Intensity Frontier
Common Offline Documentation: *art* Workbook and Users Guide

Alpha Release 0.20

July 5, 2013

Scientific Computing Division

Future Programs and Experiments Department

Scientific Software Infrastructure Group

Principal Author: Rob Kutschke
Editor: Anne Heavey

Contents

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 *art* 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 *art* framework (*art* 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 *module*

art path — a FHiCL sequence of *art* moduleLabels that specifies the work the job will do

artdaq — a toolkit that lives on top of *art* 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 *artdaq* 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 *exception* in a C++ reference

cetbuildtools — a build system developed at Fermilab

CETLIB — a utility library used by *art* (developed and maintained by the *art* 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 *classes*. Classes define types of data structures and the functions that operate on those data structures. Instances of these data types are known as *objects*. 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 *run-time configuration*

const member function — a member function of a class that does not change the value of non-mutable data members; see *mutable data member*

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 aquisition system

data handling

Data Model — see *Event Data Model*

data product — Experiment-defined class that can represent detector signals, reconstructed data, simulated events, etc. In *art*, a *data product* is the smallest unit of information that can be added to or retrieved from an event.

data type — See *type*

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 (∼), 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 *art* (e.g., for module types) meaning *event-data*

EDAnalyzer — see *analyzer module*

EDFilter — see *filter module*

EDOutput — see *output module*

EDProducer — see *producer module*

EDSource — see *source module*

Event — In HEP there are two notions of the word *event* that are in common use; see *event (unit of information)* or *event (interaction)*. In this documentation suite, unless otherwise indicated, we mean the former.

Event (interaction) — An *event (unit of data)* 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 *event* 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 *event (unit of information)*. Similarly, in a colliding-beam experiment, an *event (unit of information)* 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 *event* is a set of raw data associated in time, plus any information computed from the raw data; *event* may also refer to a simulated version of same. Within *art*, the representation of an *event (unit of information)* is the classs `art::Event`, which is the smallest unit of information that *art* can process. An `art::Event` contains an event identifier plus an arbitrary number of *data-products*; the information within the *data-products* is intrinsically experiment dependent and is defined by each experiment. For bookkeeping convenience, *art* groups events into a heirarchy: a *run* contains zero or more *subRuns* and a *subRun* 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 *art* 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 metadata that accompanies them. The HEP software community has adopted the word *event-data* to refer to the software details of dealing with the information found in *events*, 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 *catch*

experiment code — see *user code*

external product — for a given experiment, this is a software product that the experiment's software (within the *art* 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 *UPS products* or simply *products*.

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 *art* team at Fermilab to support run-time configuration for several projects, including *art*

FHiCL-CPP — the C++ toolkit used to read FHiCL documents within *art*

filter module — an *art* module that may alter the flow of processing modules within an event; it may add information to the event

framework (*art*) — The *art* 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. *art* 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., *art*, will be tailored to the given needs.

free function — a function without data members; it knows only about agruments passed to it at run time; see *function* and *member function*

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 *art* 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 *pointer*,*data product*

IF — Intensity Frontier

ifdh_sam — a UPS product that allows *art* to use *SAM* as an external runtime 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 *header* file (.h or .hh), the implementation usually resides in a separate source code file (.cc) that "#includes" the header file

instance — see *instantiation*

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 *Kerberos*

member function — (also called *method*) a function that is defined within (is a *member* 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 *art* and experiments' code that provides facilities for merging messages with a variety of severity levels, e.g., informational, error, and so on; see also *mf*

message service

method see *member function*

mf a namespace that holds classes and functions that make up the message facility used by *art* and by experiments that use *art*; see *message facility*

module a C++ class that obeys certain rules established by *art* and whose source code file gets compiled into a shared object library that can be dynamically loaded by *art*. An *art* 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 *moduleLabel*

module_type a keyword known to *art* in the parameter set describing an *art* module; it specifies the name of a shared library to be loaded

moduleLabel a user-defined identifier whose value is a parameter set that *art* will use to configure a module; see *module* and *parameter set*

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 *art* code and related code and documentation

ROOT — an HEP data management and data presentation package used by *art* and supported by CERN; *art* is designed to allow output of event-data to files in ROOT format, in fact currently it is the only output format that *art* implements

ROOT files — There are two types of ROOT files managed by *art*: (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 *art* it is supplied by a FHiCL file; see *FHiCL*

safe pointer — see *pointer*

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 *sequence* (as distinct from a *table*)

service — in *art*, a singleton-like object (type) whose lifetime and configuration are managed by *art*, and which can by accessed by module code and by other services by requesting a *service handle* to that particular service. The service *type* 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 *T File Service*

shared library

signature (of a function) — the unique identifier of a C++ a 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 *art* documentation, a *site* 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 *pointer*

source — (refers to a *data* 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 *source code*; see also *EDsource*

source code — code written in C++ (the programming language used with *art*) that requires compilation and linking to create an executable program

source module — an *art* module that can initiate an *art* 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 *public* (instead of *private*) 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 *run*; a subRun contains zero or more *events*

table (in FHiCL) — a group of FHiCL definitions delimited by braces ({ ... }) is called a *table*; within *art*, a FHiCL table gets turned into an object called a *parameter set*. Consequently, a FHiCL table is typically called a *parameter set*. See *parameter set*.

TFileService — an *art* service used by all experiments to give each module a ROOT subdirectory in which to place its own histograms, TTrees, and so on; see *TTrees* and *ROOT*

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 *truth information*, the *Monte Carlo truth* or the *God's block*.

TTrees — a ROOT implementation of a tree; see *tree* and *ROOT*

type — variables and objects in C++ must be classified into *types*, e.g., built-in types (integer, boolean, float, character, etc.), more complex user-defined classes/structures and typedefs; see *class*, *struct*, and *typedef*. The word *type* in the context of C++ and *art* is the same as *data type* unless otherwise stated.

UPS/UPD — a Fermliab-developed system for distributing software products

user code — experiment-specific and/or analysis-specific C++ code that uses the *art* framework; this includes any personal code you write that uses *art*.

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

List of Tables

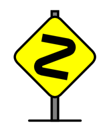
Listings

Part I

Introduction

1 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 2.4)!

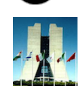


In some places, however, it will be necessary for a paragraph or two to be written for experts. Such paragraphs will be marked with a "dangerous bends" symbol in the margin, as shown at right. Beginners can skip these sections on first reading and come back to them at a later time.

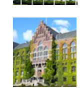
The first instance of each term that is defined in the glossary is written in *italics* followed by a γ (Greek letter gamma), e.g., *framework*(γ).

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 (< ... >).

Unix commands that are continued onto a second line use a single backslash as the last character in the first line (just before a carriage return; no spaces may follow it). This convention is used in this documentation, as well.

Computer output from a command is shown in `typewriter font`.

2 Introduction to the art Event Processing Framework

2.1 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 ($DAQ(\gamma)$) 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."

2.2 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.

2.3 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 https://cdcvs.fnal.gov/redmine/projects/gm2public/wiki/CPP2011

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

2.4 Getting Help

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

2.5 Overview of the Documentation Suite

When complete, this documentation suite will contain several principal components, or *volumes*: the introduction that you are reading now, a Workbook, a Users Guide, a Reference Manual, a Technical Reference and a Glossary. At the time of writing, drafts exist for the Workbook, the Users Guide and the Glossary. The components in the documentation suite are illustrated in Figure 2.1.

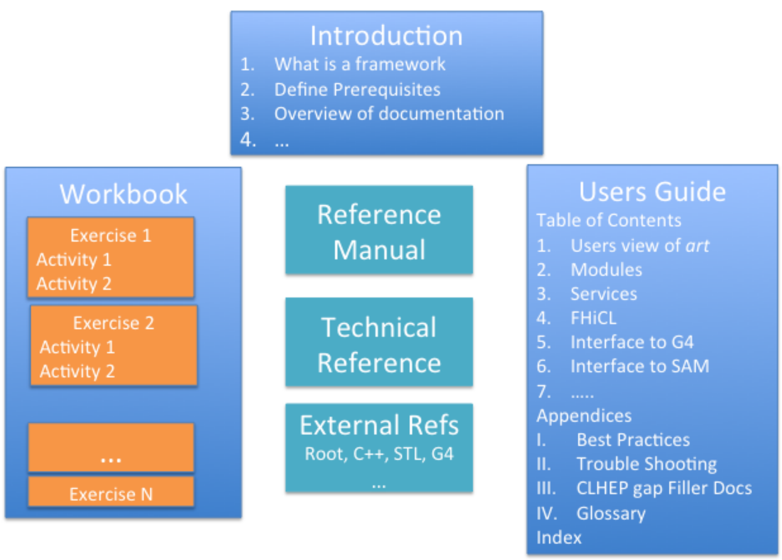


Figure 2.1: The principal components of the *art* documentation suite

2.5.1 The Introduction

This introductory volume is intended to set the stage for using *art*. It introduces *art*, provides background material, describes some of the software tools on which *art* depends, describes its interaction with related software and identifies prerequisites for successfully completing the Workbook exercises.

2.5.2 The Workbook

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

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

The Workbook assumes some basic computing skills and some basic familiarity with the C++ computing language; Chapter 5 provides a tutorial/refresher for readers whose C++ skills aren't quite up-to-speed.

