_`

25 Yes, there are two functions named `set`. Since in C++ the full name of a
 26 member function encodes all of the following information:

27 1. the name of the class it is in

28 2. the name of the member function

29 3. the argument list; that is the number, type and order of arguments

30 4. whether or not the function is `const`

31 the member functions both named `set` are completely different member func-
 32 tions. As you work through the Workbook you will encounter a lot of this and
 33 you should develop the habit of looking at the full function name (i.e., all the
 34 parts). The full name of a member function, turned into text string, is called
 35 the *mangled name* of the member function; each C++ compiler does this a little
 36 differently. All linker symbols related to C++ classes are the mangled names of
 37 the members.



1 If you want to see what mangled names are created for the class `Point`, you
2 can do the following

```
3 $ c++ -Wall -Wextra -pedantic -Werror \\  
4 -std=c++11 -c Point.cc  
5 $ nm Point.o
```

6 You can understand the output of `nm` by reading the man page for `nm`.

7 In a class declaration, if any of the keywords `public`, `private`, or `protected`
8 appear, then all members following that keyword, and before the next such
9 keyword, have the named property. In `Point.h` the two data members are
10 `private` and all other members are `public`.

11 Look at `Point.cc`. Compare it to the version in `v5`:

```
12 $ diff -wb Point.cc ../v5/
```

13 Relative to version `v5` the following changes were made:

- 14 1. the data members have been renamed from `x` and `y` to `x_` and `y_`
- 15 2. an implementation is present for each of the four new member functions

16 Inspect the code in the implementation of each of the new member functions.
17 The member function `x()` simply returns the value of the data member `x_`;
18 similarly for the member function `y()`. These are called *accessors*, *accessor*
19 *functions*, or *getters*⁶. The notion of *accessor* is often extended to include any
20 member function that returns the value of simple, non-modifying calculations
21 on a subset of the member data; in this sense, the `mag` and `phi` functions of
22 the `Point` class are considered *accessors*.

23 The two member functions named `set` copy the values of their arguments into
24 the data members of the class. These are, not surprisingly, called *setters* or
25 *setter functions*.

26 More generally, any member function that modifies the value of any member
27 data is called a *modifier*.

28 There is no requirement that there be accessors and setters for every data mem-
29 ber of a class; indeed, many classes provide no such member functions for many
30 of their data members. If a data member is important for managing internal
31 state but is of no value to a user of the class, then you should certainly not
32 provide an accessor or a setter.

33 Now that the data members of `Point` are `private`, i.e., only the code within
34 `Point` is permitted to access these data members directly. All other code must

⁶ There is a coding style in which the function `x()` would have been called something like `GetX()`, `getX()` or `get_x()`; hence the name *getters*. Almost all of the code that you will see in the Workbook omits the `get` in the names of accessors; the authors of this code view the `get` as redundant. Within the Workbook, the exception is for accessors defined by `ROOT`. The `Geant4` package also includes the `Get` in the names of its accessors.

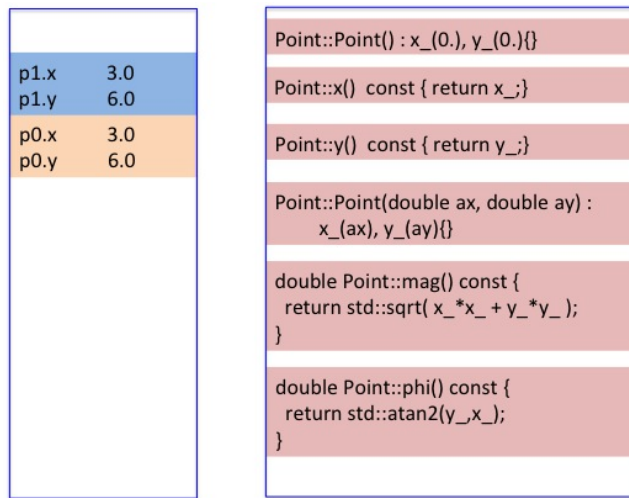


Figure 6.2: Memory diagram at the end of a run of Classes/v6/ptest.cc

- 1 access this information via the accessor and setter functions.
- 2 Look at `ptest.cc`. Compare it to the version in v5:
- 3 `$ diff -wb ptest.cc ../v5/`
- 4 Relative to version v5 the following changes were made:
 - 5 1. the printout has been changed to use the accessor functions
 - 6 2. a new section has been added to illustrate the use of the two set methods
- 7 Presumably these are clear.
- 8 Figure 6.2 shows a diagram of the computer memory at the end of running
- 9 this version of `ptest`. The two boxes with the blue outlines represent sections
- 10 of the computer memory; the part on the left represents that part that is re-
- 11 served for storing data (such as objects) and the part on the right represents
- 12 the part of the computer memory that holds the executable code. This is a
- 13 big oversimplification because, in a real running program, there are many parts
- 14 of the memory reserved for different sorts of data and many parts reserved for
- 15 executable code.
- 16 The key point in Figure 6.2 is that each object has its own member data but
- 17 there is only one copy of the code. Even if there are thousands of objects of
- 18 type `Point`, there will only be one copy of the code. When a line of code asks
- 19 for `p0.mag()`, the computer will pass the address of `p0` as an argument to
- 20 the function `mag()`, which will then do its work. When a line of code asks for
- 21 `p1.mag()`, the computer will pass the address of `p1` as an argument to the
- 22 function `mag()`, which will then do its work.

1 Initially this sounds a little weird: the previous paragraph talks about passing
 2 an argument to the function `mag()` but, according to the source code, `mag()`
 3 does not take any arguments! The answer is that all member functions have
 4 an implied argument that always must be present — the address of the object
 5 that the member function will do work on. Because it must always be there,
 6 and because the compiler knows that it must always be there, there is no point
 7 in actually writing it in the source code! It is by using this so called *hidden*
 8 *argument* that the code for `mag()` knew that `x_` means one thing for `p0` but
 9 that it means something else for `p1`.

10 Every C++ member function has a variable whose name is `this`, which is
 11 a pointer to the object on which the member function will do its work. For
 12 example, the accessor for `x()` could have been written:

```
13     double x() const { return this->x_; }
```

14 This version of the syntax makes it much clearer how there can be one copy of
 15 the code even though there are many objects in memory; but it also makes the
 16 code harder to read once you have understood how the magic works. There are
 17 not many places in which you need to explicitly use the *this* pointer, but there
 18 will be some. For further information, consult standard C++ documentation
 19 (listed in Section 6.7).

20 6.6.7.2 What's the deal with the underscore?

21 C++ will not permit you to use the same name for both a data member and
 22 its accessor. Since the accessor is part of the public interface, it should get the
 23 simple, obvious, easy-to-type name. Therefore the name of the data member
 24 needs to be decorated to make it distinct.

25 The convention used in the Workbook exercises and in the toyExperiment UPS
 26 product is that the names of member data end in an underscore character.
 27 There are some other conventions that you may encounter:

```
28     _name;  
29     __name;  
30     m_name;  
31     mName;
```



32 You may also see the choice of a leading underscore, or double underscore,
 33 followed by a capital letter. Never do this.



34 The compiler promises that all of the linker symbols it creates will begin with
 35 a leading single or double underscore, followed by a capital letter. Some of the
 36 identifiers that you define in a C++ class will be used as part of a linker symbol.
 37 If you chose identifiers that match the pattern reserved for symbols created by
 38 the compiler there is a chance you will have naming collision with a compiler

1 defined symbol. While this is a very small risk, it seems wise to adopt habits
2 that guarantee that it can never happen.

3 It is common to extend the pattern for decorating the names of member data
4 to all member data, even those without accessors. One reason for doing so is
5 just symmetry. A second reason has to do with writing member functions; the
6 body of a member function will, in general, use both member data and vari-
7 ables that are local to the member function. If the member data are decorated
8 differently than the local variables, it can make the member functions easier to
9 understand.

10 6.6.7.3 An example to motivate private data

11 This section describes a class for which it makes sense to have private data: a 2D
12 point class that has data members `r` and `phi` instead of `x` and `y`. The author
13 of such a class might wish to define a standard representation in which it is
14 guaranteed that `r` be non-negative and that `phi` be on the domain $0 \leq \phi < 2\pi$.
15 If the data is public, the class cannot make these guarantees; any code can
16 modify the data members and break the guarantee.

17 If this class is implemented with private data manipulated by member functions,
18 then the constructors and member functions can enforce the guarantees.

19 The language used in the software engineering texts is that a guaranteed re-
20 lationship among the data members is called an *invariant*. If a class has an
21 invariant then the class must have private data.

22 If a class has no invariant then one is free to choose public data. The Workbook
23 and the toyExperiment never make this choice for the reason that mixing private
24 and public data is very confusing to most beginners.

25 6.6.8 C++ Exercise 4 v7: The inline keyword

26 This section introduces the *inline keyword*.

27 To build and run this exercise, `cd` to the directory `Classes/v7` and follow the
28 same instructions as in Section 6.6.3. When you run the `pctest` program you
29 should see the following output:

```
30 $ pctest  
31 p0: ( 1, 2 ) Magnitude: 2.23607 Phi: 1.10715
```

32 Look at `Point.h` and compare it to the version in v6. The new material added
33 to this version is the implementation for the two accessors `x()` and `y()`. These
34 accessors are defined outside of the class declaration.

35 Look at `Point.cc` and compare it to the version in v6. You will see that the
36 implementation of the accessors `x()` and `y()` has been removed.

1 `Point.h` now contains an almost exact copy of the the implementation of the
 2 accessor `x()` that was previously found in the file `Point.cc`; the difference is
 3 that it is now preceded by the keyword `inline`. This keyword tells the compiler
 4 that it has two options that it may choose from at its discretion.

5 The first option is that the compiler may decline to write a callable member
 6 function `x()`; instead, whenever the member function `x()` is used, the compiler
 7 will insert the body of `x()` right into the machine code at that spot. This is
 8 called *inlining* the function. For something simple like an accessor, relative to
 9 explicitly calling a function, the inlined code is very likely to

- 10 1. have a smaller memory footprint
- 11 2. execute more quickly

12 These are both good things.

13 On the other hand, if you inline a bigger or more complex function, some nega-
 14 tive effects of inlining may appear. If the inlined function is used in many places
 15 and if the memory footprint of the inlined code is large compared to the mem-
 16 ory footprint of a function call, then the total size of the program can increase.
 17 There are various ways in which a large program might run more slowly than a
 18 logically equivalent but smaller program. So, if you inline large functions, your
 19 program may actually run more slowly!

20 When the compiler sees the `inline` keyword, it also has a second option: it can
 21 choose to ignore it. When the compiler chooses this option it will write many
 22 copies of the code for the member function — one copy for each *compilation*
 23 *unit*⁷ in which the function is called. Each compilation unit only knows about
 24 its own copy of the function and the compiler calls that copy as needed. The net
 25 result is completely negative: the function call is not actually elided so there is
 26 no time savings from that; moreover the code has become bigger because there
 27 are multiple copies of the function in memory; the larger memory footprint can
 28 further slow down execution; and compilation takes longer because multiple
 29 copies of the function must be compiled.

30 C++ does not permit you to force inlining; you may only give a hint to the
 31 compiler that a function is appropriate for inlining.

32 The bottom line is that you should always inline simple accessors and simple
 33 setters. Here the adjective *simple* means that they do not do any significant
 34 computation and that they do not contain any `if` statements or loops. The
 35 decision to inline anything else should only follow careful analysis of information
 36 produced by a profiling tool.

37 Look at the definition of the member function `y()` in `Point.h`. Compared
 38 to the definition of the member function `x()` there is small change in whites-
 39 pace. This difference is not meaningful to the compiler. You will see several

⁷ A compilation unit is the unit of code that the compiler considers at one time. For most purposes, each `.cc` file is its own compilation unit.

1 other variations on whitespace when you look at code in the Workbook and its
2 underlying packages.

3 **6.6.9 C++ Exercise 4 v8: Defining Member Functions** 4 **within the Class Declaration**

5 The version of `Point` in this section introduces the idea that you may provide
6 the definition (implementation) of a member function at the point that it is
7 declared inside the class declaration. This topic is introduced now because you
8 will see this syntax as you work through the Workbook.

9 To build and run this exercise, `cd` to the directory `Classes/v8` and follow the
10 same instructions as in Section 6.6.3. When you run the `pctest` program you
11 should see the following output:

```
12 $ pctest  
13 p0: ( 1, 2 ) Magnitude: 2.23607 Phi: 1.10715
```

14 This is the same output made by v7.

15 Look at `Point.h`. The only change relative to v7 is that the definition of the
16 accessor methods `x()` and `y()` has been moved into the class declaration.

17 The files `Point.cc` and `pctest.cc` are unchanged with respect to v7.

18 This version of `Point.h` shows that you may define any member function in-
19 side the class declaration. When you do this, the `inline` keyword is implicit.
20 Section 6.6.8 discussed some cautions about inappropriate use of inlining; those
21 same cautions apply when a member function is defined inside the class decla-
22 ration.

23 When you define a member function within the class declaration, you must not
24 prefix the function name with the class name and the scope resolution operator;
25 that is,

```
26 double Point::x() const { return x_; }
```

27 would produce a compiler diagnostic.

28 In summary, there are two ways to write inlined definitions of member functions.
29 In most cases, the two are entirely equivalent and the choice is simply a matter
30 of style. The one exception occurs when you are writing a class that will become
31 part of an *art* data product, in which case it is recommended that you write
32 the definitions of member functions *outside* of the class declaration.

33 When writing an *art* data product, the code inside that header file is parsed by
34 software that determines how to write objects of that type to the output disk
35 files and how to read objects of that type from input disk files. The software
36 that does the parsing has some limitations and we need to work around them.
37 The work arounds are easiest if any member functions definitions in the header



1 file are placed outside of the class declarations. For details see
 2
 3 [https://cdcv.s.fnal.gov/redmine/projects/art/wiki/Data_Product_Design_Guide#Issues-](https://cdcv.s.fnal.gov/redmine/projects/art/wiki/Data_Product_Design_Guide#Issues-mostly-related-to-ROOT)
 4 [mostly-related-to-ROOT](https://cdcv.s.fnal.gov/redmine/projects/art/wiki/Data_Product_Design_Guide#Issues-mostly-related-to-ROOT)

5 **6.6.10 C++ Exercise 4 v9: The stream insertion opera-** 6 **tor**

7 The version of `Point` in this section illustrates how to write a *stream insertion*
 8 *operator*. This is the piece of code that lets you print an object without having
 9 to print each data member by hand, for example:

```
10 Point p0(1,2);
11 std::cout << p0 << std::endl;
```

12 To build and run this exercise, `cd` to the directory `Classes/v9` and follow the
 13 same instructions as in Section 6.6.3. When you run the `pctest` program you
 14 should see the following output:

```
15 $ pctest
16 p0: ( 1, 2 ) Magnitude: 2.23607 Phi: 1.10715
```

17 This is the same output made by v7 and v8.

18 Look at `Point.h`. The changes relative to v7 are the following two addi-
 19 tions:

- 20 1. an include directive for the header `<iosfwd>`
- 21 2. a declaration for the stream insertion operator

22 Look at `Point.cc`. The changes relative to v7 are the following two addi-
 23 tions:

- 24 1. an include directive for the header `<iostream>`
- 25 2. the definition of the stream insertion operator.

26 Look at `pctest.cc`. The only change relative to v7 is that the printout now
 27 uses the stream insertion operator for `p0` instead of inserting each data member
 28 of `p0` by hand.

29 In `Point.h`, the stream insertion operator is declared as (shown here on two
 30 lines)

```
31 std::ostream& operator<<
32 (std::ostream& ost, Point const& p );
```



33 If the class whose type is used as second argument is declared in a namespace,
 34 then the stream insertion operator must be declared in the same namespace.

1 When the compiler sees a `<<` operator that has an object of type `std::ostream`
 2 on its left hand side and an object of type `Point` on its right hand side, then the
 3 compiler will look for a function named `operator<<` whose first argument is of
 4 type `std::ostream&` and whose second argument is of type `Point const&`.
 5 If it finds such a function it will call that function to do the work; if it cannot
 6 find such a function it will issue a compiler diagnostic.

7 The reason that the function returns a `std::ostream&` is that this is the
 8 C++ convention that permits us to chain together multiple instances of the `<<`
 9 operator:

```
10 Point p0(1,2), p1(3,4);
11 std::cout << p0 << `` `` << p1 << std::endl;
```

12 The C++ compiler parses this left to right. First it recognizes:

```
13 std::cout << p0;
```

14 and calls our stream insertion operator to do this work. Then it thinks of the
 15 rest of the line as:

```
16 std::cout << `` `` << p1 << std::endl;
```

17 Now it recognizes,

```
18 std::cout << `` ``;
```

19 and calls the appropriate stream insertion operator to do that work. And so
 20 on.

21 Look at the implementation of the stream insertion operator in `Point.cc`.
 22 The first argument, `ost`, is a reference to an object of type output stream; the
 23 name `ost` has no meaning to C++; it is just a variable. When writing this
 24 operator we don't know and don't care what the output stream is connected
 25 to; perhaps it is a file; perhaps it is standard output. In any case, you send
 26 output to `ost` just as you do to `std::cout`, which is just another object of
 27 type `std::ostream`. In this example we chose to enclose the values of `x_` and
 28 `y_` in parentheses and to separate them with a comma; this is simply our choice,
 29 not something required by C++ or by *art*.

30 In this example, the stream insertion operator does *not* end by inserting a
 31 newline into `ost`. This is a very common choice as it allows the user of the
 32 operator to have full control about line breaks. For a class whose printout is
 33 very long and covers many lines, you might decide that this operator should end
 34 by inserting newline character; it's your choice.

35 If you wish to write a stream insertion operator for another class, just follow
 36 the pattern used here.

37 If you want to understand more about why the operator is written the way that
 38 it is, consult the standard C++ references; see Section 6.7.



1 The stream insertion operator is a *free function*(γ), not a member function of
 2 the class `Point`; the tie to the class `Point` is via its second argument. Because
 3 this function is a free function, it could have been declared in its own header file
 4 and its implementation could be provided in its own `.cc` file. However that is
 5 not common practice. Instead the common practice is as shown in this example:
 6 to include it in `Point.h` and `Point.cc`.

7 The choice of whether or not to put the declaration of the stream insertion
 8 operator into its own header file is a tradeoff between the following two criteria:
 9

- 10 1. it is convenient to have it there; otherwise you would have to remember
 11 to include an additional header file when you want to use this operator
- 12 2. one can imagine many simple free functions that take an object of type
 13 `Point` as an argument. If we put them all inside `Point.h`, and if they
 14 are only infrequently used, then the compiler will waste time processing
 15 those declarations every time `Point.h` is included somewhere.

16 Ultimately this is a judgement call and the code in this example follows the
 17 recommendations made by the *art* development team. Their recommendation
 18 is that the following sorts of free functions, and only these sorts, should be
 19 included in header files containing a class declaration:

- 20 1. the stream insertion operator for that class
- 21 2. out of class arithmetic and comparison operators

22 With one exception, if including a function declaration in `Point.h` requires the
 23 inclusion of an additional header in `Point.h`, declare that function in a different
 24 header file. The exception is that it is okay to include `<iosfwd>`.

25 6.6.11 Review

26 The class `Point` is an example of a class that is primarily concerned with
 27 providing convenient access to the data it contains. Not all classes are like
 28 this; when you work through the Workbook, you will write some classes that
 29 are primarily concerned with packaging convenient access to a set of related
 30 functions.

- 31 1. class
- 32 2. object
- 33 3. identifier
- 34 4. free function
- 35 5. member function

6.7 C++ References

This section lists some recommended C++ references, both text books and online materials.

The following references describe the C++ core language,

- Stroustrup, Bjarne: “The C++ Programming Language, Special Third Edition”, Addison-Wesley, 2000. ISBN 0-201-70073-5.
- <http://www.cplusplus.com/doc/tutorial/>

The following references describe the C++ Standard Library,

- Josuttis, Nicolai M., “The C++ Standard Library: Tutorial and Reference”, Addison-Wesley, 1999. ISBN 0-201-37926-0.
- <http://www.cplusplus.com/reference>

The following contains an introductory tutorial. Many copies of this book are available at the Fermilab library. It is a very good introduction to the big ideas of C++ and Object Oriented Programming but it is not a fast entry point to the C++ skills needed for HEP.

- Andrew Koenig and Barbara E. Moo, “Accelerated C++: Practical Programming by Example” Addison-Wesley, 2000. ISBN 0-201-70353-X.

The following contains a discussion of recommended best practices,

- Herb Sutter and Andrei Alexandrescu, “C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.”, Addison-Wesley, 2005. ISBN 0-321-11358-6.

7 Using External Products in UPS

Section 3.6.8 introduced the idea of external products. For the Intensity Frontier experiments (and for Fermilab-based experiments in general), access to external products is provided by a Fermilab-developed product-management package called Unix Product Support (UPS). An important UPS feature – demanded by most experiments as their code evolves – is its support for multiple versions of a product and multiple builds (e.g., for different platforms) per version.

Another notable feature is its capacity to handle multiple databases of products. So, for example, on Fermilab computers, login scripts (see Section 4.9) set up the UPS system, providing access to a database of products commonly used at Fermilab.



The *art* Workbook and your experiment’s code will require additional products (available in other databases). For example, each experiment will provide a copy of the toyExperiment product in its experiment-specific UPS database.

In this chapter you will learn how to see which products UPS makes available, how UPS handles variants of a given product, how you use UPS to initialize a product provided in one of its databases and about the environment variables that UPS defines.

7.1 The UPS Database List: PRODUCTS

The act of setting up UPS defines a number of environment variables (discussed in Section 7.5), one of which is `PRODUCTS`. This particularly important environment variable merits its own section.

The environment variable `PRODUCTS` is a colon-delimited list of directory names, i.e., it is a path (see Section 4.6). Each directory in `PRODUCTS` is the name of a *UPS database*, meaning simply that each directory functions as a repository of information about one or more products. When UPS looks for a product, it checks each directory in `PRODUCTS`, in the order listed, and takes the first match.



1 If you are on a Fermilab machine, you can look at the value of `PRODUCTS`
 2 just after logging in, before sourcing your site-specific setup script. Run
 3 `printenv`:

4 `$ printenv PRODUCTS`

5 It should have a value of

6 `/grid/fermiapp/products/common/db`

7 This generic Fermilab UPS database contains a handful of software products
 8 commonly used at Fermilab; most of these products are used by all of the
 9 Intensity Frontier Experiments. This database does not contain any of the
 10 experiment-specific software nor does it contain products such as *ROOT*(γ),
 11 *Geant4*(γ), CLHEP or *art*. While these last few products are indeed used by
 12 multiple experiments, they are often custom-built for each experiment and as
 13 such are distributed via the experiment-specific (i.e., separate) UPS databases.

14 After you source your site-specific setup script, look at `PRODUCTS` again. It
 15 will probably contain multiple directories, thus making many more products
 16 available in your “site” environment. For example, on the DS50+Fermilab site,
 17 after running the DS50 setup script, `PRODUCTS` contains:

18 `/ds50/app/products/:grid/fermiapp/products/common/db`

19 You can see which products `PRODUCTS` contains by running `ls` on its directories,
 20 one-by-one, e.g.,

21 **`ls /grid/fermiapp/products/common/db`**

```
22
23 afs      git      ifdhc      mu2e      python     shrc      ups
24 cpn      gitflow  jobsub_tools  oracle_tnsnames  sam_web_client  upd
25 encp     gits     login      perl      setpath    upd_libs
```

26
 27 `$ ls /ds50/app/products`

```
28 art      cetpkggsupport  g4neutronxs  libxml2      totalview
29 artdaq    clhep           g4nucleonxs  messagefacility  toyExperiment
30 art_suite cmake           g4photon     mpich         upd
31 art_workbook_base  cpp0x          g4pii        mvapich2      ups
32 boost     cppunit         g4radiative  python         xerces_c
33 caencomm  ds50daq         g4surface    root           xmlrpc_c
34 caendigitizer  fftw           gcc           setup
35 caenvme    fhiclcpp        gccxml       smc_compiler
36 cetbuildtools  g4emlow        geant4       sqlite
37 cetlib     g4neutron      libsigcpp    tbb
```

38 Each directory name in these listings corresponds to the name of a UPS product.
 39 If you are on a different experiment, the precise contents of your experiment’s
 40 product directory may be slightly different. Among other things, both databases

1 contain a subdirectory named `ups`¹; this is for the UPS system itself. In this
 2 sense, all these products, including *art*, `toyExperiment` and even the product(s)
 3 containing your experiment's code, regard UPS as just another external prod-
 4 uct.

5 7.2 UPS Handling of Variants of a Product

6 An important feature of UPS is its capacity to make multiple variants of a
 7 product available to users. This of course includes different versions, but beyond
 8 that, a given version of a product may be built more than one way, e.g., for use by
 9 different operating systems (what UPS distinguishes as *flavors*). For example, a
 10 product might be built once for use with SLF5 and again for use with SLF6. A
 11 product may be built with different versions of the C++ compiler, e.g., with the
 12 production version and with a version under test. A product may be built with
 13 full compiler optimization or with the maximum debugging features enabled.
 14 Many variants can exist. UPS provides a way to select a particular build via an
 15 idea named *qualifiers*.

16 The full identifier of a UPS product includes its product name, its version, its
 17 flavor and its full set of qualifiers. In Section 7.3, you will see how to fully
 18 identify a product when you set it up.

19 7.3 The `setup` Command: Syntax and Func- 20 tion

21 Any given UPS database contains several to many, many products. To select a
 22 product and make it available for use, you use the `setup` command.

23 In most cases the correct flavor can be automatically detected by `setup` and
 24 need not be specified. However, if needed, flavor, in addition to various quali-
 25 fiers and options can be specified. These are listed in the UPS documentation
 26 referenced later in this section. The version, if specified, must directly follow
 27 the product name in the command line, e.g.,:

```
28 $ setup <options> <product-name> <product-version> -f <flavor> \  
29 -q <qualifiers>
```

30 Putting in real-looking values, it would look something like:

```
31 $ setup -R myproduct v3_2 -f SLF5 -q BUILD_A
```

32 What does the `setup` command actually do? It may do any or all of the
 33 following:

¹ups appears in both listings; as always, the first match wins!

- 1 • define some environment variables
- 2 • define some bash functions
- 3 • define some aliases
- 4 • add elements to your PATH
- 5 • setup additional products on which it depends

6 Setting up dependent products works recursively. In this way, a single `setup`
7 command may trigger the setup of, say, 15 or 20 products.

8 When you follow a given site-specific setup procedure, the `PRODUCTS` environ-
9 ment variable will be extended to include your experiment-specific UPS reposi-
10 tory.



11 `setup` is a bash function (defined by the UPS product when it was initialized)
12 that shadows a Unix system-configuration command also named `setup`, usually
13 found in `/usr/bin/setup` or `/usr/sbin/setup`. Running the right '`setup`'
14 should work automatically as long as UPS is properly initialized. If it's not,
15 `setup` returns the error message:

16 You are attempting to run ``setup`` which requires administrative
17 privileges, but more information is needed in order to do so.

18 If this happens, the simplest solution is to log out and log in again.

19 Few people will need to know more than the above about the UPS system.
20 Those who do can consult the full UPS documentation at:

21 <http://www.fnal.gov/docs/products/ups/ReferenceManual/index.html>

22 7.4 Current Versions of Products

23 For some UPS products, but not all, the site administrator may define a partic-
24 ular fully-qualified version of the product as the default version. In the language
25 of UPS this notion of default is called the *current* version. If a current version
26 has been defined for a product, you can set up that product with the com-
27 mand:

28 \$ `setup <product-name>`

29 When you run this, the UPS system will automatically insert the version and
30 qualifiers of the version that has been declared current.

31 Having a current version is a handy feature for products that add convenience
32 features to your interactive environment; as improvements are added, you au-
33 tomatically get them.

1 However the notion of a current version is very dangerous if you want to ensure
 2 that software built at one site will build in exactly the same way on all other
 3 sites. For this reason, the Workbook fully specifies the version number and
 4 qualifiers of all products that it requires; and in turn, the products used by
 5 the Workbook make fully qualified requests for the products on which they
 6 depend.



7 7.5 Environment Variables Defined by UPS

8 When your login script or site-specific setup script initializes UPS, it defines
 9 many environment variables in addition to `PRODUCTS` (Section 7.1), one of
 10 which is `UPS_DIR`, the root directory of the currently selected version of UPS.
 11 The script also adds `$UPS_DIR/bin` to your `PATH`, which makes some UPS-
 12 related commands visible to your shell. Finally, it defines the bash function
 13 `setup` (see Sections 4.8 and 7.3). When you use the `setup` command, as
 14 illustrated below, it is this bash function that does the work.

15 In discussing the other important variables, the `toyExperiment` product will be
 16 used as an example product. For a different product, you would replace “toy-
 17 Experiment” or “`TOYEXPERIMENT`” in the following text by the product’s
 18 name. Once you have followed your appropriate setup procedure (Table 5.1)
 19 you can issue the following command this is informational for the purposes of
 20 this section; you don’t need to do it until you start running the first Workbook
 21 exercise):

```
22 $ setup toyExperiment v0.00.14 -qe2:prof
```

23 The version and qualifiers shown here are the ones to use for the Workbook exer-
 24 cises. When the `setup` command returns, the following environment variables
 25 will be defined:

26 **`TOYEXPERIMENT_DIR`** defines the root `DIR`ectory of the chosen UPS
 27 product

28 **`TOYEXPERIMENT_INC`** defines the path to the root directory of the C++
 29 header files that are provided by this product (so called because the header
 30 files are `IN`cluded)

31 **`TOYEXPERIMENT_LIB`** defines the directory that contains all of the share-
 32 able object `LIB`raries (ending in `.so`) that are provided by this product

33 Almost all UPS products that you will use in the Workbook define these three
 34 environment variables. Several, including `toyExperiment`, define many more.
 35 Once you’re running the exercises, you will be able to see all of the environ-
 36 ment variables defined by the `toyExperiment` product by issuing the following
 37 command:

```
38 $ printenv | grep TOYEXPERIMENT
```



1 Many software products have version numbers that contain dot characters. UPS
2 requires that version numbers not contain any dot characters; by convention,
3 version dots are replaced with underscores. Therefore `v0.00.09` becomes
4 `v0_00_09`. Also by convention, the environment variables are all upper case,
5 regardless of the case used in the product names.

6 7.6 Finding Header Files

7 7.6.1 Introduction

8 *Header files* were introduced in Section 6.3.2. Recall that a header file typically
9 contains the “parts list” for its associated `.cc` source file and is “included” in
10 the `.cc` file.

11 The software for the Workbook depends on a large number of external products;
12 the same is true, on an even larger scale, for the software in your experiment.
13 The preceding sections in this chapter discussed how to establish a working
14 environment in which all of these software products are available for use.

15 When you are working with the code in the Workbook, and when you are
16 working on your experiment, you will frequently encounter C++ classes and
17 functions that come from these external products. An important skill is to be
18 able to identify them when you see them and to be able to follow the clues
19 back to their source and documentation. This section will describe how to do
20 that.

21 An important aid to finding documentation is the use of *namespaces*; if you are
22 not familiar with namespaces, see Section 31.6 or consult the standard C++
23 documentation.

24 7.6.2 Finding *art* Header Files

25 This subsection will use the example of the class `art::Event` to illustrate how
26 to find header files from the *art* UPS product; this will serve as a model for
27 finding header files from most other UPS products.

28 The class that holds the *art* abstraction of an HEP event is named, `art::Event`;
29 that is, the class `Event` is in the namespace `art`. In fact, all classes and func-
30 tions defined by *art* are in the namespace `art`. The primary reason for this
31 is to minimize the chances of accidental name collisions between *art* and other
32 codes; but it also serves a very useful documentation role and is one of the clues
33 you can use to find header files.

34 If you look at code that uses `art::Event` you will almost always find that the
35 file includes the following header file:

Part

```

1  #include "art/Framework/Principal/Event.h"
2
3  The art UPS product has been designed so that the relative path used to include
4  any art header file starts with the directory art; this is another clue that the
5  class or function of interest is part of art.
6
7  When you setup the art UPS product, it defines the environment variable
8  ART_INC, which points to the root of the header file tree for art. You now have
9  enough information to discover where to find the header file for art::Event;
10 it is at
11
12 $ART_INC/art/Framework/Principal/Event.h
13
14 You can follow this same pattern for any class or function that is part of art.
15 This will only work if you are in an environment in which ART_INC has been
16 defined, which will be described in Chapters 9 and 10.
17
18 If you are a C++ beginner, you will likely find this header file difficult to un-
19 derstand; you do not need to understand it when you first encounter it but, for
20 future reference, you do need to know where to find it.
21
22 Earlier in this section, you read that if a C++ file uses art::Event, it would
23 almost always include the appropriate header file. Why almost always? Because
24 the header file Event.h might already be included within one of the other
25 headers that are included in your file. If Event.h is indirectly included in this
26 way, it does not hurt also to include it explicitly, but it is not required that you
27 do so.2
28
29 We can summarize this discussion as follows: if a C++ source file uses art::Event
30 it must always include the appropriate header file, either directly or indirectly.
31
32 art does not rigorously follow the pattern that the name of file is the same as
33 the name of the class or function that it defines. The reason is that some files
34 define multiple classes or functions; in most such cases the file is named after
35 the most important class that it defines.
36
37 Finally, from time to time, you will need to dig through several layers of header
38 files to find the information you need.
39
40 There are two code browsing tools that you can use to help navigate the layering
41 of header files and to help find class declarations that are not in a file named
42 for the class:
43
44 1. use the art redmine( $\gamma$ ) repository browser:
45    https://cdcv.sfnal.gov/redmine/projects/art/repository/revisions/master/show/art
46
47 2. use the LXR code browser: http://cdcv.sfnal.gov/lxr/art/
48
49 (In the above, both URLs are live links.)

```

² Actually there is small price to pay for redundant includes; it makes the compiler do unnecessary work, and therefore slows it down. But providing some redundant includes as a pedagogical tool is often a good trade-off; the Workbook will frequently do this.

7.6.3 Finding Headers from Other UPS Products

Section 3.7 introduced the idea that the Workbook is built around a UPS product named `toyExperiment`, which describes a made-up experiment. All classes and functions defined in this UPS product are defined in the namespace `tex`, which is an acronym-like shorthand for `toyExperiment` (`ToyEXperiment`). (This shorthand makes it (a) easier to focus on the name of each class or function rather than the namespace and (b) quicker to type.)

One of the classes from the `toyExperiment` UPS product is `tex::GenParticle`, which describes particles created by the event generator, the first part of the simulation chain (see Section 3.7.2). The include directive for this class looks like

```
#include "toyExperiment/MCDataProducts/GenParticle.h"
```

As for headers included from *art*, the first element in the relative path to the included file is the name of the UPS product in which it is found. Similarly to *art*, the header file can be found using the environment variable `TOYEXPERIMENT_INC`:

```
$TOYEXPERIMENT_INC/toyExperiment/MCDataProducts/GenParticle.h
```

With a few exceptions, discussed in Section 7.6.4, if a class or function from a UPS product is used in the Workbook code, it will obey the following pattern:

1. The class will be in a namespace that is unique to the UPS product; the name of the namespace may be the full product name or a shortened version of it.
2. The lead element of the path specified in the include directive will be the name of the UPS product.
3. The UPS product setup command will define an environment variable named `<PRODUCT-NAME>_INC`, where `<PRODUCT-NAME>` is in all capital letters.

Using this information, the name of the header file will always be

```
$<PRODUCT-NAME>_INC/<path-specified-in-the-include-directive>
```

This pattern holds for all of the UPS products listed in Table 7.1.

A table listing git- and LXR-based code browsers for many of these UPS products can be found near the top of the web page:

<https://cdcvns.fnal.gov/redmine/projects/art/wiki>

7.6.4 Exceptions: The Workbook, ROOT and Geant4

There are three exceptions to the pattern described in Section 7.6.3:

Part

Table 7.1: For selected UPS Products, this table gives the names of the associated namespaces. The UPS products that do not use namespaces are discussed in Section 7.6.4. [‡]The namespace `tex` is also used by the *art* Workbook, which is not a UPS product.

UPS Product	Namespace
art	<code>art</code>
boost	<code>boost</code>
cet	<code>cetlib</code>
clhep	<code>CLHEP</code>
fhiclcpp	<code>fhicl</code>
messagefacility	<code>mf</code>
toyExperiment	<code>tex</code> [‡]

- 1 • the Workbook itself
- 2 • **ROOT**
- 3 • Geant4

4 The Workbook is so tightly coupled to the `toyExperiment` UPS product that all
5 classes in the Workbook are also in its namespace, `tex`. Note, however, that
6 classes from the Workbook and the `toyExperiment` UPS product can still be
7 distinguished by the leading element of the relative path found in the `include`
8 directives for their header files:

- 9 • `art-workbook` for the Workbook
- 10 • `toyExperiment` for the `toyExperiment`

11 The **ROOT** package is a CERN-supplied software package that is used by *art*
12 to write data to disk files and to read it from disk files. It also provides many
13 data analysis and data presentation tools that are widely used by the HEP com-
14 munity. Major design decisions for **ROOT** were frozen before namespaces were
15 a stable part of the C++ language, therefore **ROOT** does not use namespaces.
16 Instead **ROOT** adopts the following conventions:

- 17 1. All class names by defined by **ROOT** start with the capital letter T
18 followed by another upper case letter; for example, `TFile`, `TH1F`, and
19 `TCanvas`.
- 20 2. With very few exceptions, all header files defined by **ROOT** also start with
21 the same pattern; for example, `TFile.h`, `TH1F.h`, and `TCanvas.h`.
- 22 3. The names of all global objects defined by **ROOT** start with a lower case
23 letter `g` followed by an upper case letter; for example `gDirectory`, `gPad`
24 and `gFile`.

25 The rule for writing an `include` directive for a header file from **ROOT** is to write
26 its name without any leading path elements:


```
1  #include "TFile.h"
```

2 All of the ROOT header files are found in the directory that is pointed to by
3 the environment variable \$ROOT_INC. For example, to see the contents of this
4 file you could enter:

```
5 $ less $ROOT_INC/TFile.h
```

6 Or you can learn about this class using the reference manual at the CERN
7 web site: <http://root.cern.ch/root/html534/ClassIndex.html>

8 You will not see the **Geant4** package in the Workbook but it will be used
9 by the software for your experiment, so it is described here for completeness.
10 **Geant4** is a toolkit for modeling the propagation particles in electromagnetic
11 fields and for modeling the interactions of particles with matter; it is the core of
12 all detector simulation codes in HEP and is also widely used in both the Medical
13 Imaging community and the Particle Astrophysics community.

14 As with **ROOT**, **Geant4** was designed before namespaces were a stable part of
15 the C++ language. Therefore **Geant4** adopted the following conventions.

- 16 1. The names of all identifiers begin with G4; for example, G4Step and
17 G4Track.
- 18 2. All header files defined by Geant4 begin with G4; for example, G4Step.h
19 and G4Track.h.

20 Most of the header files defined by **Geant4** are found in a single directory, which
21 is pointed to by the environment variable G4INCLUDE.

22 The rule for writing an include directive for a header file from **Geant4** is to
23 write its name without any leading path elements:

```
24 #include "G4Step.h"
```

25 The workbook does not set up a version of Geant4; therefore G4INCLUDE is
26 not defined. If it were, you would look at this file by:

```
27 $ less $G4INCLUDE/G4Step.h
```

28 Both **ROOT** and **Geant4** define many thousands of classes, functions and
29 global variables. In order to avoid collisions with these identifiers, do not
30 define any identifiers that begin with any of (case-sensitive):

- 31 • T, followed by an upper case letter
- 32 • g, followed by an upper case letter
- 33 • G4

1

Part II

2

Workbook

8 Preparation for Running the Workbook Exercises

8.1 Introduction

You will run the Workbook exercises on a computer that is maintained by your experiment, either at Fermilab or at another institution. Many details of the working environment change from site to site¹ and these differences are parameterized so that (a) it is easy to establish the required environment, and (b) the Workbook exercises work the same way at all sites. In this chapter you will learn how to find and log into the right machine remotely from your local machine (laptop or desktop), and make sure it can support your Workbook work.

Note that it is possible to install the Workbook software on your local (Unix-like) machine; instructions are available at [. The instructions in this document will work whether the Workbook code is installed locally or on a remote machine.](#)



8.2 Getting Computer Accounts on Workbook-enabled Machines

In order to run the exercises in the Workbook, you will need an account on a machine that can access your site's installation of the Workbook code. The experiments provide instructions for getting computer accounts on their machines (and various other information for new users) on web pages that they maintain, as listed in Table 8.1. The URLs in the table are live hyperlinks.

Currently, each of the experiments using *art* has installed the Workbook code on one of its experiment machines in the Fermilab General Purpose Computing Farm (GPCF).



¹Remember, a *site* refers to a unique combination of experiment and institution.

Table 8.1: Experiment-specific Information for New Users

Experiment	URL of New User Page
ArgoNeut	https://cdcv.s.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Darkside	https://cdcv.s.fnal.gov/redmine/projects/darkside-public/wiki/Before_You_Arrive
LArSoft	https://cdcv.s.fnal.gov/redmine/projects/larsoftsvn
LBNE	https://cdcv.s.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
MicroBoone	https://cdcv.s.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Muon g-2	https://cdcv.s.fnal.gov/redmine/projects/g-2/wiki/NewGm2Person
Mu2e	http://mu2e.fnal.gov/atwork/general/userinfo/index.shtml#comp
NOvA	http://www-nova.fnal.gov/NOvA_Collaboration_Information/index.html

Table 8.2: Login machines for running the Workbook exercises

Experiment	Name of Login Node
ArgoNeut	<code>argoneutvm.fnal.gov</code>
Darkside	<code>ds50.fnal.gov</code>
LBNE	<code>lbnevm.fnal.gov</code>
MicroBoone	<code>uboonevm.fnal.gov</code>
Muon g-2	<code>gm2gpvm.fnal.gov</code>
Mu2e	<code>mu2evm.fnal.gov</code>
NOvA	<code>nova-offline.fnal.gov</code>

1 At time of writing, the new-user instructions for all LArSoft-based experiments
2 are at the LArSoft site; there are no separate instructions for each experi-
3 ment.

4 If you would like a computer account on a Fermilab computer in order to eval-
5 uate *art*, contact the *art* team (see Section 3.4).

6 8.3 Choosing a Machine and Logging In

7 The experiment-specific machines confirmed to host the Workbook code are
8 listed in Table 8.2 In most cases the name given is not the name of an actual
9 computer, but rather a round-robin alias for a cluster. For example, if you
10 log into `mu2evm`, you will actually be connected to one of the five computers
11 `mu2egpvm01` through `mu2egpvm05`. These Mu2e machines share all disks that
12 are relevant to the Workbook exercises, so if you need to log in multiple times,
13 it is perfectly OK if you are logged into two different machines; you will still see
14 all of the same files.

15 Each experiment's web page has instructions on how to log in to its computers
16 from your local machine.

8.4 Launching new Windows: Verify X Connectivity

Some of the Workbook exercises will launch an X window from the remote machine that opens in your local machine. To test that this works, type `xterm &`:

```
$ xterm &
```

This should, without any messages, give you a new command prompt. After a few seconds, a new shell window should appear on your laptop screen; if you are logging into a Fermilab computer from a remote site, this may take up to 10 seconds. If the window does not appear, or if the command issues an error message, contact a computing expert on your experiment.



To close the new window, type `exit` at the command prompt in the new window:

```
$ exit
```

If you have a problem with `xterm`, it could be a problem with your Kerberos and/or `ssh` configurations. Try logging in again with `ssh -Y`.



8.5 Choose an Editor

As you work through the Workbook exercises you will need to edit files. Familiarize yourself with one of the editors available on the computer that is hosting the Workbook. Most Fermilab computers offer four reasonable choices: `emacs`, `vi`, `vim` and `nedit`. Of these, `nedit` is probably the most intuitive and user-friendly. All are very powerful once you have learned to use them. Most other sites offer at least the first three choices. You can always contact your local system administrator to suggest that other editors be installed.

A future version of this documentation suite will include recommended configurations for each editor and will provide links to documentation for each editor.

9 Exercise 1: Run Pre-built *art* Modules

9.1 Introduction

In this first exercise of the Workbook, you will be introduced to the *FHiCL*(γ) configuration language and you will run *art* on several modules that are distributed as part of the toyExperiment UPS product. You will not compile or link any code.

9.2 Prerequisites

Before running any of the exercises in this Workbook, you need to be familiar enough with the material discussed in Part I (Introduction) of this documentation set and Chapter 8 to be able to find information as needed.

If you are following the instructions below on a Mac computer, and if you are reading the instructions from a PDF file, be aware that if you use the mouse or trackpad to cut and paste text from the PDF file into your terminal window, the underscore characters will be turned into spaces. You will have to fix them before the commands will work.



9.3 What You Will Learn

In this exercise you will learn:

1. when to use the site-specific setup procedure
2. how to set up the toyExperiment UPS product
3. how to run an *art* job
4. how to control the number of events to process
5. how to select different input files

- 1 6. how to start at an event that is not the first event in the file
- 2 7. how to concatenate input files
- 3 8. how to write an output file
- 4 9. some basics about the grammar and structure of a FHiCL file
- 5 10. a little bit about the *art* run-time environment

6 9.4 Running the Exercise

7 9.4.1 The Pieces

8 Several event-data input files have been provided for use by the Workbook
 9 exercises. These input files are packaged as part of the toyExperiment UPS
 10 product. Table 9.1 lists the range of event IDs found in each file. You will need
 11 to refer back to this table as you proceed.

Table 9.1: The input files provided by for the Workbook exercises

File Name	Run	SubRun	Range of Event Numbers
input01_data.root	1	0	1...10
input02_data.root	2	0	1...10
input03_data.root	3	0	1...5
	3	1	1...5
	3	2	1...5
input04_data.root	4	0	1...1000

12 A run-time configuration (FHiCL) file has also been provided, `hello.fcl`.

13 9.4.2 Log In, Set Up and Execute *art*

14 The intent of this section is for the reader to start from “zero” and execute
 15 an *art* job, without necessarily understanding each step, just to get familiar
 16 with the process. A detailed discussion of what these steps do will follow in
 17 Section 9.8.

18 Some steps are written as statements, others as commands to issue at the
 19 prompt. Notice that *art* takes the argument `-c hello.fcl`; this points *art*
 20 to the run-time configuration file that will tell it what to do and where to find
 21 the “pieces” on which to operate.

22 Most readers: Follow the steps in Section 9.4.2.1, then proceed directly to Sec-
 23 tion 9.6.

- 1 If you wish to manage your working directory yourself, skip Section 9.4.2.1,
- 2 follow the steps in Section 9.4.2.2, then proceed to Section 9.6.
- 3 If you log out and wish to log back in, follow the procedure outlined in Sec-
- 4 tion 10.6.



5 9.4.2.1 Standard Procedure

- 6 1. Log in to the computer you chose in Section 8.3.
- 7 2. Follow the site-specific setup procedure; see Table 5.1.
- 8 3. `$ mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built`
9 In the above and elsewhere as indicated, substitute your kerberos principal
10 for the string `<username>`.
- 11 4. `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built`
- 12 5. `$ setup toyExperiment v0.00.14 -q$ART_WORKBOOK_QUAL:prof`
- 13 6. `$ cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .`
- 14 7. `$ source makeLinks.sh`
- 15 8. `$ art -c hello.fcl >& output/hello.log`
- 16 Proceed to Section 9.6.

17 9.4.2.2 Procedure allowing Self-managed Working Directory

- 18 1. Log in to the computer you chose in Section 8.3.
- 19 2. Follow the site specific setup procedure; see Table 5.1
- 20 3. Make a working directory and `cd` to it.
- 21 4. `setup toyExperiment v0.00.14 -q$ART_WORKBOOK_QUAL:prof`
- 22 5. `cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .`
- 23 6. Make a subdirectory named `output`. If you prefer you can make this
24 on some other disk and put a symbolic link to it, named `output`, in the
25 current working directory.
- 26 7. `ln -s $TOYEXPERIMENT_DIR/inputFiles .`
- 27 8. `art -c hello.fcl`

9.5 Logging In Again

If you log out and later wish to log in again to work on this or any other exercise, you need to do the following:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Section 5.
3. `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built`
4. `$ setup toyExperiment v0.00.14 -q$ART_WORKBOOK_QUAL:prof`

Compare this with the list given in Section 9.4.2. You will see that three steps are missing because they only need to be done the first time.

You are now ready to run *art* as you were before.

9.6 Examine Output

Compare the output you produced against Listing 9.1; the only differences should be the timestamps. It also processed the first file listed in Table 9.1.

Listing 9.1: Sample output from running `hello.fcl`

```

1 MSG-i MF_INIT_OK: art 27-Apr-2013 21:22:13 CDT JobSetup
2 MessageLogger initialization complete.
3 MSG
4 27-Apr-2013 21:22:14 CDT Initiating request to open file
5 inputFiles/input01_data.root
6 27-Apr-2013 21:22:14 CDT Successfully opened file
7 inputFiles/input01_data.root
8 Begin processing the 1st record. run: 1 subRun: 0 event: 1 at
9 27-Apr-2013 21:22:14 CDT
10 Hello World! This event has the id: run: 1 subRun: 0 event: 1
11 Begin processing the 2nd record. run: 1 subRun: 0 event: 2 at
12 27-Apr-2013 21:22:14 CDT
13 Hello World! This event has the id: run: 1 subRun: 0 event: 2
14 Hello World! This event has the id: run: 1 subRun: 0 event: 3
15 Hello World! This event has the id: run: 1 subRun: 0 event: 4
16 Hello World! This event has the id: run: 1 subRun: 0 event: 5
17 Hello World! This event has the id: run: 1 subRun: 0 event: 6
18 Hello World! This event has the id: run: 1 subRun: 0 event: 7
19 Hello World! This event has the id: run: 1 subRun: 0 event: 8
20 Hello World! This event has the id: run: 1 subRun: 0 event: 9
21 Hello World! This event has the id: run: 1 subRun: 0 event: 10
22 27-Apr-2013 21:22:14 CDT Closed file inputFiles/input01_data.root
23
24
25 TrigReport ----- Event Summary -----
26 TrigReport Events total = 10 passed = 10 failed = 0
27
28 TrigReport ----- Modules in End-Path: el -----

```

Part

```

29 TrigReport  Trig Bit#    Visited    Passed    Failed    Error Name
30 TrigReport      0      0         10         10         0         0 hi
31
32 TimeReport  ----- Time  Summary ---[sec]----
33 TimeReport CPU = 0.004000 Real = 0.002411
34
35 Art has completed and will exit with status 0.

```

8 Every time you run *art*, the first thing to check is the last line in your out-
 9 put or log file. It should be Art has completed and will exit with
 10 status 0. If the status is not 0, or if this line is missing, it is an error; please
 11 contact the *art* team as described in Section 3.4.

12 *A future version of these instructions will specify how much disk space is needed,*
 13 *including space for all ouptut files.*

14 9.7 Understanding the Configuration File `hello.fcl`

15 The file `hello.fcl` gives *art* its run-time configuration. This file is writ-
 16 ten in the Fermilab Hierarchical Configuration Language (FHiCL, pronounced
 17 “fickle”), a language that was developed at Fermilab to support run-time config-
 18 uration for several projects, including *art*. By convention, files written in FHiCL
 19 end in `.fcl`. As you work through the Workbook, the features of FHiCL that
 20 are relevant for each exercise will be explained.

21 The full details of the FHiCL language, plus the details of how it is used by *art*,
 22 are given in the Users Guide, Chapter 24. Most people will find it much easier
 23 to follow the discussion in the Workbook documentation than to digest the full
 24 documentation up front.



25 9.7.1 Some Bookkeeping Syntax

26 In a FHiCL file, the start of a comment is marked by the hash sign character
 27 (`#`); a comment may begin in any column.

28 The hash sign has one other use, however. If the first eight characters of a line
 29 are exactly `#include`, followed by whitespace and a quoted list of file paths,
 30 then the line will be interpreted as an *include directive* and the line containing it
 31 will be replaced by the contents of the file named in the include directive.

32 The basic element of FHiCL is the *definition*, which has the form

```
33 name : value
```

34 A group of FHiCL definitions delimited by braces `{}` is called a *table*(γ). Within
 35 *art*, a FHiCL table gets turned into a C++ object called a *parameter set*(γ);
 36 this document set will often refer to a FHiCL table as a parameter set.



- 1 The fragment of `hello.fcl` shown in Listing 9.2 contains the FHiCL table that
 2 configures the *source*(γ) of events that *art* will read in and operate on.

Listing 9.2: The source parameter set from `hello.fcl`

```

1 source : {
2   module_type : RootInput
3   fileNames   : [ "inputFiles/input01_data.root" ]
4 }
```



- 7 The name *source* is a *keyword* in *art*; i.e., the name *source* has no special
 8 meaning to FHiCL but it does have a special meaning to *art*. To be precise, it
 9 only has a special meaning to *art* if it is at the outermost *scope*(γ) of a FHiCL
 10 file; i.e., not inside any braces `{}` within the file. The notion of *scope* in FHiCL is
 11 discussed further in Chapter 12. When *art* sees a parameter set named *source*
 12 at the outermost scope, then *art* will interpret that parameter set to be the
 13 description of the source of events for this run of *art*.

- 14 In the *source* parameter set, the identifier *module_type* is a keyword in *art*
 15 that tells *art* the name of a module that it should load and run, *RootInput* in
 16 this case. *RootInput* is one of the standard source modules provided by *art*
 17 and it reads disk files containing event-data written in an *art*-defined ROOT-
 18 based format. The default behaviour of the *RootInput* module is to start at
 19 the first event in the first file and read to the end of the last event in the last
 20 file.¹

- 21 The identifier *fileNames* is again a keyword, but this time defined in the
 22 *RootInput* module, that gives the module a list of filenames from which to read
 23 events. The list is delimited by square brackets and contains a comma-separated
 24 list of filenames. This example shows only one filename, but the square brackets
 25 are still required. The proper FHiCL name for a comma-separated list delimited
 26 by square brackets is a *sequence*(γ).

- 27 In most cases the filenames in the sequence must be enclosed in quotes. FHiCL,
 28 like many other languages has the following rule: if a string contains white
 29 space or any special characters, then quoting it is required, otherwise quotes are
 30 optional.

- 31 FHiCL has its own set of special characters; these include anything *except* all
 32 upper and lower case letters, the numbers 0 through 9 and the underscore char-
 33 acter. *art* restricts the use of the underscore character in some circumstances;
 34 these will be discussed as they arise.

- 35 It is implied in the foregoing discussion that a FHiCL *value* need not be a
 36 simple thing, such as a number or a quoted string. For example, in Listing 9.2,

¹ In the Workbook, the only source *module_type* that you will see will be *RootInput*. Your experiment may have a source module that reads events from the live experiment and other source modules that read files written in experiment-defined formats; for example *Mu2e* has a source module that reads single particle events from a text file written by *G4beamline*.

1 the source value is a parameter set (of two parameters) and the value of
 2 `fileNames` is a (single-item) sequence.

3 9.7.2 Some Physics Processing Syntax

4 The identifier *physics*(γ), when found at the outermost scope, is a keyword in
 5 *art*. The *physics* parameter set is so named because it contains most of the
 6 information needed to describe the physics workflow of an *art* job.

7 The fragment of `hello.fcl` shown in Listing 9.3 shows a rather long-winded
 8 way of telling *art* to find a module named `HelloWorld` and execute it.

Listing 9.3: The *physics* parameter set from `hello.fcl`

```
1 physics :{
2   analyzers: {
3     hi : {
4       module_type : HelloWorld
5     }
6   }
7   e1      : [ hi ]
8   end_paths : [ e1 ]
9 }
```

18 Why so long-winded? *art* has very powerful features that enable execution
 19 of multiple complex chains of modules; the price is that specifying something
 20 simple takes a lot of keystrokes.

21 Within the *physics* parameter set, notice the identifier *analyzers*. When
 22 found as a top-level identifier within the *physics* scope, it is recognized as a
 23 keyword in *art*. The *analyzers* parameter set defines the run-time configura-
 24 tion for all of the analyzer modules that are part of the job – only `HelloWorld`
 25 in this case.

26 For our current purposes, the module `HelloWorld` does only one thing of
 27 interest, namely for every event it prints one line:

28 Hello World! This event has the id: run: <RR> subRun: <SS> event: <EE>

29 where RR, SS and EE are substituted with the actual run, subRun and event
 30 number of each event.

31 If you look back at Listing 9.1, you will see that this line appears ten times,
 32 once each for events 1 through 10 of run 1, subRun 0 (as expected, according
 33 to Table 9.1). The remainder of the listing is standard output generated by
 34 *art*.

35 Listing 9.4 shows the remainder of the lines in `hello.fcl`. The line starting
 36 with *process.name*(γ) tells *art* that this job has a name and that the name
 37 is “hello”; it has no real significance in these simple exercises. It becomes
 38 important when an *art* job creates new data products (described in User Guide



Chapter 25) and writes them to a file; each data product will be uniquely identified by a four-part name, one part of which is the name of the process that created the data product. This imposes a constraint on `process_name` values: *art* joins the four parts of a data product name into a single string, with the underscore (`_`) as a separator between fields; none of the parts (e.g., the process name) may contain additional underscores.



In an *art* event-data file, each data product is stored as a TBranch of a *TTree*(γ); the string containing the full name of the data product is used as the name of the TBranch. On readback, *art* must parse the name of the TBranch to recover the four individual pieces of the data product name. If one of the four parts internally contains an underscore, then *art* cannot reliably recover the four parts.

Listing 9.4: The remainder of `hello.fcl`

```
1b #include "fcl/minimalMessageService.fcl"
1c
1d process_name : hello
1e
1f services : {
1g     message : @local::default_message
1h }
```

Listing 9.4 also contains the `services` parameter set, which provides run-time configuration information for all *art* services. For our present purposes, it is sufficient to know that the configuration for the message service is found inside the file that is included via the `#include` line. The message service controls the limiting and routing of debug, informational, warning and error messages generated by *art* or by user code. The message service does not control information written directly to `std::cout` or `std::cerr`.

9.7.3 Command line Options

art supports some command line options. To see what they are, type the following command at the bash prompt

```
$ art --help
```

Note that some options have both a short form and a long form. This is a common convention for Unix programs; the short form is convenient for interactive use and the long form makes scripts more readable.

9.7.4 Maximum Number of Events to Process

By default *art* will read all events from all of the specified input files. You can set a maximum number of events in two ways, one way is from the command line:

Part

```
1 $ art -c hello.fcl -n 5
2 $ art -c hello.fcl --nevents 4
3 Run each of these commands and observe their output.
4 The second way is within the FHiCL file. Start by making a copy of hello.fcl:
5 $ cp hello.fcl hi.fcl
6 Edit hi.fcl and add the following line anywhere in the source parameter
7 set:
8 maxEvents      : 3
9 By convention this is added after the fileNames definition but it can go anywhere
10 inside the source parameter set because the order of parameters within a FHiCL
11 table is not important. Run art again, using hi.fcl:
12 $ art -c hi.fcl
13 You should see output from the HelloWorld module for only the first three
14 events.
15 To configure the file for art to process all the events, i.e., to run until art reaches
16 the end of the input files, either leave off the maxEvents parameter or give it
17 a value of -1.
18 If the maximum number of events is specified both on the command line and in
19 the FHiCL file, then the command line takes precedence. Compare the outputs
20 of the following commands:
21 $ art -c hi.fcl
22 $ art -c hi.fcl -n 5
23 $ art -c hi.fcl -n -1
```



24 9.7.5 Changing the Input Files

25 For historical reasons, there are multiple ways to specify the input event-data
26 file (or the list of input files) to an *art* job:

- 27 • within the FHiCL file's `source` parameter set
- 28 • on the *art* command line via the `-s` option (you may specify one input
29 file only)
- 30 • on the *art* command line via the `-S` option (you may specify a text file
31 that lists multiple input files)
- 32 • on the *art* command line, after the last recognized option (you may specify
33 one or more input files)

34 If input file names are provided both in the FHiCL file and on the command
35 line, the command line takes precedence.



1 Let's run a few examples.

2 We'll start with the `-s` command line option (second bullet). Run *art* without
3 it (again), for comparison (or recall its output from Listing 9.1):

4 `$ art -c hello.fcl`

5 To see what you should expect given the following input file, check Table 9.1,
6 then run:

7 `$ art -c hello.fcl -s inputFiles/input02_data.root`

8 Notice that the 10 events in this output are from run 2 subRun 0, in contrast
9 to the previous printout which showed events from run 1. Notice also that the
10 command line specification overrode that in the FHiCL file. The `-s` (lower case)
11 command line syntax will only permit you to specify a single filename.

12 This time, edit the source parameter set inside the `hi.fcl` file (first bullet);
13 change it to:

```
14     source : {
15         module_type : RootInput
16         fileNames   : [ "inputFiles/input01_data.root",
17                        "inputFiles/input02_data.root" ]
18         maxEvents   : -1
19     }
```

20 (Notice that you also added `maxEvents : -1`.) The names of the two input
21 files could have been written on a single line but this example shows that, in
22 FHiCL, newlines are treated simply as white space.

23 Check Table 9.1 to see what you should expect, then rerun *art* as follows:

24 `$ art -c hi.fcl`

25 You will see 20 lines from the HelloWorld module; you will also see messages
26 from *art* at the open and close operations on each input file.

27 Back to the `-s` command-line option, run:

28 `$ art -c hi.fcl -s inputFiles/input03_data.root`

29 This will read only `inputFiles/input03_data.root` and will ignore the
30 two files specified in the `hi.fcl`. The output from the HelloWorld module
31 will be the 15 events from the 3 subRuns of run 3.

32 There are several ways to specify multiple files at the command line. One
33 choice is to use the `-S` (upper case) [`--source-list`] command line option
34 (third bullet) which takes as its argument the name of a text file containing the
35 filename(s) of the input event-data file(s). An example of such a file has been
36 provided, `inputs.txt`. Look at the contents of this file:

```
37 $ cat inputs.txt
38 inputFiles/input01_data.root
```

Part

```
1  inputFiles/input02_data.root
2  inputFiles/input03_data.root

3  Now run art using inputs.txt to specify the input files:

4  $ art -c hi.fcl -S inputs.txt

5  You should see the HelloWorld output from the 35 events in the three files;
6  you should also see the messages from art about the opening and closing of the
7  three files.

8  Finally, you can list the input files at the end of the art command line (fourth
9  bullet):

10 $ art -c hi.fcl inputFiles/input02_data.root \
11     inputFiles/input03_data.root

12 ( Remember the unix convention about a trailing backslash marking a command
13  that continues on another line; see Chapter 2. ) In this case you should see the
14  HelloWorld output from the 25 events in the two files.

15 In summary, there are three ways to specify input files from the command line;
16 all of them override any input files specified in the FHiCL file. Do not try to
17 use two or more of these methods on a single art command line; the art job will
18 run without issuing any messages but the output will likely be different than
19 you expect.
```



9.7.6 Skipping Events

The source parameter set supports a syntax to start execution at a given event number or to skip a given number of events at the start of the job. Look, for example, at the file *skipEvents.fcl*, which differs from *hello.fcl* by the addition of two lines to the source parameter set:

```
25  firstEvent   : 5
26  maxEvents    : 3
```

art will process events 5, 6, and 7 of run 1, subRun 0. Try it:

```
28 $ art -c skipEvents.fcl
```

An equivalent operation can be done from the command line in two different ways. Try the following two commands and compare the output:

```
31 $ art -c hello.fcl -e 5 -n 3
32 $ art -c hello.fcl --nskip 4 -n 3
```

1 You can also specify the initial event to process relative to a given event ID
 2 (which, recall, contains the run, subRun and event number). Edit `hi.fcl` and
 3 edit the source parameter set as follows:

```

4   source : {
5       module_type : RootInput
6       fileNames   : [ ``inputFiles/input03_data.root`` ]
7       firstRun     : 3
8       firstSubRun  : 1
9       firstEvent   : 6
10  }
```

11 When you run this job, *art* will process events starting from run 3, subRun 2,
 12 event 1, – because there are only 5 events in subRun 1.

```

13 $ art -c hi.fcl
```

14 9.7.7 Identifying the User Code to Execute

15 Recall from Section 9.7.2 that the *physics* parameter set contains the physics
 16 content for the *art* job. Within this parameter set, *art* must be able to determine
 17 which (user code) modules to process. These must be referenced via *module*
 18 *labels*(γ), which as you will see, represent the pairing of a module name and a
 19 run-time configuration.

20 Look back at Listing 9.3, which contains the *physics* parameter set from
 21 `hello.fcl`. The analyzer parameter set, nested inside the *physics* pa-
 22 rameter set, contains the definition:

```

23 hi : {
24     module_type : HelloWorld
25 }
```

26 The identifier *hi* is a module label (defined by the user, not by FHiCL or *art*)
 27 whose value must be a parameter set that *art* will use to configure a module.
 28 The parameter set for a module label must contain (at least) a FHiCL definition
 29 of the form:

```

30 module_type : <module-name>
```

31 Here *module_type* is a keyword in *art* and *<module-name>* tells *art* the
 32 name of the module to load and execute. (Since it is within the analyzer
 33 parameter set, the module must be of type *EDAnalyzer*; i.e. the *base type* of
 34 *<module-name>* must be *EDAnalyzer*.)

35 Module labels are fully described in Section 24.5.

36 In this example *art* will look for a module named *HelloWorld*, which it will
 37 find as part of the toyExperiment UPS product. Section 9.9 describes how *art*
 38 uses *<module-name>* to find the shareable library that contains code for the

1 HelloWorld module. A parameter set that is used to configure a module may
 2 contain additional lines; if present, the meaning of those lines is understood by
 3 the module itself; those lines have no meaning either to *art* or to FHiCL.

4 Now look at the FHiCL fragment in Listing 9.5. We will use it to reinforce some
 5 of the ideas discussed in the previous paragraph.

6 *art* allows you to write a FHiCL file that uses a given module more than once.
 7 For example you may want to run an analysis twice, once with a loose mass
 8 cut on some intermediate state and once with a tight mass cut on the same
 9 intermediate state. In *art* you can do this by writing one module and mak-
 10 ing the cuts “run-time configurable.” This idea will be developed further in
 11 Chapter 13.

Listing 9.5: A FHiCL fragment illustrating module labels

```

11 analyzers : {
12   loose : {
13     module_type : MyAnalysis
14     mass_cut    : 20.
15   }
16   tight : {
17     module_type : MyAnalysis
18     mass_cut    : 15.
19   }
20 }
```

22 When *art* processes this fragment it will look for a module named `MyAnalysis`
 23 and instantiate it twice, once using the parameter set labeled (i.e. with mod-
 24 ule label) `tight` and once using the parameter set labeled `loose`. The two
 25 instances of the module `MyAnalysis` are distinguished by the module labels
 26 `tight` and `loose`.

27 *art* requires that module labels be unique within a FHiCL file. Module label
 28 may contain only upper- and lower-case letters and the numerals 0 to 9.



29 In the FHiCL files in this exercise, all of the modules are analyzer modules. Since
 30 analyzers do not make data products, these module labels are nothing more
 31 than identifiers inside the FHiCL file. For producer modules, however, which
 32 *do* make data products, the module label becomes part of the data product
 33 identifier and as such has a real significance. All module labels must conform to
 34 the same naming rules.

35 Within *art* there is no notion of reserved names or special names for module
 36 labels; however your experiment will almost certainly have established some
 37 naming conventions.

38 9.7.8 Paths

39 In the physics parameter set for `hello.fcl` there are two parameters that
 40 represent paths (discussed in Section 4.6 :

```

1   e1           : [ hi ]
2   end_paths    : [ e1 ]

```

3 The path defined by the parameter `e1` takes a value that is a FHiCL sequence
 4 of module labels. The name of a path is an arbitrary identifier that must be
 5 unique within a FHiCL file; it has no persistent significance and can be any legal
 6 FHiCL name.



7 Sometimes this documentation uses the word *path* in the sense of an *art path*(γ)
 8 (a sequence of module labels), other times *path* is used as a path in a file system
 9 and in yet other situations, it is used as a colon-delimited set of directory names.
 10 The use should be clear from the context.

11 The name `end_paths`, in contrast to `e1`, is a keyword in *art*. Its value must be
 12 a FHiCL sequence of paths – here it is a sequence of one path, `e1`. *reference the*
 13 *rules when available* When *art* processes the `end_paths` definition it combines
 14 all of the path definitions and forms the set of unique module labels from all
 15 paths defined in the parameter set. In other words, it is legal in *art* for a
 16 module label to appear in more than one path; if it does, *art* will recognize this
 17 and will ensure that the module is executed only once per event.

18 If you put the name of a module label into the definition of `end_paths`, *art*
 19 will issue an error and stop processing.



20 The paths listed in `end_paths` may only contain module labels for analyzer
 21 and/or output modules; they may *not* contain module labels for producer or fil-
 22 ter modules. The reason for this restriction will be discussed in Section .

23 What about the order of module labels in a path? Since analyzer and output
 24 modules may neither add new information to the event nor communicate with
 25 each other except via the event, the processing order is not important for the
 26 event. By definition, then, *art* may run analyzer and output modules in any
 27 order. In a simple *art* job with a single path, *art* will, in fact, run the modules
 28 in the order of appearance in the path, but do not write code that depends on
 29 execution order because *art* is free to change it.



30 It may seem that `end_paths` could more simply have been defined as a set of
 31 module labels, eliminating the layer of the path altogether, but there is a reason.
 32 We will defer this discussion to Section .

33 If the `end_paths` parameter is absent or defined as:

```

34   end_paths    : [ ]

```

35 *art* will understand that this job has no analyzer modules and no filter modules
 36 to execute. It is legal to define a path as an empty FHiCL sequence.

37 As is standard in FHiCL, if the definition of `end_paths` appears more than
 38 once, the last definition takes precedence.

1 9.7.9 Writing an Output File

2 The file `writeFile.fcl` gives an example of writing an output file. This file
3 introduces the parameter set named `outputs`:

```
4   outputs : {
5       output1 : {
6           module_type : RootOutput
7           fileName    : "output/writeFile_data.root"
8       }
9   }
```

10 When it appears at the outermost scope of a FHiCL file, the identifier `outputs`
11 is a keyword reserved to *art*. In this case the value of `outputs` must be a param-
12 eter set (e.g., `output1`) of parameter sets (e.g., `module_type` and `fileName`);
13 each of the inner parameter sets provides the configuration of one output mod-
14 ule.

15 An *art* job may have zero or more output modules.

16 The name `RootOutput` is the name of a standard *art* output module; it writes
17 the events in memory to a disk file in an *art*-defined, ROOT-based format. Files
18 written by the module `RootOutput` can be read by the module `RootInput`.
19 The identifier `output1` is just another module label that obeys the same rules
20 discussed in Section 9.7.7. The identifier `fileName` is a keyword known to the
21 `RootOutput` module; its value is the name of the output file that this instance
22 of `RootOutput` will write.

23 There are many more optional parameters that can be used to configure an
24 output module. For example, an output module can be configured to write
25 out only selected events and/or to write out only a subset of the available data
26 products. Optional parameters are described in Chapter .

27 Notice in `writeFile.fcl` that the path `e1` has been extended to include the
28 module label of the output module:

```
29   e1 : [ hi, output1 ]
```

30 Finally, the source parameter set of `writeFile.fcl` is configured to read only
31 events 4, 5, 6, and 7.

32 To run `writeFile.fcl` and check that it worked correctly:

```
33 $ art -c writeFile.fcl
34 $ ls -s output/writeFile_data.root
35 $ art -c hello.fcl -s output/writeFile_data.root
```

36 The first command will write the output file; the second will check the size of
37 the output file and the last one will read back the output file and print the event
38 IDs for all of the events in the file. You should see the `HelloWorld` printout
39 for events 4, 5, 6 and 7.

9.8 Understanding the Process for Exercise 1

Section 9.4.2 contained a list of steps needed to run this exercise; this section will describe each of those steps in detail. When you understand what is done in these steps, you will understand the run-time environment in which *art* runs. As a reminder, the steps are listed again here:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Chapter 5
3. `mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built`
In the above and elsewhere as indicated, substitute your kerberos principal for the string `<username>`.
4. `cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built`
5. `setup toyExperiment v0-00-14 -q$ART_WORKBOOK_QUAL:prof`
6. `cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .`
7. `source makeLinks.sh`
8. Run *art*:
`art -c hello.fcl >& output/hello.log`

Steps 1 and 4 should be self explanatory and will not be discussed further.



When reading this section, you do not need to run any of the commands given here; this is a commentary on commands that you have already run.

9.8.1 Follow the Site-Specific Setup Procedure (Details)

The site-specific startup procedure, described in Chapter 5, ensures that the UPS system is properly initialized and that the UPS database (containing all of the UPS products needed to run the Workbook exercises) is present in the `PRODUCTS` environment variable.