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

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

2.5.3 Users Guide

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

2.5.4 Reference Manual

The Reference Manual will be targeted at physicists who already understand the major ideas underlying *art* and who need a compact reference to the Application Programmer Interface (*API*(γ)). The Reference Manual will likely be generated from annoted source files, possibly using *Doxygen*(γ).

2.5.5 Technical Reference

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

2.5.6 Glossary

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

2.6 Some Background Material

This section defines some language and some background material about the *art* framework that you will need to understand before starting the Workbook.

2.6.1 Events and Event IDs

In almost all HEP experiments, the core idea underlying all bookkeeping is the *event*(γ). 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 [1]. Typically these time intervals are a few nanoseconds to a few tens of mircoseconds. 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 the smallest unit of data that *art* can process at one time.

[1] There 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.

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 a critical requirement imposed by *art* that each event be uniquely identifable by its event ID. This requirement also applies to simulated events.

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

- *run*(γ) number
- *subRun*(γ) number
- *event*(γ) 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 (0 not 1 for historical reasons) and the run number will be incremented.

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

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

When an experiment takes data, events read from the DAQ are typically written to disk files, with copies made on tape. *art* imposes only weak constraints on the event sequence within a file. The events in a single subRun may be spread over several files; conversely a single file may contain many runs, each of which contains many subRuns.

2.6.2 *art* Modules and the Event Loop

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

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

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

The simplest command to run *art* looks like:

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

The *run-time configuration file*(γ) is a text file that tells one run of *art* what it should do. Run-time configuration files for *art* are written in the Fermilab Hierarchical Configuration Language *FHiCL* (γ), pronounced “fickle”) and the filenames end in `.fcl`. As you progress through the Workbook, this language and the conventions used in the run-time configuration file will be explained; the full details are available in Chapter 23 of the Users Guide. (The run-time configuration file is often referred to as simply the *configuration file* or even more simply as just the *configuration*(γ).)

When *art* starts up, it reads the configuration file to learn what input files it should read, what user code it should run and what output files it should write. As mentioned above, an experiment’s code (including any code written by individual experimenters) is provided in units called *art modules*. A module is simply a C++ class, provided by the experiment or user, that obeys a set of rules defined by *art* and whose *source code*(γ) file gets compiled into a shared *object*(γ) library that can be dynamically loaded by *art*. These rules will be explained as you work through the Workbook and they are summarized in Section 30.3. [2]

The code base of a typical experiment will contain many C++ classes. Only a small fraction of these will be modules; most of the rest will be ordinary C++ classes that are used within modules[3].

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

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

- at the start of the *art* job
- at the end of the *art* job
- at the start of each run
- at the end of each run
- at the start of each SubRun

[2] Many programming languagues 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 2.6.4 and 2.6.5.

- at the end of each SubRun

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

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

1. calls the *constructor*(γ) of every module in the configuration
2. calls the beginJob *member function*(γ) of every module that provides one
3. reads one event from the input source, and for that event
   (a) determines if it is from a run different from that of the previous event (true for first event in loop)
   (b) if so, calls the beginRun member function of each module that provides one
   (c) determines if the event is from a subRun different from that of the previous event (true for first event in loop)
   (d) if so, calls the beginSubRun member function of each module that provides one
   (e) calls each module's (required) per-event member function
4. moves to the next event and repeats the above per-event steps until it encounters a new subRun
5. closes out the current subRun by calling the endSubRun method of each module that provides one
6. repeats steps 4 and 5 until it encounters a new run
7. closes out the current run by calling the endRun method of each module that provides one
8. repeats steps 3 through 7 until it reaches the end of the source
9. calls the endJob method of each module that provides one
10. calls the *destructor*(γ) of each module

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

2.6.3 Module Types

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

analyzer module(γ) May inspect information found in the event but may not add new information to the event; described in Chapter 26

producer module(γ) May inspect information found in the event and may add new information to the event; described in Chapter 25

filter module(γ) Same functions as a Producer module but may also tell *art* to skip the processing of some, or all, modules for the current event; may also control which events are written to which output; described in Chapter 27.

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

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

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

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

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

2.6.4 *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.

2.6.5 *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 2.6.2 and data products in Section 2.6.4. *art services*(γ) are yet another example of this.

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

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

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

After *art* completes its initialization phase and before it constructs any modules (see Section 2.6.2), it

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

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

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

For those of you familiar with the C++ Singleton Design Pattern, an *art* service has some differences and some similarities to a Singleton. The most important difference is that the lifetime of a service is managed by *art*, which calls the constructors of all services at a well-defined time in a well-defined order. Contrast this with the behavior of Singletons, for which the order of initialization is undefined by the C++ standard and which is an accident of the implementation details of the loader. *art* also includes services under the umbrella of its powerful run-time configuration system; in the Singleton Design pattern this issue is simply not addressed.

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

Table 2.1: 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

2.6.6 Shareable Libraries and *art*

When code is executed within the *art* framework, *art*, not the experiment, provides the main executable. The experiment provides its code to the *art* executable in the form of shareable object libraries that *art* loads dynamically at run time; these libraries are also called *dynamic load libraries* or *plugins* and their filenames are required to end in `.so`. For more information about shareable libraries, see Section 30.5.

2.6.7 Build Systems and *art*

To make an experiment's code available to *art*, the source code must be compiled and linked (i.e., *built*) to produce shareable object libraries (Section 2.6.6). The tool that creates the `.so` files from the C++ source files is called a *build system*(γ).

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

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

The compiler options corresponding to the three levels are listed in Table 2.1.

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

2.6.8 External Products

As you progress through the Workbook, you will see that the exercises use some software packages that are part of neither *art* nor the toy experiment's code. The Workbook code, *art* and the software for your experiment all rely heavily on some external tools and, in order to be an effective user of *art*-based HEP software, you will need at least some familiarity with them; you may in fact need to become expert in some.

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

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

art the event processing framework

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

CETLIB a utility library used by *art*

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

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

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

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

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

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

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

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

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

jobusub_tools(γ) tools for submitting jobs to the Fermigrid batch system and monitoring them.

ifdh_sam(γ) allows *art* to use SAM as an external run-time agent that can deliver remote files to local disk space and can copy output files to tape.

SAM is a Fermilab-supplied resource that provides the functions of a file catalog, a replica manager and some functions of a batch-oriented workflow manager x

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

The external products are made available to your code via a mechanism called UPS, which will be described in Section 6. UPS is, itself, just another external product. From the point of view of your experiment, *art* is an external product. From the point of view of the Workbook code, both *art* and the code for the toy experiment are external products.

Finally, it is important to recognize an overloaded word, *products*. When a line of documentation simply says *products*, it may be refering either to data products or to external products. If it is not clear from the context which is meant, please let us know (see Section 2.4).

2.6.9 The Event-Data Model and Persistency

Section 2.6.4 introduced the idea of *art* data products. In a small experiment, a fully reconstructed event may contain on the order of ten data products; in a large experiment there may be hundreds.

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

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

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

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

A few other related terms that you will encounter include:

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

2.6.10 Event-Data Files

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

When you simulate an experiment and write a disk file that holds the information produced by the simulation, what should you call the file? The Particle Data Group has recommended that this not be called a "data file" or a "simulated data file;" they prefer that the word "data" be strictly reserved for information that comes from an actual experiment. They recommend that we refer to these files as "files of simulated events" or "files of Monte Carlo events" [6]. Note the use of "events", not "data."

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

2.6.11 Files on Tape

Many experiments do not have access to enough disk space to hold all of their event-data files, ROOT files and log files. The solution is to copy a subset of the disk files to tape and to read them back from tape as necessary.

At any given time, a snapshot of an experiment's files will show some on tape only, some on tape with copies on disk, and some on disk only. For any given file, there may be multiple copies on disk and those copies may be distributed

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

across many *sites*(γ), some at Fermilab and others at collaborating laboratories or universities.

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

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

2.7 The Toy Experiment

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

The software for the toy experiment is designed around a toy detector, which is shown in Figure 2.2. The *toyExperiment* code contains many C++ classes: some modules, some data products, some services and some plain old C++ classes. About half of the modules are producers that individually perform either one step of the simulation process or one step of the reconstruction/analysis process. The other modules are analyzers that make histograms and ntuples of the information produced by the producers.

2.7.1 Toy Detector Description

The toy detector is a central detector made up of 15 concentric shells, with their axes centered on the z axis; the left hand part of Figure 2.2 shows an xy view of these shells and the right shows the radius vs z view. The inner five shells are closely spaced radially and are short in z; the ten outer shells are more widely spaced radially and are longer in z. The detector sits in a uniform magnetic field of 1.5 T oriented in the $+z$ direction. The origin of the coordinate system is at the center of the detector. The detector is placed in a vacuum.

Each shell is a detector that measures (φ, z), where φ is the azimuthal angle of a line from the origin to the measurement point. Each measurement has perfectly gaussian measurement errors and the detector always has perfect separation of hits that are near to each other. The geometry of each shell, its efficiency and resolution are all configurable at run-time.