This procedure also defines two environment variables that are defined by your experiment to allow you to run the Workbook exercises on their computer(s):

ART_WORKBOOK_WORKING_BASE the top-level directory in which users create their working directory for the Workbook exercises

ART_WORKBOOK_OUTPUT_BASE the top-level directory in which users create their output directory for the Workbook exercises; this is used by the script `makeLinks.sh`

If these environment variables are not defined, ask a system admin on your experiment.

1 9.8.2 Make a Working Directory (Details)

2 On the Fermilab computers the home disk areas are quite small so most ex-
3 periments ask that their collaborators work in some other disk space. This is
4 common to sites in general, so we recommend working in a separate space as a
5 best practice. The Workbook is designed to require it.

6 This step given as:

```
7 $ mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built
```

8 creates a new directory to use as your working directory. It is defined relative
9 to an environment variable described in Section 9.8.1. It only needs to be done
10 the first time that you log in to work on Workbook exercises – once it's there,
11 it's there!

12 If you follow the rest of the naming scheme, you will guarantee that you have
13 no conflicts with other parts of the Workbook.

14 As discussed in Section 9.4.2.2, you may of course choose your own working
15 directory on any disk that has adequate disk space.

16 9.8.3 Setup the toyExperiment UPS Product (Details)

17 This step is the main event in the eight-step process.

```
18 $ setup toyExperiment v0_00_14 -q$ART_WORKBOOK_QUAL:prof
```

19 This command tells UPS to find a product named *toyExperiment*, with the
20 specified version and qualifiers, and to *setup* that product, as described in Sec-
21 tion 7.3.

22 The required qualifiers may change from one experiment to another and even
23 from one site to another within the same experiment. To deal with this, the site
24 specific setup procedure defines the environment variable `ART_WORKBOOK_QUAL`,
25 whose value is the qualifier string that is correct for that site.

26 The complete ups qualifier for *toyExperiment* has two components, separated by
27 a colon: the string defined by `ART_WORKBOOK_QUAL` plus a qualifier describing
28 the compiler optimization level with which the product was built, in this case
29 “prof”; see Section 3.6.7 for information about the optimization levels.

30 Each version of the *toyExperiment* product knows that it requires a particular
31 version and qualifier of the *art* product. In turn, *art* knows that it depends
32 on particular versions of ROOT, CLHEP, boost and so on. When this recur-
33 sive setup has completed, over 20 products will have been setup. All of these
34 products define environment variables and about two-thirds of them add new
35 elements to the environment variables `PATH` and `LD_LIBRARY_PATH`.

1 If you are interested, you can inspect your environment before and after doing
 2 this setup. To do this, log out and log in again. Before doing the setup, run the
 3 following commands:

```
4 $ printenv > env.before
5 $ printenv PATH | tr : \\n > path.before
6 $ printenv LD_LIBRARY_PATH | tr : \\n > ldpath.before
```

7 Then setup toyExperiment and capture the environment afterwards (*env.after*).
 8 Compare the before and after files: the after files will have many, many additions
 9 to the environment.

10 9.8.4 Copy Files to your Current Working Directory (De- 11 tails)

12 The step:

```
13 $ cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .
```

14 only needs to be done only the first time that you log in to work on the Work-
 15 book.

16 In this step you copied the files that you will use for the exercises into your
 17 current working directory. You should see these files:

```
18 hello.fcl makeLinks.sh skipEvents.fcl writeFile.fcl
```

19 9.8.5 Source makeLinks.sh (Details)

20 This step:

```
21 $ source makeLinks.sh
```

22 only needs to be done only the first time that you log in to work on the Work-
 23 book. It created some symbolic links that *art* will use.

24 The FHiCL files used in the Workbook exercises look for their input files in the
 25 subdirectory *inputFiles*. This script made a symbolic link, named *inputFiles*,
 26 that points to:

```
27 $TOYEXPERIMENT_DIR/inputFiles
```

28 in which the necessary input files are found.

29 This script also ensures that there is an output directory that you can write
 30 into when you run the exercises and adds a symbolic link from the current
 31 working directory to this output directory. The output directory is made under
 32 the directory *\$ART_WORKBOOK_OUTPUT_BASE*; this environment variable was
 33 set by the site-specific setup procedure and it points to disk space that will have
 34 enough room to hold the output of the exercises.

1 9.8.6 Run *art* (Details)

2 Issuing the command:

```
3 $ art -c hello.fcl
```

4 runs the *art* main program, which is found in \$ART_FQ_DIR/bin. This direc-
5 tory was added to your PATH when you setup toyExperiment. You can inspect
6 your PATH to see that this directory is indeed there.

7 9.9 How does *art* find Modules?

8 When you ran `hello.fcl`, how did *art* find the module HelloWorld?

9 It looked at the environment variable LD_LIBRARY_PATH, which is a colon-
10 delimited set of directory names defined when you setup the toyExperiments
11 product. We saw the value of LD_LIBRARY_PATH in Section 9.8.3; to see it
12 again, type the following:

```
13 $ printenv LD_LIBRARY_PATH | tr : \\n
```

14 (The fragment `| tr : \\n` tells the bash shell to take the output of `print-`
15 `env` and replace every occurrence of the colon character with the newline charac-

16 `ter`; this makes the output much easier to read.) The output should look similar
17 to that shown in Listing 9.6.

Listing 9.6: Example of the value of LD_LIBRARY_PATH

```
1b /ds50/app/products/tbb/v4_1_2/Linux64bit+2.6-2.12-e2-prof/lib
2b /ds50/app/products/sqlite/v3_07_16_00/Linux64bit+2.6-2.12-e2-prof/lib
3b /ds50/app/products/libsigcpp/v2_2_10/Linux64bit+2.6-2.12-e2-prof/lib
4b /ds50/app/products/cppunit/v1_12_1/Linux64bit+2.6-2.12-e2-prof/lib
5b /ds50/app/products/clhep/v2_1_3_1/Linux64bit+2.6-2.12-e2-prof/lib
6b /ds50/app/products/python/v2_7_3/Linux64bit+2.6-2.12-gcc47/lib
7b /ds50/app/products/libxml2/v2_8_0/Linux64bit+2.6-2.12-gcc47-prof/lib
8b /ds50/app/products/fftw/v3_3_2/Linux64bit+2.6-2.12-gcc47-prof/lib
9b /ds50/app/products/root/v5_34_05/Linux64bit+2.6-2.12-e2-prof/lib
10b /ds50/app/products/boost/v1_53_0/Linux64bit+2.6-2.12-e2-prof/lib
11b /ds50/app/products/cpp0x/v1_03_15/slf6.x86_64.e2.prof/lib
12b /ds50/app/products/cetlib/v1_03_15/slf6.x86_64.e2.prof/lib2
13b /ds50/app/products/fhiclcpp/v2_17_02/slf6.x86_64.e2.prof/lib
14b /ds50/app/products/messagefacility/v1_10_16/slf6.x86_64.e2.prof/lib
15b /ds50/app/products/art/v1_06_00/slf6.x86_64.e2.prof/lib
16b /ds50/app/products/toyExperiment/v0_00_14/slf6.x86_64.e2.prof/lib
17b /grid/fermiapp/products/common/prd/git/v1_8_0_1/Linux64bit-2/lib
```

35 Of course the leading element of each directory name, `/ds50/app` will be
36 replaced by whatever is correct for your experiment. The last element in
37 LD_LIBRARY_PATH is not relevant for running *art* and it may or may not
38 be present on your machine, depending on details of what is done inside your
39 site-specific setup procedure.

1 If you compare the names of the directories listed in `LD_LIBRARY_PATH` to the
 2 names of the directories listed in the `PRODUCTS` environment variable, you will
 3 see that all of these directories are part of the UPS products system. Moreover,
 4 for each product, the version, flavor and qualifiers are embedded in the directory
 5 name. In particular, both *art* and *toyExperiment* are found in the list.

6 If you `ls` the directories in `LD_LIBRARY_PATH` you will find that each directory
 7 contains many shareable object libraries (`.so` files).

8 When *art* looks for a module named `HelloWorld`, it looks through the directe-
 9 ries defined in `LD_LIBRARY_PATH` and looks for a file whose name matches
 10 the pattern,

```
11 lib*HelloWorld_module.so
```

12 where the asterisk matches (zero or) any combination of characters. *art* finds
 13 that, in all of the directories, there is exactly one file that matches the pattern,
 14 and it is found in the directory:

```
15 /ds50/app/products/toyExperiment/v0_00_14/slf6.x86_64.e2.prof/lib/
```

16 The name of the file is:

```
17 libtoyExperiment_Analyzers_HelloWorld_module.so
```

18 If *art* had found no files that matched the pattern, it would have printed an
 19 error message and tried to shutdown as gracefully as possible. If *art* had found
 20 more than one file that matched the pattern, it would have printed a different
 21 error message and tried to shut down as gracefully as possible.



22 One of the important features of *art* is that, whenever it detects an error condi-
 23 tion that is serious enough to stop execution, it always attempts to shut down as
 24 gracefully as possible. Among other things this means that it tries to properly
 25 close all output files. This feature is not so important when an error occurs
 26 at the start of a job but it ensures that, when an error occurs after hours of
 27 execution, your results up to the error are correct and available.

28 9.10 The *art* Run-time Environment

29 This discussion is aimed to help you understand the process described in this
 30 chapter as a whole and how the pieces fit together in the *art* run-time environ-
 31 ment. This environment is summarized in Figure 9.1. In this figure the boxes
 32 refer either to locations in memory or to files on a disk.

33 At the center of the figure is a box labelled “*art* executable;” this represents
 34 the *art* main program resident in memory after being loaded. When the *art*
 35 executable starts up, it reads its run-time configuration (FHiCL) file, repre-
 36 sented by the box to its left. Following instructions from the configuration
 37 file, *art* will load shared libraries from *toyExperiment*, from *art*, from `ROOT`,

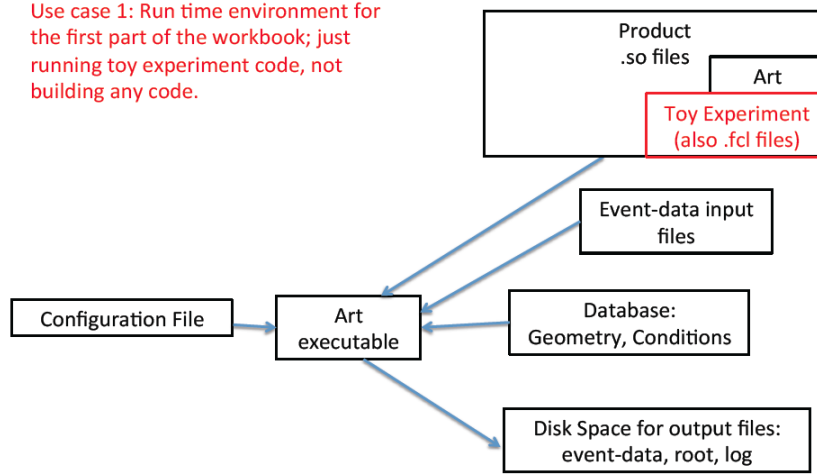


Figure 9.1: Elements of the *art* run-time environment for the first Workbook exercise

1 from CLHEP and from other UPS products. All of these shared librares (.so
 2 files) will be found in the appropriate UPS products in `LD_LIBRARY_PATH`,
 3 which points to directories in the UPS products area (box at upper right). Also
 4 following instructions from the FHiCL file, *art* will look for input files (box la-
 5 beled “Event-data input files” at right). The FHiCL file will tell *art* to write
 6 its event-data and histogram output files to a particular directory (box at lower
 7 right).

8 One remaining box in the figure (at right, second from bottom) is not encoun-
 9 tered in the first Workbook exercise but has been provided for completeness. In
 10 most *art* jobs it is necessary to access experiment-related geometry and condi-
 11 tions information; in a mature experiment, these are usually stored in a database
 12 that stands apart from the other elements in the picture.

13 The arrows in Figure 9.1 show the direction in which information flows. Every-
 14 thing but the output flows into the *art* executable.

15 9.11 Finding FHiCL files: `FHICL_FILE_PATH`

16 This section will describe where *art* looks for FHiCL files. There are two cases:
 17 looking for the file specified by the command line argument `-c` and looking
 18 for files that have been included by a `#include` directive within a FHiCL
 19 file.

1 9.11.1 The `-c` command line argument

2 When you issued the command

```
3 $ art -c hello.fcl
```

4 *art* looked for a file named `hello.fcl` in the current working directory and
 5 found it. You may specify any absolute or relative path as the argument of
 6 the `-c` option. If *art* had not found `hello.fcl` in this directory it would
 7 have looked for it relative to the path defined by the environment variable
 8 `FHICL_FILE_PATH`. This is just another path-type environment variable, like
 9 `PATH` or `LD_LIBRARY_PATH`. You can inspect the value of `FHICL_FILE_PATH`
 10 by:

```
11 $ printenv FHICL_FILE_PATH
12  .:$TOYEXPERIMENT_DIR
```

13 Actually the output will show the translated value of the environment variable
 14 `TOYEXPERIMENT_DIR`. The presence of the current working directory (`.`) in
 15 the path is redundant when processing the command line argument but it is
 16 significant in the case discussed in the next section.



17 Some experiments have chosen to configure their version of the *art* main pro-
 18 gram so that it will not look for the command line argument `FHiCL` file in
 19 `FHICL_FILE_PATH`. It is also possible to configure *art* so that only relative
 20 paths, not absolute paths, are legal values of the `-c` argument. This last op-
 21 tion can be used to help ensure that only version-controlled files are used when
 22 running production jobs. Experiments may enable or disable either of these
 23 options when their main program is built.

24 9.11.2 `#include` Files

25 Section 9.7 discussed Listing 9.4, which contains the fragments of `hello.fcl`
 26 that are related to configuring the message service. The first line in that listing
 27 is an include directive. *art* will look for the file named by the include directive
 28 relative to `FHICL_FILE_PATH` and it will find it in:

```
29 $TOYEXPERIMENT_DIR/fcl/minimalMessageService.fcl
```

30 This is part of the `toyExperiment` UPS product.

31 The version of *art* used in the Workbook does not consider the argument of the
 32 include directive as an absolute path or as a path relative to the current working
 33 directory; it only looks for files relative to `FHICL_FILE_PATH`. This is in contrast
 34 to the choice made when processing the `-c` command line option.



35 When building *art*, one may configure *art* to first consider the argument of
 36 the include directive as a path and to consider `FHICL_FILE_PATH` only if that
 37 fails.

- 1 *Add a section called Review that looks at trigger paths, end paths, etc and works*
- 2 *backwards*

10 Exercise 2: Build and Run Your First Module

10.1 Introduction

In this exercise you will build and run a simple *art* module. Section 3.6.7 introduced the idea of a build system, a software package that compiles and links your source code to turn it into machine code that the computer can execute. In this chapter you will be introduced to the *art* development environment, which adds the following to the run-time environment (discussed in Section 9.10):

1. a build system
2. a source code repository
3. a working copy of the Workbook source code
4. a directory containing shared libraries created by the build system

In this and all subsequent Workbook exercises, you will use the build system used by the *art* development team, **cetbuildtools**. This system will require you to open two shell windows your local machine and, in each one, to log into the remote machine ¹. The windows will be referred to as the *source window* and the *build window*:

- In the *source window* you will check out and edit source code.
- In the *build window* you will build and run code.

Exercise 2 and all subsequent Workbook exercises will use the setup instructions found in this chapter.

Most readers: Follow the setup steps in Section 10.4.1, and skip Section 10.5.

If you are an advanced user and wish to manage your working directory yourself, skip Section 10.4.1, and follow the steps in Section 10.5, then go back to Section 10.4.2 and 10.4.4 to examine the directories' contents.

¹**cetbuildtools** requires what are called *out-of-source builds*; this means that the source code and the working space for the build system must be in separate directories.



10.2 Prerequisites

Before running this exercise, you need to be familiar with the material in Part I (Introduction) of this documentation set and Chapter 9 from Part II (Workbook).

- namespace
- `#include` directives
- header file
- class
- base class
- derived class
- constructor
- destructor
- what does the compiler do if you do not provide a destructor?
- the C preprocessor
- member function (aka method)
- const vs non-const member function
- argument list of a function
- signature of a function
- virtual function
- pure virtual function
- virtual class
- pure virtual class
- concrete class
- *declaration* vs *definition* of a class
- arguments passed by reference
- arguments passed by const reference
- notion of *type*: e.g., a class, a struct, a free function or a typedef
- how to write a C++ main program

In this chapter you will also encounter the C++ idea of *inheritance*. Understanding inheritance is not a prerequisite; it will be described as you encounter it in the Workbook exercises.

10.3 What You Will Learn

In this exercise you will learn:

- how to establish the *art* development environment
- how to checkout the Workbook exercises from the git source code management system
- how to use the **cetbuildtools** build system to build the code for the Workbook exercises
- how include files are found
- what a *link list* is
- where the build system finds the link list
- what the `art::Event` is and how to access it
- what the `art::EventID` is and how to access it
- what makes a class an *art module*
- where the build system puts the `.so` files that it makes

10.4 Setting up to Run Exercises: Standard Procedure

10.4.1 “Source Window” Setup

In your source window do the following:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Table 5.1
3. `$ mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook`
In the above and elsewhere as indicated, substitute your kerberos principal for the string `<username>`.
4. `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook`
5. Set up the source code management system git; check the output for each step in Section 10.4.2.1:
 - (a) `$ setup git`
 - (b) `$ git clone http://cdcvns.fnal.gov/projects/art-workbook`

```

1      (c) $ cd art-workbook
2          This will be referred to as your source directory.
3      (d) $ git checkout -b v0_00_15 v0_00_15
4      6. $ source ups/setup_deps -p $ART_WORKBOOK_QUAL

```

Up through step 4, the results should look similar to those of Exercise 1. Note that the directory name chosen here is different than that chosen in the first exercise; this is to avoid file name collisions.

10.4.2 Examine Source Window Setup

10.4.2.1 About git and What it Did

git is a source code management system² that is used to hold the source code for the Workbook exercises. A source code management system is a tool that helps to look after the bookkeeping of the development of a code base; among many other things it keeps a complete history of all changes and allows one to get a copy of the source code as it existed at some time in the past. Because of git's many advanced features, many HEP experiments are moving to git. git is fully described in the git manual.

Some experiments set up git in their site-specific setup procedure; others do not. In running setup git, you have ensured that a working copy of git is in your PATH³.

The git clone and git checkout commands produce a working copy of the Workbook source files in your source directory; git clone should produce the following output:

```
Cloning into 'art-workbook'...
```

Executing the git checkout command should produce the following output:

```
Switched to a new branch 'v0_00_15'
```

If you do not see the expected output, contact the *art* team as described in Section 3.4. If you wish to learn about git branches, consult a git manual.

The final step sources a script that defines a lot of environment variables (the same set that will be defined in the build window).

²Other source code management systems with which you may be familiar are cvs and svn.

³No version needs to be supplied because the git UPS product has a current version declared; see Section 7.4.

1 10.4.2.2 Contents of the Source Directory

2 At the end of the setup procedure, see what your source directory contains:

```
3 $ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook
4 $ ls
5 admin art-workbook CMakeLists.txt ups
```

6 (Yes, it contains a subdirectory of the same name as its parent, `art-workbook`.)

- 7 • The `admin` directory contains some scripts used by **cetbuildtools** to
- 8 customize the configuration of the development environment.
- 9 • The `art-workbook` directory contains the main body of the source code.
- 10 • The file `CMakeLists.txt` is the file that the build system reads to learn
- 11 what steps it should do.
- 12 • The `ups` directory contains information about what UPS products this
- 13 product depends on; it contains additional information used to configure
- 14 the development environment.

15 Look inside the `art-workbook` (“junior”) directory (via `ls`) and see that it

16 contains several files and subdirectories. The file `CMakeLists.txt` contains

17 more instructions for the build system. Actually every directory contains a

18 `CMakeLists.txt`; each contains additional instructions for the build system.

19 The subdirectory `FirstModule` contains the files that will be used in this ex-

20 ercise; the remaining subdirectories contain files that will be used in subsequent

21 Workbook exercises.

22 If you look inside the `FirstModule` directory, you will see

```
23 CMakeLists.txt      FirstAnswer01_module.cc  First_module.cc
24 firstAnswer01.fcl   first.fcl
```

25 The file `CMakeLists.txt` in here contains yet more instructions for the build

26 system and will be discussed later. The file `First_module.cc` is the first

27 module that you will look at and `first.fcl` is the FHiCL file that runs it. This

28 exercise will suggest that you try to write some code on your own; the answer is

29 provided in `FirstAnswer01_module.cc` and the file `firstAnswer01.fcl`

30 runs it. These files will be discussed at length during these exercises.

31 10.4.3 “Build Window” Setup

32 Now go to your build window and do the following:

- 33 1. Log in to the computer you chose in Section 8.3.
- 34 2. Follow the site-specific setup procedure; see Chapter 5
- 35 3. `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook`

```

1    4. $ mkdir build-prof
2
3    5. $ cd build-prof
4      This new directory will be your build directory.
5
6    6. $ source ../art-workbook/ups/setup_for_development \
7      -p $ART_WORKBOOK_QUAL
8      The output from this command will tell you to take some additional steps;
9      do not do those steps.
10
11   7. buildtool
12     This step may take a few minutes.

```

10.4.4 Examine Build Window Setup

Logging in and sourcing the site-specific setup script should be clear by now. Notice that next you are told to `cd` to the same *workbook* directory as in Step 4 of the instructions for the source window. From there, you make a directory in which you will run builds (your *build* directory), and `cd` to it. (The name `build-prof` can be any legal directory name but it is suggested here because this example performs a profile build; see Section 3.6.7)

Step 6 sources a script called `setup_for_development` found in the `ups` sub-directory of the source directory. This script, run exactly as indicated, defines `build-prof` to be your build directory. This command selects a profile build (via the option `-p`); it also requests that the `ups` qualifiers defined in the environment variable `ART_WORKBOOK_QUAL` be used when requesting the `ups` products on which it depends; this environment variable was discussed in Section 9.8.3. The expected output is shown in Listing 10.1.



Check that there are no error messages in the indicated block. The listing concludes with a request for you to run a `cmake` command; do NOT run `cmake` (this line is an artifact of layering **cetbuildtools** on top of **cmake**).

Listing 10.1: Example of output created by `setup_for_development`

```

21 The working build directory is /ds50/app/user/kutschke/workbook/build-prof
22 The source code directory is /ds50/app/user/kutschke/workbook/art-workbook
23 ----- check this block for errors -----
24 -----
25 /ds50/app/user/kutschke/workbook/build-prof/lib has been added to LD_LIBRARY_PATH
26 /ds50/app/user/kutschke/workbook/build-prof/bin has been added to PATH
27
28 CETPKG_SOURCE=/ds50/app/user/kutschke/workbook/art-workbook
29 CETPKG_BUILD=/ds50/app/user/kutschke/workbook/build-prof
30 CETPKG_NAME=art_workbook
31 CETPKG_VERSION=v0_00_15
32 CETPKG_QUAL=e2:prof
33 CETPKG_TYPE=Prof
34
35 Please use this cmake command:

```

Part

```
17 cmake -DCMAKE_INSTALL_PREFIX=/install/path -DCMAKE_BUILD_TYPE=$CETPKG_TYPE $CETPKG_SOURCE
```

2 This script sets up all of the UPS products on which the Workbook depends; this
 3 is analogous to the actions taken by Step 5 in the first exercise (Section 9.4.2.1)
 4 when you were working in the *art* run-time environment. This script also creates
 5 several files and directories in your build-prof directory; these comprise the
 6 working space used by **cetbuildtools**.

7 After sourcing this script, the contents of build-prof will be

```
8 art_workbook-v0_00_15 bin cetpkg_variable_report diag_report lib
```

9 At this time the two subdirectories bin and lib will be empty. The other files
 10 are used by the build system to keep track of its configuration.

11 Step 7 (buildtool) tells **cetbuildtools** to build everything found in the source
 12 directory; this includes all of the Workbook exercises, not just the first one. The
 13 build process will take two or three minutes on an unloaded (not undergoing
 14 heavy usage) machine. Its output should end with the lines:

```
15 -----
16 INFO: Stage build successful.
17 -----
```

18 After the build has completed do an *ls* on the directory lib; you will see that
 19 it contains a large number of shared library (.so) files; for v0_00_15 there
 20 will be 29 .so files; these are the files that *art* will load as you work through
 21 the exercises.

22 Also do an *ls* on the directory bin; these are scripts that are used by **cetbuild-**
 23 **tools** to maintain its environment; if the Workbook contained instructions to
 24 build any executable programs, they would have been written to this direc-
 25 tory.

26 After running buildtool, the build directory will contain:

```
27 admin                CMakeFiles                fcl
28 art-workbook          cmake_install.cmake        inputFiles
29 art_workbook-v0_00_15 CPackConfig.cmake         lib
30 bin                  CPackSourceConfig.cmake    Makefile
31 cetpkg_variable_report CTestTestfile.cmake        output
32 CMakeCache.txt        diag_report                ups
```

33 Most of these files are standard files that are explained in the **cetbuildtools**
 34 documentation. . However, three of these items need special attention here
 35 because they are customized for the Workbook.

36 An *ls -l* on the files fcl, inputFiles and output will reveal that they
 37 are symbolic links to

```
38 inputFiles -> ${TOYEXPERIMENT_DIR}/inputFiles
39 output -> ${ART_WORKBOOK_OUTPUT_BASE}/<username>/art_workbook_output
```

1 fcl -> <your source directory>/art-workbook

2 These links are present so that the FHiCL files for the Workbook exercises can
3 be machine-independent.

- 4 • The link `inputFiles` points to the directory `inputFiles` present in the
5 toyExperiment UPS product; this directory contains the input files that
6 *art* will read when you run the first exercise. These are the same files used
7 in the first exercise; if you need a reminder of the contents of these files,
8 see Table 9.1. These input files will also be used in many of the subsequent
9 exercises.
- 10 • The link `outputFiles` points to a directory that was created to hold your
11 output files; the environment variable `ART_WORKBOOK_OUTPUT_BASE` was
12 defined by your site-specific setup procedure.
- 13 • The link `fcl` points into your source directory hierarchy; it allows you to
14 access the FHiCL files that are found in that hierarchy with the convenience
15 of tab completions.

16 10.5 Setting up to Run Exercises: Self-managed 17 Working Directory

18 If you have worked through Section 10.4, skip this section and proceed to Section 10.6.
19

20 The explanation for the steps in these procedures that are the same as for the
21 “standard” procedures are found in Section 10.4.2 (for the source window) and
22 Section 10.4.4 (for the build window).

23 In your source window do the following:

- 24 1. Log in to the computer you chose in Section 8.3.
- 25 2. Follow the site-specific setup procedure; see Chapter 5
- 26 3. Make a working directory to hold the checked out source
- 27 4. `cd` to the directory made in the previous step
- 28 5. Ensure that `git` is in your `PATH`
- 29 6. `$ git clone http://cdcvcs.fnal.gov/projects/art-workbook`
- 30 7. `$ cd art-workbook`
- 31 In the following, this will be referred to as your source directory.
- 32 8. `$ git checkout -b v0.00.15 v0.00.15`

33 Now go to your build window and do the following:

Part

- 1 1. Log in to the computer you chose in Section 8.3.
- 2 2. Follow the site-specific setup procedure; see Chapter 5
- 3 3. Make a directory to hold the code that you will build; this is your build
- 4 directory in your build window. In the following, this will be referred to as
- 5 your build directory.
- 6 4. `cd` to your build directory
- 7 5. Make a directory, *outside of the hierarchy rooted at your build directory*,
- 8 to hold output files created by the workbook exercises.
- 9 6. `$ ln -s <directory-made-in-previous-step> output`
- 10 7. `$ source <your-source-directory>/ups/setup_for_development`
- 11 `-p $ART_WORKBOOK_QUAL`
- 12 The output from this command will tell you to take some additional steps;
- 13 do not do those steps.
- 14 8. `$ buildtool`

15 10.6 Logging In Again

16 If you log out and later wish to log in again to work on this or any other exercise,
17 you need to do the following:

18 In your source window:

- 19 1. Log in to the computer you chose in Section 8.3.
- 20 2. Follow the site-specific setup procedure; see Table 5.1
- 21 3. `cd` to your source directory
- 22 `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook`
- 23 4. `source ups/setup_deps -p`

24 In your build window:

- 25 1. Log in to the computer you chose in Section 8.3.
- 26 2. Follow the site-specific setup procedure; see Chapter 5
- 27 3. `cd` to your build directory
- 28 `$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/build-prof`
- 29 4. `$ source ../art-workbook/ups/setup_for_development -p $ART_WORKBOOK_QUAL`

30 If you chose to manage your own directory names (ie you followed Section 10.5),
31 then the names of your source and build directories will be different than those
32 shown.

- 1 Compare these steps with those given in Sections 10.4.1 and Section 10.4.3. You
- 2 will see that five steps are missing from the source window instructions and
- 3 three steps are missing from the build window instructions. The missing steps
- 4 only needed to be executed the first time.

5 10.7 The *art* Development Environment

- 6 In the preceeding sections of this chapter you established what is known as the
- 7 *art development environment*; this is a superset of the *art* run-time environment,
- 8 which was described in Section 9.10. This section summarizes the new elements
- 9 that are part of the development environment but not part of the run-time
- 10 environment.

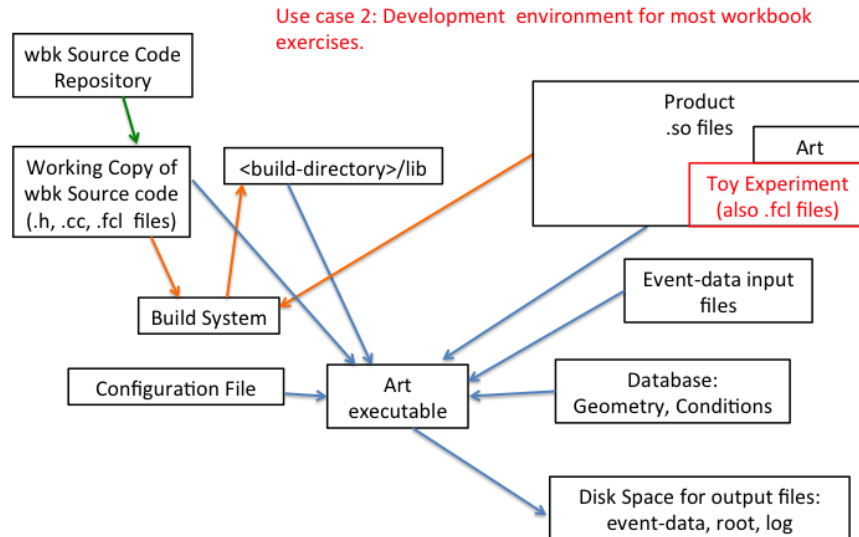


Figure 10.1: Elements of the *art* development environment as used in most of the Workbook exercises; the arrows denote information flow, as described in the text.

- 11 In Section 10.4.1, step 5b (`git clone ...`) was to contact the source code
- 12 repository and make a clone of the repository in your disk space; step 5d `git`
- 13 `checkout ...`) was to check out the correct version of the code from the
- 14 clone and to put it into your source directory. The repository is hosted on a
- 15 central Fermilab server and is accessed via the network. The upper left box in
- 16 Figure 10.1 denotes the repository and the box below it denotes your working
- 17 copy of the Workbook code. The flow of information during the clone and
- 18 checkout processes is indicated by the green arrow in the figure.

1 In step 7 of Section 10.4.3, you ran `buildtool`, which read the source code
 2 files from your working copy of the Workbook code and turned them into shared
 3 libraries. The script `buildtool` is part of the build system, which is denoted
 4 as the box in the center left section of the figure. When you ran `buildtool`,
 5 it wrote shared library files to the `lib` subdirectory of your build directory;
 6 this directory is denoted in the figure as the box in the top center labeled
 7 `<build-directory>/lib`. The orange arrows in the figure denote the in-
 8 formation flow at build-time. In order to perform this task, `buildtool` also
 9 needed to read header files and shared libraries found in the UPS products area,
 10 hence the orange arrow leading from the UPS Products box to the build system
 11 box.

12 In the figure, information flow at run-time is denoted by the blue lines. When
 13 you ran the `art` executable, it looked for shared libraries in the directories defined
 14 by `LD_LIBRARY_PATH`. In the `art` development environment, `LD_LIBRARY_PATH`
 15 contains

- 16 1. the `lib` subdirectory of your build directory.
- 17 2. all of the directories previously described in Section 9.9

18 In all environments, the `art` executable looks for FHiCL files in

- 19 1. in the file specified in the `-c` command line argument
- 20 2. in the directories specified in `FHICL_FILE_PATH`

21 The first of these is denoted in the figure by the box labeled “Configuration
 22 File.” In the `art` development environment, `FHICL_FILE_PATH` contains

- 23 1. some directories found in your checked out copy of the source
- 24 2. all of the directories previously described in Section 9.11

25 The remaining elements in Figure 10.1 are the same as described for Fig-
 26 ure 9.1.

27 10.8 Running the Exercise

28 10.8.1 Run *art* on `first.fcl`

29 In your build window, make sure that your current working directory is your
 30 build directory. From here, run the first part of this exercise by typing the
 31 following:

```
32 $ art -c fcl/FirstModule/first.fcl > output/first.log
```

33 (We suggest you get in the habit of routing your output to the output directory.)
 34 The output of this step will look much like that in Listing 9.1, but with two

1 significant differences. The first difference is that the output from `first.fcl`
 2 contains an additional line

3 `Hello from First::constructor.`

4 The second difference is that the words printed out for each event are a little
 5 different; the printout from `first.fcl` looks like

6 `Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1`
 7 while that from `hello.fcl` looked like

8 `Hello World! This event has the id: run: 1 subRun: 0 event: 1`

9 The reason for changing this printout is so that you can identify, from the
 10 printout, which module was run.

11 10.8.2 The FHiCL File `first.fcl`

12 Compare the FHiCL file used in this exercise, `fcl/FirstModule/first.fcl`,
 13 with `hello.fcl` from the first exercise (i.e., run `cat` or `diff` on them). Other
 14 than comments, the only difference is that the `module_type` has changed from
 15 `HelloWorld` to `First`:

```
16 $ diff $TOYEXPERIMENT_DIR/HelloWorldScripts/hello.fcl \
17     fcl/FirstModule/first.fcl
18 ...
19 <      module_type : HelloWorld
20 ---
21 >      module_type : First
```

22 The file `first.fcl` tells *art* to run a module named `First`. As described in
 23 Section 9.9, *art* looks through the directories defined in `LD_LIBRARY_PATH`
 24 and looks for a file whose name matches the pattern `lib*First_module.so`.
 25 This module happens to be found at this location, relative to your build direc-
 26 tory:

```
27 lib/libart-workbook_FirstModule_First_module.so
```

28 This shared library file was created when you ran `buildtool`.

29 10.8.3 The Source Code File `First_module.cc`

30 This section will describe the source code for the module `First` and will use it
 31 as a model to describe modules in general. The source code for this module is
 32 found in the following file, relative to your source directory (go to your source
 33 window!):

```
34 art-workbook/FirstModule/First_module.cc
```

1 For convenience, the contents of the file is also shown in Listing 10.2. When
 2 you ran `builddtool`, it compiled and linked this source file into the following
 3 shared library (relative to your your build directory):

4 `lib/libart-workbook_FirstModule_First_module.so`

5 This is the shared library that was loaded by *art* when you ran code for this
 6 exercise, in Section 10.8.2.

7 In broad strokes, the file `First_module.cc`:

- 8 • declares a class named `First`
- 9 • provides the implementation for the class
- 10 • contains a call to the C-Preprocessor macro named `DEFINE_ART_MODULE`,
 11 discussed in Section 10.8.3.7

12 All module files that you will see in the Workbook share these “broad strokes.”
 13 Some experiments that use *art* have chosen to split the source code for one
 14 module into three separate files; the *art* team does not recommend this practice,
 15 but it is in use and it will be discussed in Section 10.11.2.