All of the code in the toyExperiment product works in the set of units described in Table 2.2. Because the code in the Workbook is built on toyExperiment, it

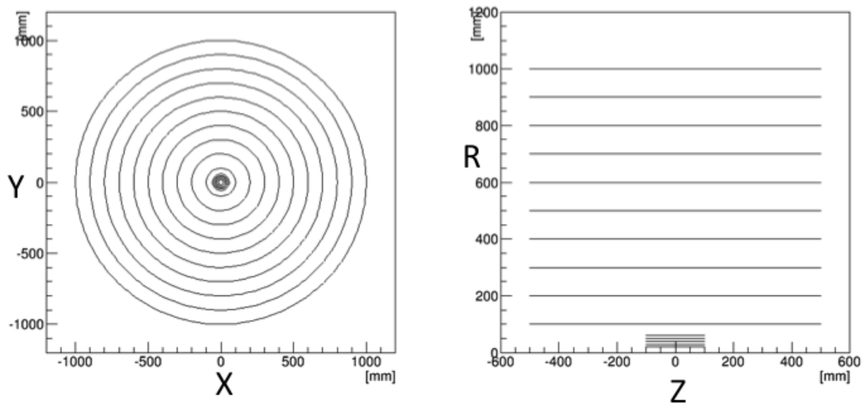


Figure 2.2: The geometry of the toy detector; the figures are described in the text. A uniform magnetic field of strength 1.5 T is oriented in the $+z$ direction.

Table 2.2: 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

uses the same units. *art* itself is not unit aware and places no constraints on which units your experiment may use.

The first six units listed in Table 2.2 are the base units defined by the `CLHEP SystemOfUnits` package. These are also the units used by `Geant4`.

2.7.2 Workflow for Running the Toy Experiment Code

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

1. event generation
2. detector simulation
3. hit-making
4. track reconstruction

5. analysis of the mass resolution

For each event, the event generator creates one particle with the following properties:

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

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

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

Figure 2.3 shows an event display of a typical simulated event. In this event the ϕ meson was travelling almost at 90° to the z axis and it decayed nearly symmetrically; both tracks intersect all 15 detector cylinders. The left-hand figure shows an xy view of the event; the solid lines show the trajectory of the kaons, red for K^+ and blue for K^-; the solid dots mark the intersections of the trajectories with the detector shells. The right-hand figure shows the same event but in an rz view.

Figure 2.4 shows an event display of another simulated event. In this event the K^- is produced with a very shallow trajectory and it does not intersect any detector shells while the K^+ makes five hits in the inner detector and seven in the outer detector. Why does the trajectory of the K^+ end where it does? In order to keep the exercises focused on *art* details, not geometric corner cases, the simulation stops a particle when it completes its outward-going arc and starts to curl back towards the z axis; it does this even if the the particle is still inside the detector.

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

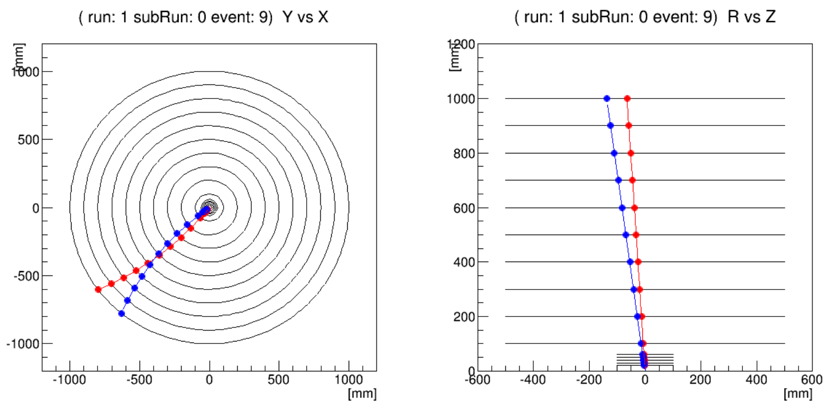


Figure 2.3: Event display of a typical simulated event in the toy detector.

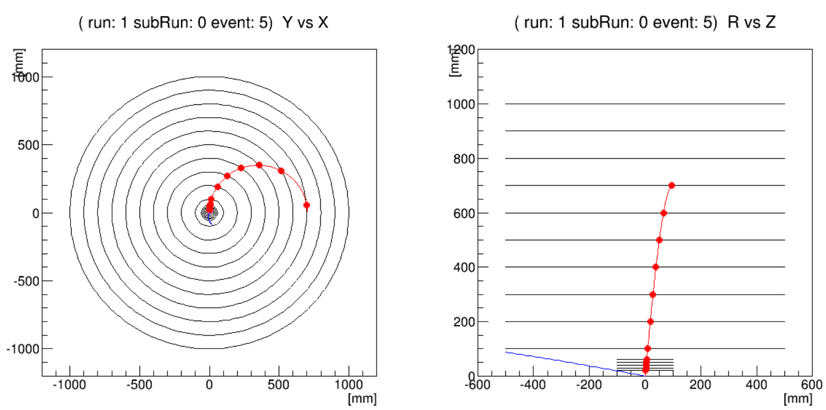


Figure 2.4: Event display of another simulated event in the toy detector; a K^- (blue) is produced with a very shallow trajectory and it does not intersect any detector shells while the K^+ (red) makes five hits in the inner detector and seven in the outer detector

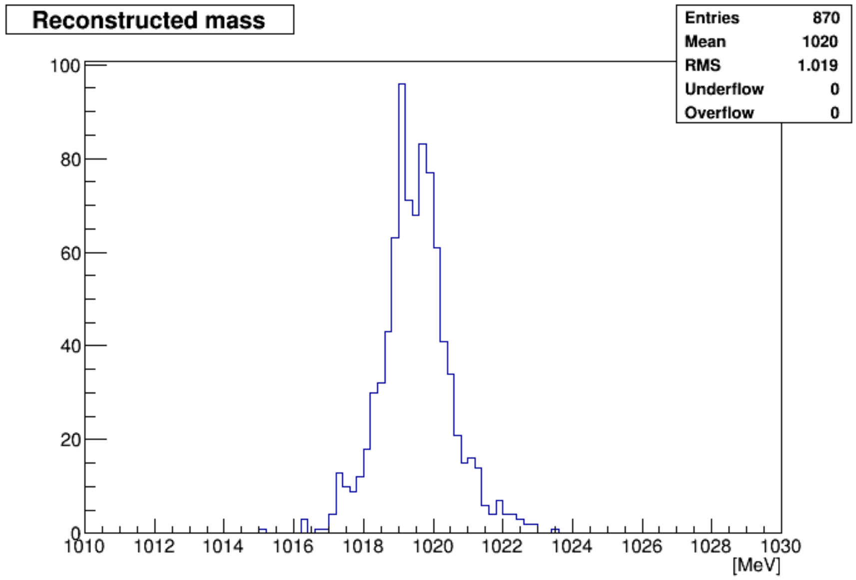


Figure 2.5: The final plot showing 870 reconstructed events out of 1000 generated events

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

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

The fifth step in the workflow does a simulated analysis using the fitted helices from the reconstruction step. It forms all distinct pairs of tracks and requires that they be oppositely charged. It then computes the invariant mass of the pair, under the assumption that both fitted helices are kaons. This module is an analyzer module and does not make any output data product. But it does make some histograms, one of which is a histogram of the reconstructed invariant mass of all pairs of oppositely charged tracks; this histogram is shown in Figure 2.5. When you run the Workbook exercises, you will make this plot and can compare it to Figure 2.5.

2.8 Rules, Best Practices, Conventions and Style

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

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

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

3 Unix Prerequisites

3.1 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.

3.2 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 3.10 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

[1] Remember 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 1.

others:

```
$ <command-name> --help
```

or

```
$ <command-name> -?
```

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

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

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

- filename vs pathname
- absolute path vs relative path
- directories and subdirectories (equivalent to folders in the Windows and Mac worlds)
- current working directory
- home directory (aka login directory)
- `../` notation for viewing the directory above your current working directory
- environment variables (discussed briefly in Section 3.5)
- *paths*(γ) (in multiple senses; see Section 3.6)
- file protections (read-write-execute, owner-group-other)
- symbolic links
- stdin, stdout and stderr
- redirecting stdin, stdout and stderr
- putting a command *in the background* via the `&` character
- pipes

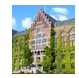
3.3 Shells

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

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

```
$ echo $SHELL
```

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



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

3.4 Scripts: Part 1

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

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

If you would like to learn more about *bash*, some references are listed in Section 3.10.

3.5 Unix Environments

3.5.1 Layering Environments

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

- 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.

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

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

Particular programs (e.g., *art*) usually require extra information (i.e., another environment *layer*) on top of a standard working environment, e.g., paths to the program's executable(s) and to its dependent programs, paths indicating where it can find input files and where to direct its output, and so on. In addition to environment variables, the *art*-enabled computing environment includes some aliases and *bash* functions that have been defined; these are discussed in Section 3.8.

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

Given the different experiments using *art* and the variety of laboratories and universities at which the users work, a *site*(γ) in *art* is a unique combination of *experiment* and *institution*. It is used to refer to a set of computing resources configured for use by a particular experiment at a particular institution. Setting up your site-specific environment will be discussed in Section 3.7.

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

This concept of environment layering is illustrated in Figure 3.1.

3.5.2 Examining and Using Environment Variables

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

```
$ printenv HOME
```

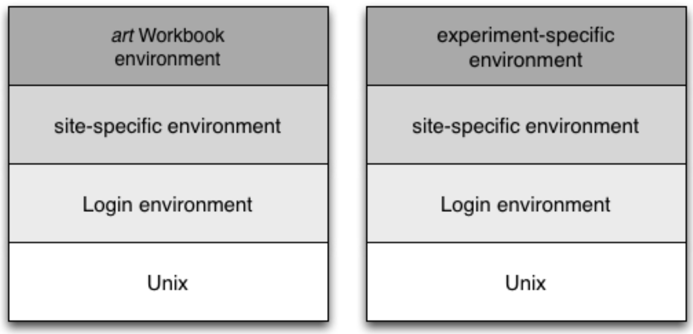


Figure 3.1: Layers in the *art* Workbook (left) and experiment-specific (right) computing environments

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

```
$ echo $HOME
```

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

By convention, environment variables are virtually always written in all capital letters[3].

There may be times when the Workbook instructions tell you to set an environment variable to some value. To do so, type the following at the command prompt:

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

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

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

[3]Another 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.

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

3.7 Scripts: Part 2

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

```
$ file.sh
$ source file.sh
$ . file.sh
```

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

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

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

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

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

If you would like to learn more about bash, some references are listed in Section 3.10.

3.8 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 6, 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`?

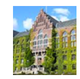
To learn how bash will resolve a particular command, give the bash command:

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

3.9 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 3.5, these scripts modify your `PATH` and define `bash` functions, aliases and environment variables. All of these become part of your environment.

When your account on a Fermilab computer was first created, you were given standard versions of the files `.profile` and `.bashrc`; these files are used by `bash`[4]. You can read about login scripts in any standard bash reference. You may add to these files but you should not remove anything that is present.



If you are working on a non-Fermilab computer, inspect the login scripts to understand what they do.

It can be useful to inspect the login scripts of your colleagues to find useful customizations.

If you read generic Unix documentation, you will see that there are other login scripts with names like, `.login`, `.cshrc` and `.tcshrc`. These are used by the `csh` family of shells and are not relevant for the Workbook exercises, which require the `bash` shell.

3.10 Suggested Unix and bash References

The following cheat sheet provides some of the basics:

- http://mu2e.fnal.gov/atwork/computing/UnixHints.shtml

A more comprehensive summary is available from:

- http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/GNU-Linux-Tools-Summary.html

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

- BASH Programming - Introduction HOW-TO
  http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
- Bash Guide for Beginners
  http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
- Advanced Bash Scripting Guide
  http://www.tldp.org/LDP/abs/html/abs-guide.html

The first of these is a compact introduction and the second is a more comprehensive introduction.

The above guides were all found at the Linux Documentation Project: Workbook:

- http://www.tldp.org/guides.html

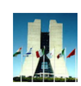
[4]These files are used by the `sh` family of shells, which includes`bash`.

4 Site-Specific Setup Procedure

Section 3.5 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 (NOνA being a notable exception), the procedure involves only sourcing a shell script (recall the discussion in Section 3.7). 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 (generally it is a single command to source a script) 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 4.1 lists the site-specific setup command for each experiment. You will run the command when you get to Section 8.4.

The table gives two options for Mu2e; both are equivalent. The first option,

```
$ setup mu2e
```

simply redirects to

```
$ source /grid/fermiapp/products/mu2e/setupmu2e-art.sh
```

We recommend that Mu2e members adopt the habit of using `setup mu2e`. The other Intensity Frontier experiments will be making a similar change in the near future.

Table 4.1: Site-specific setup procedure for $IF(\gamma)$ Experiments at Fermilab

Experiment	Site-Specific Setup Command
ArgoNeut	
Darkside	`source /ds50/app/ds50/ds50.sh`
LBNE	(in development) `setup lbne`
MicroBoone	`. /grid/fermiapp/products/uboone/etc/setups.sh`
Muon g-2	`source /gm2/app/software/prod/g-2/setup`
Mu2e	`setup mu2e` (preferred) `source /grid/fermiapp/products/mu2e/setupmu2e-art.sh` (deprecated)

NOνA	`source /grid/fermiapp/nova/novaart/novasvn/srt/srt.sh`
	`export EXTERNALS=/nusoft/app/externals`
	`source $SRT_DIST/setup/setup_novasoft.sh -b maxopt`

5 Get your C++ up to Speed

5.1 Introduction

’

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

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

A typical program consists of many source code files, each of which contains a human-readable description of one component of the program. In the Workbook, you will see source code files written in the C++ computer language; these files have names that end in `.cc`. In C++, there is a second sort of source code file, called a *header file* that ends in `.h`; in most, but not all, cases for every file ending in `.cc` there is another file with the same name but ending in `.h`. Header files can be thought of as the "parts list" for the corresponding `.cc` file; you will see how these are used in Section 5.4.

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

It is often convenient to collect related groups of `.o` files and put them into *libraries*. There are two kinds of library files, static libraries, whose names end in `.a` and shared libraries whose names end in `.so`. Putting many `.o` files into a single library allows you to use them as a single coherent entity. We will defer further discussion of libraries until more background information has been provided.

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

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

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

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

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

This chapter is designed around a handful of exercises, each of which you will first build and run, then "pick apart" to understand how the results were obtained.

5.2 Establishing the Environment

5.2.1 Initial Setup

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

1. Log into the node that you will use for Workbook exercises.
2. Follow the site-specific setup procedure from Table 4.1.
3. Create an empty working directory and `cd` to it.
4. Run these commands to copy a gzipped tar file from the web, unpack it, and get a directory listing:

```
$ wget http://artdoc.fnal.gov/C++UpToSpeed.tar.gz
$ tar xzf C++UpToSpeed.tar.gz
$ rm C++UpToSpeed.tar.gz
$ ls
BasicSyntax Build Classes Libraries setup.sh
```

5. Source the `setup.sh` script to select the correct compiler version and define a few environment variables that will be used later in these exercises:
   `$ source setup.sh`

After these steps, you are ready to begin the exercise in Section 5.3.

5.2.2 Subsequent Logins

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

1. Log into the node that you will normally use.
2. Follow the site-specific setup procedure.
3. `cd` to the working directory you created in Section 5.2.1.
4. `$ source setup.sh`
5. `cd` to the directory that contains the exercise you want to work on.

5.3 C++ Exercise 1: The Basics

5.3.1 Concepts to Understand

This section provides a program that illustrates the parts of C++ that are assumed knowledge for the Workbook material. If you do not understand some of the code in this example program, consult any standard C++ reference; several are listed in Section 5.7.

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

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

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

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

The above list explicitly does not include classes, objects and inheritance, which will be discussed in Sections 5.6 and 30.9.

5.3.2 How to Compile, Link and Run

In this section you will learn how to compile, link and run the small C++ program that illustrates the features of C++ that are considered prerequisites. The main discussion of the details of compiling and linking will be deferred until Section 5.4.

We don't offer a lot of details up front; more will follow in Sections 5.3.5 and 5.3.4. The idea here is to get used to the steps and see what results you get.

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

1. If not yet done, log in and establish the working environment (Section 5.2).
2. List the starting set of files:
   ```
   $ cd BasicSyntax/v1/
   $ ls
   build t1.cc t1_example.log
   ```
   The file `t1.cc` contains the source code of the *main program*, which is a function called `main() { ...}`. The file `build` is a script that will

compile and link the code. The file t1_example.log is an example of the output expected when you run t1.

3. Compile and link the code; then look at a directory listing:
   ```
   $ build
   t1.cc:  In function int main():
   t1.cc:43:26:  warning:  k may be used uninitialized in
   this function [-Wuninitialized]
   $ ls
   build t1 t1.cc t1_example.log
   ```

   The script named build compiles and links the code, and produces the executable file t1. The warning message, issued by the compiler, also comes during this step.

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

5.3.3 Suggested Homework

1. Compare your output with the standard example:
   ```
   $ diff t1.log t1_example.log
   ```

   There will almost certainly be a handful of differences.

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

If you don't understand something, consult a standard C++ reference; see Section 5.7. A few of your questions might also be answered in Section 5.3.4.

5.3.4 Discussion

Why do we expect several of the lines of the output to be different from those in t1_example.log? There are two classes of answers: (1) an uninitialized variable and (2) variation in variable addresses from run to run.

In t1.cc, the line

```
int k;
```

declares that k is a variable whose type is int but it does not initialize the variable. Therefore the value of the variable k is whatever value happened to be sitting in the memory location that the program assigned to k. Each time that the program runs, the operating system will put the program into whatever region of memory makes sense to the operating system; therefore the address of any variable, and thus the value returned, may change unpredictably from run to run.

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

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

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

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

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

5.3.5 How was this Exercise Built?

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

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

This turned the source file `t1.cc` into an executable program, named `t1` (the argument to the `-o` (for "output") option). We will discuss compiling and linking in Section 5.4.

5.4 C++ Exercise 2: About Compiling and Linking

5.4.1 What You Will Learn

In the previous exercise, the entire program was found in a single file and the build script performed compiling and linking in a single step. For all but the smallest programs, this is not practical. It would mean, for example, that you would need to recompile and relink everything when you made even the smallest change anywhere in the code; generally this would take much too long. To address this, some computer languages, including C++, allow you to break up a large program into many smaller files and rebuild only a small subset of files when you make changes in one.

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

5.4.2 The Source Code for this Exercise

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

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

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

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

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

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

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

```
float function( float );
```

This line is called the *declaration*(γ) of the function. It says (1) that the identifier `function` is the name of a function that (2) takes an argument of type `float` (the "float" inside the parentheses) and (3) returns a value of type `float` (the “float” at the start of the line). The file `t1.cc` includes this header file, thereby giving the compiler these three pieces of information it needs to know about `function`.