Listing 10.2: The contents of `First_module.cc`

```

1b
12 #include ``art/Framework/Core/EDAnalyzer.h``
13 #include ``art/Framework/Core/ModuleMacros.h``
14 #include ``art/Framework/Principal/Event.h``
15
16 #include <iostream>
17
18 namespace tex {
19
20     class First : public art::EDAnalyzer {
21     public:
22
23         explicit First(fhicl::ParameterSet const& );
24
25         void analyze(art::Event const& event) override;
26
27     };
28 }
29
30 tex::First::First(fhicl::ParameterSet const& ){
31     std::cout << ``Hello from First::constructor.`` << std::endl;
32 }
33
34 void tex::First::analyze(art::Event const& event){
35     std::cout << ``Hello from First::analyze. Event id: ``
36         << event.id()
37         << std::endl;
38 }
39
40 DEFINE_ART_MODULE(tex::First)

```

1 Those of you with some C++ experience will have noticed that there is no
 2 file named `First_module.h` in the directory `art-workbook/FirstModule`.
 3 The explanation for this will be given in Section 10.11.1.

4 10.8.3.1 The `#include` Files

5 The first three non-blank lines in Listing 10.2 are three *include directives* that
 6 include header files. All three of these files are included from the *art* UPS
 7 product (where to find included header files is discussed in Section 7.6).



8 If you are a C++ beginner you will likely find these files difficult to understand;
 9 you do not need to understand them at this time but you do need to know
 10 where to find them for future reference.

11 The next non-blank line, `#include <iostream>`, includes the C++ header
 12 that enables this code to write output to the screen; for details, see any standard
 13 C++ documentation.

14 10.8.3.2 The Declaration of the Class `First`

15 The next portion of Listing 10.2 starts with the line “`namespace tex {`”
 16 which opens the namespace `tex` (the namespace is closed with a “`}`” about half
 17 way down the listing). If you are not familiar with namespaces, consult the
 18 standard C++ documentation.

19 All of the code in the *toyExperiment* UPS product was written in a namespace
 20 named `tex`; the name `tex` is an acronym-like shorthand for the *toyExperiment*
 21 (*ToyEXperiment*) UPS product. In order to keep things simple, all of the classes
 22 in the Workbook are also declared in the namespace `tex`. For more information
 23 about this choice, see Section 7.6.4.

24 The namespace contains the declaration of a class named `First`, which has
 25 only two members:

- 26 1. a constructor, described in Section 10.8.3.3
- 27 2. a member function, named `analyze`, described in Section 10.8.3.5

28 *art* will call the constructor once at the start of each job and it will call `analyze`
 29 once for each event.

30 The first line of the class `First`’s declaration is:

```
31 class First : public art::EDAnalyzer {
```

32 The fragment (`: public art::EDAnalyzer`) tells the C++ compiler that
 33 the class `First` is a (public⁴) *derived class* that *inherits* from a *base class*
 34 named `art::EDAnalyzer`. At this time it is not necessary to understand

⁴The members of this class can be accessed by member and nonmember functions.

1 C++ inheritance, base classes or derived classes; just follow the pattern when
 2 you write you own modules.

3 Section 3.6.3 discussed the idea of *module types*: analyzer, producer, filter and
 4 so on. If a class inherits from `art::EDAnalyzer` then the class is an analyzer
 5 module and it will have the properties of an analyzer module that were discussed
 6 in Section 3.6.3.

7 For a class to be a valid *art* analyzer module, it must follow a set of rules defined
 8 by *art*:

- 9 1. It must inherit from `art::EDAnalyzer`.
- 10 2. It must provide a constructor with the argument list:
 11 `fhicl::ParameterSet const&`
- 12 3. It must provide a member function named `analyze`, with the signature⁵:
 13 `analyze( art::Event const&)`
- 14 4. If the name of a module class is `<ClassName>` then the source code for
 15 the module must be in a file named `<ClassName>.module.cc` and this
 16 file must contain the lines:
 17 `#include ``art/Framework/Core/ModuleMacros.h```
 18 `DEFINE_ART_MODULE( <namespace>::<ClassName>`
- 19 5. It may optionally provide other member functions with signatures pre-
 20 scribed by *art*; if these member functions are present in a module class,
 21 then *art* will call them at the appropriate times. Some examples are pro-
 22 vided in Chapter 11.

23 You can see from Listing 10.2 that the class `First` follows all of these rules and
 24 that it does not contain any of the optional member functions.

25 A module may also contain any other member data and any other member
 26 functions that are needed to do its job.

27 The next line of the class declaration is:

28 `public:`

29 which tells the compiler that *art* is permitted to call the constructor `First` and
 30 the member function `analyze`⁶.

31 The next line of the class declaration declares a constructor with the argument
 32 list prescribed by *art*:

33 `First(fhicl::ParameterSet const& );`

⁵ In C++ the *signature* of a member function is the name of the class of which the function is a member, the name of the function, the number, types and order of the arguments, and whether the member function is marked as `const` or `volatile`. The signature does not include the return type.

⁶ Actually, in standard C++ this line says that any code may call these member functions; but one of the design rules of *art* stipulates that nothing besides *art* itself may call them.

1 The requirement that the class name match the filename (minus the `_module.cc`
 2 portion) is enforced by *art*'s system for dynamically loading shared libraries.
 3 The requirement that the class provide the prescribed constructor is enforced
 4 by the macro `DEFINE_ART_MODULE`, which will be described in Section 10.8.3.7.
 5 And the last line of the class declaration declares the member function, `analyze`
 6 with the argument list required by *art*:

```
7   analyze( art::Event const &) override;
```

8 The *override* contextual keyword is a feature that is new in C++ 11 so older
 9 references will not discuss it. It is a new safety feature that we recommend you
 10 use; we cannot give a proper explanation until we have had a chance to discuss
 11 inheritance further. For now, just consider it a rule that, in all analyzer modules,
 12 you should provide this keyword as part of the declaration of `analyze`.



13 For those who are knowledgeable about C++, the base class `art::EDAnalyzer`
 14 declares the member function `analyze` to be pure virtual; so it must be pro-
 15 vided by the derived class. The optional member functions of the base class
 16 are declared virtual but not pure virtual; do-nothing versions of these member
 17 functions are provided by the base class.

18 *In a future version of this documentation suite, more information will be avail-*
 19 *able in the Users Guide in Chapter .*

20 10.8.3.3 The Constructor for the Class First

21 In Listing 10.2, following the class declaration and the closing brace of the
 22 namespace, is the *definition* of the constructor:

```
23 tex::First::First(fhicl::ParameterSet const& ){  
24     std::cout << ``Hello from First::constructor.`` << std::endl;  
25 }
```

26 It has the argument required by *art* (`fhicl::ParameterSet const&`).
 27 This constructor simply prints some information (via `std::cout`) to let the
 28 user know that it has been called.

29 The fragment `tex::First::First` should be parsed as follows: the part
 30 `First::First` says that this definition is for a constructor of the class `First`.
 31 In principle there might be many classes named `First`, each in a different
 32 namespace; the leading `tex::` says that this is the constructor for the class
 33 named `First` that is found in the namespace `tex`.

34 The argument to the constructor is of type `fhicl::ParameterSet const&`;
 35 the class `ParameterSet`, found in the namespace `fhicl`, is a C++ represen-
 36 tation of a FHiCL parameter set (aka FHiCL *table*). This argument is not used
 37 in this exercise; you will see how it is used in Chapter 12.

1 You will also notice that the argument to the constructor is passed by `const`
2 reference, `const&`. This is a requirement specified by *art*; if you write a con-
3 structor that does not have exactly the correct argument type, then the com-
4 piler will issue a diagnostic and will stop compilation. Because the argument
5 is `const`, your code may not modify it; because it is passed by reference, it is
6 efficient to pass a large parameter set. If you are not familiar with `const`'ness
7 or with passing arguments by reference, consult the standard C++ documenta-
8 tion.

9 10.8.3.4 Aside: Unused Formal Parameters

10 You have probably noticed that neither the declaration of the constructor nor
11 the definition of the constructor provided a name for the argument of the con-
12 structor; both only provided the type. This section describes why the name was
13 omitted.

14 Each argument of a function (remember that a constructor is just a special kind
15 of function) has a type and a *formal parameter*; in casual use most of us refer
16 to the *formal parameter* as the name of the argument.

17 In a function definition, if a formal parameter is unused in the body of the func-
18 tion (i.e., between the braces `{}`) then the C++ standard says that the formal
19 parameter is optional; it is common to provide formal parameters in function
20 declarations as a form of documentation but the compiler always ignores these
21 formal parameters. Even when the formal parameter is omitted, the type is still
22 required because the full name of the function includes the number, type and
23 order of its arguments.

24 In the case of the Workbook, however, **cetbuildtools** has been configured to
25 go one step further. It enforces the following rule:

- 26 • If a function has a formal parameter that is not used by the definition of
27 the function, and if you *intend* that it not be used, then you must omit
28 that formal parameter when writing the argument list in the definition.

29 Consequently, if the compiler sees a formal parameter that is not used by the
30 definition of the function, it will presume that this is an error and it will issue
31 a diagnostic that stops compilation.

32 **cetbuildtools** is configured this way because an unused formal parameter is
33 frequently an indication of an error and the authors of the Workbook recommend
34 that we make full use of all safety features provided by the compiler. It is easy
35 enough to indicate to the compiler what your intention is; so we say “Just do
36 it!”

37 Your experiment's build system might or might not be configured to follow this
38 rule. It might permit unused formal parameters in function definitions or it

1 might consider this situation to warrant a warning level diagnostic, not an error
2 level diagnostic.

3 10.8.3.5 The Member Function `analyze` and `art::Event`

4 In Listing 10.2, following the definition of the constructor, you will find the
5 definition of the member function `analyze`:

```
6 void tex::First::analyze(art::Event const& event){
7     std::cout << ``Hello from First::analyze. Event id: ``
8                 << event.id()
9                 << std::endl;
10 }
```

11 The `override` contextual keyword that was present in the declaration of this
12 member function is not present in its definition; this is standard C++ us-
13 age.

14 This function has the argument list required by `art` (`art::Event const&`
15 `event`). If the type of the argument is not exactly correct, including the the
16 `const&`, the compiler will issue a diagnostic and stop compilation. The compiler
17 is able to do this because of one of the features of *inheritance*: it requires
18 that the member function named `analyze` have exactly the signature specified
19 by the base class (the details of how this works is beyond the scope of this
20 discussion).



21 Section 3.6.1 discussed the HEP idea of an event and the *art* idea of a three-
22 part event identifier. The class `art::Event` is the representation within *art*
23 of the HEP notion of an event. For the present discussion it is safe to consider
24 the following over-simplified view of an event: it contains an event identifier
25 plus a collection of data products (see Section 3.6.4). The name of the formal
26 parameter `event` has no meaning either to *art* or to the compiler – it is just
27 an identifier – but your code will be easier to read if you choose a meaningful
28 name.

29 At any given time in a running *art* program there is only ever one `art::Event`
30 object; in the rest of this paragraph we will call this object *the event*. It is owned
31 and managed by *art*, but *art* lets analyzer modules see the contents of the event;
32 it does so by passing the event by `const` reference when it calls the `analyze`
33 member function of analyzer modules. Because the event is passed by reference
34 (indicated by the `&`), the member function `analyze` does not get a copy of the
35 event; instead it is told where to find the event. This makes it efficient to pass
36 an event object even if the event contains a lot of information. Because the
37 argument is a `const` reference, if your code tries to change the contents of the
38 event, the compiler will issue a diagnostic and stop compilation.

39 As described in Section 3.6.3, analyzer modules may only inspect data in `event`,
40 not modify it. This section has shown how *art* institutes this policy as a hard

1 rule that will be enforced rigorously by the compiler:

- 2 1. The compiler will issue an error if an analyzer module does not contain
- 3 an member function named `analyze` with exactly the correct signature.
- 4 2. In the correct signature, the formal parameter `event` is a `const` refer-
- 5 ence.
- 6 3. Because `event` is `const`, the compiler will issue an error if the module
- 7 tries to call any member function of `art::Event` that will modify the
- 8 event.



9 You can find the header file for `art::Event` by following the guidelines de-
 10 scribed in Section 7.6.2. A future version of this documentation will con-
 11 tain a chapter in the Users Guide that provides a complete explanation of
 12 `art::Event`. Here, and in the rest of the Workbook, the features of `art::Event`
 13 will explained as needed.

14 The body of the function is almost trivial: it prints some information to let the
 15 user know that it has been called. In Section 10.8.1, when you ran *art* using
 16 `first.fcl`, the printout from the first event was

17 `Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1`

18 If you compare this to the source code you can see that the fragment

19 `<< event.id()`

20 creates the following printout

21 `run: 1 subRun: 0 event: 1`

22 This fragment tells the compiler to do the following:

- 23 1. In the class `art::Event`, find the member function named `id()` and
- 24 call this member function on the object `event`.
- 25 2. Whatever is returned by this function call, find its stream insertion oper-
- 26 ator and call it.

27 From this description you can probably guess that the member function
 28 `art::Event::id()` returns an object that represents the three part event
 29 identifier. In Section 10.8.3.6 you will learn that this guess is correct.

30 10.8.3.6 `art::EventID`

31 Before you work through this section, you may wish to review Section 7.6 which
 32 discusses how to find header files.

33 Section 3.6.1 discussed the idea of an event identifier, which has three compo-
 34 nents, a run number, a subRun number and event number. In this section you
 35 will learn where to find the class that *art* uses to represent an event identifier.

1 Rather than simply telling you the answer, this section will guide you through
 2 the process of discovering the answer for yourself.

3 In the previous section you looked at some code and the printout that it made;
 4 this strongly suggested that the member function `art::Event::id()` returns
 5 an object that represents the event identifier. To follow up on this suggestion,
 6 look at the header file for `art::Event`:

```
7 $ less $ART_INC/art/Framework/Principal/Event.h
```

8 Or use one of the code browsers discussed in 7.6.2. In this file you will find the
 9 definition of the member function `id()`:⁷

```
10     EventID  
11     id() const {return aux_.id();}
```

12 The important thing to look at here is the return type, `EventID`, which looks
 13 like a good candidate to be the class that holds the event identifier; you do not
 14 need to (or want to) know anything about the data member `aux_`. If you look
 15 near the beginning of `Event.h` you will see that it has the line:

```
16 #include "art/Persistency/Provenance/EventID.h"
```

17 which looks like a good candidate to be the header file for `EventID`. Look at
 18 this header file,

```
19 $ less $ART_INC/art/Persistency/Provenance/EventID.h
```

20 In this file you will discover that it is indeed the header file for `EventID`; you
 21 will also see that the class `EventID` is within the namespace `art`, making
 22 its full name `art::EventID`. Near the top of the file you will also see the
 23 comments:

```
24 // An EventID labels an unique readout of the data acquisition system,  
25 // which we call an ``event``.
```

26 This is another clue that `art::EventID` is the class we are looking for. Look
 27 again at `EventID.h`; you will see that it has accessor methods that permit you
 28 see the three components of the an event identifier:

```
29     RunNumber_t    run()    const;  
30     SubRunNumber_t subRun() const;  
31     EventNumber_t  event()  const;
```

32 Earlier in `EventID.h` the C++ *type*⁸ `EventNumber_t` was defined as:

```
33 namespace art {  
34     typedef std::uint32_t EventNumber_t;
```

⁷In C++, newlines are treated the same as any other white space; so this could have been written on a single line but the authors of `Event.h` have adopted a style in which return types are always written on their own line.

⁸In C++ the collective noun *type*, refers to both the built-in types, such as `int` and `float`, plus user defined types, which include classes, structs and typedefs.

1 }
 2

3 meaning that the event number is represented as a 32-bit unsigned integer. If you
 4 are not familiar with the C++ concept of *typedef*, or if you are not familiar with
 5 the definite-length integral types defined by the `<stdint.h>` header, consult
 6 any standard C++ documentation. If you dig deeper into the layers included
 7 in the `art::EventID` header, you will see that the run number and subRun
 8 number are also implemented as 32-bit unsigned integers.

9 At this point you can be sure that `art::EventID` is the class that *art* uses to
 10 represent the three part event identifier: the class has the right functionality.
 11 It's also true that the comments agree with this hypothesis but comments are
 12 often ill-maintained; be wary of comments and always read the code. This is a
 13 fairly typical tour through the layers of software.

14 The authors of *art* might have chosen an alternate definition of `EventNumber_t`

```
15 namespace art {  
16     typedef unsigned EventNumber_t;  
17 }
```

18 The difference is the use of `unsigned` rather than `std::uint32_t`. This
 19 alternate version was not chosen because it runs the risk that some computers
 20 might consider this type to have a length of 32 bits while other computers might
 21 consider it to have a length of 16 or 64 bits. In the definition that is used by *art*,
 22 an event number is guaranteed to be exactly 32 bits on all computers.

23 Why did the authors of *art* insert the extra level of indirection and not simply
 24 define the following member function inside `art::EventID`?

```
25     std::uint32_t event() const;
```

26 The answer is that it makes it easy to change the definition of the type should
 27 that be necessary. If, for example, an experiment requires that event numbers be
 28 of length 64 bits, only one change is needed, followed by a recompilation.

29 It is good practice to use typedefs for every concept for which the underlying
 30 data type is not absolutely certain.



31 It is a very common, but not universal, practice within the HEP C++ com-
 32 munity that typedefs that are used to give context-specific names to the C++
 33 built-in types (`int`, `float`, `char` etc) end in `_t`.

33 10.8.3.7 DEFINE_ART_MACRO: The Module Maker Macros

34 The final line in `First_module.cc` invokes a C preprocessor macro:

```
35 DEFINE_ART_MODULE(tex::First)
```

36 This macro is defined in the header file that was included by:

1 `#include ``art/Framework/Core/ModuleMacros.h```

2 If you are not familiar with the C preprocessor, don't worry; you do not need
3 to look under the hood. But if you would like to learn more about the C pre-
4 processor, consult any standard C++ reference.

5 The `DEFINE_ART_MODULE` macro instructs the compiler to put some additional
6 code into the shared library made by `builddtool`. This additional code provides
7 the glue that allows *art* to create instances of the class `First` without ever
8 seeing the header or the source for the class; it only gets to see the `.so` and
9 nothing else.



10 The `DEFINE_ART_MODULE` macro adds two pieces of code to the `.so` file. It
11 adds a factory function that, when called, will create an instance of `First` and
12 return a pointer to the base classes `art::EDAnalyzer`. In this way, *art* never
13 sees the derived type of any analyzer module; it sees all analyzer modules via
14 pointer to base. When *art* calls the factory function, it passes as an argument
15 the parameter set specified in the FHiCL file for this module instance. The
16 factory function passes this parameter set through to the constructor of `First`.
17 The second piece of code put into the `.so` file is a static object that will be
18 instantiated at load time; when this object is constructed, it will contact the
19 *art* module registry and register the factory function under the name `First`.
20 When the FHiCL file says to create a module of type `First`, *art* will simply
21 call the registered factory function, passing it the parameter set defined in the
22 FHiCL file. This is the last step in making the connection between the source
23 code of a module and the *art* instantiation of a module.

24 10.8.3.8 Some Alternate Styles

25 C++ allows some flexibility in syntax, which can be seen as either powerful or
26 confusing, depending on your level of expertise. Here we introduce you to a few
27 alternate styles that you will need to recognize and may want to use.

28 Look at the `std::cout` line in the `analyze` method of Listing 10.2:

```
29     std::cout << ``Hello from First::analyze. Event id: ``
30               << event.id()
31               << std::endl;
32 }
```

33 This could have been written:

```
34     art::EventID id = event.id();
35     std::cout << "Hello from First::analyze. Event id: "
36               << id
37               << std::endl;
```

38 This alternate version explicitly creates a temporary object of type `art::EventID`,
39 whereas the original version created an *implicit* temporary object. When you

1 are first learning C++ it is often useful to break down compound ideas by introducing *explicit temporaries*. However, the recommended best practice is to not introduce explicit temporaries unless there is a good reason to do so.

4 Another style that you will certainly encounter is to write the first line of the above as:

```
6 art::EventID id(event.id());
```

7 Here `id` is initialized using *constructor syntax* rather than using *assignment syntax*. For almost all classes these two syntaxes will produce exactly the same result.

10 You may also see the argument list of the `analyze` function written a little differently,

```
12 void analyze( const art::Event& );
```

13 instead of

```
14 void analyze( art::Event const& );
```

15 The position of the `const` has changed. These mean exactly the same thing and the compiler will permit you to use them interchangeably. In most cases, small differences in the placement of the `const` keyword have very different meanings but, in a few cases, both variants mean the same thing. When C++ allows two different syntaxes that mean the same thing, this documentation suite will point it out.



21 Finally, Listing 10.3 shows the same information as Listing 10.2 but using a style in which the namespace remains open after the class declaration. In this style, the leading `tex::` is no longer needed in the definitions of the constructor and of `analyze`. Both layouts of the code have the same meaning to the compiler. You are likely to encounter this style in the source code of many experiments.

Listing 10.3: An alternate layout for `First_module.cc`

```
21 #include ``art/Framework/Core/EDAnalyzer.h``
22 #include ``art/Framework/Core/ModuleMacros.h``
23 #include ``art/Framework/Principal/Event.h``
24
25 #include <iostream>
26
27 namespace tex {
28
29     class First : public art::EDAnalyzer {
30
31     public:
32
33         explicit First(fhicl::ParameterSet const& );
34
35         void analyze(art::Event const& event) override;
36     }
```

```

18     };
19
20     First::First(fhicl::ParameterSet const& ){
21         std::cout << ``Hello from First::constructor.`` << std::endl;
22     }
23
24     void First::analyze(art::Event const& event){
25         std::cout << ``Hello from First::analyze. Event id: ``
26                 << event.id()
27                 << std::endl;
28     }
29
30 }
31
32 DEFINE_ART_MODULE(tex::First)

```

10.9 What does the Build System Do?

10.9.1 The Basic Operation

In Section 10.4.3 you issued the command `buildtool`, which *built* `First_module.cc`. The purpose of this section is to provide some more details about building modules.

When you ran `buildtool` it performed the following steps:

1. It *compiled* `First_module.cc` to create an object file (ending in `.o`).
2. It *linked* the object file against the libraries on which it depends and inserted the result into a shared library (ending in `.so`).

The object file contains the machine code for the class `tex::First` and the machine code for the additional items created by the `DEFINE_ART_MODULE` C preprocessor macro. The shared library contains the information from the object file plus some additional information that is beyond the scope of this discussion. This process is called *building* the module.

The verb *building* can mean different things, depending on context. Sometimes is just means compiling; sometimes is just means linking; more often, as in this case, it means both.

To be complete, when you ran `buildtool` it built all of code in the Workbook, both modules and non-modules, but this section will only discuss how it built `First_module.cc`.

How did `buildtool` know what to do? The answer is that it looked in your source directory, where it found a file named `CMakeLists.txt`; this file contains instructions for `cetbuildtools`. Yes, when you ran `buildtool` in your build directory, it did look in your source directory; it knew to do this because, when you sourced `setup_for_development`, it saved the name of the source

1 directory. The instructions in `CMakeLists.txt` tell `cetbuildtools` to look
 2 for more instructions in the subdirectory `ups` and in the file `art-workbook/CMakeLists.txt`,
 3 which, in turn, tells it to look for more instructions in the `CMakeLists.txt`
 4 files in each subdirectory of `art-workbook`.

5 When `cetbuildtools` has digested these instructions it knows the rules to
 6 build everything that it needs to build.

7 The object file created by the compilation step is a temporary file and, once it
 8 has been inserted into the shared library, it is not used any more. Therefore the
 9 name of the object file is not important.

10 On the other hand, the name of the shared library file is very important. *art*
 11 requires that for every module source file (ending in `_module.cc`) the build
 12 system must create exactly one shared library file (ending in `_module.so`). It
 13 also requires that the name of each `_module.so` file conform to a pattern. Con-
 14 sider the example of the file `First_module.cc`; *art* requires that the shared
 15 library for this file match the pattern

16 `lib*First_module.so`

17 where the `*` wildcard matches 0 or more characters.

18 When naming shared libraries, `buildtool` uses the following algorithm, which
 19 satisfies the *art* requirements and adds some additional features; the algorithm
 20 is illustrated using the example of `First_module.cc`:

- 21 1. find the relative path to the source file, starting from the source directory
 22 `art-workbook/FirstModule/First_module.cc`
- 23 2. replace all slashes with underscores
 24 `art-workbook_FirstModule_First_module.cc`
- 25 3. change the trailing `.cc` to `.so`
 26 `art-workbook_FirstModule_First_module.so`
- 27 4. add the prefix `lib`
 28 `libart-workbook_FirstModule_First_module.so`
- 29 5. put the file into the directory `lib`, relative to the build directory
 30 `lib/libart-workbook_FirstModule_First_module.so`

31 You can check that this file is there by issuing the following command from your
 32 build directory:

33 `$ ls -l lib/libart-workbook_FirstModule_First_module.so`

34 This algorithm guarantees that every module within `art-workbook` will have
 35 a unique name for its shared library.

36 The experiments using *art* have a variety of build systems. Some of these follow
 37 the minimal *art*-conforming pattern, in which the wildcard is replaced with

1 zero characters. If the Workbook had used such a build system, the name of
 2 the shared library file would have been

3 `lib/libFirst_module.so`

4 Both names are legal. Some additional features that are enabled by including
 5 the full path in the name will be discussed in Section .

6 10.9.2 Incremental Builds and Complete Rebuilds

7 When you edit a file in your source area you will need to rebuild that file in
 8 order for those changes to take effect. If any other files in your source area
 9 depend on the file that you edited, they too will need to be rebuilt. To do this,
 10 reissue the `buildtool` command:

11 `$ buildtool`

12 Remember that the `buildtool` command must be executed from your build
 13 directory and that, before running `buildtool`, you must have setup the envi-
 14 ronment in your build window. When you run this command, **cetbuildtools**
 15 will automatically determine which files need to be rebuilt and will rebuild them;
 16 it will not waste time rebuilding files that do not need to be rebuilt. This is
 17 called an *incremental build* and it will usually complete much faster than the
 18 initial build.

19 If you want to clean up everything in your build area and rebuild everything
 20 from scratch, use the following command:

21 `$ buildtool -c`

22 This command will give you five seconds to abort it before it starts removing
 23 files; to abort, type `ctrl-C` in your build window. It will take about the same
 24 time to execute as did your initial build of the Workbook. The name of the
 25 option `-c` is a mnemonic for “clean”.

26 When you do a clean build it will remove all files in your build directory that
 27 are not managed by **cetbuildtools**. For example, if you redirected the output
 28 of *art* as follows,



29 `$ art -c fcl/FirstModule/first.fcl >& first.log`

30 then, when you do a clean build, the file `first.log` will be deleted. This is
 31 why the instructions earlier in this chapter told you to redirect output to a log
 32 file by

33 `$ art -c fcl/FirstModule/first.fcl >& output/first.log`

34 When you ran `buildtool`, it created a directory to hold your output files and
 35 you created a symbolic link, named `output`, from your build directory to this
 36 new directory. Both the other directory and the symbolic link survive clean

1 builds and your output files will be preserved. The Workbook exercises write
 2 all of their root and event-data output files to this directory.

3 If you edit certain files in the `ups` subdirectory of your source directory, rebuild-
 4 ing requires an extra step. If you edit one of these files, the next time that you
 5 run `buildtool`, it will issue an error message saying that you need to re-source
 6 `setup_for_development`. If you get this message, make sure that you are in
 7 your build directory, and

```
8 $ source ../art-workbook/ups/setup_for_development -p $ART\_WORKBOOK\_QUAL
9 $ buildtool
```

10 10.9.3 Finding Header Files at Compile-time

11 When `setup_for_development` establishes the working environment for the
 12 build directory, it does a UPS setup on the UPS products that it requires;
 13 this triggers a chain of additional UPS setups. As each UPS product is
 14 set up, that product defines many environment variables, two of which are
 15 `<PRODUCT-NAME>_INC` and `<PRODUCT-NAME>_LIB`. The first of these points
 16 to a directory that is the root of the header file hierarchy for that version of
 17 that UPS product. The second of these points to a single directory that holds
 18 all of the shared library files for that UPS product.

19 You can spot-check this by doing, for example,

```
20 $ ls $TOYEXPERIMENT_INC/*
21 $ ls $TOYEXPERIMENT_LIB
22 $ ls $ART_INC/*
23 $ ls $ART_LIB
```

24 You will see that the `_INC` directories have a subdirectory tree underneath them
 25 while the `_LIB` directories do not.

26 There are a few small perturbations on this pattern. The most visible is that the
 27 `ROOT` product puts most of its header files into a single directory, `$ROOT_INC`
 28 and does not clone the directory heirarchy of its source files. The `Geant4` product
 29 does the same thing.

30 When the compiler compiles a `.cc` file, it needs to know where to find the files
 31 specified by the `#include` directives. The compiler looks for included files by
 32 first looking for arguments on the command line of the form

```
33 -I<absolute-path-to-a-directory>
```

34 There may be many such arguments on one command line. The compiler as-
 35 sembles the set of all `-I` arguments and uses it as an include path; that is, it
 36 looks for the header files by trying the first directory in the path and if it does
 37 not find it there, it tries the second directory in the path, and so on. The choice
 38 of `-I` for the name of the argument is a mnemonic for Include.

1 When `buildtool` compiles a `.cc` file it adds many `-I` options to the com-
 2 mand line; it adds one for each UPS product that was set up when you sourced
 3 `setup_for_development`. When building `First_module.cc`, `buildtool`
 4 added `-I$ART_INC`, `-I$TOYEXPERIMENT_INC` and many more.



5 A corollary of this discussion is that when you wish to include a header file
 6 from a UPS product, the `#include` directive must contain the relative path
 7 to the desired file, starting from the `_INC` environment variable for that UPS
 8 product.

9 This system illustrates how the Workbook can work the same way on many dif-
 10 ferent computers at many different sites. As the author of some code, you only
 11 need to know paths of include files relative to the relevant `_INC` environment
 12 variable. This environment variable may have different values from one com-
 13 puter to another but the setup and build systems will ensure that the site specific
 14 information is communicated to the compiler using environment variables and
 15 the `-I` option.

16 This system has the potential weakness that if two products each have a header
 17 file with exactly the same relative path name, the compiler will get confused.
 18 Should this happen, the compiler will always choose the file from the earlier
 19 of the two `-I` arguments on the command line, even when the author of the
 20 code intended the second choice to be used. To mitigate this problem, the *art*
 21 and UPS teams have adopted the convention that, whenever possible, the first
 22 element of the relative path in an `#include` directive will be the UPS package
 23 name. It is the implementation of this convention that led to the repeated
 24 directory name `art-workbook/art-workbook` that you saw in your source
 25 directory. There are a handful of UPS products for which this pattern is not
 26 followed and they will be pointed out as they are encountered.

27 The convention of having the UPS product name in the relative path of `#include`
 28 directives also tells readers of the code where to look for the included file.

29 10.9.4 Finding Shared Library Files at Link-time

30 The module `First_module.cc` needs to call methods of the class `art::Event`.
 31 Therefore the compiler left a notation in the object file saying “to use this ob-
 32 ject file you need to tell it where to find `art::Event`.”⁹ The technical way to
 33 say this is that the object file contains a list of *undefined symbols* or *undefined*
 34 *external references*. When the linker makes the shared library

35 `libart-workbook_FirstModule_First_module.so`

36 it must resolve all of the undefined symbols from all of the object files that go
 37 into the library. To resolve a symbol, the linker must learn what shared library

⁹ This is a deliberate vague statement; the next level of detail is too much for this discussion.

1 defines that symbol. When it discovers the answer, it will write the name of
2 that shared library into something called the *dependency list* of

3 `libart-workbook_FirstModule_First_module.so`

4 A dependency list is kept inside each shared library. **cetbuildtools** tells the
5 linker that the dependency list should contain only the filename of each shared
6 library, not the full path to it. If, after the linker has finished, there remain
7 unresolved symbols, then the linker will issue an error message and the build
8 will fail.

9 Dependency lists are not recursive. If library A depends on library B and library
10 B depends on library C, then the dependency list library A needs to contain
11 only library B. Sometimes this is discussed by saying that the dependency list
12 needs to contain only *direct dependencies* or *first order* dependencies.

13 To learn where to look for symbol definitions, the linker looks at its command
14 line to find something called the *link list*. The link list can be specified in several
15 different ways and the way that **cetbuildtools** uses is simply to write the link
16 list as the absolute path to every `.so` file that the linker needs to know about.
17 The link list can be different for every shared library that the build system
18 builds. However it is very frequently true that if a directory contains several
19 modules, then all of the modules will require the same link list. The bottom
20 line is that the author of a module needs to know the link list that is needed to
21 build the shared library for that module.

22 For these Workbook exercises, the author of each exercise has determined the
23 link list for each shared library that will be built for that exercise. In the
24 **cetbuildtools** system, the link list for `First_module.cc` is located in the
25 `CMakeLists.txt` file from same directory as `First_module.cc`; the con-
26 tents of this file are shown in Listing 10.4. This `CMakeLists.txt` file says
27 that all modules found in this directory should be built with the same link list
28 and it gives the link list; the link list is the seven lines that begin with a dollar
29 sign; these lines each contain one `cmake` variable. Recall that **cetbuildtools**
30 is a build system that lives on top of `cmake`, which is another build system. A
31 `cmake` variable is much like an environment variable except that is only defined
32 within the environment of the running build system; you cannot look at it with
33 `printenv`.

34 The five `cmake` variables beginning with `ART_` were defined when `buildtool`
35 set up the UPS *art* product. Each of these variables defines an absolute path to
36 a shared library in `$ART_LIB`. For example `${ART_FRAMEWORK_CORE}` resolves
37 to

38 `$ART_LIB/libart_Framework_Core.so`

39 Almost all *art* modules will depend on these five libraries. Similarly the other
40 two variables resolve to shared libraries in the **fhiclcpp** and **cetlib** UPS prod-
41 ucts.

1 When **cetbuildtools** constructs the command line to run the linker, it copies
 2 the link list from the `CMakeLists.txt` file to the command linker line.

3 The experiments that use *art* use a variety of build systems. Some of these
 4 build systems do not require that all external symbols be resolved at link-time;
 5 they allow some external symbols to be resolved at run-time. This is legal but
 6 it can lead to certain difficulties. *A future version of this documentation suite*
 7 *will contain a chapter in the Users Guide that discusses linkage loops and how*
 8 *use of closed links can will prevent them. This section will then just reference*
 9 *it.*

10 Consult the `cmake` and **cetbuildtools** documentation to understand the re-
 11 maining details of this file.

Listing 10.4: The file `art-workbook/FirstModule/CMakeLists.txt`

```
1b art_make(MODULE_LIBRARIES
12   ${ART_FRAMEWORK_CORE}
13   ${ART_FRAMEWORK_PRINCIPAL}
14   ${ART_PERSISTENCY_COMMON}
15   ${ART_FRAMEWORK_SERVICES_REGISTRY}
16   ${ART_FRAMEWORK_SERVICES_OPTIONAL}
17   ${FHICL_CPP}
18   ${CETLIB}
19 )
```

21 10.9.5 Build System Details



22 This section provides the next layer of details about the build system; *in a*
 23 *future version of this documentation set, the Users Guide will have a chapter*
 24 *with all of the details.* This entire section contains expert material.

25 If you want to see what `buildtool` is actually doing, you can enable verbose
 26 mode by issuing the command:

```
27 $ buildtool VERBOSE=TRUE
```

28 For example, if you really want to know the name of the object file, you can
 29 find it in the verbose output. For this exercise, the object file is

```
30 ./art-workbook/FirstModule/CMakeFiles/  
31 art-workbook_FirstModule_First_module.dir/First_module.cc.o
```

32 where the above is really just one line.

33 Also, you can read the verbose listing to discover the flags given to the compiler
 34 and linker. The compiler and linker flags, valid at time of writing are given in
 35 Table 10.1; actually a few of them are not present in the table because they
 36 take a lot of space but don't provide critical functionality. The C++ 11 features
 37 are selected by the presence of the `-std=c++11` flag and a high level of error
 38 checking is specified. The linker flag,

Part

Table 10.1: Compiler and Linker Flags for a Profile Build

Step	Flags
Compiler	-Dart_workbook_FirstModule.First_module_EXPORTS -DNDEBUG
Linker	-Wl,--no-undefined -shared
Both	-O3 -g -fno-omit-frame-pointer -Werror -pedantic -Wall -Werror=return-type -Wextra -Wno-long-long -Winit-self -Woverloaded-virtual -std=c++11 -D_GLIBCXX_USE_NANOSLEEP -fPIC

1 -Wl,--no-undefined

2 tells the linker that it must resolve all external references at link time. This is
3 sometime referred to as a *closed link*.

4 10.10 Suggested Activities

5 This section contains some suggested exercises in which you will make your
6 own modules and in which you will learn more about how to use the class
7 `art::EventID`.