The other three lines in `function.h` are *code guards*, described in Section 30.8. In brief, they deal with the following scenario: suppose that we have two header files, `A.h` and `B.h`, and that `A.h` includes `B.h`; there are many scenarios in which it makes good sense for a third file, either `.h` or `.cc`, to include both `A.h` and `B.h`. The code guards ensure that, when all of the includes have been expanded, the compiler sees exactly one copy of `B.h`.

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

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

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

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

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

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

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

5.4.3 Compile, Link and Run the Exercise

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

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

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

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

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

```
$ build
$ ls
build build2 function.cc function.h function.o t1 t1.cc
t1.o
```

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

3. Run the exercise:

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

This matches the expected printout.

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

1. It compiles the main program, `t1.cc`, into the object file (with the default name) `t1.o` (which will now be the thing that the term *main program* refers to):
   `c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c t1.cc`
2. It (separately) compiles `function.cc` into the object file `function.o`:
   `c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c function.cc`
3. It links `t1.o` and `function.o` to form the executable program `t1` (the name of the main program is the argument of the `-o` option):
   `c++ -std=c++11 -o t1 t1.o function.o`

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

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

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

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

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

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

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

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

The job of the linker (also invoked by the command `c++`) is to play matchmaker. First it inspects the symbol tables inside all of the object files listed on the command line and looks for a linker symbol that defines the location of the main program. If it cannot find one, or if it finds more than one, it will issue an error message and stop. In this example

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

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

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

5.4.4 Alternate Script `build2`

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

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

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

```
$ build2
$ ls
build build2 function.cc function.h t1 t1.cc
```

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

4. Execute the program that you just built

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

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

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

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

5.4.5 Suggested Homework

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

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

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

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

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

The first homework exercise will issue the diagnostic:

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

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

The second homework exercise will issue the diagnostic:

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

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

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

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

```
  float tmp = a;
```

The compiler will then use this temporary variable as the argument of the function. Consult the standard C++ documentation to understand when automatic type conversions may occur; see Section 5.7.

5.5 C++ Exercise 3: Libraries

Multiple compiled object code files can be grouped into a single file known as a *library*, obviating the need to specify each and every object file when linking; you can reference the libraries instead. This simplifies the multiple use and sharing of software components. Components that are large can be created for dynamic use, thus allowing the library to remain separate from the executable, reducing its size and thus the disk space used. The library components are called when needed. [1]

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

[1] The text in this section's introduction is abridged from http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html.

- static libraries of object code (filenames for which end in `.a`) that are linked with, and become part of, the application (*art* does not use static libraries)
- dynamically linked, shared object libraries (filenames end in `.so`): These can be used in two ways.
  - Dynamically linked at run time but statically aware. The libraries must be available during the compile/link phase. The shared objects are not included in the executable component but are tied to the execution.
  - Dynamically loaded/unloaded and linked during execution (i.e., similar to browser plug-in) using the dynamic linking/loader system functions.

5.5.1 What You Will Learn

In this section you will repeat the example of Section 5.4 with a variation. You will create an object library, insert `function.o` into that library and use that library in the link step. This pattern generalizes easily to the case that you will encounter in your experiment software, where object libraries will typically contain many object files.

5.5.2 Building and Running the Exercise

To perform this exercise, do the following:

1. Log in and establish your working environment (Section 5.2).
2. `cd` to your working directory.
3. `cd` to the directory for this exercise and get a directory listing:
   ```
   $ cd Libraries/v1
   $ ls
   build build2 build3 function.cc function.h t1.cc
   ```
   The three files, `function.cc`, `function.h` and `t1.cc` are identical to those from the previous exercise. The three files, `build`, `build2` and `build3` are scripts that show three different ways to build the main program in this exercise.
4. Compile and link this exercise using `build`, then compare the directory listing to that taken pre-build:
   ```
   $ build
   $ ls
   build build3 function.h libpackage1.a t1.cc
   ```

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

5. Execute the main program:
```
$ ./t1
a = 3
function(a) 6
```

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

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

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

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

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

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

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

1. remove files built by `build1`
   ```
   $ rm function.o t1.o libpackage1.a t1
   ```
2. build the code with `build2` and look at the directory contents
   ```
   $ build2
   $ ls
   ```

```
build build3 function.h libpackage1.a t1.cc
build2 function.cc function.o t1 t1.o
```

3. run `t1` as before

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

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

while that from `build2` is:

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

In the script `build`, the path to the library, relative or absolute, is written explicitly on the command line. In the script `build2`, two new elements are introduced. The command line may contain any number of `-L` options; the argument of each option is the name of a directory. The ensemble of all of the `-L` options forms a search path to look for named libraries; the path is searched in the order in which the `-L` options appear on the line. The names of libraries are specified with the `-l` options (this is a lower case letter L, not the numeral one); if a `-l` option has an argument of `XXX` (or `package1`), then the linker with search the path defined by the `-L` options for a file with the name `libXXX.a` (or `libpackage1.a`).

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

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

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

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

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

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

2. build the code with `build2` and look at the directory contents
   ```
   $ build3
   $ ls
   build build3 function.h libpackage1.a t1.cc
   build2 function.cc function.o t1
   ```
3. run `t1` as before

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

5.6 Classes

5.6.1 Introduction

The comments in the sample program used in Section 5.3 emphasized that every variable has a type: `int`, `float`, `std::string`, `std::vector<std::string>`, and so on. One of the basic building blocks of C++ is that users may define their own types; user-defined types may be built-up from all types, including other user-defined types.

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

In general, a class contains both a *declaration* (what it consists of) and an *instantiation*(γ) (what to do with the parts). The declaration contains some data (called *data members* or *member datum*) plus some functions (called *member functions*) that will (when instantiated) operate on that data, but it is legal for a class declaration (and therefore, a class) to contain only data or only functions. A class *declaration* has the form shown in Listing 5.1.

Listing 5.1: The form of a class declaration

```
class MyClassName{

  // required: declarations of all members of the class
  // optional: definitions of some members of the class

};
```

The string *class* is a keyword that is reserved to C++ and may not be used for any user-defined *identifiers*.[2] This construct tells the C++ compiler that `MyClassName` is the name of a class; everything that is between the braces is part of the class declaration. The remainder of Section 5.6 will give many examples of *members* of a class.

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

The upcoming sections will illustrate some features of classes, with an emphasis on features that will be important in the earlier Workbook exercises. This is not indended to be a comprehensive description of classes. To illustrate, we will show nine versions of a class named `Point` that represents a point in a plane. The first version will be simple and each subsequent version will add features.

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

5.6.2 C++ Exercise 4 v1: The Most Basic Version

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

To build and run this example:

1. Log in and follow the follow the steps in Section 5.2.
2. `cd` to the directory for this exercise and examine it
   ```
   $ cd Classes/v1/
   $ ls
   Point.h ptest.cc
   ```
   Within the subdirectory `v1` the main program for this exercise is the file `ptest.cc`. The file `Point.h` contains the first version of the class `Point`; shown in Listing 5.2.
3. Build the exercise.
   ```
   $ ../build
   $ ls
   Point.h ptest ptest.cc
   ```
   The file named `ptest` is the executable program.
4. Run the exercise.
   ```
   $ ./ptest
   p0:  (2.31827e-317, 0)
   p0:  (1, 2)
   ```

[2] 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.

```
    p1:  (3, 4)
    p2:  (1, 2)
    Address of p0:  0x7fff883fe680
    Address of p1:  0x7fff883fe670
    Address of p2:  0x7fff883fe660
```

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

Look at the header file `Point.h` which shows the basic version of the class `Point`. The three lines starting with # make up a code guard, described in Section 30.8.

Listing 5.2: The contents of `v1/Point.h`

```
#ifndef Point_h
#define Point_h

class Point {
 public:
  double x;
  double y;
};

#endif /* Point_h */
```

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

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

Look at the function `main()`) (the *main program*) in `ptest.cc`, which illustrates the use of the class `Point`; see Listing 5.3. This file includes `Point.h` so that the compiler will know about the class `Point` when it begins execution. It also includes the C++ header `<iostream>` which enables printing with `std::cout`.

Listing 5.3: The contents of `v1/ptest.cc`

```
#include "Point.h"

#include <iostream>

```

```
5  int main() {
6
7    Point p0;
8    std::cout << "p0:␣(" << p0.x << ",␣" << p0.y << ")" << std::endl;
9
10   p0.x = 1.0;
11   p0.y = 2.0;
12   std::cout << "p0:␣(" << p0.x << ",␣" << p0.y << ")" << std::endl;
13
14   Point p1;
15   p1.x = 3.0;
16   p1.y = 4.0;
17   std::cout << "p1:␣(" << p1.x << ",␣" << p1.y << ")" << std::endl;
18
19   Point p2 = p0;
20   std::cout << "p2:␣(" << p2.x << ",␣" << p2.y << ")" << std::endl;
21
22   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 }
```

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

```
Point p0;
```

This declares that `p0` is the name of a variable whose type is (the class) `Point`. When this line of code is executed, the program will ensure that memory has been allocated[3] to hold the data members of `p0`. If the class `Point` contained code to initialize data members then the program would also run that, but `Point` does not have any such code. Therefore the data members take on whatever values happened to preexist in the memory that was allocated for them.