8 10.10.1 Create Your Second Module

9 In this exercise you will create a new module by copying `First_module.cc`
10 and making the necessary changes; you will build it using `buildtool`; you make
11 a FHiCL file to run the new module by copying `first.fcl` and making the
12 necessary changes; and you will run the new module using the new FHiCL
13 file.

14 Go to your source window and `cd` to your source directory. If you have logged
15 out, remember to re-establish your working environment; see Section 10.6
16 Type the following commands:

```
17 $ cd art-workbook/First
18 $ cp First_module.cc Second_module.cc
19 $ cp first.fcl second.fcl
```

20 Edit the files `Second_module.cc` and `first.fcl`. In both files, change every
21 occurrence of the string “First” to “Second”; there are eight places in the source
22 file and two in the FHiCL file, one of which is in a comment.

23 The new module needs the same link list as did `First_module.cc` so there is
24 no need to edit `CMakeLists.txt`; the instructions in `CMakeLists.txt` tell
25 `buildtool` to build all modules that it finds in this directory and to use the
26 same link list for all modules.

1 Go to your build window and `cd` to your build directory. If you have logged, out
 2 remember to re-establish your working environment; see Section 10.6. Rebuild
 3 the Workbook code:

4 `$ buildtool`

5 This should complete with the message:

```
6 -----
7 INFO: Stage build successful.
8 -----
```

9 If you get an error message, consult a local expert or consult the *art* team as
 10 described in Section 3.4.

11 When you run `buildtool` it will perform an incremental build (see Sec-
 12 tion 10.9.2), during which it will detect `Second_module.cc` and build it.

13 You can verify that `buildtool` created the expected shared library:

```
14 $ ls lib/*Second*.so
15 lib/libart-workbook_FirstModule_Second_module.so
```

16 Stay in your build directory and run the new module:

```
17 $ art -c fcl/FirstModule/second.fcl >& output/second.log
```

18 Compare `output/second.log` with `output/first.log`. You should see
 19 that the printout from `First_module.cc` has been replaced by that from
 20 `Second_module.cc`.

21 10.10.2 Use `artmod` to Create Your Third Module

22 This exercise is much like the previous one; the difference is that you will use a
 23 tool named `artmod` to create the source file for the module.

24 Go to your source window and `cd` to your source directory. If you have logged
 25 out, remember to re-establish your working environment; see Section 10.6

26 The command `artmod` creates a file containing the skeleton of a module. It
 27 is supplied by the UPS product **cetpkgsupport**, which was set up when you
 28 performed the last step of establishing the environment in the source window,
 29 sourcing `setup_deps`. You can verify that the command is in your path by
 30 using the bash built-in command `type`:

```
31 $ type artmod
32 artmod is hashed (/ds50/app/products/cetpkgsupport/v1_02_00/bin/artmod)
```

33 In general the leading elements of the directory name will be different on your
 34 computer; they will be the leading elements of your UPS products area.

35 From your source directory, type the following commands:

Part

```
1 $ cd art-workbook/First
2 $ artmod analyzer tex::Third
3 $ cp first.fcl third.fcl
```

4 The second argument tells artmod to create a file named `Third_module.cc`
5 that contains the skeleton for a module named `Third` in the namespace `tex`;
6 the first argument tells artmod that it should write the skeleton for an analyzer
7 module.

8 If you compare `Third_module.cc` to `First_module.cc` you will see a few
9 differences:

- 10 1. the layout of the class is a little different but the two layouts are equivalent
- 11 2. there are some extra `#include` directives
- 12 3. the include for `<iostream>` is missing
- 13 4. a formal parameter is supplied in the definition of the constructor
- 14 5. in the `analyze` member function, the name of the formal parameter is
15 different.
- 16 6. artmod supplies the skelton of a destructor

17 The `#include` directives provided by artmod are a best guess, made by the
18 author of artmod, about what `#include` directives will be needed in a “typ-
19 ical” module. Other than slowing down the compiler by an amount you won’t
20 notice, the extra `#include` directives do no harm; keep them or leave them as
21 you see fit.

22 Edit `Third_module.cc`

- 23 1. add the `#include` directive for `<iostream>`
- 24 2. copy the bodies of the constructor and the `analyze` member function
25 from `First_module.cc`; change the string “First” to “Third”.
- 26 3. delete the formal parameter from the definition of the constructor
- 27 4. in the definition of the member function `analyze`, change the name of
28 the formal parameter to `event`.

29 When you built `First_module.cc`, the compiler wrote a destructor for you
30 that is identical to the destructor written by artmod; so you can leave the
31 destructor as artmod wrote it. This class has no work to do in the destructor
32 so the one written by artmod has an empty body.

33 Edit `third.fcl` Change every occurence of the string “First” to “Third”; there
34 are two places, one of which is in a comment.

35 Go to your build window and `cd` to your build directory. If you have logged, out
36 remember to re-establish your working environment; see Section 10.6. Rebuild
37 the Workbook code:


```

1  $ buildtool
2  Refer to the previous section to learn how to identify a successful build and how
3  to verify that the expected library was created.
4  Stay in your build directory and run the third module:
5  $ art -c fcl/FirstModule/third.fcl >& output/third.log
6  Compare output/third.log with output/first.log. You should see
7  that the printout from First_module.cc has been replaced by that from
8  Third_module.cc.
9  artmod has many options that you can explore by typing:
10 $ artmod --help

```

11 10.10.3 Running Many Modules at Once

12 In this exercise you will run four modules at once, the three made in this exercise
 13 plus the HelloWorld module from Chapter 9.

14 Go to your source window and `cd` to your source directory. Type the following
 15 commands:

```

16 $ cd art-workbook/First
17 $ cp first.fcl all.fcl

```

18 Edit the file `all.fcl` and replace the `physics` parameter set with that found
 19 in Listing 10.5. This parameter set:

- 20 1. defines four module labels
- 21 2. puts all four module labels into the `end_paths` sequence.

22 When you run `art` on this FHiCL file, `art` will first look at the definition of
 23 `end_paths` and learn that you would like it to run four module labels. Then it
 24 will look in the `analyzers` parameter set to find the definition of each module
 25 label; in each definition `art` will find the class name of the module that it should
 26 run. Given the class name and the environment variable `LD_LIBRARY_PATH`,
 27 `art` can find the right shared library to load. If you need a refresher on module
 28 labels and `end_paths`, refer to Sections 9.7.7 and 9.7.8.

Listing 10.5: The `physics` parameter set for `all.fcl`

```

2b physics :{
3b   analyzers: {
4b     hello : {
5b       module_type : HelloWorld
6b     }
7b     first : {
8b       module_type : First
9b     }
10b    second : {

```

Part

```

10     module_type : Second
11   }
12   third : {
13     module_type : Third
14   }
15 }
16
17 el      : [ hello, first, second, third ]
18 end_paths : [ el ]
19
20 }

```

Go to your build window and `cd` to your build directory. If you have logged, out remember to re-establish your working environment; see Section 10.6. You do not need to build any code for this exercise.

Run the exercise:

```
$ art -c fcl/FirstModule/all.fcl >& output/all.log
```

Compare `output/all.log` with the log files from the previous exercises. The new log file should contain printout from each of the four modules. Once, near the start of the file, you should see the printout from the three constructors; remember that the `HelloWorld` module does not make any printout in its constructor. For each event you should see the printout from the four `analyze` member functions.

Remember that `art` is free to run analyzer modules in any order; this was discussed in Section 9.7.8.

10.10.4 Access Parts of the EventID

In this exercise, you will access the individual parts of the event identifier.

Before proceeding with this section, review the material in Section 10.8.3.6 which discusses the class `art::EventID`. The header file for this class is:

```
$ART_INC/art/Persistency/Provenance/EventID.h"
```

In this exercise, you are asked to rewrite the file `Second_module.cc` so that the printout made by the `analyze` method looks like the following:

```

32 Hello from FirstAnswer01::analyze.  run number: 1 sub run number: 0 event number: 1
33 Hello from FirstAnswer01::analyze.  run number: 1 sub run number: 0 event number: 2

```

and so on for each event.

To do this, you will need to reformat the text in the `std::cout` statement and you will need to separately extract the run, subRun and event numbers from the `art::EventID` object.

You will do the editing in your source window, in the subdirectory `art-workbook/FirstModule`.

1 When you think that you have successfully rewritten the module, you can test it
2 by going to your build window and cd'ing to your build directory. Then:

```
3 $ buildtool
4 $ art -c fcl/FirstModule/second.fcl >& output/eventid.log
```

5 Work on this for 15 minutes or so. If you have not figured out how to do it,
6 you can look at the file `FirstAnswer01.module.cc`, in the same directory
7 as `First.module.cc`. This file is one possible answer.

8 To run the answer module, to verify that it makes the requested output:

```
9 $ art -c fcl/FirstModule/firstAnswer01.fcl >& output/firstAnswer01.log
```

10 You did not need to build this module because it was already built the first time
11 that you ran `buildtool`; that run of `buildtool` built all of the modules in
12 the Workbook.

13 There is second correct answer to this exercise. If you look at the header file for
14 `art::Event`, you will see that this class also has member functions

```
15     EventNumber_t    event()    const {return aux_.event();}
16     SubRunNumber_t   subRun()   const {return aux_.subRun();}
17     RunNumber_t      run()      const {return id().run();}
```

18 So you could have called these directly,

```
19     std::cout << ``Hello from FirstAnswer01::analyze. ``
20               << `` run number: ``      << event.run()
21               << `` sub run number: `` << event.subRun()
22               << `` event number: ``    << event.event()
23               << std::endl;
```

24 Instead of

```
25     std::cout << ``Hello from FirstAnswer01::analyze. ``
26               << `` run number: ``      << event.id().run()
27               << `` sub run number: `` << event.id().subRun()
28               << `` event number: ``    << event.id().event()
29               << std::endl;
```

30 But the point of this exercise was to learn a little about how to dig down into
31 nested header files to find the information you need.

32 10.11 Final Remarks

33 10.11.1 Why is there no `First.module.h` File?

34 When you performed the exercises in this chapter, you saw, for example, the
35 file `First.module.cc` but there was no corresponding `First.module.h` file.

1 This section will explain why.

2 In a typical C++ programming environment there is a header file (`.h`) for each
3 source file (`.cc`). For definiteness, consider the examples of `Point.h` and
4 `Point.cc` that you saw in Section 6.6.10.

5 The reason for having `Point.h` is that the implementation of the class, `Point.cc`,
6 and the users of the class, need to agree on what the class `Point` is; in this
7 example the only user of the class is the main program, `pctest.cc`. The file
8 `Point.h` serves as the unique, authoritative declaration of what the class is;
9 both `Point.cc` and `pctest.cc` rely on on this declaration.

10 If you think carefully, you are already aware of a very common exception to the
11 pattern of one `.h` file for each `.cc` file: there is never header file for a main pro-
12 gram. For example, in the examples that exercised the class `Point`, there was
13 never a header file for the main program `pctest.cc`. The reason for this is that
14 there is no other piece of user written code that needs to know about the decla-
15 ration of any classes or functions declared or defined inside `pctest.cc`.

16 The reason that there is no `First_module.cc` file is simply that every entity
17 that needs to see the declaration of the class `First` is already inside the file
18 `First_module.cc`. Therefore there is no reason to have a separate header
19 file. There was a dangerous bend paragraph at the end of Section 10.8.3.7 that
20 described how *art* is able to use modules without needing to know about the
21 declaration of the module class.

22 The architecture of *art* says that only *art* may construct instances of module
23 classes and only *art* may call member functions of module classes. In particular,
24 modules may not construct other modules and may not call member functions of
25 other modules. The absence of a `First_module.h`, provides a physical barrier
26 that enforces this design.

27 10.11.2 The Three File Module Style

28 In this chapter, the source for the module `First` was written in a single file. You
29 may also write it using three files, `First.h`, `First.cc` and `First_module.cc`.
30 The authors of *art* do not recommend this style because it exposes the decla-
31 ration of `First` in a way that permits it to be misused (as was discussed in
32 Section 10.11.1).

33 However some experiments do use this style. Therefore this section has been
34 provided.

35 In this style, `First.h` contains the class declaration plus any necessary `#include`
36 directives; it also now requires code guards (see Section 31.8); this is shown in
37 Listing 10.6. The file `First.cc` contains the definitions of the constructor and
38 the `analyze` member function, plus the necessary `#include` directives; this
39 is shown in Listing 10.7. And `First_module.cc` is now stripped down to the

- 1 invocation of the `DEFINE_ART_MODULE` macro plus the necessary `#include`
- 2 directives; this is shown in Listing 10.8.
- 3 The build system distributed with the Workbook has not been configured to
- 4 build modules written in this style.

Listing 10.6: The contents of `First.h` in the 3 file model

```

1
2 #ifndef art-workbook_FirstModule_First_h
3 #define art-workbook_FirstModule_First_h
4
5 #include ``art/Framework/Core/EDAnalyzer.h``
6 #include ``art/Framework/Principal/Event.h``
7
8 namespace tex {
9
10     class First : public art::EDAnalyzer {
11
12     public:
13
14         explicit First(fhicl::ParameterSet const& );
15
16         void analyze(art::Event const& event) override;
17
18     };
19
20 }
21 #endif

```

Listing 10.7: The contents of `First.cc` in the 3 file model

```

22
23 #include ``art-workbook/FirstModule/First.h``
24
25 #include <iostream>
26
27 tex::First::First(fhicl::ParameterSet const& ){
28     std::cout << ``Hello from First::constructor.`` << std::endl;
29 }
30
31 void tex::First::analyze(art::Event const& event){
32     std::cout << ``Hello from First::analyze. Event id: ``
33         << event.id()
34         << std::endl;
35 }

```

Listing 10.8: The contents of `First_module.cc` in the 3 file model

```

46
47 #include ``art-workbook/FirstModule/First.h``
48 #include ``art/Framework/Core/ModuleMacros.h``
49
50 DEFINE_ART_MODULE(tex::First)

```

1 10.12 Review

2 10.12.1 What Makes a class an Analyzer Module

3 This section reviews the properties that a class must have in order to be an
4 analyzer module. These were first given in Section 10.8.3.2 but are repeated here
5 for easy reference:

- 6 1. it must inherit from `art::EDAnalyzer`
- 7 2. it must provide a constructor with the argument list
8 `fhicl::ParameterSet const&`
- 9 3. it must provide a member function named `analyze`, with the signature:
10 `analyze( art::Event const& )`
- 11 4. if the name of a module class is `<ClassName>` then the source code for
12 the module must be in a file named `<ClassName>.module.cc` and this
13 file must contain the lines:
14 `#include ``art/Framework/Core/ModuleMacros.h```
15 `DEFINE_ART_MODULE( tex::<ClassName> )`
- 16 5. it may, optionally, provide other member functions with signatures pre-
17 scribed by *art*; if these member functions are present in a module class,
18 then *art* will call them at the appropriate times. Some examples are pro-
19 vided in Chapter 11

20 The *art* team recommends that you write modules using the one file style, not
21 the three file style; the above list is written presuming that you use the one file
22 style.

23 10.12.2 Flow from source to .fcl

24 This section reviews how the source code found in `First_module.cc` is exe-
25 cuted by *art*:

- 26 1. the script `setup_for_development` defines many environment variables
27 that are used by `buildtool` and by *art* and `toyExperiment`.
- 28 2. one of the important environment variables is `LD_LIBRARY_PATH`. This
29 contains the directory `lib` in your build area plus the `lib` directories
30 from many UPS products, including *art*.
- 31 3. `buildtool` compiles `First_module.cc` to a temporary object file.
- 32 4. `buildtool` links the temporary object file to create a shared library in the
33 `lib` subdirectory of your build area:
34 `lib/libart-workbook.FirstModule.First_module.so`

- 1 5. when you run *art* using file `first.fcl`, the FHiCL file tells *art* to find
- 2 and load a module with the `module_type First`.
- 3 6. in response to this request, *art* will search the directories in `LD_LIBRARY_PATH`
- 4 to find a shared library file whose name matches the pattern:
- 5 `lib*First_module.so`
- 6 7. if *art* finds zero matches to this pattern, or more than one match to this
- 7 pattern, it will issue an error message and stop
- 8 8. if *art* finds exactly one match to this pattern, it will load the shared library.
- 9 9. after *art* has loaded the shared library, it has access to a function that
- 10 can, on demand, create instances of the class `First`.



- 11 The last bullet really means that the shared library contains a factory function
- 12 that can construct instances of `First` and return a pointer to the base class,
- 13 `art::EDANalyzer`. The shared library also contains a static object that, at
- 14 load-time, will contact the *art* module registry and register the factory function
- 15 under the `module_type First`.

11 Exercise 3: The Optional Member Functions of *art* Modules

11.1 Introduction

In this exercise you will build and execute an analyzer module that illustrates three of the optional member functions of an *art* module: `beginJob`, `beginRun` and `beginSubRun`. These member functions are called, respectively, once at the start of the *art* job, once for each new run and once for each new subRun. These member functions are optional functions in all types of modules, not just analyzer modules.

You will also be given a suggested exercise to add three more of the optional member functions, `endJob`, `endRun` and `endSubRun`. The Workbook provides a solution for this suggested exercise.

11.2 Prerequisites

The prerequisites for this chapter is all of the material in Part I (Introduction) and all of the material up to this point in Part II (Workbook).

In particular make sure that you understand the idea of the event loop, that was described in Section 3.6.2.

11.3 What You Will Learn

You will learn about the optional member functions of an *art* module.

1. `beginJob()`
2. `beginRun( art::Run const&)`
3. `beginRun( art::SubRun const&)`


```

1    4. endJob()
2    5. endRun( art::Run const&)
3    6. endRun( art::SubRun const&)
4    You will also learn about the classes,
5    1. art::RunID
6    2. art::Run
7    3. art::SubRunID
8    4. art::SubRun

```

9 11.4 Setting up to Run this Exercise

10 To run this exercise, you need to be logged in to the computer on which you ran
 11 Exercise 2 (in Chapter 10). If you are continuing on from the previous exercise,
 12 you need to keep both your source and build windows open.

13 If you are logging back in, follow the instructions in Section 10.6 to reestablish
 14 your source and build windows.

15 In your source window, `cd` to your source directory. Then `cd` to the directory
 16 for this exercise and look at its contents

```

17 $ cd art-workbook/OptionalMethods
18 $ ls
19 CMakeLists.txt          OptionalAnswer01_module.cc  Optional_module.cc
20 optionalAnswer01.fcl    optional.fcl

```

21 The source code for the module you will run is `Optional_module.cc` and the
 22 FHiCL file to run it is `first.fcl`. The file `CMakeLists.txt` is identical that
 23 used by the previous exercise; this is because the new features introduced by
 24 this module do not require any modifications to the link list. The other two files
 25 are the answers to the exercise you will be asked to do in Section 11.9.

26 In your build window, make sure that you are in your build directory. In this
 27 exercise you do not need to build any code because all code for the Workbook
 28 was built the first time that you ran `buildtool`.

29 11.5 The Source File `Optional_module.cc`

30 In your source window, look at the source file, `Optional_module.cc` and
 31 compare it to `First_module.cc`. The differences are

32 1. the name of the class has changed from `First` to `Optional`

- 1 2. it has two new include directives, for `Run.h` and `SubRun.h`
- 2 3. the class declaration declares three new member functions
- 3 `void beginJob () override;`
- 4 `void beginRun ( art::Run const& run ) override;`
- 5 `void beginSubRun( art::SubRun const& subRun ) override;`
- 6 4. the text printed by the constructor and the analyze member functions has
- 7 changed
- 8 5. the file contains the definitions of the three new member functions, each
- 9 of which simply makes some identifying printout

10 Each of the member functions introduced in this section must have exactly the
 11 argument list prescribed by *art*. The `override` keyword instructs the compiler
 12 to do the following: if a member function has a name that is spelled incorrectly
 13 or it if has an incorrect argument list, the compiler will issue an error message
 14 and stop.

15 This is a very handy feature. If the `override` keyword were absent, and if
 16 the function were spelled incorrectly or the argument list were incorrect, then
 17 the compiler would assume that it was your intention to define a new member
 18 function that is unrelated to one of the optional *art* defined member functions.
 19 The result would be a difficult to diagnose run-time error: *art* would simply not
 20 recognize your member function and would never call it.

21 Always provide the `override` keyword when your class provides one of the
 22 optional *art* defined member functions.



23 For those with some C++ background, the three member functions `beginJob`,
 24 `beginRun` and `beginSubRun` are declared as virtual in the base class,
 25 `art::EDAnalyzer`. The `override` keyword is new in C++-11 and will not be
 26 described in older text books. It instructs the compiler that this member func-
 27 tion is intended to override a virtual function from the base class; if the compiler
 28 cannot find such a function in the base class, it will issue an error.



29 As described in Section 3.6.2, *art* will call the `beginJob` method of each module
 30 once at the start of the job; it will call the `beginRun` method of each module at
 31 the start of each run and it will call the `beginSubRun` method of each module
 32 at the start of each `sunRun`.

33 11.6 The classs `art::Run`, `art::RunID`, `art::SubRun` 34 and `art::SubRunID`

35 In Section 10.8.3.6 you learned about the class `art::EventID`, which describes
 36 the three-part event identifier. *art* also provides two related classes:

- 37 1. `art::RunID`, which is an one-part identifier for a run number

1 2. `art::SubRunID`, which is a two-part identifier for a subRun

2 You can find the header files for these classes at:

```
3   $ less $ART_INC/art/Persistency/Provenance/RunID.h
4   $ less $ART_INC/art/Persistency/Provenance/SubRunID.h
```

5 Remember that you type a lower case letter “q” to exit less. Or you can
6 look at these header files using one of the code browsers described in Sec-
7 tion 7.6.2.

8 The argument to the `beginRun` method is a `const` reference to an object
9 of type `art::Run`. This object is similar to an `art::Event`: a simplified
10 picture is that it holds an `art::RunID` plus a collection of data products. The
11 purpose of the `art::Run` object is to hold information that describes an entire
12 run; some of that information might be available at the start of the run but
13 some of it might only be added at the end of the run.

14 You can find the header file for `art::Run` at:

```
15   $ less $ART_INC/art/Framework/Principal/Run.h
```

16 If you take a snapshot of a running *art* job you will see that there is exactly one
17 object of type `art::Run`. This object is owned by *art* and *art* gives modules
18 access to it when it calls their `beginRun` and `endRun` methods. Because it is
19 passed by reference, the `beginRun` member function does not get a copy of the
20 `art::Run` object; instead it has access to the unique `art::Run` object that is
21 owned by *art*. Because it is passed by `const` reference, your analyzer module
22 may look at information in the run object but it may not add information to
23 the run object.

24 In your `analyze` member function, if you have an `art::Event`, named `event`,
25 you can access the associated run information by:

```
26   art::Run const& run = event.getRun();
```

27 You may sometimes see this written as:

```
28   auto const& run = event.getRun();
```

29 Both version mean exactly the same thing. When a type is long and awkward to
30 write, the `auto` keyword is very useful; however it is likely to be very confusing
31 to beginners. When you encounter it, check the header files for the classes on
32 right hand side of the assignment; from there you can learn the return type of
33 the member function that returned the information.



34 In both cases the `const&` is very important. If you omit the reference part, the
35 `&`, then the variable `run` will contain a *copy* of the run object that is owned by
36 *art*. This is a waste of both CPU time and memory and in some circumstances
37 it can be a significant waste.

38 If you omit the `const`, but remember the `&`, then you will get a compiler
39 error because the `getRun()` method only permits you `const` access to the run

1 object.

2 There is a very important habit that you need to develop as a user of *art*. Many
 3 methods in *art*, in the Workbook code and very likely in your experiment's code,
 4 will return information by `&` or by `const&`. If you receive these by value, not by
 5 reference, then you will make copies that waste both CPU and memory; in some
 6 cases these can be significant wastes. Unfortunately there is no way to tell the
 7 compiler to catch this mistake. The only solution is your own vigilance.



8 In the call to `beginSubRun` the argument is of type `art::SubRun const&`.
 9 A simplified description of this object is that it contains an `art::SubRunID`
 10 plus a collection of data products that describe the subRun. All of the com-
 11 ments about the class `art::Run` in the preceding few paragraphs apply to
 12 `art::SubRun`. You can find the header file for `art::SubRun` at:

```
13 $ less $ART_INC/art/Framework/Principal/SubRun.h
```

14 If you have an `art::Event`, named `event`, you can access the associated
 15 subRun object by,

```
16 art::SubRun const& subRun = event.getSubRun();
```

17 If you have an `art::SubRun` object, named `subRun`, you can access the asso-
 18 ciated `art::Run` object:

```
19 art::Run const& run = subRun.getRun();
```

20 11.7 Running this Exercise

21 Look at the file `optional.fcl`. This FHiCL file runs the module `Optional`
 22 on the the input file `inputFiles/input03_data.root`. Consult Table 9.1
 23 and you will see that this file contains 15 events, all from run 3. It contains
 24 events 1 through 5 from each of subRuns 0, 1 and 2. With this knowledge,
 25 and the knowledge of the source file `Optional.module.cc`, you should have
 26 a clear idea of what this module will print out.

27 In your build directory, run the following command

```
28 $ art -c fcl/OptionalMethods/optional.fcl >& output/optional.log
```

29 The part of the printed output that comes from the module `Optional` is given
 30 in Listing 11.1. Is this what you expected to see? If not, understand why this
 31 module made the printout that it did. If you did not get this printout, double
 32 check that you followed the instructions carefully; if that still does not fix it,
 33 ask for help (see Section 3.4).

Listing 11.1: The output produced by `Optional_module.cc` when run using `optional.fcl`

```

1
2 Hello from Optional::constructor.
3 Hello from Optional::beginJob.
4 Hello from Optional::beginRun: run: 3
5 Hello from Optional::beginSubRun: run: 3 subRun: 0
6 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 1
7 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 2
8 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 3
9 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 4
10 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 5
11 Hello from Optional::beginSubRun: run: 3 subRun: 1
12 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 1
13 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 2
14 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 3
15 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 4
16 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 5
17 Hello from Optional::beginSubRun: run: 3 subRun: 2
18 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 1
19 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 2
20 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 3
21 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 4
22 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 5

```

11.8 The Member Function `beginJob`

The member function `beginJob` gets called once at the start of the job. The constructor of the each module is also called once at the start of the job. This brings up the question, what code belongs in the constructor and what code belongs in the `beginJob` member function.

The answer to this question is partly clean and partly fuzzy. *art* does require that some tasks be done in the constructor, not in the `beginJob` member function, but the examples that you have seen so far do not have enough richness to illustrate this. These tasks will be pointed out as you encounter them.

The second part of the answer is that we strongly encourage you to initialize as many of your data members as possible using the colon initializer syntax. This is simply a C++ best practice: if at all possible, do not allow uninitialized or incompletely initialized variables of any kind.

Other tasks can be done in the constructor or the `beginJob` member function as you see fit. One reasonable guideline is that physics related tasks belong in the `beginJob` member function while computer science related tasks belong in the constructor. Your experiment may have additional guidelines.

For those of you familiar with ROOT, we can provide an example of something physics related. We suggest that you create histograms, ntuples or trees in one of

1 the `begin` methods, perhaps `beginJob` or `beginRun`. Other constraints may
2 enter into this decision. If booking a histogram requires geometry information
3 to set the limits correctly, that information may be run dependent and you will
4 need to book these histograms in `beginRun`, not `beginJob`.



5 11.9 Suggested Activities

6 11.9.1 Add the Matching `end` Member functions

7 *art* defines the following three member functions:

```
8     void endJob      () override;  
9     void endRun      ( art::Run const&      run      ) override;  
10    void endSubRun    ( art::SubRun const& subRun ) override;
```

11 Go to your source window. In the file `Optional_module.cc`, add these meth-
12 ods to the declaration of the class `Optional` and provide an implementation
13 for each. In your implementation, just copy the printout created in the corre-
14 sponding `begin` function and, in that printout, change the string “begin” to
15 “end”.

16 Then go to your build window and make sure that your current directory is your
17 build directory. Then rebuild this module and run it:

```
18 $ buildtool  
19 $ art -c fcl/OptionalMethods/optional.fcl >& output/optional2.log
```

20 Consult Chapter 10 if you need to remember how to indentify that the build
21 completed successfully. Compare the output from this run of *art* with that of
22 the previous run: do you see the additional printout from the methods that you
23 added?

24 The solution to this activity is provided as the file `OptionalAnswer01_module.cc`.
25 It is already built. You can run it with:

```
26 $ art -c fcl/OptionalMethods/optionalAnswer01.fcl >& output/optionalAnswer01.log
```

27 Does the output of your code match the output from this code?

1 11.9.2 Run on Multiple Input Files

2 In a single run of *art*, run your modified version of the module `Optional` on
3 all of the three of the following input files:

4 `inputFiles/input01_data.root`
5 `inputFiles/input02_data.root`
6 `inputFiles/input03_data.root`

7 If you need a reminder about how to tell *art* to run on three input files in one
8 job, consult Section 9.7.5.

9 Make sure that the printout from this job matches the description of the event
10 loop found in Section 3.6.2; if it does not ask for help (see Section 3.4).

12 Exercise 4: A First Look at Parameter Sets

12.1 Introduction

In the previous few chapters you used FHiCL files to configure *art*. Among other things, you learned how to define a module label and its corresponding parameter set:

```
moduleLabel : {  
    module_type : ClassName  
}
```

where `module_type` is a name that is reserved to *art*, `ClassName` is the name of a module class and where the `moduleLabel` is an identifier that you get to define.

When you define a module label, you may add additional FHiCL definitions between the braces. For example:

```
moduleLabel : {  
    module_type      : ClassName  
    thisParameter    : 1  
    thatParameter    : 3.14159  
    anotherParameter : "a string"  
    arrayParameter   : [ 1, 3, 5, 7, 11] }  
    nestedPSet       : {  
                        a : 1  
                        b : 2  
                    }  
}
```

These additional definitions have the following properties:

1. Each definition must be a legal FHiCL definition.
2. These definitions have no meaning to *art* or to FHiCL.

- 1 3. The module specified by the `module_type` parameter defines which pa-
2 rameters must be present and which parameters may be provided option-
3 ally.
 - 4 4. Each definition may use the full power of FHiCL and may contain nested
5 parameter sets to arbitrary depth.
- 6 This functionality allows you to write modules whose behaviour is run-time
7 configurable. If, for example, you have a reconstruction algorithm that depends
8 on some cuts, the values of those cuts can be made run-time configurable.

9 12.2 Prerequisites

10 The prerequisite for this chapter is all of the material in Part I (Introduction)
11 and the material in Part II (Workbook) up to and including Chapter 10. You
12 can read this chapter without having read Chapter 11.

13 12.3 What You Will Learn

14 In all of the previous exercises, you saw that the constructor of a module takes
15 an argument of type `fhiCL::ParameterSet const&`. The modules that
16 you have seen so far have not used this argument.

17 In this chapter you will learn how to use this argument to read the additional
18 FHiCL definitions described in Section 12.1. In particular, you will learn:

- 19 1. About the class `fhiCL::ParameterSet`.
- 20 2. How to read parameters from the parameter set.
- 21 3. How to require that a parameter be present in a parameter set.
- 22 4. How to specify that a parameter has a default value that will be used if
23 the parameter is not present in a parameter set.
- 24 5. How to print a parameter set.
- 25 6. The two special parameters automatically provided by *art*.

26 You will also be introduced to C++ templates and C++ exceptions; these will
27 only be described in the minimum required detail.

28 12.4 Setting up to Run this Exercise

29 To run this exercise, you need to be logged in to the computer on which you ran
30 Exercise 2 (in Chapter 10). If you are continuing on from a previous exercise,

Listing 12.1: The definition of `psetTester` from `pset01.fcl`

```

1
2   psetTester : {
3     module_type : PSet01
4     a : "this is quoted string"
5     b : 42
6     c : 3.14159
7     d : true
8     e : [ 1, 2, 3 ]
9     f : {
10      a : 4
11      b : 5
12    }
13  }

```

- 1 you need to keep both your source and build windows open.
- 2 If you are logging back in, follow the instructions in Section 10.6 to reestablish
- 3 your source and build windows.
- 4 In your source window, `cd` to your source directory. Then `cd` to the directory
- 5 for this exercise and look at its contents
- 6 `$ cd art-workbook/ParameterSets`
- 7 `$ ls`
- 8 `CMakeLists.txt` `pset02.fcl` `PSet03_module.cc`
- 9 `pset01.fcl` `PSet02_module.cc` `pset04.fcl`
- 10 `PSet01_module.cc` `pset03.fcl` `PSet04_module.cc`
- 11 The source code for the first module you will run is `PSet01_module.cc` and
- 12 the FHiCL file to run it is `pset01.fcl`. The file `CMakeLists.txt` is identical
- 13 that used by the previous two exercises. The remaining files are the source and
- 14 FHiCL files for additional steps in this exercise.
- 15 In your build window, make sure that you are in your build directory. At this
- 16 time you do not need to build any code because all code for the Workbook was
- 17 built the first time that you ran `buildtool`.

18 12.5 The File `pset01.fcl`

- 19 The FHiCL file that you will run in this exercise is `pset01.fcl`. In your source
- 20 window, look at this file. You will see that `pset01.fcl` contains the definition
- 21 of a module label named `psetTester`; Listing 12.1 shows the definition of this
- 22 parameter set.
- 23 This parameter set tells *art* to run the module named `PSet01`. The parameter
- 24 set then supplies some additional parameter definitions that will be read by
- 25 that module. The parameter names and values have no meaning; they are

1 just name-value pairs used to illustrate the interaction between FHiCL and a
 2 module.

3 12.6 Running the Exercise

4 In your build directory, run the following command

```
5 $ art -c fcl/ParameterSets/pset01.fcl >& output/pset01.log
```

6 The expected output from this command is shown in Listing 12.2; for clarity,
 7 the printout made by *art* has been elided.

8 You will see in Section 12.7 that the module reads in the parameter set and then
 9 prints out each of the values in several different ways. Check that the printout
 10 matches the definitions of the parameters from `pset01.fcl`.

11 12.7 The Source code file `PSet01_module.cc`

12 The source code for this exercise is found in the file `PSet01_module.cc`. The
 13 new features seen in this exercise are all in the definition of the constructor,
 14 which is shown in Listing 12.3.

15 The first change is that the constructor now uses its argument; therefore the
 16 argument must have a name, which, in this case, is `pset`. You can choose any
 17 name that you want for this argument.

18 The type of the argument is `fhiCL::ParameterSet const&`. The class
 19 `ParameterSet` is in the namespace `fhiCL`; all identifiers in the namespace
 20 `fhiCL` are found in the UPS package named **fhiCLcpp**.¹ All header files from
 21 that package are found in the directory heirarchy rooted at `$FHICL_CPP_INC`.
 22 You can look at the header file for `ParameterSet` by typing:

```
23 $ less $FHICL_CPP_INC/fhiCLcpp/ParameterSet.h
```

24 Or you can use the code browser described in near the end of Section 7.6.3. For
 25 a review of UPS products, see Chapter 7.

26 The argument `pset` is passed by `const` reference. So you may not modify it,
 27 only read it. And it is efficient to pass as an argument because it is passed by
 28 reference, not value.

¹ In many, but not all, cases the name of the namespace is the same as that of the UPS product; this is one of the exceptions. There are, or will soon be, bindings of the FHiCL parser for several different languages; hence the package name is **fhiCLcpp**, where the **cpp** denotes that this UPS product is the C++ binding. The authors of **fhiCLcpp** decided that the full product name was too long to be used as the name of a namespace that must be typed frequently; so they chose a shorter name for the namespace, `fhiCL`.