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

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

An important take-away from the above is that a *variable* is an identifier in a source code file while an *object* is something that exists in the computer memory. Most of the time a one-to-one correspondence exists betweeen variables in the

[3] 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.

source code and objects in memory. There are exceptions, however, for example, sometimes a compiler needs to make anonymous temporary objects that do not correspond to any variable in the source code, and sometimes two or more variables in the source code can refer to the same object in memory.

Line 8 (shown split here):

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

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

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

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

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

The last section of the main program (and of `ptest.cc` itself), lines 22-24, prints the address of each of the three objects, `p0`, `p1` and `p2`. The addresses are represented in hexadecimal (base 16) format. On almost all computers, the length of a `double` is eight bytes. Therefore an object of type `Point` will have a length of 16 bytes. If you look at the printout made by `ptest` you will see that the addresses of `p0`, `p01` and `p2` are separated by 16 bytes; therefore the three objects are contiguous in memory.

Figure 5.1 shows a diagram of the computer memory at the end of running `ptest`; the outer box (blue outline) represents the memory of the computer; each filled colored box represents one of the three objects in this program. The diagram shows them in contiguous memory locations, which is not necessary; there could have been gaps between the memory locations in Figure 5.1.

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

1. a *state*, given by the values of its data members
2. the ability to have operations performed on it: e.g., setting/reading in value of a data member, assigning value of object of a given type to another of the same type
3. an *identity*: a unique address in memory

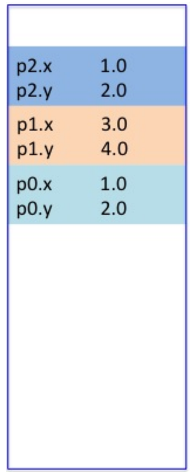


Figure 5.1: Memory diagram at the end of a run of Classes/v1/ptest.cc

5.6.3 C++ Exercise 4 v2: The Default Constructor

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

To build and run this example:

1. Log in and follow the follow the steps in Section 5.2.
2. Go to the directory for this exercise:
   ```
   $ cd Classes/v2
   $ ls
   Point.cc Point.h ptest.cc
   ```
   In this example, `Point.cc` is a new file.
3. Build the exercise:
   ```
   $ ../build
   $ ls
   Point.cc Point.h ptest ptest.cc
   ```
4. Run the exercise:
   ```
   $ ptest
   p0:  (0, 0)
   p0:  (3.1, 2.7)
   ```

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

Look at `Point.h`. There is one new line in the body of the class declaration:

```
Point();
```

The parentheses tell you that this new member is some sort of function. A C++ class may have several different kinds of functions. A function that has the same name as the class itself has a special role and is called a *constructor*; if a constructor takes no arguments it is called a *default constructor*. In informal written material, the word constructor is sometimes written as *c'tor*.

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

Look at the file `Point.cc`. It "includes" the header file `Point.h` because the compiler needs to know all about this class before it can compile the code that it finds in `Point.cc`. The rest of the file contains a *definition* of the constructor. The syntax `Point::` says that the function to the right of the `::` is part of (a member of) the class `Point`. The body of the constructor gives initial values to the two data members, `x` and `y`.

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

```
Point p0;
```

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

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

In the previous example, `Classes/v1/ptest.cc`, the following steps formally took place. When a class does not contain a default constructor, the compiler will write one for you; this default constructor simply default constructs each of the data members. The default constructor of the built-in type `double` does nothing, leaving the data member uninitialized. The compiler knew all of this and almost certainly did not waste time writing and calling do-nothing constructors; it simply made sure that the memory was allocated. This discussion is presented here since it would have sounded silly to say all of that before giving you an example of a real default constructor.

5.6.4 C++ Exercise 4 v3: Constructors with Arguments

This exercise introduces three new ideas:

1. constructors with arguments
2. the copy constructor

3. single phase construction vs two phase construction

To build and run this exercise, cd to the directory `Classes/v3` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program, you should see the following output:

```
$ ptest
p0: (1, 2)
p1: (1, 2)
```

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

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

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

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

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

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

This line declares the variable `p0` and initializes it by calling the new constructor defined in this section. The next line prints the value of the data members.

The next line of code

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

introduces the *copy constructor*, which is another constructor that can be written by the compiler if the user chooses not to provide one. This exercise did not provide a copy constructor so the compiler-written one was used; that version simply does a copy, data member by data member, from `p0` to `p1`. The next line prints the values of the data members of `p1` and you can see that the copy constructor worked as expected.

For any class whose data members are either built-in types or simple aggregates of built-in types, you should usually let the compiler write the copy constructor for you. `Point` is an example of such a class. If your class has data members that are pointers, or data members that manage some external resource, such as a file that you are writing to, then you will very likely need to write your own copy constructor. There are some other cases in which you should write your own copy constructor, but discussing them here is beyond the scope of this document. When you need to write your own copy constructor, you can learn how to do so from any standard C++ reference; see Section 5.7.

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

```
Point p0;
p0.x = 3.1;
p0.y = 2.7;
```

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

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

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

1. Single-phase construction more clearly conveys the intent of the programmer: the intent is to initialize the object `p0`. The second version says this directly. In the first version you needed to do some extra work to recognize that the three lines quoted above formed a logical unit distinct from the remainder of the program. This is not difficult for this simple class, but it can become so with even a little additional complexity.

2. Two-phase construction is less robust. It leaves open the possibility that a future maintainer of the code might not recognize all of the follow-on steps that are part of construction and will use the object before it is fully constructed. This can lead to difficult-to-diagnose run-time errors.

5.6.5 C++ Exercise 4 v4: Colon Initializer Syntax

This version of the class `Point` introduces *colon initializer syntax* for constructors.

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

```
$ ptest
p0: (1, 2)
p1: (1, 2)
```

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

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 rewritten using colon initializer syntax. The rules for the colon-initializer syntax are:

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

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

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

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

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

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

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

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

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

5.6.6 C++ Exercise 4 v5: Member functions

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

To build and run this exercise, cd to the directory `Classes/v5` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program you should see the following output:

```
$ ptest
Before p0: (1, 2)  Magnitude: 2.23607  Phi: 1.10715
After  p0: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
```

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

```
  double mag() const;
  double phi() const;
  void scale( double factor );
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. In `ptest.cc`, change the first line to
   `Point const p0(1,2);`
   Then build the code again; you should see the following diagnostic message:

   ```
   ptest.cc: In function int main():
   ptest.cc:13:14: error: no matching function for call to
   Point::scale(double) const
   ptest.cc:13:14: note: candidate is:
   In file included from ptest.cc:1:0:
   Point.h:13:8: note: void Point::scale(double) <near match>
   Point.h:13:8: note:   no known conversion for implicit this
   parameter from const Point* to Point*
   ```

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

   ```
   double mag();
   ```

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

   ```
   Point.cc:12:8: error: prototype for double Point::mag() const
   does not match any in class Point
   In file included from Point.cc:1:0:
   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:

   ```
   Point.cc:12:8: error: prototype for double Point::mag()
   does not match any in class Point
   In file included from Point.cc:1:0:
   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 of `Point::mag()` and that is absent in the definition of the member function `Point::scale()`. The contract says that the definition of `Point::mag()` may not modify the values of any data members of the class `Point`; users of the class `Point` may count on this behaviour. The contract also says that the definition of the member function `Point::scale()` may modify the values of data members of the class `Point`; users of the class `Point` must assume that `Point::scale()` will indeed modify member data and act accordingly.[4]

In the first homework exercise, the value of a member datum is modified, thereby breaking the contract. The compiler detects it and issues a diagnostic message.

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

The third and fourth homework exercises illustrate that the compiler considers two member functions that are identical except for the presence of the `const` keyword to be different functions[5]. In homework exercise 3, when the compiler tried to compile `Point::mag() const` in `Point.cc`, it looked at the class declaration in `Point.h` and could not find a matching member function declaration; these was a close, but not exact match. Therefore it issued a diagnostic message, telling us about the close match, and then stopped. Similarly, in homework exercise 4, it also could not find a match.

5.6.7 C++ Exercise 4 v6: Private Data and Accessor Methods

5.6.7.1 Setters and Getters

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

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

[4] C++ has another keyword, *mutable*, that one can use to exempt individual data members from this contract. It's 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.

A 2D point class, with member data in Cartesian coordinates, is not a good example of *why* it is often a good idea to have private data. But it does have enough richness to illustrate the mechanics, which is the purpose of this section. Section 5.6.7.3 discusses an example in which having private data makes obvious sense.

To build and run this exercise, `cd` to the directory `Classes/v6` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program you should see the following output:

```
$ ptest
Before p0: (1, 2)  Magnitude: 2.23607  Phi: 1.10715
After  p0: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
p1: (0, 1)  Magnitude: 1  Phi: 1.5708
p1: (1, 0)  Magnitude: 1  Phi: 0
p1: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
```

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

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

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

1. four new member functions have been declared,
   (a) `double x() const;`
   (b) `double y() const;`
   (c) `void set( double ax, double ay);`
   (d) `void set( Point const& p);`
2. the data members have been declared private
3. the data members have been renamed from `x` and `y` to `x_` and `y_`

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

1. the name of the class it is in
2. the name of the member function
3. the argument list; that is the number, type and order of arguments
4. whether or not the function is `const`

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

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

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

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

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

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

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

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

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

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

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

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

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

Now that the data members of `Point` are private, i.e., only the code within `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.

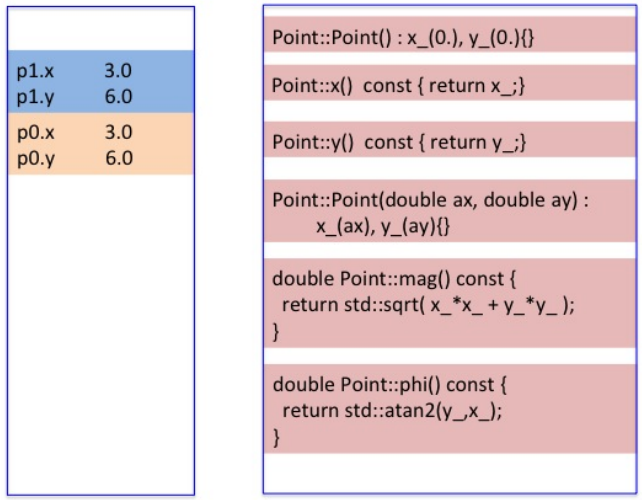


Figure 5.2: Memory diagram at the end of a run of Classes/v6/ptest.cc

access this information via the accessor and setter functions.

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

```
$ diff -wb ptest.cc ../v5/
```

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

1. the printout has been changed to use the accessor functions
2. a new section has been added to illustrate the use of the two set methods

Presumably these are clear.

Figure 5.2 shows a diagram of the computer memory at the end of running this version of `ptest`. The two boxes with the blue outlines represent sections of the computer memory; the part on the left represents that part that is reserved for storing data (such as objects) and the part on the right represents the part of the computer memory that holds the executable code. This is a big oversimplification because, in a real running program, there are many parts of the memory reserved for different sorts of data and many parts reserved for executable code.

The key point in Figure 5.2 is that each object has its own member data but there is only one copy of the code. Even if there are thousands of objects of type `Point`, there will only be one copy of the code. When a line of code asks for `p0.mag()`, the computer will pass the address of `p0` as an argument to the function `mag()`, which will then do its work. When a line of code asks for `p1.mag()`, the computer will pass the address of `p1` as an argument to the function `mag()`, which will then do its work.

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

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

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

This version of the syntax makes it much clearer how there can be one copy of the code even though there are many objects in memory; but it also makes the code harder to read once you have understood how the magic works. There are not many places in which you need to explicitly use the *this* pointer, but there will be some. For further information, consult standard C++ documentation (listed in Section 5.7).

5.6.7.2 What's the deal with the underscore?

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

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

```
 _name;
__name;
m_name;
 mName;
```

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

The compiler promises that all of the linker symbols it creates will begin with a leading single or double underscore, followed by a capital letter. Some of the identifiers that you define in a C++ class will be used as part of a linker symbol. If you chose identifiers that match the pattern reserved for symbols created by the compiler there is a chance you will have naming collision with a compiler defined symbol. While this is a very small risk, it seems wise to adopt habits that guarantee that it can never happen.

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

5.6.7.3 An example to motivate private data

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

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

The language used in the software engineering texts is that a guaranteed relationship among the data members is called an *invariant*. If a class has an invariant then the class must have private data.

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

5.6.8 C++ Exercise 4 v7: The inline keyword

This section introduces the *inline keyword*.

To build and run this exercise, cd to the directory `Classes/v7` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program you should see the following output:

```
$ ptest
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

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

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

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

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

1. have a smaller memory footprint
2. execute more quickly

These are both good things.

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

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

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

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

Look at the definition of the member function `y()` in `Point.h`. Compared to the definition of the member function `x()` there is small change in whitespace. 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.

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

5.6.9 C++ Exercise 4 v8: Defining Member Functions within the Class Declaration

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

To build and run this exercise, `cd` to the directory `Classes/v8` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program you should see the following output:

```
$ ptest
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

This is the same output made by `v7`.

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

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

This version of `Point.h` shows that you may define any member function inside the class declaration. When you do this, the inline keyword is implicit. Section 5.6.8 discussed some cautions about inappropriate use of inlining; those same cautions apply when a member function is defined inside the class declaration.

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

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

would produce a compiler diagnostic.

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

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

https://cdcvs.fnal.gov/redmine/projects/art/wiki/Data_Product_Design_Guide#Issues-mostly-related-to-ROOT

5.6.10 C++ Exercise 4 v9: The stream insertion operator

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

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

To build and run this exercise, `cd` to the directory `Classes/v9` and follow the same instructions as in Section 5.6.3. When you run the `ptest` program you should see the following output:

```
$ ptest
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

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

Look at `Point.h`. The changes relative to `v7` are the following two additions:

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

Look at `Point.cc`. The changes relative to `v7` are the following two additions:

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

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

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

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

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

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

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

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

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

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

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

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

Now it recognizes,

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

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

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

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

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

If you want to understand more about why the operator is written the way that it is, consult the standard C++ references; see Section 5.7.

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

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

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

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

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

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

5.6.11 Review

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

1. class
2. object
3. identifier
4. free function
5. member function

5.7 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.

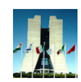
The following contains a discussion of recommended best practices,

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

6 Using External Products in UPS

Section 2.6.8 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 3.9) 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.

6.1 The UPS Database List: PRODUCTS

The act of setting up UPS defines a number of environment variables (discussed in Section 6.5), 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 3.6). 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.

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

```
$ printenv PRODUCTS
```

It should have a value of

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

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

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

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

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

ls /grid/fermiapp/products/common/db

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

$ ls /ds50/app/products
art                 cetpkgsupport  g4neutronxs  libxml2          totalview
artdaq              clhep          g4nucleonxs  messagefacility  toyExperiment
art_suite           cmake          g4photon     mpich            upd
art_workbook_base   cpp0x          g4pii        mvapich2         ups
boost               cppunit        g4radiative  python           xerces_c
caencomm            ds50daq        g4surface    root             xmlrpc_c
caendigitizer       fftw           gcc          setup
caenvme             fhiclcpp       gccxml       smc_compiler
cetbuildtools       g4emlow        geant4       sqlite
cetlib              g4neutron      libsigcpp    tbb
```

Each directory name in these listings corresponds to the name of a UPS product. If you are on a different experiment, the precise contents of your experiment's product directory may be slightly different. Among other things, both databases contain a subdirectory named `ups`[1]; this is for the UPS system itself. In this sense, all these products, including *art*, toyExperiment and even the product(s) containing your experiment's code, regard UPS as just another external product.

6.2 UPS Handling of Variants of a Product

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

The full identifier of a UPS product includes its product name, its version, its flavor and its full set of qualifiers. In Section 6.3, you will see how to fully identify a product when you set it up.

6.3 The `setup` Command: Syntax and Function

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

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

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

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

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

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

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

- define some environment variables
- define some bash functions
- define some aliases
- add elements to your `PATH`
- setup additional products on which it depends

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

When you follow a given site-specific setup procedure, the `PRODUCTS` environment variable will be extended to include your experiment-specific UPS repository.

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

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

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

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

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

6.4 Current Versions of Products

For some UPS products, but not all, the site administrator may define a particular fully-qualified version of the product as the default version. In the language of UPS this notion of default is called the *current* version. If a current version has been defined for a product, you can set up that product with the command:

```
$ setup <product-name>
```

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

Having a current version is a handy feature for products that add convenience features to your interactive environment; as improvements are added, you automatically get them.

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

6.5 Environment Variables Defined by UPS

When your login script or site-specific setup script initializes UPS, it defines many environment variables in addition to `PRODUCTS` (Section 6.1), one of which is `UPS_DIR`, the root directory of the currently selected version of UPS. The script also adds `$UPS_DIR/bin` to your `PATH`, which makes some UPS-related commands visible to your shell. Finally, it defines the bash function `setup` (see Sections 3.8 and 6.3). When you use the `setup` command, as illustrated below, it is this bash function that does the work.

In discussing the other important variables, the toyExperiment product will be used as an example product. For a different product, you would replace "toy-Experiment" or "TOYEXPERIMENT" in the following text by the product's name. Once you have followed your appropriate setup procedure (Table 4.1) you can issue the following command this is informational for the purposes of this section; you don't need to do it until you start running the first Workbook exercise):

```
$ setup toyExperiment v0_00_14 -qe2:prof
```

The version and qualifiers shown here are the ones to use for the Workbook exercises. When the `setup` command returns, the following environment variables will be defined:

TOYEXPERIMENT_DIR defines the root DIRectory of the chosen UPS product

TOYEXPERIMENT_INC defines the path to the root directory of the C++ header files that are provided by this product (so called because the header files are INCluded)

TOYEXPERIMENT_LIB defines the directory that contains all of the shareable object LIBraries (ending in `.so`) that are provided by this product

Almost all UPS products that you will use in the Workbook define these three environment variables. Several, including toyExperiment, define many more. Once you're running the exercises, you will be able to see all of the environment variables defined by the toyExperiment product by issuing the following command:

```
$ printenv | grep TOYEXPERIMENT
```

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

6.6 Finding Header Files

6.6.1 Introduction

Header files were introduced in Section 5.3.2. Recall that a header file typically contains the “parts list” for its associated `.cc` source file and is “included” in the `.cc` file.

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

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

An important aid to finding documentation is the use of *namespaces*; if you are not familiar with namespaces, see Section 30.6 or consult the standard C++ documentation.

6.6.2 Finding *art* Header Files

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

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

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