Listing 12.2: The output from PSet01 when run using pset01.fcl

```
1  -----
2  Part 1:
3  a : this is quoted string
4  b : 42
5  c : 3.14159
6  d : 1
7  e : 1 2 3
8  f.a : 4
9  f.b : 5
10 module_type: PSet01
11 module_label: psetTester
12
13  -----
14  Part 2:
15  f as string:          a:4 b:5
16  f as indented-string:
17  a: 4
18  b: 5
19
20
21  -----
22  Part 3:
23  pset:
24  a: "this is quoted string"
25  b: 42
26  c: 3.14159
27  d: true
28  e: [ 1
29      , 2
30      , 3
31      ]
32  f: { a: 4
33      b: 5
34      }
35 module_label: "psetTester"
36 module_type: "PSet01"
```

Listing 12.3: The Constructor of PSet01_module.cc

```

1
2 tex::PSet01::PSet01(fhicl::ParameterSet const& pset ){
3
4     std::string          a = pset.get<std::string>("a");
5     int                  b = pset.get<int>    ("b");
6     double               c = pset.get<double>("c");
7     bool                 d = pset.get<bool>   ("d");
8     std::vector<int>      e = pset.get<std::vector<int>>("e");
9     fhicl::ParameterSet f = pset.get<fhicl::ParameterSet>("f");
10
11     int                  fa = f.get<int>("a");
12     int                  fb = f.get<int>("b");
13
14     std::string module_type = pset.get<std::string>("module_type");
15     std::string module_label = pset.get<std::string>("module_label");
16
17     std::cout << "\n-----\nPart_1:\n";
18     std::cout << "a_:_" << a << std::endl;
19     std::cout << "b_:_" << b << std::endl;
20     std::cout << "c_:_" << c << std::endl;
21     std::cout << "d_:_" << d << std::endl;
22
23     std::cout << "e_:";
24     for ( int i: e ){
25         std::cout << "_" << i;
26     }
27     std::cout << std::endl;
28
29     std::cout << "f.a_:_" << fa << std::endl;
30     std::cout << "f.b_:_" << fb << std::endl;
31
32     std::cout << "module_type:_" << module_type << std::endl;
33     std::cout << "module_label:_" << module_label << std::endl;
34
35     std::cout << "\n-----\nPart_2:\n";
36     std::cout << "f_as_string:_" << f.to_string() << std::endl;
37     std::cout << "f_as_indented-string:\n" << f.to_indented_string() << std::endl;
38
39     std::cout << "\n-----\nPart_3:\n";
40     std::cout << "pset:\n" << pset.to_indented_string() << std::endl;
41
42 }

```

1 The variable `pset` is a C++ representation of the information in the parameter
2 set `psetTester` from the file `pset01.fcl`, plus one additional definition.
3 After copying `psetTester` into `pset`, *art* adds a parameter with the name
4 `module_label`; the value of this parameter is set to the module label, in this
5 case `psetTester`. This extra step is not done for every parameter set defined
6 within a FHiCL file; it is only done for parameter sets that are used to configure
7 modules.

8 When you and I look at Listing 12.1 we can infer that the parameter `a` has a
9 value that is a string of text, parameter `b` has a value that is an integer number,
10 parameter `c` has a value that is a floating point number, parameter `d` has a
11 value that is one of the two possible boolean values, parameter `e` has a value
12 that is an array of integers and that parameter `f` has a value that is another
13 parameter set. However the object `pset` is not aware of any of this: internally it
14 represents the value of each parameter as a string. The computer-science-speak
15 for this is that FHiCL is a type-free language.

16 Therefore, when your code asks `pset` to return the value of a parameter, it
17 must tell `pset` two things:

- 18 1. The name of the parameter
- 19 2. The type of the variable that will receive the returned value.

20 The syntax used to specify the return type uses a feature of C++ called *tem-*
21 *plates*(γ), which will be discussed below.

22 If you ask that the value of a parameter be returned as a string, `pset` will
23 simply return a copy of its internal representation of that parameter.

24 On the other hand, if you ask that the value of a parameter be returned as
25 any other type, then `pset` needs to do some additional work. For example,
26 if you ask that a parameter be returned as an `int`, then `pset` must first find
27 its internal string representation of that parameter; it must then convert that
28 string into a temporary variable of type `int` and return the temporary variable.
29 Similarly for other types.

30 Look at the line 4 of Listing 12.3:

```
31 std::string a = pset.get<std::string>("a");
```

32 The angle brackets, `<>`, are the signature of a C++ template. *art* and FHiCL
33 use templates in several prominent places.

34 As mentioned above, to get the value of a parameter from `pset`, you must
35 specify both the name of the parameter and the return type. The name of the
36 parameter is specified as a familiar function argument while the return type is
37 specified between the angle brackets. The name between the angle brackets is
38 called a template argument. If you do not supply a template argument, then
39 your code will not compile.

1 Line 4 does the following: it declares a local variable named `a` that is of type
2 `std::string` and then does the following:

- 3 1. It asks `pset` if it has a parameter named `a`.
- 4 2. If it does have this parameter, `pset` will return it as a string.
- 5 3. The returned value is used to initialize the local variable named `a`.

6 Section 12.9 will describe what happens if `pset` does not have a parameter
7 named `a`.

8 It is not required that the local variable, `a`, have the same name as the FHiCL
9 parameter `a`. But, with rare exceptions, it is a good practice to make them
10 either exactly the same or, at a minimum, obviously related.

11 Look at the line 5 of Listing 12.3:

```
12 int b = pset.get<int>("b");
```

13 This line is similar to line 4; the main difference is that `pset` will convert the
14 string to an `int` before returning it. `pset` knows that it must perform the
15 conversion to `int` because the template argument tells it to.

16 It is beyond the scope of this chapter to discuss how the template mechanism
17 is used to trigger automatic type conversions. It is sufficient to always re-
18 member the following: when you use the `get` member function of the class
19 `fhiCL::ParameterSet`, the template argument must always match the type
20 of the variable on the left hand side.

21 The authors of FHiCL could have designed a different interface, such as

```
22 std::string a = pset.get_as_string("a");  
23 std::string b = pset.get_as_int    ("b");
```

24 Instead they chose to write it using templates. The reason for this choice is that
25 it allows one to add new types to FHiCL without needing to recompile FHiCL.
26 How you do this is beyond the scope of this chapter. You now know everything
27 that you need to know about templates in order to use `fhiCL::ParameterSet`
28 effectively.

29 Lines 6 through 15 of Listing 12.3 extract the remaining parameters from `pset`
30 and make copies of them in local variables. Lines 17 through 33 print the
31 values of these parameters; these make the printout in lines 1 through 11 of
32 Listing 12.2.

33 Your code ask for the values of parameters from a `ParameterSet` in any order;
34 your code may ask for a parameter multiple times or not at all.

35 Lines 29 and 30 of Listing 12.3 show one way to print the two values, `a` and `b`
36 from the parameter set `f`. Lines 35 through 37 show two other ways, using the
37 `to_string()` and `to_indented_string()` member functions of the class

1 `fhicl::ParameterSet`. These lines make the printout found in lines 13
2 through 18 of Listing 12.2.

3 Lines 39 and 40 of Listing 12.3 use the `to_indented_string()` member func-
4 tion to print everything found in the parameter set `psetTester`. These lines
5 make the printout found in lines 21 through 36 of Listing 12.2.

6 One final comment on `PSet01_module.cc`: the `analyze` member function is
7 empty. Never-the-less it must be present because *art* requires all analyzer mod-
8 ules to provide a member function named `analyze`. If we removed this member
9 function from the class `PSet01`, then the module would not compile.

10 12.8 A Comment About Templates

11 Now that you have seen templates, we can introduce some more language that
12 you will encounter and should know. In the above examples, `get<std::string>`
13 and `get<int>` are *member functions* of the class `ParameterSet`.

14 On the other hand, `get`, on its own, is called a *member function template*; this
15 means that `get` is a set of rules to write a member function but the member
16 function can only be written once the template argument has been specified. In
17 the future, when we refer to `get`, we will call it by its proper name:

18 `ParameterSet::get<T>`

19 or, sometimes, just `get<T>`. In the notation `<T>`, the angle brackets indicate
20 that `get` is a template and the capital letter `T` is a dummy argument that
21 indicates that if you want to use the template, you must supply one template
22 argument. The choice of the letter `T` as the name of the dummy argument is
23 mnemonic for *Type*, indicating that the template argument must be the name
24 of a type.

25 12.9 Exceptions

26 There are two sorts of error conditions that may occur when reading parameters
27 from a parameter set:

- 28 1. The requested parameter is not present in the parameter set.
- 29 2. The requested parameter is present but cannot be converted into the re-
30 quested type.

31 To give an example of the second sort, suppose that on line 6 of Listing 12.1
32 someone changed the definition of the parameter `c` from `3.14159` to “test”.
33 Now consider what happens when line 6 of Listing 12.3 is executed: the code

1 will correctly find that parameter `c` exists but it will produce an error when it
 2 tries to convert the string “test” to a `double`.

3 What happens when one of these errors occurs?

4 The C++ language has a feature called *exceptions*(γ). As for templates, this is
 5 an advanced feature that you do not need to understand in detail but you do
 6 need to be aware of it in broad strokes. In this case, exceptions are used behind
 7 the scenes and you will not see any code that explicitly uses exceptions.

8 When an error of this sort occurs, *art*’s default response is to abort the pro-
 9 cessing of your module and to return control to *art*, which will then attempt an
 10 orderly shutdown. By orderly shutdown we mean the following:

- 11 1. Record that your module had an error condition that stopped processing;
 12 this information will be written to all output event-data files.
- 13 2. Write a message to the log file that describes what happened.
- 14 3. Call the `endSubRun` member function of every module.
- 15 4. Call the `endRun` member function of every module.
- 16 5. Call the `endJob` member function of every module.
- 17 6. Properly flush and close all output and log files.
- 18 7. Perform a few other clean up and shutdown actions for parts of *art* that
 19 have not yet been discussed.
- 20 8. Return a non-zero status code to the parent process; the status code is
 21 the number that appears on the last line of your *art* output, the line that
 22 begins “Art has completed ...”.

23 For most sorts of errors, the orderly shutdown will be successful and your work
 24 up to the exception will be preserved.

25 But there are circumstances for which the orderly shutdown will fail. One
 26 example of this is if you have reached your disk quota and there is no disk space
 27 to hold more output.



28 *art* provides a mechanism that allows the user to specify, at run time, alternate
 29 behaviour. For example, there may be some errors for which you wish to write
 30 the offending event to a separate output file and to continue with the processing
 31 of the next event. For some other errors you may wish that *art* continue the
 32 current event and proceed to the next module. If you do not specify an alternate
 33 behaviour, *art* will perform its default action, which is to attempt an orderly
 34 shutdown. A detailed discussion of these features is beyond the scope of this
 35 chapter. .

1 12.9.1 Suggested Exercises

2 In `pset01.fcl`, remove the definition of the parameter `b`. Rerun *art*. You
 3 should see an error message like that shown in Listing 12.4. Read the error
 4 message and understand what it is telling you so that you will recognize the
 5 error message if you make this mistake yourself.

Listing 12.4: The expected output when the parameter `b` is removed from
`pset01.fcl`

```
1b %MSG-s ArtException: PSet01:psetTester@Construction 14-Jul-2013 19:38:19 CDT
7 ModuleConstruction
2 cet::exception caught in art
3 ---- Can't find key BEGIN
14 b
5 ---- Can't find key END
6 %MSG
17 Art has completed and will exit with status 8001.
```

14 In `pset01.fcl`, restore the definition of `b` and change the definition of `c` to
 15 "test" . Rerun *art*. You should see an error message like that shown in List-
 16 ing 12.5. Read the error message and understand what it is telling you so that
 17 you will recognize the error message if you make this mistake yourself.

Listing 12.5: The expected output when the parameter `c` misdefined in
`pset01.fcl`

```
1b %MSG-s ArtException: PSet01:psetTester@Construction 14-Jul-2013 19:42:54 CDT
19 ModuleConstruction
2 cet::exception caught in art
3 ---- Type mismatch BEGIN
4 c
5 ---- Type mismatch BEGIN
6 error in float string:
7 test
8 at or before:
9 ---- Type mismatch END
10 ---- Type mismatch END
11 %MSG
12 Art has completed and will exit with status 8001.
```

31 12.10 Parameters and Data Members

32 Very often information from the parameter set is needed in a member function of
 33 the module class. The way to propagate this information from parameter set to
 34 the member function is to store the values of these parameters as data members
 35 of the module class. This is illustrated in the two files `PSet02_module.cc`
 36 and `pset02.fcl`.

- 1 The source code for this example should be self explanatory. If you are not
 2 already familiar with data members or with the colon initializer syntax, see
 3 Section 6.6.5.
- 4 To run this example,
- 5 `$ art -c fcl/ParameterSets/pset02.fcl >& output/pset02.log`
- 6 The expected output from this is given in Listing 12.6.

Listing 12.6: The output from PSet02 when run using pset02.fcl

```
1 Event number: run: 1 subRun: 0 event: 1 b: 42 c: 3.14159 f: a:4 b:5
2 Event number: run: 1 subRun: 0 event: 2 b: 42 c: 3.14159 f: a:4 b:5
3 Event number: run: 1 subRun: 0 event: 3 b: 42 c: 3.14159 f: a:4 b:5
```



- 10 You should only do this for parameters that are actually used in one or more
 11 of the member functions. If a parameter is used only inside the constructor, do
 12 not store it as a data member; instead you should store it as a local variable of
 13 the constructor. This is an example of a general principal of good programming
 14 hygiene: always declare a variable in the narrowest scope that works.

15 12.11 Optional Parameters with Default Values

- 16 It is sometimes convenient to provide a default value for a parameter and, in
 17 FHiCL, default values may be provided in the source code that reads the pa-
 18 rameter set. This mechanism is illustrated by the files PSet03_module.cc
 19 and pset03.fcl.

- 20 You have already seen that the member function template `ParameterSet::get<T>`
 21 takes one function argument, the name of the parameter. For example,

```
22 int b = pset.get<int>("b");
```

- 23 It also takes an optional, second, function argument, a default value for the
 24 parameter. For example,

```
25 int b = pset.get<int>("b", 0);
```

- 26 If the second argument is present, there two cases:

- 27 1. If the parameter is not defined in pset, then the second argument is
 28 returned as the value of the call to `get`.
- 29 2. If the parameter is defined in pset, then the second argument is ignored
 30 and the value read from the FHiCL file is returned as the value of the call
 31 to `get`.

- 32 When reading the code in this example you will encounter the expression:

```
33 std::vector<double>(5, 1.0)
```

Listing 12.7: The output from PSet03 when run using `pset03.fcl`

```
1 debug level: 0
2 g:  1 1 1 1 1
```

1 This tells the compiler to instantiate an object of type `std::vector<double>`,
2 set its size to 5 and initialize elements 0 through 4 to have the value 1.0. If you
3 are not familiar with this syntax, you can read about it in the STL documen-
4 tation (see Section 6.7).

5 This expression appears as the second argument of a call to `pset.get`. There-
6 fore the compiler will create an unnamed temporary object (the vector of dou-
7 bles) and pass that object as the second argument; once the function call has
8 completed, the temporary object will be deleted.

9 With the above explanations, the source code for this example should be rea-
10 sonably self explanatory; it looks for two parameters named `debugLevel` and
11 `g` and supplies default values for each of them. Look at the file `pset03.fcl`;
12 you will see that the parameters `debugLevel` and `g` are not present in the
13 file.

14 To run this example,

```
15 $ art -c fcl/ParameterSets/pset03.fcl >& output/pset03.log
```

16 The expected output from this is given in Listing 12.7.

17 12.11.1 Suggested Exercises

18 Edit `pset03.fcl` and, in the parameter set `testPSet`, provide definitions for the
19 parameters `debugLevel` and `g`. Make their values different from the default
20 values. Rerun *art* and verify that the module has correctly read in and printed
21 out the values you defined.

22 12.11.2 Policies About Optional Parameters

23 Allowing optional parameters is important for developing, debugging and test-
24 ing; if it is required that all parameters be supplied all of the time the complete
25 list of parameters can become unwieldy. On the other hand, the use of optional
26 parameters can make it difficult to audit the physics content of a job. Therefore
27 many experiments have policies for what sort of parameters may have defaults
28 and what sort of parameters may not have defaults. Frequently the policy is
29 that parameters that define the physics behaviour must not have default values,
30 while other parameters may have default values.

31 Consult your experiment to learn what policies you should follow.

1 FHiCL has several features that make it easier to deal with large parameter
2 sets. Your experiment may define a standard definition for a parameter set; your
3 FHiCL file may start with that standard definition and modify a few values. The
4 syntax for doing this will be explained in a future chapter.

5 12.12 Numerical Types, Precision and Canonical 6 Forms

7 FHiCL recognizes numbers in both fixed point and exponential notation, for
8 example `123.4` and `1.234e2`; the letter `e` that separates the exponent can be
9 written in either upper or lower case.

10 In the preceding exercises you defined some numerical values in a FHiCL file,
11 read them into your code and printed them out; the printed values exactly
12 matched the input values. The values used in those exercises were carefully
13 chosen to avoid a few surprises: there are cases in which the printed value with
14 be an equivalent, but not identical, form. This section discusses those cases and
15 provides some examples.

16 When FHiCL recognizes that a parameter value is a number it converts the
17 number into a *canonical form* and stores the canonical form as a string. The
18 transformation to the canonical form preserves the full precision of the number
19 and involves the following steps:

- 20 1. The canonical form has no insignificant characters:
 - 21 (a) no insignificant trailing zeros
 - 22 (b) no insignificant trailing decimal point
 - 23 (c) no insignificant leading plus sign
 - 24 (d) no insignificant leading plus sign in the exponent
- 25 2. If a number is specified in exponential notation and if the number can
26 be represented as a integer, without loss of precision, and if the resulting
27 integer has 6 or fewer digits, then the canonical form is the integer. For
28 example, the canonical form of `1.23456E5` is `123456` but the canonical
29 form of `1.23456E6` is `1.23456e6`.
- 30 3. The canonical form of all other floating point numbers is exponential no-
31 tation with a single, non-zero digit to the left of the decimal point.
- 32 4. The canonical form of all strings, includes beginning and ending quotes;
33 this is true even if the string contains no embedded whitespace or other
34 special characters.

35 Some examples of numbers and their canonical forms are given in Table 12.1.

Table 12.1: caption

Number	Canonical Form	Number	Canonoical Form
2	2	1.234E2	1.234e2
2.	2	1.23456E5	123456
2.0	2	1.23456E6	1.23456e6
2.1E2	210	1234567	1.234567e6
+210	210	0.01	1E-2

1 If a numerical value, when expressed as a fixed point number, has no fractional
 2 part, your code may ask for the parameter to be returned as either a floating
 3 point type (such as `double` or `float`) or as an integral type (such as `int`,
 4 `short unsigned` or `std::size_t`). For example, line 6 of Listing 12.3 could
 5 have been written:

```
6 double b = pset.get<double>("b");
```

7 This code will do the expected thing: in the example `pset01.fcl`, it will read
 8 the value 42 into a variable of type `double`.

9 On the other hand, if a numerical value, when expressed as a fixed point num-
 10 ber, does have a fractional part, you may only ask for the parameter to be
 11 returned as a floating point type. If you ask for such a value as an `int`, the
 12 `ParameterSet::get<int>` member function will throw an exception; sim-
 13 ilarly for all other integral types. This behaviour may not be intuitive: the
 14 authors of *art* could have decided, instead, to discard the fractional part and
 15 return the integer part. They chose not to do this on the the following grounds:
 16 when this situations occurs, it is almost always an error.

17 12.12.1 Suggested Exercises

18 The above ideas are illustrated by the files `PSet04.module.cc` and `pset04.fcl`.
 19 To run this example,

```
20 $ art -c fcl/ParameterSets/pset04.fcl >& output/pset04.log
```

21 The expected output from this is given in Listing 12.8. Read the source code
 22 and the FHiCL file; then understand the output. The first three lines show three
 23 different printed formats of the parameter `a`, with the first being the canonical
 24 form. While all forms are equal to the number found in the FHiCL file, none
 25 have the same format. Understand why each line has the format it does.

26 Line four shows the canonical form of the parameter `b`. Line five shows what
 27 is printed using the default C++ settings; the two least significant characters
 28 were dropped. The code that produces line six shows how the use the STL
 29 precision function to tell C++ to print more significant figures.

Listing 12.8: The output from PSet04 when run using `pset04.fcl`

```

1 parameter a as a string: 1.23456e6
2 parameter a as a double: 1.23456e+06
3 parameter a as int:      1234560
4 parameter b as a string: 3.1415926
5 parameter b as a double: 3.14159
6 parameter b as a double with more significant figures: 3.1415926
7 parameter c as a string: 1
8 parameter c as an int:   1

```

Listing 12.9: The output from PSet04 when run using the modified version of `pset04.fcl`, which has an intentional error

```

1 %MSG-s ArtException: PSet04:pset@Construction 28-Jul-2013 23:39:31 CDT
  ModuleConstruction
2 cet::exception caught in art
3 ---- Type mismatch BEGIN
4   c
5   narrowing conversion
6 ---- Type mismatch END
7 %MSG
8 Art has completed and will exit with status 8001.

```

1 If you modify the precision `cout`, it will change the format of the printout
 2 for the rest of the run of the program; usually this is a bad thing. To avoid
 3 this, `PSet04_module.cc` illustrates how to save and restore the precision of
 4 `cout`.

5 Line seven shows the canonical form of the parameter `c` and line eight shows
 6 the default C++ printed form of the integer.

7 For the next exercise, edit `pset04.fcl` and change the value of `c` to something
 8 with a fractional part. Rerun `art`; you should see that it throws an exception
 9 because it is illegal to read a numeric value with a fractional part into a variable
 10 of integral type. The error message from `art` is shown in Listing 12.9. Read the
 11 error message and understand what it is telling you so that you will recognize
 12 the error message if you make this mistake yourself.

13 12.13 Review

14 After working through the exercises in this section, you should know:

- 15 1. How to read parameter values from a FHiCL file into a module.
- 16 2. How to use data members to communicate information from the construc-
 17 tor to other member functions of a module.
- 18 3. How to use the colon initializer syntax.

Part

- 1 4. How to provide a default value for a parameter.
- 2 5. That you should learn your experiment's policy about what sorts of pa-
- 3 rameters are allowed to have default values.
- 4 6. What are the canonical forms of parameters.
- 5 7. How to modify the precision of the printout of floating point types.
- 6 8. How to recognize the error messages for a missing parameter or for a value
- 7 that cannot be converted to the requested type.

13 Exercise 5: Making Multiple Instances of a Module

13.1 Introduction

In a typical HEP experiment is often necessary to repeat one analysis several times, with each version differing only in the values of some cuts; this is frequently done to tune cuts or to study systematic errors. Very often it is both convenient and efficient to run all of the variants of the analysis in a single job.

A powerful feature of *art* is that it permits you to run an *art* job in which you define and run many instances of the same module; when you do this, each instance of the module gets its own parameter set. In this chapter you will learn how to use this feature of *art*.

13.2 Prerequisites

The prerequisite for this chapter is all of the material in Part I (Introduction) and the material in Part II (Workbook) up to and including Chapter 12, but excluding Chapter 11.

13.3 What You Will Learn

In this chapter you will learn how to run an *art* job in which you run the same module more than once. This exercise will make it clear why *art* needs to distinguish the two ideas of *module label* and *module.type*.

1 13.4 Setting up to Run this Exercise

2 To run this exercise, you need to be logged in to the computer on which you ran
 3 Exercise 2 (in Chapter 10). If you are continuing on from a previous exercise,
 4 you need to keep both your source and build windows open.

5 If you are logging back in, follow the instructions in Section 10.6 to reestablish
 6 your source and build windows.

7 In your source window, `cd` to your source directory. Then `cd` to the directory
 8 for this exercise and look at its contents

```
9 $ cd art-workbook/ModuleInstances
10 $ ls
11 CMakeLists.txt  magic.fcl  MagicNumber_module.cc
```

12 The source code for the first module you will run is `MagicNumber_module.cc`
 13 and the FHiCL file to run it is `magic.fcl`. The file `CMakeLists.txt` is
 14 identical that used by the previous two exercises.

15 In your build window, make sure that you are in your build directory. At this
 16 time you do not need to build any code because all code for the Workbook was
 17 built the first time that you ran `buildtool`.

18 13.5 The Source File `Magic_module.cc`

19 The source code for this exercise is found in the file `Magic_module.cc`. Look
 20 at this file and you should see the following features, all of which you have seen
 21 before.

- 22 1. The file declares and defines a class named `MagicNumber` that follows
 23 the rules to be an *art* analyzer module.
- 24 2. The class has a constructor and an `analyze` method.
- 25 3. The class has a data member named `magicNumber_`, of type `int`.
- 26 4. The class initializes `magicNumber_` by reading a value from its parameter
 27 `set`; the name of the parameter is `magicNumber` (without the underscore).
- 28 5. The parameter `magicNumber` is a required parameter.
- 29 6. Both the constructor and the `analyze` method print an informational
 30 message that includes the value of `magicNumber_`.

1 13.6 The FHiCL File `magic.fcl`

2 The FHiCL file used to run this exercise is `magic.fcl`. Look at this file and
3 you should see the following features:

- 4 1. Compared to previous exercises, The FHiCL names `process_name`, `source`
5 and `services` have no important differences.
- 6 2. In the `analyzers` parameter set, inside the `physics` parameter set, you
7 will see the definition of four module labels, `boomboom`, `rocket`, `flower`
8 and `bigbird`¹. The value of each definition is a parameter set.
- 9 3. The first three of these parameter sets tell *art* to run the module `MagicNumber`
10 and each provides a value for the required `magicNumber` parameter²
- 11 4. The last parameter set tells *art* to run the module `First`, the source
12 for which was discussed in Chapter 10; this module does not need any
13 additional parameters.
- 14 5. The path `e1` contains the names of all of the module labels from the
15 `analyzers` parameter set.

16 13.7 Running the Exercise

17 In your build directory, run the following command

```
18 $ art -c fcl/ModuleInstances/magic.fcl >& output/magic.log
```

19 The expected output from this command is shown in Listing 13.1; for clarity, the
20 printout made by *art* has been elided. Compare this printout to the printout
21 from your run; it should be exactly the same. Inspect the printout and the
22 files `MagicNumber_module.cc` and `../FirstModule/First_module.cc`;
23 understand why the printout is what it is.

24 13.8 Discussion

25 13.8.1 Order of Analyzer Modules is not Important

26 As it happens, *art* runs the four analyzer modules in the order specified in the
27 path definition `e1`. But you must not count on this behaviour! Two of the
28 design rules of *art* are:



¹ All of these are nicknames of ice hockey players who played for the Montreal Canadiens ice hockey team; all of them have had their sweater number retired

²In each case the magic number is the sweater number of the hockey player whose nickname is the module label.

Listing 13.1: The output from running `magic.fcl`

```

1 MagicNumber::constructor: magic number: 9
2 MagicNumber::constructor: magic number: 5
3 Hello from First::constructor.
4 MagicNumber::constructor: magic number: 10
5 MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 9
6 MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 5
7 Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
8 MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 10
9 MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 9
10 MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 5
11 Hello from First::analyze. Event id: run: 1 subRun: 0 event: 2
12 MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 10
13 MagicNumber::analyze: event: run: 1 subRun: 0 event: 3 magic number: 9
14 MagicNumber::analyze: event: run: 1 subRun: 0 event: 3 magic number: 5
15 Hello from First::analyze. Event id: run: 1 subRun: 0 event: 3

```

- 1 1. Modules may only communicate with each other by putting information
2 into, and reading information from, the `art::Event`.
- 3 2. Analyzer modules may not put information into the `art::Event`.
- 4 Therefore *art* is free to run analyzer modules in any order.
- 5 For producer modules, which may add information to the event, the order of ex-
6 ecution is often very important. When you reach the exercises that run producer
7 modules, you will be told how to specify the order of execution.
- 8 You may wish to review some of the other ideas about *art* paths that are de-
9 scribed in Section 9.7.8.

10 13.8.2 Two Meanings of *Module Label*

11 In the preceeding discussion, the name *module label* was used in two subtly
12 different ways, as is illustrated by the module label `rocket`:

- 13 1. `rocket` identifies a parameter set that is used to configure an instance of
14 the module `MagicNumber`.
- 15 2. `rocket` is also used as the name of the module instance that is configured
16 using this parameter set; the elements in the path `e1` are all the names of
17 module instances.

18 Clearly these two meanings are very closely related, which is why the same name,
19 *module label*, is used for both ideas. Throughout the remainder of this document
20 suite the name *module label* will be used for both meanings; the authors believe
21 it will be clear from the context which meaning is intended. This is standard
22 usage within the *art* community.

1 13.9 Suggested Exercise

2 Edit `magic.fcl` and do the following:

- 3 1. Add a new analyzer module label that configures an instance of the module
4 Optional from Chapter 11.
- 5 2. Add the new module label to `e1`.

6 Then re-run `magic.fcl`. Do you see the expected additional printout?

7 13.10 Review

8 After working through this exercise, you should:

- 9 1. Know how to run multiple instances of the same module within one *art*
10 job.
- 11 2. Understand that *art* does not guarantee the order in which analyzer mod-
12 ules will be run.
- 13 3. Understand the two senses in which the name *module label* is used: as the
14 name of a parameter set and as the name of the corresponding instance
15 of a module.

14 Accessing Data Products

14.1 Prerequisites

14.2 What You Will Learn

14.3 Running the Exercise

14.4 Discussion

14.5 Suggested Activities

15 Making Histograms and TFileService

15.1 Prerequisites

15.2 What You Will Learn

15.3 Running the Exercise

15.4 Discussion

15.5 Suggested Activities

1 16 Looping Over Collections

2 16.1 Prerequisites

3 16.2 What You Will Learn

4 16.3 Running the Exercise

5 16.4 Discussion

6 16.5 Suggested Activities

17 The Geometry Service

17.1 Prerequisites

17.2 What You Will Learn

17.3 Running the Exercise

17.4 Discussion

17.5 Suggested Activities

18 The Particle Data Table

18.1 Prerequisites

18.2 What You Will Learn

18.3 Running the Exercise

18.4 Discussion

18.5 Suggested Activities

19 GenParticle: Properties of Generated Particles

19.1 Prerequisites

19.2 What You Will Learn

19.3 Running the Exercise

19.4 Discussion

19.5 Suggested Activities

1

Part III

2

Users Guide

20 Obtaining Credentials to Access Fermilab Computing Resources

To request your Fermilab computing account(s) and permissions to log into the your experiment's nodes, fill out the form Request for Fermilab Visitor ID and Computer Accounts. Typically, experimenters that are not Fermilab employees are considered *visitors*. You will be required to read the Fermilab Policy on Computing.

After you submit the form, an email from the Fermilab Service Desk should arrive within a week (usually more quickly), saying that your Visitor ID (an identifying number), Kerberos Principal and Services Account have been created. You will need to change the password for both Kerberos and Services.

20.1 Kerberos Authentication

Your Kerberos Principal is effectively a username for accessing nodes that run Kerberos in what's called the FNAL.GOV *realm* (all non-PC Fermilab machines).¹

To change your Kerberos password, first choose one (minimum 10 characters with mixture of upper/lower case letters and numbers and/or symbols such as !, , #, \$, %, &, *, %). From your local machine, log into the machine using `ssh` or `slogin` and run the `kpasswd` command. Respond to the prompts, as follows:

```
$ kpasswd <username>@FNAL.GOV
Password for username@FNAL.GOV: <--- type your current password here
New password: <--- type your new password here
New password (again): <--- type your new password here for confirmation
```

¹The FERMI.WIN.FNAL.GOV realm is available for PCs.

1
2 Kerberos password changed.
3 Your Kerberos password will remain valid for 400 days.

4 **20.2 Fermilab Services Account**

5 The Services Account enables you to access a number of important applica-
6 tions at Fermilab with a single username/password (distinct from your Kerberos
7 username/password). Applications available via the Services Account include
8 SharePoint, Redmine, Service Desk, VPN and others.

9 To get your initial Services Account password, a user must first contact the
10 Service Desk to get issued a first time default password. Once a default password
11 is issued, users can access <http://password-reset.fnal.gov/> to change it.

12 If you are not on-site or connected to the Fermi VPN, call the Service Desk
13 at 630-840-2345. You will be given a one-time password and a link to change
14 it.

1 21 Using git

2 The source code for the exercises in the *art* workbook is stored in a source
3 code management system called *git* and maintained in a repository managed by
4 Fermilab. Think of *git* as an enhanced *svn* or (a VERY enhanced) *cvs* system.
5 The repository is located at . You will be shown how to access it with *git*.

6 If you want some background on *git*, we suggest the Git Reference.

7 You will need to know how to install *git*, download the workbook exercise files
8 initially to your system and how to download updates. You will not be checking
9 in any code.

10 To install *git* on a Mac:

11 \$ `http://git-scm.com/download/mac`

12 This will automatically download a disk image. Open the disk image and click
13 on the .pkg file.

14 In your home directory, edit the file `.bash_profile` and add the line:

15 \$ `export PATH=/usr/local/git/bin:${PATH}`

16 \$ `git clone ssh://p-art-workbook@cdcvcs.fnal.gov/cvs/projects/art-workbook`

17 and how to download updates as the developers make them:

18 \$ `git pull`

22 *art* Run-time and Development Environments

22.1 The *art* Run-time Environment

Your *art* run-time environment consists of:

- your current working directory
- all of the directories that you can see and that contain relevant files, including system directories, project directories, product directories, and so on
- the files in the above directories
- the environment variables in your environment (not sure how to say this nicely)
- any aliases or shell functions that are defined

Figures 22.1, 22.2 and 22.3 show the elements of the run-time environment in various scenarios, and a general direction of information flow for job execution.

When you are running *art*, there are three environment variables that are particularly important:

- PATH
- LD_LIBRARY_PATH
- FHICL_FILE_PATH

They are colon-separated lists of directory names. When you type a command at the command prompt, or in a shell script, the (bash) shell splits the line using whitespace and the first element is taken as the name of a command. It looks in three places to figure out what you want it to do. In order of precedence:

1. it first looks at any aliases that are defined

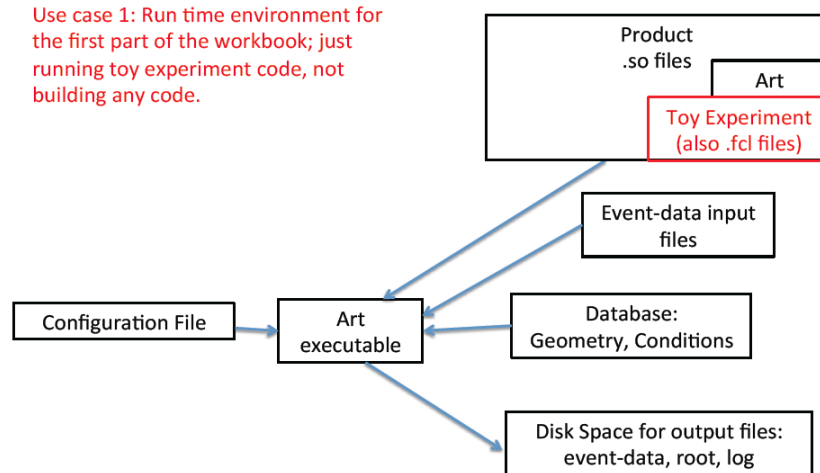


Figure 22.1: Elements of the *art* run-time environment, just for running the Toy Experiment code for the Workbook exercises

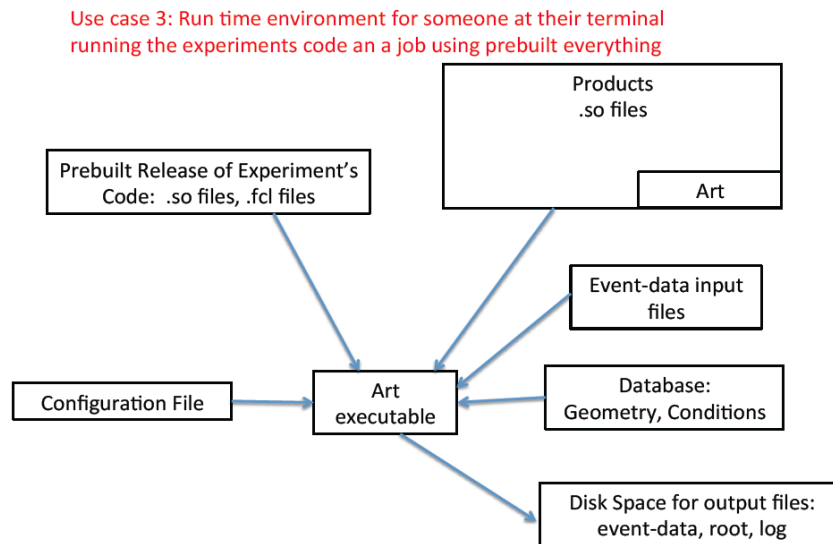


Figure 22.2: Elements of the *art* run-time environment for running an experiment's code (everything pre-built)

Use case 4: Run time environment for someone running a production job with officially tracked inputs.

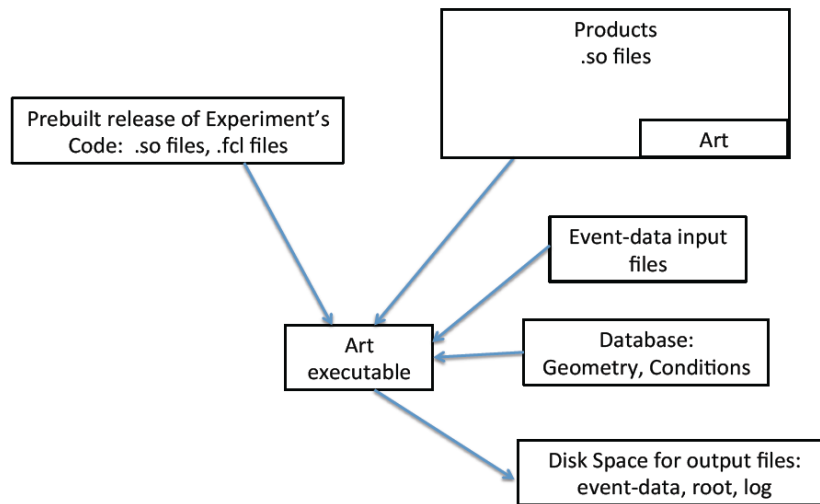


Figure 22.3: Elements of the *art* run-time environment for a production job with officially tracked inputs

- 1 2. secondly, it looks for shell keywords in your environment with the com-
- 2 mand name you provide
- 3 3. thirdly, it looks for shell functions in your environment with that name
- 4 4. then it looks for shell built-ins in your environment with that name
- 5 5. finally, it looks in the first directory defined in `PATH` and looks for a
- 6 file with that name; if it does not find a match, it continues with the
- 7 next directory, and so on, followed by the paths defined in the other two
- 8 variables.

9 Some parts of the run-time environment will be established at login time by your
 10 login scripts. This is highly site-dependent. We will describe what happens at
 11 Fermilab - consult your site experts to find out if anything is provided for you
 12 at your remote site.

13 When running the workbook, the interesting parts of your environment are
 14 established in two steps:

- 15 • source a site-specific setup script
- 16 • source a project-specific setup script

17 The Workbook, and the software suites for most IF experiments, are designed
 18 so that all site dependence is encoded in the site-specific setup script; that script

- 1 adds information to your environment so that the project-specific scripts can be
- 2 written to work properly on any site.

3 22.2 The *art* Development Environment

4 The development environment includes the run-time environment in Section 22.1
5 plus the following.

- 6 • the source code repository
- 7 • the build tools (these are the tools that know how to turn `.h` and `.cc`
8 files in to `.so` files)
- 9 • additional environment variables and `PATH` elements that simplify the use
10 of the above

11 Figures 22.4, 22.5 and 22.6 illustrate the development environment for various
12 scenarios.

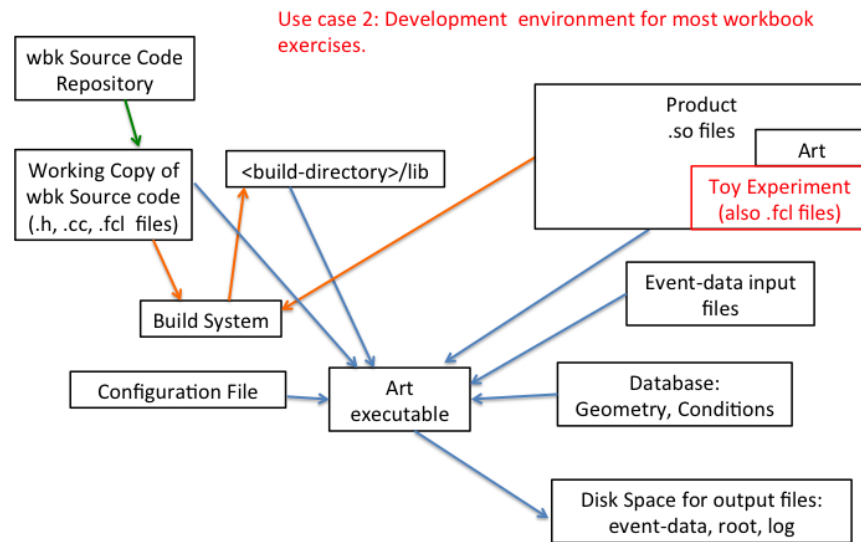


Figure 22.4: Elements of the *art* development environment as used in most of the Workbook exercises

13 In some experiments the run-time and development environments are identi-
14 cal.

15 It turns out that there is no perfect solution for the job that build tools do.
16 As a result, several different tools are widely used. Every tool has some pain
17 associated with it. You never get to avoid pain entirely but you do get to pick
18 where you will take your pain.

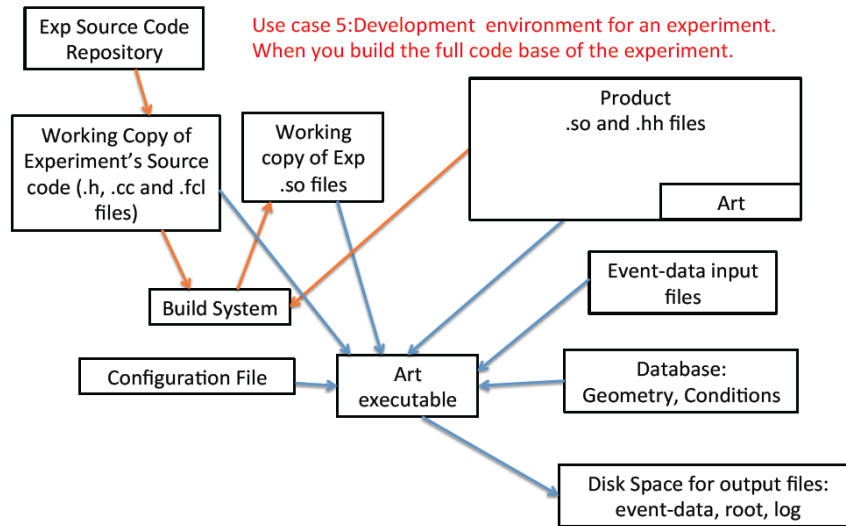


Figure 22.5: Elements of the *art* development environment for building the full code base of an experiment

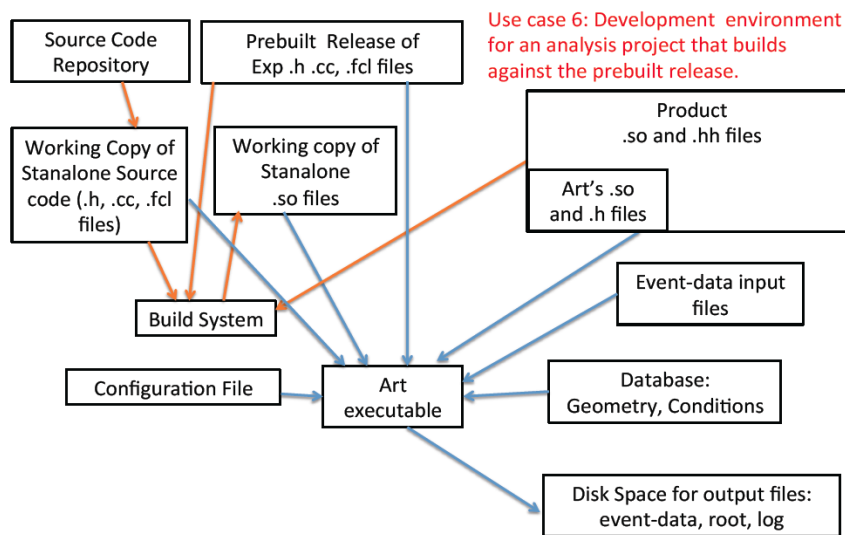


Figure 22.6: Elements of the *art* development environment for an analysis project that builds against prebuilt release

- 1 The workbook uses a build tool named **cetbuildtools**. Other projects have
- 2 chosen make, cmake, scons and Software Release Tools (SRT). Here is some-
- 3 thing to watch out for: “build tools” written as two words refers generically to
- 4 the above set of tools; but “buildtools” written as one word is the name of the
- 5 executable that runs the build for **cetbuildtools**.

23 art Framework Parameters

This chapter describes all the parameters currently understood by the *art* framework, including by framework-provided services and modules. The parameters are organized by category (module, service or miscellaneous), and preceded by a general introduction to the expected overall structure of an *art* FHiCL configuration document.

23.1 Parameter Types

The parameters are described in tables for each module. The type of a defined parameter may be:

- TABLE: A nested parameter set, e.g., `set: { par1: 3 }`
- SEQUENCE: A homogeneous sequence of items, e.g., `list: [ 1, 1, 2, 3, 5, 8 ]`
- STRING: A string (enclosing double quotes not required when the string matches `[A-Za-z_-][A-Za-z0-9_]*`). (Note: Special keywords when quoted are no longer keywords.) E.g.,

```
simpleString: g27
harderString: "a-1"
sneakysting1: "nil"
sneakysting2: "true"
sneakysting3: "false"
```

- COMPLEX: A complex number; e.g., `cnum: (3, 5)`
- NUMBER: A scalar (integer or floating point), e.g., `num: 2.79E-8`
- BOOL: A boolean, e.g.,

```
tbool: true
fbool: false
```


23.2 Structure of *art* Configuration Files

The expected structure of an *art* configuration file

Note, any parameter set is optional, although certain parameters or sets are expected to be in particular locations if defined.

```

1  # Prolog (as many as desired, but they must all be contiguous with only
2  # whitespace or comments inbetween.
3  BEGIN_PROLOG
4  pset:
5  {
6      nested_pset:
7      {
8          v1: [ a, b, "c-d" ]
9          b1: false
10         c1: 29
11     }
12 }
13 END_PROLOG
14
15 # Defaulted if missing: you should define it in most cases.
16 process_name: PNAME
17
18 # Descriptions of service and general configuration.
19 services:
20 {
21     # Parameter sets for known, built-in services here.
22     # ...
23
24     # Parameter sets for user-provided services here.
25     user:
26     {
27     }
28
29     # General configuration options here.
30     scheduler:
31     {
32     }
33 }
34
35 # Define what you actually want to do here.
36 physics:
37 {
38     # Parameter sets for modules inheriting from EDProducer.
39     producers:

```

Part

```
1  {
2    myProducer:
3    {
4      module_type: MyProducer
5      nested_pset: @local::pset.nested_pset
6    }
7  }
8
9  # Parameter sets for modules inheriting from EDFilter.
10 filters:
11 {
12   myFilter: { module_type: SomeFilter }
13 }
14
15 # Parameter sets for modules inheriting from EDAnalyzer.
16 analyzers:
17 {
18 }
19
20 # Define parameters which are lists of names of module sets for
21 # inclusion in end_paths and trigger_paths.
22
23 p1: [ myProdroducer, myFilter ]
24 e1: [ myAnalyzer, myOutput ]
25
26 # Compulsory for now: will be computed automatically in a future
27 # version of ART.
28
29 trigger_paths: [ p1 ]
30 end_paths: [ e1 ]
31 }
32
33 # The primary source of data: expects one and only one input source
34 parameter set.
35 source:
36 {
37 }
38
39 # Parameter sets for output modules should go here.
40 outputs:
41 {
42 }
43 }
```

23.3 Services

23.3.1 System Services

These services are always loaded regardless of whether a configuration is specified.

23.3.2 FloatingPointControl

These parameters control the behavior of floating point exceptions in different modules.

Table 23.1: *art* Floating Point Parameters

Enclosing Table Name	Parameter Name	Type	Default	Notes
services	floating_point_control	TABLE	{}	Top-level parameter set for the service
floating_point_control	setPrecisionDouble	BOOL	false	
	reportSettings moduleNames	BOOL SEQUENCE	false []	Each module name listed should also have its own parameter set within floating_point_control. One may also specify a module name of, "default" to provide default settings for the following items:
{module-name}	enableDivByZeroEx	BOOL	false	
	enableInvalidEx	BOOL	false	
	enableOverFlowEx	BOOL	false	
	enableUnderFlowEx	BOOL	false	

1 **23.3.3 Message Parameters**

2 These parameters configure the behavior of the message logger (this is a pseudo-
3 service – not accessible via ServiceHandle).

Table 23.2: *art* Message Parameters

Enclosing Ta- ble Name	Parameter Name	Type	Default	Notes
services	message	TABLE		Top-level pa- rameter set for the service
message				

3

4 **23.3.4 Optional Services**

5 These services are only loaded if a configuration is specified (although it may
6 be empty).

7 **23.3.5 Sources**

8 **23.3.6 Modules**

9 Output modules

24 Job Configuration in *art*: FHiCL

Run-time configuration for *art* is written in the Fermilab Hierarchical Configuration Language (FHiCL, pronounced “fickle”), a language that was developed at Fermilab to support run-time configuration for several projects, including *art*. For this reason, this chapter will need to discuss FHiCL both as a standalone language and as used by *art*.

By convention, the names of FHiCL files end in `.fcl`. Job execution is performed by running *art* on a FHiCL configuration file, which is specified via an argument for the `-c` option:

```
$ art -c run-time-configuration-file.fcl
```

See Figure ?? in Section 22.1 to see how the configuration file fits into the run-time environment.

The FHiCL concept of *sequence*, as listed in brackets `[]`, maps onto the C++ concept of `std::vector`, which is a sequence container representing an array that can change in size. Similarly, the FHiCL idea of *table*, as listed in curly brackets `{}`, maps onto the idea of `fhicl::ParameterSet`. . Note that `ParameterSet` is not part of *art*; it is part of a utility library *used* by *art*, FHiCL-CPP, which is the C++ toolkit used to read FHiCL documents within *art*. FHiCL files provide the parameter sets to the C++ code, specified via module labels and paths, that is to be executed.

24.1 Basics of FHiCL Syntax

24.1.1 Specifying Names and Values

A FHiCL file contains a collection of definitions of the form

```
name : value
```

where “name” is a parameter that is assigned the value “value.” Many types of values are possible, from simple atomic values (a number, string, etc., with no internal whitespace) to highly structured table-like values; a value may also

1 be a reference to a previously defined value. The white space on either side of
 2 the colon is optional. However, to include whitespace within a string, the string
 3 must be quoted (single or double quotes are equivalent in this case).

4 The fragment below will be used to illustrate some of the basics of FHiCL
 5 syntax:

```

6 # A comment.
7 // Also a comment.
8
9
10 name0 : 123           # A numeric value. Trailing comments
11                        # work, too.
12 _name0 : 123          # Names can begin with underscores
13
14 name00 : "A quoted comment prefix, # or //, is just part of a
15                        # quoted string, not a comment"
16
17 name1:456.            # Another numeric value; whitespace is
18                        # not important within a definition
19 name2 : -1.e-6
20 name3 : true          # A boolean value
21 NAME3 : false         # Other boolean value; names are case-
22                        # sensitive.
23 name4 : red           # Simple strings need not be quoted
24 name5 : "a quoted string"
25 name6 : 'another quoted string'
26
27 name7 : 1 name8 : 2   # Two definitions on a line, separated by
28                        # whitespace.
29 name9
30 :                     # Same as name9:3 ; newlines are just
31 3                     # whitespace, which is not important.
32
33
34 namea : [ abc, def, ghi, 123 ] # A sequence of atomic values.
35                                # FHiCL allows heterogeneous
36                                # sequences, which are not,
37                                # however, usable via the C++ API.
38
39 nameb :                 # A table of definitions; tables may nest.
40 {
41     name0: 456
42     name1: [7, 8, 9, 10 ]
43     name2:
44     {
45         name0: 789

```

Part

```

1      }
2  }
3
4  namec : [ name0:{ a:1 b:2 } name1:{ a:3 c:4 } ]
5          # A sequence of tables.
6
7  named : []          # An empty sequence
8  namee : {}          # An empty table
9
10 namef : nil          # An atomic value that is undefined.
11
12 abc : 1              # If a definition is repeated twice within
13 abc : 2              # the same scope, the second definition
14 def : [ 1, 2, 3 ]    # will win (e.g., "abc" will be 2 and
15 def : [ 4, 5, 6 ]    # "def" will be [4,5,6])
16 name : {
17     abc : 1
18     abc : 2
19 }
20
21 cont1:{x: 1.0 y: 2.0 z: 3.0} # Hierarchical (compound) names denote
22 cont1.x : 5                # levels of scope; here set x in cont1 to 5.
23 OR
24 cont2:[1, 2, 3]
25 cont2[0] : 1              # Here, redefine the first (atomic) value
26                            # for cont2, assign it the value 1. I.e., here,
27                            # no action. Indices of PHiCL sequences
28                            # begin with 0. \fixme{right?}
29
30 name0:{ a:1 b:2 }
31 x : @local::name0.a      # Using reference notation "@local," this assigns
32                            # to x the value of a in table name0, in the
33                            # line above, this value is 1.

```

24.1.2 FHiCL-reserved Characters and Keywords

Several keywords, symbols and strings are *reserved to* FHiCL. What does this mean? Whenever FHiCL encounters a *reserved* string, FHiCL will interpret it according to the *reserved* meaning. Nothing prevents you from using these reserved strings in a name or value, but if you do, it is likely to confuse FHiCL. FHiCL may produce an error or warning, or it may silently do something different than what you intended. Bottom line: don't use reserved strings or symbols in the FHiCL environment for other than their intended uses.

The following characters, including the two-character sequence ::, are reserved

1 to FHiCL:

2 `, : :: @ [ ] { } ( )`

3 The following keywords have special meaning to FHiCL. They can be used
4 as parameter values to pass to classes, e.g., to initialize a variable within a
5 program, but their uses will not be fully described here because of subtleties
6 and variations. As you work with C++ and FHiCL, the way to use them will
7 become clearer.

8 **true, false** These values convert to a boolean

9 **nil** This value is associated with no data type. E.g. if `a : nil`, then `a` can't
10 be converted to any data type, and it must be redefined before use

11 **infinity, +infinity, -infinity** These values initialize a variable to positive (the
12 first two) or negative (the third) infinity

13 **BEGIN_PROLOG, END_PROLOG** ()

14 The first six keywords (three lines) above are only keywords when entered as
15 lower case and unquoted; the last two keywords (the last line) are only keywords
16 when they are in upper case, unquoted and at the start of a line. Otherwise
17 these are just strings. You may include any of the above reserved characters
18 and keywords in a “quoted” string to prevent them from being recognized as
19 keywords.

20 24.2 FHiCL Keywords Reserved to *art*

21 FHiCL supports run-time configuration for several projects, not only for *art*.
22 *art* reserves certain FHiCL names as keywords that it uses in well-defined ways.
23 (Other projects may use FHiCL names differently.) Within FHiCL files used by
24 *art*, these FHiCL names obey scoping rules similar to C++. These keywords
25 appear in the FHiCL file with a scope, i.e.,

26 `keyword : {`
27 `...`
28 `}`

29 if they define a list of modules or a processing block, or with square brackets
30

31 `keyword :[`
32 `...`
33 `]`

34 if they define a list of paths.

35 The following is a list of the keywords reserved to *art* and their meanings. In the
36 outermost scope within a FHiCL file, the following keywords can appear:

Part

1 **process_name** A user-given name to identify the configuration defined by the
 2 FHiCL file (it is recommended to make it similar to the FHiCL file name).
 3 This must appear at the top of the file. It may not contain the underscore
 4 character (`_`).

5 **source** Identifies the data source, e.g., a file in ROOT format containing HEP
 6 events.

7 **services** Identifies ...

8 **physics** Identifies the block of code that configures the scientific work to be
 9 done on every event (as contrasted with the “bookkeeping” portions).

10 **outputs** List of output modules.

11 The following may appear within the `physics` scope:

12 **producers** Sets the list and order of producer modules; see Chapter 26

13 **analyzers** Sets the list and order of analyzer modules; see Chapter 27

14 **filters** Sets the list and order of filter modules; see Chapter 28

15 **trigger_paths** List of producer and/or filter module paths; for each event,
 16 *art* executes all these module paths. The paths may only contain the
 17 module labels of producer and filter modules that are in the list of defined
 18 module labels. *art* can identify module labels that are common to several
 19 `trigger_paths` and will execute them only once per event. The various
 20 paths within the `trigger_paths` may be executed in any order.

21 **end_paths** List of analyzer and/or output module paths; for each event, *art*
 22 executes all these module paths exactly once. The various paths within
 23 the `end_paths` may be executed in any order.

24 The keyword `process_name` is really only reserved to *art* within the outermost
 25 scope (but it would seem to be needlessly confusing to use `process_name` as
 26 the name of a parameter within some other scope). The names `trigger_paths`
 27 and `end_paths` are artifacts of the first use of the CMS framework, to simulate
 28 the several hundred parallel paths within the CMS trigger; their meaning should
 29 be come clear after reading the remainder of this page.

30 24.3 Structure of a FHiCL Run-time Configu- 31 ration File for *art*

32 Here is a sample FHiCL file called `ex01.fcl` that will do a physics analysis
 33 using the code in the *art* module `Ex01_module.so` (the object file of the C++
 34 source file `Ex01_module.cc`). In this configuration, *art* will operate sequen-
 35 tially on the first three events contained in the source file `inputFiles/input01.data.root`.

```

1 #include "fcl/minimalMessageService.fcl"
2
3 process_name : ex01
4
5 source : {
6     module_type : RootInput
7     fileNames   : [ "inputFiles/input01_data.root" ]
8     maxEvents   : 3
9 }
10
11 services : {
12     message : @local::default_message
13 }
14
15 physics :{
16     analyzers: {
17         hello : {
18             module_type : Ex01
19         }
20     }
21
22     e1          : [ hello ]
23     end_paths  : [ e1 ]
24 }

```

25 Let's look at it step-by-step.

```

26 #include "fcl/minimalMessageService.fcl"

```

27 Similar to C++ syntax, this effectively replaces the ‘#include’ line with
 28 the contents of the named file. This particular file sets up a messaging
 29 service.

```

30 process_name : ex01

```

31 The value of the parameter `process_name` (`ex01`, here, the same as the
 32 FHiCL file name) identifies this *art* job. It is used as part of the identifier
 33 for data products produced in this job. For this reason, the value that you
 34 assign may not contain underscore (`_`) characters. If the `process_name`
 35 is absent, *art* substitutes a default value of “DUMMY.”

```

36 source : {
37     module_type : RootInput
38     fileNames   : [ "inputFiles/input01_data.root" ]
39     maxEvents   : 3
40 }

```

41 This `source` parameter describes where events come from. There may
 42 be at most one source module declared in an *art* configuration. At present

1 there are two options for choosing a source module:

2 **module_type : RootInput** *art*::Events will be read from an input file or from
 3 a list of input files; files are specified by giving their pathname within the
 4 file system.

5 **module_type : EmptyEvent** Internally *art* will start the processing of each
 6 event by incrementing the event number and creating an empty *art*::Event.
 7 Subsequent modules then populate the *art*::Event. This is the normal
 8 procedure for generating simulated events.

9 Here *RootInput* is used; the data input file, in ROOT format, is assigned
 10 to the variable *fileNames*. The *maxEvents* parameter says: Look at
 11 only the first three events in this file. (A value of -1 here would mean
 12 “read them all.”)

13 Note that if no source parameter set is present, *art* substitutes a default
 14 parameter set of:

```
15 source : {
16   module_type : EmptyEvent
17   maxEvents : 1
18 }
```

19 See the web page about configuring input and output modules for details about
 20 what other parameters may be supplied to these parameter sets.

```
21 services : {
22   message : @local::default_message
23 }
```

24 Before starting processing, this puts the message logger in the recom-
 25 mended configuration.

```
26 physics :{
27   analyzers: {
28     hello : {
29       module_type : Ex01
30     }
31   }
```

32 In *art*, *physics* is the label for a portion of the run-time configuration
 33 of a job. It contains the “meat” of the configuration, i.e., the scientific
 34 processing instructions, in contrast to the more administrative or book-
 35 keeping information. The *physics* block of code may contain up to
 36 five sections, each labeled with a reserved keyword (that together form a
 37 parameter set within the FHiCL language); the keywords are *analyzers*,
 38 *producers*, *filters*, *trigger-paths* and *end-paths*. In our example it’s set to
 39 *analyzers*.

1 The `analyzers` keyword takes values that are FHiCL tables of param-
 2 eter sets (this is true also for `filters` and `producers`). Here it takes
 3 the value `hello`, which is defined as a table with one parameter, namely
 4 `module_type`, set to the value `Ex01`. The setup defined a variable called
 5 `LD_LIBRARY_PATH`; *art* knows to match the value defined by the name
 6 `module_type` to a C++ object file with the name `Ex01.module.so`
 7 somewhere in the path defined by `LD_LIBRARY_PATH`.

8 We will expand on the `physics` portion of the FHiCL configuration in
 9 Section 24.5.

```
10  el           : [ hello ]
11  end_paths   : [ el ]
```

12 24.4 Order of Elements in a FHiCL Run-time 13 Configuration File for *art*

14 In FHiCL files there are very, very few places in which order is important. Here
 15 are the places where it matters:

- 16 • A `#include` must come before lines that use names found inside the
 17 `#include`.
- 18 • A later definition of a name overrides an earlier definition of the same
 19 name.
- 20 • The definition of a name resolved using `@local` needs to be earlier in the
 21 file than the place(s) where it is used.
- 22 • Within a trigger path, the order of module labels is important.



23 Here is a list of *a few places* (of many) where order does not matter. This list
 24 is by no means exhaustive.

- 25 • Inside the `physics` scope, the order in which modules are defined does NOT
 26 matter for `filters` and `analyzers` blocks. These blocks define the run-time
 27 configurations of *instances* of modules.
- 28 • The five *art*-reserved words that appear in the outermost scope of a FHiCL
 29 file can be in any order. You could put `outputs` first and `process_name`
 30 last, as far as FHiCL cares. It may be more difficult for humans to follow,
 31 however.
- 32 • Within the `services` block, the `services` may appear in any order.

33 Regarding `trigger_paths` and `end_paths`, the following is a conceptual description
 34 of how *art* processes the FHiCL file:

Part

1. *art* looks at the `trigger_paths` sequence. It expands each trigger path in the sequence, removes duplicate entries and turns the result into an ordered list of module labels. The final list has to obey the order of each contributing trigger path, but there are no other ordering constraints.
 2. It does the same for the `end_paths` sequence but there is no constraint on order.
 3. It makes one big sequence that contains everything in 1 followed by everything in 2.
 4. It looks throughout the file to find parameter sets to match to each module label in the big list in 3.
 5. It gives warning messages if there are left over parameter set definitions not matched to any module label in 3.
 6. It then parses the rest of the physics block to make a “dictionary” that matches module labels to their configuration.
- A conceptual description for the porcessing of services is as follows:
1. *art* first makes a list of all services, sorted alphabetically.
 2. It makes a dictionary that matches service names to their parameter sets. A collorary is that service names must be unique within an *art* job.
 3. *art* has some “magic” services that it knows about internally. It loads the `.so` file for each of them and constructs the services.
 4. It loads the `.so` files for all of the services and calls their constructors, passing each service its proper parameter set.
 5. It works through its list of modules in 5 - it loads the `.so` and calls the constructor, passing the constructor the right parameter set.
 6. It gives warning messages if there are left-over parameter set definitions not matched to any module label in 3.



When one service relies on another, things get a bit more complicated. If service A requires that service B be constructed first, then the constructor of service A must ask *art* for a handle to service B. When this happens, *art* will start to construct service A since it is alphabetically first. When the constructor of A asks for a handle to B, *art* will interrupt the construction of service A, construct service B, and return to finish service A. Next, *art* will see that the next thing in the list is B, but it will notice that B has already been constructed and will skip to the next one.

Got that? Whew!

24.5 The *physics* Portion of the FHiCL Configuration

art looks for the experiment code in *art* modules. These must be referenced in the FHiCL file via *module labels*, which are just variable names that take particular values, as this section will describe. The structure of the FHiCL file – or a portion thereof – therefore defines the event loop for *art* to execute. The event loop, as defined in the FHiCL file, is collected into a scope labeled *physics*.

For a module label you may choose any name, as long as it is unique within a job, contains no underscore (`_`) characters and is not one of the names reserved to *art*. In the sample physics scope code below, we define `aProducer`, `bProducer`, `checkAll`, `selectMode0` and `selectModel` as module labels.

```
physics: {
  producers : {
    aProducer: { module_type: MakeA }
    bProducer: { module_type: MakeB }
  }
  analyzers : {
    checkAll: { module_type: CheckAll }
  }
  filter : {
    selectMode0: {
      module_type: Filter1
      mode: 0
    }
    selectModel: {
      module_type: Filter1
      mode: 1
    }
  }
}
```

The minimum configuration of a module is:

```
<moduleLabel> : { module_type : <ClassName> }
```

for example, in our code above:

```
aProducer: { module_type: MakeA }
```

`aProducer` is the module label and `MakeA` corresponds to a module of experiment code (i.e., an *art* module) named `MakeA.module.so`, which in turn was

1 built from `MakeA_module.cc`. Since it falls within the scope producers, it
 2 must be a module of type `EDProducer`.

3 Let's take this a step farther, and assume that this `EDProducer`-type module
 4 `MakeA` accepts four arguments that we want to provide to *art*. The configura-
 5 tion may look like this:

```

6 moduleLabel : {
7   module_type : MakeA
8   pname0 : 1234.
9   pname1 : [ abc, def]
10  pname2 : {
11      name0: {}
12  }
13 }
```

14 This list under `module_type : MakeA` represents parameters that will be
 15 formed into a `fhicl::ParameterSet` object and passed to the module `MakeA`
 16 as an argument in its constructor. `pname0` is a double, `pname1` is a sequence
 17 of two atomic character values, `pname2` consists of a single table named `name0`
 18 with undefined contents.

19 Note that *paths* are lists of module labels, while the two reserved names, `trigger_paths`
 20 and `end_paths` are lists of paths.

21 24.6 Choosing and Using Module Labels and 22 Path Names

23 For a module label or a path name, you may choose any name so long as it is
 24 unique within a job, contains no underscore (`_`) characters and is not one of the
 25 names reserved to *art* (see Section 24.2).

26 Any name that is a top-level name inside of the `physics` parameter set is either
 27 a reserved name or the name of a path.

28 It is important to recognize which identifiers are module labels and which are
 29 path names in a FHiCL file. It is also important to distinguish between a class
 30 that is a module and instances of that module class, each uniquely identified by
 31 a module label.

32 *art* has several rules that were recommended practices in the old framework but
 33 which were not strictly enforced by that framework. *art* enforces some of these
 34 rules and will, soon, enforce all of them:

- 35 • A path may go into either the `trigger_paths` list or into the `end_paths`
 36 list, but not both.

- 1 • A path that is in the `trigger_paths` list may only contain the module
- 2 labels of producer modules and filter modules.
- 3 • A path that is in the `end_paths` list may only contain the module labels
- 4 of analyzer modules and output modules.
- 5 Analyzer modules and the output modules may be separated into different paths;
- 6 that might be convenient at some times but it is not necessary. On the other
- 7 hand, keeping trigger paths separate has real meaning.

8 24.7 Scheduling Strategy in *art*

9 A set of scheduling rules is enforced in *art*. (Some of the details are remnants
 10 of compromises and conflicting interests with CMS.) One of the top-level rules
 11 in the scheduler is that all producers and filters must be run first, using the
 12 ordering rules specified below. After that, all analyzer and output modules will
 13 be run. Recall that analyzer modules and output modules may not modify the
 14 event, nor may they produce side effects that influence the behavior of other
 15 analyzer or output modules. Therefore, *art* is free to run analyzer and output
 16 modules in any order.

17 The full description of the scheduler strategy is given below:

- 18 • If a module name appears in the definition of a path name but it is not
- 19 found among the the list of defined module labels, FHiCL will issue an
- 20 error.
- 21 • One each event, before executing any of the paths, execute the source
- 22 module.
- 23 • On each event, execute all of the paths listed in the `trigger_paths`.
 - 24 – Within one path, the order of modules listed in the path is followed
 - 25 strictly; at present there is one exception to this: see the discussion
 - 26 about the remaining issues
 - 27 – *art* can identify module labels that are in common to several trig-
 - 28 ger_paths and will execute them only once per event. In the above
 - 29 example, `aProducer` and `bProducer` are executed only once per event.
 - 30 – The various paths within the `trigger_paths` may be executed in any
 - 31 order, subject to the above constraints.
 - 32 – If a path contains a filter, and if the filter return false, then the
 - 33 remainder of the path is skipped.
 - 34 – The module name of a filter can be negated in path using, `!module-`
 - 35 Label; in this case the path will continue if the filter returns false
 - 36 and will be aborted if the filter returns true.

- 1 – If the module label of a filter appears in two paths, negated in one
- 2 path and not negated in the other, *art* will only run the instance of
- 3 filter module once and will use the result in both places.
- 4 – If a module in a trigger path throws, the default behaviour of *art* is to
- 5 stop all processing and to shut down the job as gracefully as possible.
- 6 *Art* can be configured, at run time, so that, for selected exceptions,
- 7 it behaves differently. For example it can be configured to continue
- 8 with the current trigger path, skip to the next trigger path, skip to
- 9 the next event, and so on.
- 10 • On each event, execute all of the paths listed in the `end_paths`.
- 11 – The module labels listed in `end_paths` are executed exactly once per
- 12 event, regardless of how many paths there are in the `trigger_paths`
- 13 and regardless of any filters that failed.
- 14 – If a module label appears multiple times among the end paths, it is
- 15 executed only once. No warning message is given.
- 16 – Even if all `trigger_paths` have filters that fail, all module labels in the
- 17 end path will be run.
- 18 – `End_path` is free to execute the modules in the `end_path` in any order.
- 19 – If a module in the `end_path` throws, the default response of *art* is to
- 20 make a best effort to complete all other modules in the end path and
- 21 then to shutdown the job in an orderly fashion. This behaviour can
- 22 be changed at run-time by adding the appropriate parameter set to
- 23 the top level `.fcl` file.
- 24 • One can ask that an output module be run only for events that pass a
- 25 given `trigger_path`; this is done using the `SelectEvents` parameter set,
- 26 • At present there is no syntax to ask that an analyzer module be run
- 27 only for events that pass or fail some of the trigger paths. A planned
- 28 improvement to *art* is to give analyzer modules a `SelectEvents` parameter
- 29 that behaves as it does for output modules.
- 30 • If a path appears in neither the `trigger_paths` nor the `end_paths`, there is
- 31 no warning given.
- 32 • If a module label appears in no path, a warning will be given.

33 In the above there is a lot of focus on which groups of modules are free to be
 34 run in an arbitrary order. This is laying the groundwork for module-parallel
 35 execution: *art* is capable of identifying which modules may be run in parallel
 36 and, on a multi-core machine, *art* could start separate threads for each module.
 37 At present both ROOT and G4 are not thread-safe so this is not of immediate
 38 interest. But there are efforts underway to make both of these thread-safe and
 39 we may one day care about module-parallel execution; our interest in this will

1 depend a great deal on the future evolution of the relative costs of memory and
2 CPU.

3 For simple cases, in which there is one trigger path with only a few modules
4 in the path, and one end path with only a few modules in the path, the extra
5 level of bookkeeping is just extra typing with no obvious benefit. The benefit
6 comes when many work groups wish to run their modules on the same events
7 during one art job; perhaps this is a job skimming off many different calibration
8 samples or perhaps it is a job selecting many different streams of interesting
9 Monte Carlo events. In such a case, each work group needs only to define their
10 own trigger path and their own end path, without regard for the requirements
11 of other work groups; each work group also needs to ensure that their paths are
12 added to the `end_paths` and `trigger_paths` variables. Art will then automatically,
13 and correctly, schedule the work without redoing any work twice and without
14 skipping work that must be done. This feature came for free with art and,
15 while it imposes a small burden for novice users doing simple jobs, it provides
16 an enormously powerful feature for advanced users. Therefore it was retained
17 in art when some other features were removed.

18 24.8 Scheduled Reconstruction using Trigger Paths

19 Consider the following problem. You wish to run a job that has:

- 20 • Two producers `MakeA_module.cc` and `MakeB_module.cc`. You want to
21 run both producers on all events.
- 22 • One analyzer module that you want to run on all events, `CheckAll_module.cc`.
- 23 • You have a filter module, `Filter1_module.cc` that has two modes, 0 and 1;
24 the mode can be selected at run time via the parameter set.
- 25 • You wish to write all events that pass mode 0 of the filter to the file
26 `file0.root` and you wish to write all events that pass mode 1 of the filter to
27 `file1.root`

28 Here is code that would accomplish this:

```
29 process_name: filter1
30
31 source: {
32     # Configure some source here.
33 }
34
35 physics: {
36
37     producers : {
38         aProducer: { module_type: MakeA }
```

Part

```

1      bProducer: { module_type: MakeB }
2    }
3
4    analyzers : {
5      checkAll: { module_type: CheckAll }
6    }
7
8    filter : {
9      selectMode0: {
10        module_type: Filter1
11        mode: 0
12      }
13      selectModel: {
14        module_type: Filter1
15        mode: 1
16      }
17    }
18
19    mode0: [ aProducer, bProducer, selectMode0 ]
20    model: [ aProducer, bProducer, selectModel ]
21    analyzermods: [ checkAll ]
22    outputFiles: [ out0, out1 ]
23
24    trigger_paths : [ mode0, model ]
25    end_paths : [ analyzermods, outputFiles ]
26  }
27
28  outputs: {
29    out0: {
30      module_type: RootOutput
31      fileName: "file0.root"
32      SelectEvents: { SelectEvents: [ mode0 ] }
33    }
34
35    out1: {
36      module_type: RootOutput
37      fileName: "file1.root"
38      SelectEvents: { SelectEvents: [ model ] }
39    }
40
41  }

```

Recall that the names `process_name`, `source`, `physics`, `producers`, `analyzers`, `filters`, `trigger_paths`, `end_paths` and `outputs` are reserved to *art*. The names `aProducer`, `bProducer`, `checkAll`, `selectMode0`, `selectModel1`, `out0` and `out1` are module labels, and these are names of paths: `mode0`, `model`, `outputFiles`, `ana-`

1 lyzermodes.

2 24.9 Reconstruction On-Demand

3 24.10 Bits and Pieces

4 What variables are known to art? physics (which has the five reserved keywords
5 fi

6 lters, analyzers, producers, trigger paths and end paths), what else? input file
7 type RootInput

8 I know that trigger path are // different from end paths, they can contain
9 different types of modules; // event gets frozen after trigger path.

10 art knows to match the value defi

11 ned by the name 'module_name' to a C++ object fi

12 le with the name module_name_module.so" somewhere in the path defi

13 ned by LD LIBRARY PATH.

14 Further information on the FHiCL language and usage can be found at the
15 mu2e FHiCL page.

25 Data Products

25.1 Overview

A *data product* is anything that you can add to an event or see in an event. Examples include the generated particles, the simulated particles produced by Geant4, the hits produced by Geant4, tracks found by the reconstruction algorithms, clusters found in the calorimeters and so on.

25.2 The Full Name of a Data Product

Each data product within an event is uniquely identified by a four-part identifier that includes all namespace information. The four parts are separated by underscores:

`DataType_ModuleLabel_InstanceName_ProcessName`

DataType identifies the data type that is stored in the product. It is a “friendly” identifier in the way that its syntax has been standardized to deal with *collection* types, as follows:

- If a product is of type `T`, then the friendly name is “`T`”.
- If a product is of type `mu2e::T`, then the friendly name is “`mu2e::T`”.
- If a product is of type `std::vector<mu2e::T>`, then the friendly name is “`mu2e::Ts`”.
- If a product is of type `std::vector<std::vector< mu2e::T > >`, then the friendly name is “`mu2e::Tss`”.
- If a product is of type `cet::map_vector<mu2e::T>`, then the friendly name is “`mu2e::Tmv`”. See below for a discussion about where underscores may not be used; this example is safe because of the substitution of “mv” for `map_vector`.

1 *ModuleLabel* identifies the module that created the product; this is the module
 2 label , which distinguishes multiple instances of the same module within a
 3 produces . It is *not* the class name of the module.

4 *InstanceName* is a label for the data product that distinguishes two or more
 5 data products of the same type that were produced by the same module, in
 6 the same process . If a data product is already unique within this scope, it is
 7 legal to leave this field blank . The instance label is the optional argument of
 8 the call to “produces” in the constructor of the module (xxxx below) :

```
9 produces<T> ("xxxx") ;
```

10 *ProcessName* is the name of the process that created this product. It is specified
 11 in the FHiCL file that specifies the run-time configuration for the job (shown
 12 as *ReadBack02* below):

```
13 process_name : ReadBack02
```

14 Because the full name of the product uses the underscore character to delimit
 15 fields, it is forbidden to use underscores in any of the names of the fields. There-
 16 fore, none of the following items may contain underscores:

- 17 • the class name of a class that is a data product; the exception is the
 18 `cet::map_vector` template; when creating the friendly name, *art* internally
 19 recognizes this case and protects against it
- 20 • the namespace in which a data product class lives
- 21 • module labels
- 22 • data product instance names
- 23 • process names

24 It is important to know which names need to match each other; see Sec-
 25 tion 31.1.

26 Producer Modules

27 Analyzer Modules

Analyzer modules request data products, do not create new ones; make histograms, etc. .

An analyzer interface looks like the following.

```
class EDAnalyzer {
    // explicit EDAnalyzer(ParameterSet const&)
    virtual void analyze(Event const&) = 0
    virtual void reconfigure(ParameterSet const&)
    virtual void beginJob()
    virtual void endJob()
    virtual bool beginRun(Run const &)
    virtual bool endRun(Run const &)
    virtual bool beginSubRun(SubRun const &)
    virtual bool endSubRun(SubRun const &)
    virtual void respondToOpenInputFile(FileBlock const& fb)
    virtual void respondToCloseInputFile(FileBlock const& fb)
    virtual void respondToOpenOutputFiles(FileBlock const& fb)
    virtual void respondToCloseOutputFiles(FileBlock const& fb)
}
```


28 Filter Modules

Filter modules request data products and can alter further processing using return values .

A filter interface looks like the following.

```
class EDFilter {  
    // explicit EDFilter(ParameterSet const&)  
  
    virtual bool filter(Event&) = 0  
    virtual void reconfigure(ParameterSet const&)  
  
    virtual void beginJob()  
    virtual void endJob()  
    virtual bool beginRun(Run &)  
    virtual bool endRun(Run &)  
    virtual bool beginSubRun(SubRun &)  
    virtual bool endSubRun(SubRun &)  
  
    virtual void respondToOpenInputFile(FileBlock const& fb)  
    virtual void respondToCloseInputFile(FileBlock const& fb)  
    virtual void respondToOpenOutputFiles(FileBlock const& fb)  
    virtual void respondToCloseOutputFiles(FileBlock const& fb)  
}
```


29 *art* Services

Several types of *art* services exist:

- TFile: Controls the ROOT directories (one per module) and manages the histogram file.
- Timing: Tracks CPU and wall clock time for each module for each event
- Memory: Tracks increases in overall program memory on each module invocation
- FloatingPointControl: Allows configuration of FPU hardware “exception” processing
- (RandomNumberService): Manages the state of a random number stream for each interested module
- (MessageFacility): Routes user-emitted messages from modules based on type and severity to destinations

An access interface looks like the following.

```
#include "art/Framework/Services/Optional/TFileService.h"
...
art::ServiceHandle<art::TFileService> tfs;
fFinalVtxX = tfs->make<TH1F>("fFinalVtxX",
                             "Circe Vertex X; Xfit-Xmc (cm); Events",
                             200, -50.0, 50.0);
```

FHiCL configuration of services

```
services:
{
  TFileService:
  {
    fileName: "tfile_output.root"
  }
  user:
  {
```

```
1      # experiment- or user-defined plugin service
2      }
3      ...
4      }
```

1 30 art Input and Output

2 30.1 Input Modules

3 30.1.1 Configuring Input Modules to Read from Files

4 When reading from an existing file, *art* allows you to select the input files, the
5 starting event, the number of events to read, etc., either from the command
6 line or from the FHiCL file. If a particular quantity is controlled from both
7 the command line and the FHiCL file, the value on the command line takes
8 precedence.

9 The following code fragment tells *art* to read event data from the file of type
10 “ROOT,” named “file01.root” and to start at the beginning of the file. A value of
11 “-1” for maxEvents tells *art* to read events until the end of file is reached:

12 To tell *art* to read 100 events, or until the end of file, which ever comes first,
13 change the parameter maxEvents to 100. This also shows how to specify a list
14 of input files:

15 The number of files in the list of input files is arbitrary. The following fragment
16 tells *art* to skip the first two events (and thus start with the third):

17 The fragment below shows some other parameters that can be included in the
18 source parameter set:

19 The parameters whose names start with *first* specify that the first event to be
20 processed will be the first event that has an EventID greater than or equal to

Listing 30.1: Reading in a ROOT data file

```
1 source :{  
2   module_type : RootInput  
3   fileNames   : [ "file01.root" ]  
4   maxEvents   : -1  
5 }
```


Listing 30.2: Reading in a ROOT data file

```

1 source : {
2   module_type : RootInput
3   fileNames   : [ "file01.root", "file02.root", "file03.root" ]
4   maxEvents   : 100
5 }

```

Listing 30.3: Reading in a ROOT data file

```

1 source : {
2   module_type : RootInput
3   fileNames   : [ "file01.root", "file02.root", "file03.root" ]
4   maxEvents   : 100
5   skipEvents  : 2
6 }

```

1 the specified event. If one of the first* parameters is not specified, it takes a
 2 default value of -1 and is excluded from the comparison.

3 If a file of unsorted events is read in, *art* will, by default, present the events
 4 for processing in order of increasing event number. As a corollary to this, the
 5 output file will contain the events in sorted order. This sorting occurs one input
 6 file at a time; *art* does not sort across file boundaries in a list of input files. If
 7 the noEventSort parameter is set to *true*, the sorting is disabled, which will, in
 8 most cases, yield a minor performance improvement.

9 I have not yet learned the precise meaning of the skipBadFiles and the fileMatch-
 10 Mode parameters.

11 The inputCommands parameter tells *art* to delete certain data products from
 12 the copy of the event in memory after reading the input file. In other words, the
 13 input file itself is not modified but data products are removed from the copy of
 14 the event in memory before any modules are called. The syntax of this language
 15 is the same as for outputCommands, described .

16 In the pre-*art* versions of the framework, there were methods to select ranges of
 17 events or ranges of SubRuns. This is not yet working in *art*; the *art* developers
 18 will add this feature back once we decided exactly what we mean by "ranges of

Listing 30.4: Reading in a ROOT data file

```

1   firstRun           : 0
2   firstSubRun        : 0
3   firstEvent         : 0
4   noEventSort        : false
5   skipBadFiles        : false
6   fileMatchMode       : "permissive"
7   inputCommands       : ""

```

Listing 30.5: Reading in a ROOT data file

```

1 source :{
2   module_type : EmptyEvent
3   maxEvents   : 200
4 }
```

Listing 30.6: Reading in a ROOT data file

```

1 source :{
2   module_type      : EmptyEvent
3   firstRun         : 2
4   firstSubRun      : 1
5   firstEvent       : 1
6   numberEventsInRun : 1000
7   numberEventsInSubRun : 100
8   maxEvents        : 200
9   resetEventOnSubRun : true
10 }
```

1 events”.

2 Specifying Many Input Files In the pre-art, python based, configuration lan-
 3 guage, the standard syntax to initialize a list of input files was limited to 255
 4 files, after which an alternate syntax was required. This is no longer necessary;
 5 the length of a fhicl list is limited only by available memory.

6 Empty Source

7 In many simulation applications one wishes to start with an empty event, run
 8 one or more event generators, pass the generated particles through the Geant4,
 9 and so on. In art the first step in this chain is accomplished using a source
 10 module named EmptySource, as follows:

11 Instead of reading event-data from a file, the empty source increments the event
 12 number and presents an empty event to the modules that will do the work. One
 13 may configure EmptySource to specify the EventId of the first event, to specify
 14 the maximum number of events in a SubRun or SubRuns in a run.

15 The last option tells art to reset event numbers to start at 1 whenever art
 16 starts a new SubRun begins; this is the default behaviour and is opposite to the
 17 behaviour we inherited from CMS.

18 30.2 Output Filtering

19 Any output module can be configured to write out only those events passing a
 20 given trigger path.

- 1 The parameter set that configures the output module uses a parameter Se-
 2 lectEvents to control the output, as shown in the example below:

```

3   # this is only a fragment of a full configuration ...
4   physics:
5   {
6       pathA: [ ... ] # producers and filters are put in this path
7       pathB: [ ... ] # other producers, other filters are put in this path
8
9       outA: [ passWriter ] # output modules and analyzers are put in this pa
10      outB: [ failWriter ] # output modules and analyzers are put in this pa
11      outC: [ exceptWriter ] # output modules and analyzers are put in this
12
13      trigger_paths: [ pathA, pathB ] # declare that these are "trigger path
14      end_paths: [ outA outB outC ] # declare these are "end paths"
15  }
16
17  outputs:
18  {
19      passWriter:
20      {
21          module_type: RootOutput
22          fileName: "pathA_passes.root"
23          # Write all the events for which pathA ended with 'true' from filter
24          # Events which caused an exception throw will not be written.
25          SelectEvents: { SelectEvents: [ "pathA&noexception" ] }
26      }
27      failWriter:
28      {
29          module_type: RootOutput
30          fileName: "pathA_failures.root"
31          # Write all the events for which pathA ended with 'false' from filter
32          # Events which caused an exception throw will not be written.
33          SelectEvents: { SelectEvents: [ "!pathA&noexception" ] }
34      }
35      exceptWriter:
36      {
37          module_type: RootOutput
38          fileName: "pathA_exceptions.root"
39          # Write all the events for which pathA or pathB ended because an exc
40          SelectEvents: { SelectEvents: [ "exception@pathA", "exception@pathB"
41      }
42  }

```

1 30.3 Configuring Output Modules

31 *art* Misc Topics that Will Find Home

31.0.1 The Bookkeeping Structure and Event Sequencing Imposed by *art*

In almost all HEP experiments, the core idea underlying all bookkeeping is the *event*. In a triggered experiment, an event is defined as all of the information associated with a single trigger; in an untriggered spill-oriented experiment, an event is defined as all of the information associated with a single spill of the beam from the accelerator. Another way of saying this is that an event contains all of the information associated with some time interval, but the precise definition of the time interval changes from one experiment to another. Typically these time intervals are a few nano-seconds to a few tens of mirco-seconds. The information within an event includes both the raw data read from the Data Acquisition System (DAQ) and all information that is derived from that raw data by the reconstruction and analysis algorithms. An event is smallest unit of data that *art* can process at one time.

In a typical HEP experiment, the trigger or DAQ system assigns an event identifier (event ID) to each event; this ID uniquely identifies each event. The simplest event ID is a monotonically increasing integer. A more common practice is to define a multi-part ID.

art has chosen to use a three-part ID. In *art*, the parts are named

- run number
- subRun number
- event number

In a typical experiment the event number will be incremented every event. When some condition occurs, the event number will be reset to 1 and the subRun number will be incremented, keeping the run number unchanged. This cycle will repeat until some other condition occurs, at which time the event number will be reset to 1, the subRun number will be reset to 0 and the run number will be incremented.



art does not define what conditions cause these transitions; those decisions are

1 left to each experiment. Typically, experiments will choose to start new runs or
 2 new subRuns when any of the following happen:

- 3 • a preset number of events have been acquired
- 4 • a preset time interval has expired
- 5 • a disk file holding the output has reached a preset size
- 6 • certain running conditions change

7 *art* requires only that a subRun contain zero or more events and that a run
 8 contain zero or more subRuns.

9 As runs are collections of subRuns, and subRuns are collections of events, events
 10 in turn are collections of *data products*. A data product is the smallest unit of
 11 data that can be added to or retrieved from a given event. Each experiment
 12 defines types (classes and structs) for its own data products. These include types
 13 that describe the raw data, and types to define the reconstructed data and the
 14 information produced by simulations. *art* knows nothing about the internals of
 15 any experiment's data products; for *art*, the data product is a “fundamental
 16 particle.”

17 At the outside shell of the Russian doll that is the bookkeeping structure in *art*,
 18 runs are collected into the *event-data*, defined as all of the data products in an
 19 experiment's files; plus the metadata that accompanies them.

20 When an experiment takes data, events read from Data Acquisition System
 21 (DAQ) are typically written to disk files, with copies made on tape. *art* imposes
 22 only weak constraints on the event sequence within a file. The events in a single
 23 subRun may be spread over several files; conversely a single file may contain
 24 many runs, each of which contains many subRuns.

25 A critical feature of *art*'s design is that each event must be uniquely identifiable
 26 by its event ID. This requirement also applies to simulated events.

27 31.1 Rules for Module Names

28 Within any experiment's software, sometimes names of files, classes, libraries,
 29 etc., must follow certain rules. Other times, conventions are just conventions.
 30 This section is concerned with actual rules only.

31 Consider a class named `MyClass` that you wish to make into an *art* module.
 32 First, your class must inherit from one of the module base classes, `EDAnalyzer`,
 33 `EDProducer` or `EDFilter`. Secondly, it must obey the following rules, all of
 34 which are case-sensitive.

- 35 1. it must be in a file named `MyClass_module.cc`
 36 The build system will make this into a file named `lib/libMyClass_module.so`.

Listing 31.1: Module source sample

```

1 namespace xxxx {
2
3     class MyClass : public art::EDAnalyzer {
4
5     public:
6         explicit MyClass(fhicl::ParameterSet const& pset);
7         // Compiler generated destructor is OK.
8
9         void analyze( art::Event const& event );
10
11     };
12
13     MyClass::MyClass(fhicl::ParameterSet const& pset){
14         // Body of the constructor. You can access information
15         in the parameter set here.
16     }
17
18     void MyClass::analyze(art::Event const& event){
19         mf::LogVerbosim("test")
20         << "Hello, _world._From_analyze._"
21         << event.id();
22     }
23
24 } // end namespace xxxx
25
26 using xxxx::MyClass;
27 DEFINE_ART_MODULE(MyClass);

```

- 1 2. the module source file must look like Listing 31.1 (where your experiment's
- 2 namespace replaces xxxx):

3 This example is for an analyzer. To create a producer or a filter mod-

4 ule, you must inherit from either `art::EDProducer` or `art::EDFilter`, re-

5 spectively. The last line (`DEFINE_ART_MODULE(MyClass);`) invokes a

6 macro that inserts additional code into the `.so` file.

7 For the experts: it inserts a factory method to produce an instance of the

8 class and it inserts an auto-registration object that registers the factory

9 method with *art*'s module registry.

10 To declare this module to the framework you need to have a fragment like

11 the following in your FHiCL file:

```

12 1
13 2     physics :
14 3     {
15 4         analyzers:
16 5         {
17 6             looseCuts : { module_type : MyClass }
18 7
19 8             // Other analyzer modules listed here ...
20 9         }

```




```
1 10          }
```

2 where the string `looseCuts` is called a *module label* and is defined below.

3 3. the previous item was for the case that your module is an analyzer. If it is
4 a producer or filter, then the label *analyzers* needs to be either *producers*
5 or *filters*.

6 4. When you put a data product into an event, the data provenance system
7 records the module label of the module that did the “put.”

8 31.2 Data Products and the Event Data Model

9 The part of *art* that deals with the bookkeeping of the data products is called
10 the *Event Data Model*, which concerns itself with the following ideas:

11 1. what a data product looks like when it is in the memory of a running
12 program

13 2. what it looks like on disk

14 3. how it moves between memory and disk

15 4. how a data product refers to another piece of event-data within the same
16 event

17 5. how a given piece of experiment code accesses a data product

18 6. how the experiment code adds a new data product to the event

19 7. metadata that describes, for each data product,

20 • what piece of code was used to create it

21 • what is the run-time configuration of that code

22 • what data products were read in by this experiment code



23 8. The mechanism by which the metadata is “married” to the data

24 One of the core principles of *art* is that experiment code modules may commu-
25 nicate with each other only via the event.

26 31.3 Basic *art* Rules

27 *art* prescribes that your classes (i.e., your *art modules*) always contain a *member*
28 *function* that has a particular name, takes a particular set of arguments, and
29 operates on every event; *art* will call this member function for every event
30 read from the data source (input). If no member function with these attributes

exists, then at execution time *art* will print an error message and stop execution.

If your module provides any optional functions, then *art* requires a name and a set of arguments for each. For each of these that is present in a given class, *art* will make sure that it is called at the right time.

The details of the *art* rules will be discussed in .

31.4 Compiling, Linking, Loading and Executing C++ Classes and *art* Modules

When you write code to be executed by *art*, you provide it to *art* as a group of C++ functions. To make this group of functions visible to *art*, you write a C++ class that obeys a set of rules defined by *art* (summarized in Section ??). Such a class is called an *art module*, or just *module* in this documentation (this should not be confused with the notion of a *module* as defined more generally in the programming world). The container source code file for an *art* module gets compiled into a shared object library that can be dynamically loaded by *art*.

The *experiment's shared code libraries* in Figures ?? and ?? may include libraries containing standard C++ classes as well as *art* modules.

Experiments typically have many, many C++ classes for offline processing, and physicists add to them all the time. Classes from many files can be linked into a single library, as shown in Figure 31.1. The shared libraries may have one-way dependencies on each other; i.e. if library 'a' depends on library 'b', then the reverse cannot be true.

art modules, as mentioned above, follow a special structure, illustrated in Figure 31.2. They do not use header (.h) files (everything for a module is contained within a single .cc file), a single module builds a single shared library, and the name (as recognized by *art*) for each file in the build chain must end in `_module`, e.g., `MyCoolMod_module.cc`. Moreover, *art* recognizes `MyCoolMod_module.cc` as the source for `libxxx.MyCoolMod_module.so`. (Discussion of the *xxx* will be deferred.)

31.5 Shareable Libraries and *art*

When you execute code within the *art* framework, the main executable is provided by *art*, not by your experiment. Your experiment provides its code to the

¹Actually the loader that loads the shareable library, rather than *art* itself, will figure this out.

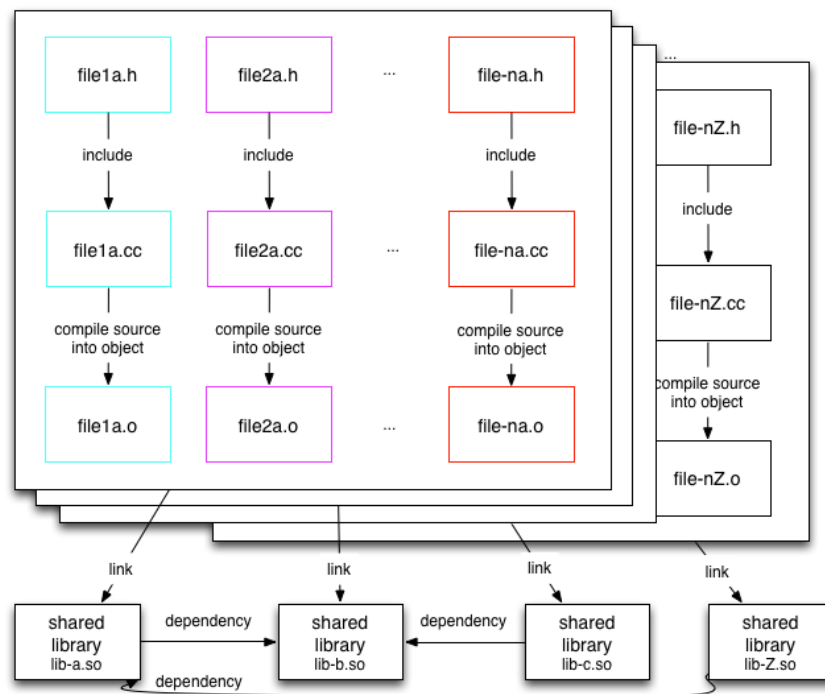


Figure 31.1: Illustration of compiled, linked “regular” C++ classes (not *art* modules) that can be used within the *art* framework. Many classes can be linked into a single shared library.

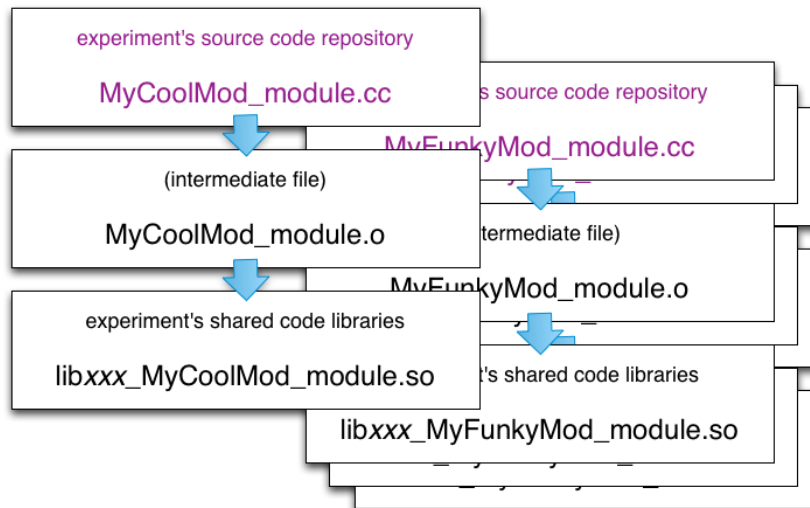


Figure 31.2: Illustration of compiled, linked *art* modules; each module is built into a single shared library for use by *art*

- 1 executable in the form of shareable object libraries that *art* loads dynamically at
- 2 run time; these libraries are also called *dynamic load libraries* or *plugins*.
- 3 Your experiment will likely have many “regular” C++ classes (as distinct from
- 4 the C++ classes that are modules, aka “*art* modules”). These “regular” classes
- 5 get built into a set of shareable libraries, where each library contains object
- 6 code for multiple classes.
- 7 Your experiment will likely have many modules, too. In fact you will likely be
- 8 writing some for your own analyses. A module must be compiled into its own
- 9 shareable object library, i.e., there is a one-to-one correspondance between the
- 10 .cc file and the .so file for a given module. When the configuration file tells *art*
- 11 to run a particular module, *art* finds the corresponding .so file, loads it, and
- 12 calls the appropriate member function at each stage of the event loop.

31.6 Namespaces, *art* and the Workbook

A *namespace* is a prefix that is used to keep different subsets of code distinguishable from one another; i.e., if the same identifier (variable name or type name) is used within multiple namespaces, each will remain distinguishable via its namespace prefix. The otherwise ambiguous identifier should be written

as

```
<namespace> :: <identifier>
```



1 The notion of *namespace* is related to that of *scope*: Within a C++ source
 2 file (.cc files) a scope is designated by a set of curly braces ({ ... }). Once
 3 a namespace is defined within a given scope, any identifiers within that scope
 4 that “belong to” that namespace no longer need to be written with the pref.
 5 E.g., the following fragment uses the `analyze` defined in the namespace `tex`
 6 (i.e., `tex :: analyze`):

```

7 namespace tex {
8     class First : public art :: EDAnalyzer {
9     public :
10         explicit First ( fhicl :: ParameterSet const & );
11         void analyze ( art :: Event const & event ) override ;
12     };
13 }
```

14 Note that `EDAnalyzer` is defined in the namespace `art`, as is `Event`, and
 15 `ParameterSet` is in `fhicl`.

16 Note also that namespaces are often associated with UPS product code, al-
 17 though the product and the namespace names may not always be identical. E.g.,
 18 code associated with the UPS product *fhiclcpp* is in the namespace `fhicl`.

19 All of the code in the toyExperiment UPS product was written in a namespace
 20 named `tex`; the name `tex` is an acronym-like shorthand for the toyExperiment
 21 (ToyEXperiment) UPS product. Because all of the Workbook code builds on top
 22 of the toyExperiment code, this code has been placed in the same namespace.
 23 The `tex` namespace has no special meaning to *art*, it is just a convenience.
 24 (Note that the *art* code itself is in a separate namespace called `art`.)

25 If you need more information about the C++ notion of *namespaces*, see a stan-
 26 dard C++ reference.

27 31.7 Orphans

28 A best practice: define ids in the narrowest scope possible to avoid accidental
 29 name collisions

30 During processing, derived information in the event may be changed, added to
 31 or deleted; the raw data is not modified. The *event* is the smallest unit of data
 32 that art can process at one time.

33 How bash shell scripts work

34 If you would like to understand how they work, the following will be use-
 35 ful:

- 36 • BASH Programming - Introduction HOW-TO
- 37 <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

- 1 • Bash Guide for Beginners
- 2 <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html>
- 3 • Advanced Bash Scripting Guide
- 4 <http://www.tldp.org/LDP/abs/html/abs-guide.html>
- 5 The first of these is a compact introduction and the second is a more compre-
- 6 hensive introduction.
- 7 The above guides were all found at the Linux Documentation Project: Work-
- 8 book:
- 9 • <http://www.tldp.org/guides.html>

10 31.8 Code Guards

11 All of the header files that you will see in the Workbook wrap their contents
12 with the following three lines:

```
13  #ifndef path_to_this_header_file_h
14  #define path_to_this_header_file_h
15      // contents of the header file
16  #endif /* path_to_this_header_file_h */
```

17 The three lines beginning with # are macros that will be processed by the
18 C preprocessor at the start of compilation. These lines are called *code guards*
19 and they address the following issue.

20 Suppose that you have a main program that includes two header files A.h and
21 B.h; further suppose that both of A.h and B.h include a third header file C.h.
22 When you compile the main program, the C preprocessor will expand all of the
23 include directives to create a temporary .cc file on which the compiler will do
24 its work. This temporary file must contain exactly one copy of the header file
25 C.h; if it contains either zero copies or more than one copy (as it would in this
26 case), the compiler will issue an error. The C preprocessor, by itself, is not
27 smart enough to skip the second inclusion of C.h but it does provide the tools
28 for us to help it do so.

29 In the first two lines, the text `path_to_this_header_file_h` is the name of a
30 C preprocessor variable; the choice of the variable name will be described later
31 but the important feature is that it must be unique within the compilation unit
32 (the file being compiled). When the C preprocessor encounters the included file
33 C.h, the line `#ifndef` tells the preprocessor to check to see if the C prepro-
34 cessor variable with this name is defined. If the variable is not defined then the
35 lines between the `#ifndef` line and the `#endif` line will be included in the
36 output of the C preprocessor. If it has already been defined, these lines will
37 be excluded from the output.

1 The first time that the preprocessor encounters `C.h` within a compilation unit,
2 the variable will not have been defined and the contents of the header will be
3 included in the output of the preprocessor. At the same time the second line
4 of the above fragment will be executed; it is a preprocessor directive that tells
5 the preprocessor to define the variable. In either case, when the preprocessor
6 encounters the second inclusion of `C.h`, the `#ifndef` test will fail and the body
7 of the header will not be copied into the output of the preprocessor. And so on
8 for subsequent inclusions of `C.h`.

9 If every header file in a code base correctly uses code guards, then every header
10 file can safely include all other header files on which it depends and one need
11 not worry about this causing compiler errors due to multiple declarations of a
12 class or function.

13 The full syntax of the `#define` directive allows one to specify a value for the
14 variable but that is not important here; the `#ifndef` test only cares that the
15 variable is defined, not what its value is.

16 For code guards to work, each header file must choose a C preprocessor variable
17 name that is unique within every compilation unit in which it might be included,
18 either directly included or indirectly included. The convention that is used by
19 *art*, by other libraries managed by the *art* team, by the toyExperiment UPS
20 product and by the Workbook is that the name of the variable is the name of
21 the path to the header file, starting from the root of the code base and with
22 the slash and dot characters changed to underscores; the reason for this change
23 is that slash and dot characters are not legal in the name of a C preprocessor
24 variable. This works because all of these products also adopt the convention
25 that the path to their header file starts with the product name. While this is
26 not perfect security it is a very high level of security.

27 31.9 Inheritance

28 31.9.1 Introduction

29 This section introduces a few of the ideas behind *inheritance* and *polymorphism*.
30 There are many, many different ways to use *inheritance* and *polymorphism* but
31 you only need to understand the small subset that are relevant for the Workbook
32 exercises. You can read about inheritance and polymorphism at the following
33 url:

34 <http://www.cplusplus.com/doc/tutorial/inheritance/>

35 Skip the section on Friendship and start at the section on inheritance. When
36 you get to the bottom of the page, continue to the next page by clicking on the
37 arrow for “Polymorphism”. You can skip the discussion of protected and private
38 inheritance because you will only need to know about public inheritance.

1 After you have learned this material, return to this section and work through
2 the following example which serves as a test that you have learned the necessary
3 material. This example is motivated by the Polygon example given in the ref-
4 erenced material. In this example there is a base class named Shape and three
5 derived classes, Circle, Triangle and Rectangle. The main program that
6 exercises these four classes is `itest.cc`.

7 31.9.2 Homework

8 To build and run this example:

9 1. log in and follow the steps in Section ??

10 2. cd to the directory for this exercise
11 \$ cd Inheritance/v1
12 \$ ls
13 build Circle.h Rectangle.cc Shape.cc Triangle.cc
14 Circle.cc itest.cc Rectangle.h Shape.h Triangle.h

15 3. build the exercise
16 \$../build
17 This will create the executable file `itest`

18 4. run the exercise
19 \$ itest
20 Area of circle c1 is: 3.14159
21 Area of circle c2 is: 12.5664
22 Area of rectangle r1 is: 4
23 Area of triangle t1 is: 0.5
24 Area of triangle t2 is: 2
25 This circle has an area of 3.14159 and a color of undefined
26 This circle has an area of 12.5664 and a color of red
27 Unknown shape has color: blue
28 This triangle has an area of 0.5 and a color of green
29 This triangle has an area of 2 and a color of yellow
30

31 When you run the code, all of the printout should match the above printout
32 exactly.

33 Read the code in the example and apply what you learned from the `cplusplus.com`
34 website. Understand why the example prints out what it does.

35 The next subsection contains some discussion about the example. In particular
36 it will discuss the *explicit* and *override* keywords.

31.9.3 Discussion

The heart of this example is the base class `Shape`, found in `Shape.h` and `Shape.cc`. This class illustrates the following ideas:

1. it has a data member named `color_`, which describes an attribute that is common to all shapes. This data member is protected so it is visible to derived classes.
2. the two constructors guarantee that the `color_` data member will be initialized whenever a derived class is instantiated.
3. the class has two virtual functions, one of which is pure virtual. Therefore you cannot instantiate an object of type `Shape`.
4. the class provides an implementation for the virtual method `print`.
5. the class provides an accessor for `color_`.

The one argument constructor of `Shape` is declared `explicit`. Since `shape` cannot be constructed, we will use an imaginary class named `T` to illustrate.

The derived class `Circle`:

1. has one data member, the radius of the circle.

31.10 Inheritance Relic

The first line of the class `First`'s declaration is:

```
class First : public art::EDAnalyzer {
```

The fragment `(: public art::EDAnalyzer)` tells the C++ compiler that the class `First` *inherits* from the class named `art::EDAnalyzer` via *public* inheritance^{c0}. “Inheritance is a way of creating new classes which extend the facilities of existing classes by including new data and functions. The class which is extended is known as the *base class* and the result of an extension is known as the *derived class*; the derived class inherits the data and function members of the base class^{c0}.” In the current example `art::EDAnalyzer` is a base class and `First` is a derived class.

The idea of *inheritance* is a very powerful feature of C++ that has many uses, only a few of which are relevant for *art* modules. This discussion should help

^{c0}Inheritance can be either *public* or *private*; the Workbook exercises always use public inheritance.

^{c0}D.M. Capper's *Introducing C++ for Scientists, Engineers and Mathematicians*, Springer-Verlag Limited 1994, Chapter 11

1 you focus on the relevant information if you need to consult C++ references on
2 inheritance.

3 31.11 Pointers

4 C++, like many other computer languages, allows you to define variables that
5 are pointers to information held in other variables. The value of a pointer is
6 the memory address of the information held by the given variable. A native
7 C++ pointer is often referred to as a *bare pointer*. While pointers provide great
8 flexibility for producing fast, efficient algorithms, they are also easy to misuse.
9 *art* has been designed so that user code will rarely, if ever, interact with *art*
10 via bare pointers; when pointer-like behaviour is required, *art* will provide that
11 information inside a wrapper that is generically referred to as a *smart pointer*
12 or a *safe pointer*; *art* defines different sorts of smart pointers for use in different
13 circumstances. The job of a smart pointer is to recognize misuse and to protect
14 against it. One commonly used type of smart pointer is called a *handle*.

15 31.12 RootOutput and table of event IDs

16 When RootOutput writes a file, it writes the event information to the file and
17 it also writes a table of event IDs that allows it to random access a single event
18 without needing to read all of the events before it. This table is kept in order of
19 increasing event id. When you open a file and read it, RootInput starts reads
20 events in the order found in the table.

21 31.13 Troubleshooting

22 (Section 7.3) `setup` returns the error message

23 You are attempting to run ```setup``` which requires administrative
24 privileges, but more information is needed in order to do so.

25 The simplest solution is to log out and log in again.

1

Part IV

2

Index