```
#include "art/Framework/Principal/Event.h"
```

The *art* UPS product has been designed so that the relative path used to include any *art* header file starts with the directory `art`; this is another clue that the class or function of interest is part of *art*.

When you setup the *art* UPS product, it defines the environment variable `ART_INC`, which points to the root of the header file tree for *art*. You now have enough information to discover where to find the header file for `art::Event`; it is at

```
$ART_INC/art/Framework/Principal/Event.h
```

You can follow this same pattern for any class or function that is part of *art*. This will only work if you are in an environment in which `ART_INC` has been defined, which will be described in Chapters 8 and 9.

If you are a C++ beginner, you will likely find this header file difficult to understand; you do not need to understand it when you first encounter it but, for future reference, you do need to know where to find it.

Earlier in this section, you read that if a C++ file uses `art::Event`, it would *almost always* include the appropriate header file. Why *almost* always? Because the header file `Event.h` might already be included within one of the other headers that are included in your file. If `Event.h` is indirectly included in this way, it does not hurt also to include it explicitly, but it is not required that you do so.[2]

We can summarize this discussion as follows: if a C++ source file uses `art::Event` it must always include the appropriate header file, either directly or indirectly.

art does not rigorously follow the pattern that the name of file is the same as the name of the class or function that it defines. The reason is that some files define multiple classes or functions; in most such cases the file is named after the most important class that it defines.

Finally, from time to time, you will need to dig through several layers of header files to find the information you need.

There are two code browsing tools that you can use to help navigate the layering of header files and to help find class declarations that are not in a file named for the class:

1. use the *art redmine*(γ) repository browser:
   https://cdcvs.fnal.gov/redmine/projects/art/repository/revisions/master/show/art
2. use the LXR code browser: http://cdcvs.fnal.gov/lxr/art/

(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 pedagodical tool is often a good trade-off; the Workbook will frequently do this.

6.6.3 Finding Headers from Other UPS Products

Section 2.7 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 2.7.2). 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 6.6.4, 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 6.1.

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://cdcvs.fnal.gov/redmine/projects/art/wiki

6.6.4 Exceptions: The Workbook, ROOT and Geant4

There are three exceptions to the pattern described in Section 6.6.3:

Table 6.1: 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 6.6.4. ‡The namespace `tex` is also used by the *art* Workbook, which is not a UPS product.

UPS Product	Namespace
art	`art`
boost	`boost`
cet	`cetlib`
clhep	`CLHEP`
fhiclcpp	`fhicl`
messagefacility	`mf`
toyExperiment	`tex`‡

- the Workbook itself
- ROOT
- Geant4

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

- `art-workbook` for the Workbook
- `toyExperiment` for the toyExperiment

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `T`, followed by an upper case letter
- `g`, followed by an upper case letter
- `G4`

Part II

Workbook

7 Preparation for Running the Workbook Exercises

7.1 Introduction

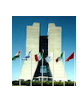
You will run the Workbook exercises on a computer that is maintained by your experiment, either at Fermilab or at another institution. Many details of the working environment change from site to site[1] and these differences are parameterized so that (a) it is easy to establish the required environment, 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 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.

7.2 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 7.1. 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).



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

Table 7.1: Experiment-specific Information for New Users

Experiment	URL of New User Page
ArgoNeut	https://cdcvs.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Darkside	https://cdcvs.fnal.gov/redmine/projects/darkside-public/wiki/Before_You_Arrive
LArSoft	https://cdcvs.fnal.gov/redmine/projects/larsoftsvn
LBNE	https://cdcvs.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
MicroBoone	https://cdcvs.fnal.gov/redmine/projects/larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Muon g-2	https://cdcvs.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 7.2: Login machines for running the Workbook exercises

Experiment	Name of Login Node
ArgoNeut	`argoneutvm.fnal.gov`
Darkside	`ds50.fnal.gov`
LBNE	`lbnevm.fnal.gov`
MicroBoone	`uboonevm.fnal.gov`
Muon g-2	`gm2gpvm.fnal.gov`
Mu2e	`mu2evm.fnal.gov`
NOνA	`nova-offline.fnal.gov`

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

If you would like a computer account on a Fermilab computer in order to evaluate *art*, contact the *art* team (see Section 2.4).

7.3 Choosing a Machine and Logging In

The experiment-specific machines confirmed to host the Workbook code are listed in Table 7.2 In most cases the name given is not the name of an actual computer, but rather a round-robin alias for a cluster. For example, if you log into `mu2evm`, you will actually be connected to one of the five computers `mu2egpvm01` through `mu2egpvm05`. These Mu2e machines share all disks that are relevant to the Workbook exercises, so if you need to log in multiple times, it is perfectly OK if you are logged into two different machines; you will still see all of the same files.

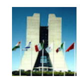
Each experiment's web page has instructions on how to log in to its computers from your local machine.

7.4 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`.

7.5 Choose an Editor

As you work through the Workbook exericses 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.

8 Exercise 1: Run Pre-built art Modules

8.1 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.

8.2 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 7 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.

8.3 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

6. how to start at an event that is not the first event in the file
7. how to concatenate input files
8. how to write an output file
9. some basics about the grammar and structure of a FHiCL file
10. a little bit about the *art* run-time environment

8.4 Running the Exercise

8.4.1 The Pieces

Several event-data input files have been provided for use by the Workbook exercises. These input files are packaged as part of the toyExperiment UPS product. Table 8.1 lists the range of event IDs found in each file. You will need to refer back to this table as you proceed.

Table 8.1: The input files provided by for the Workbook exercises

File Name	Run	SubRun	Range of Event Numbers
input01_data.root	1	0	$1 \ldots 10$
input02_data.root	2	0	$1 \ldots 10$
input03_data.root	3	0	$1 \ldots 5$
	3	1	$1 \ldots 5$
	3	2	$1 \ldots 5$
input04_data.root	4	0	$1 \ldots 1000$

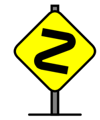
A run-time configuration (FHiCL) file has also been provided, `hello.fcl`.

8.4.2 Log In, Set Up and Execute *art*

The intent of this section is for the reader to start from "zero" and execute an *art* job, without necessarily understanding each step, just to get familiar with the process. A detailed discussion of what these steps do will follow in Section 8.8.

Some steps are written as statements, others as commands to issue at the prompt. Notice that *art* takes the argument `-c hello.fcl`; this points *art* to the run-time configuration file that will tell it what to do and where to find the "pieces" on which to operate.

Most readers: Follow the steps in Section 8.4.2.1, then proceed directly to Section 8.6.



If you wish to manage your working directory yourself, skip Section 8.4.2.1, follow the steps in Section 8.4.2.2, then proceed to Section 8.6.

If you log out and wish to log back in, follow the procedure outlined in Section 9.6.

8.4.2.1 Standard Procedure

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Table 4.1.
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. `$ art -c hello.fcl >& output/hello.log`

Proceed to Section 8.6.

8.4.2.2 Procedure allowing Self-managed Working Directory

1. Log in to the computer you chose in Section 7.3.
2. Follow the site specific setup procedure; see Table 4.1
3. Make a working directory and cd to it.
4. `setup toyExperiment v0_00_14 -q$ART_WORKBOOK_QUAL:prof`
5. `cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .`
6. Make a subdirectory named `output`. If you prefer you can make this on some other disk and put a symbolic link to it, named `output`, in the current working directory.
7. `ln -s $TOYEXPERIMENT_DIR/inputFiles .`
8. `art -c hello.fcl`

8.5 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 folllowing:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Section 4.
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 8.4.2. 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.

8.6 Examine Output

Compare the ouptut you produced against Listing 8.1; the only differences should be the timestamps. It also processed the first file listed in Table 8.1.

Listing 8.1: Sample output from running hello.fcl

```

%MSG-i MF_INIT_OK:  art 27-Apr-2013 21:22:13 CDT JobSetup
Messagelogger initialization complete.
%MSG
27-Apr-2013 21:22:14 CDT  Initiating request to open file
inputFiles/input01_data.root
27-Apr-2013 21:22:14 CDT  Successfully opened file
inputFiles/input01_data.root
Begin processing the 1st record. run: 1 subRun: 0 event: 1 at
27-Apr-2013 21:22:14 CDT
Hello World! This event has the id: run: 1 subRun: 0 event: 1
Begin processing the 2nd record. run: 1 subRun: 0 event: 2 at
27-Apr-2013 21:22:14 CDT
Hello World! This event has the id: run: 1 subRun: 0 event: 2
Hello World! This event has the id: run: 1 subRun: 0 event: 3
Hello World! This event has the id: run: 1 subRun: 0 event: 4
Hello World! This event has the id: run: 1 subRun: 0 event: 5
Hello World! This event has the id: run: 1 subRun: 0 event: 6
Hello World! This event has the id: run: 1 subRun: 0 event: 7
Hello World! This event has the id: run: 1 subRun: 0 event: 8
Hello World! This event has the id: run: 1 subRun: 0 event: 9
Hello World! This event has the id: run: 1 subRun: 0 event: 10
27-Apr-2013 21:22:14 CDT  Closed file inputFiles/input01_data.root

TrigReport ---------- Event  Summary ------------
TrigReport Events total = 10 passed = 10 failed = 0

TrigReport ------ Modules in End-Path: e1 ------------
```

```
TrigReport  Trig Bit#    Visited     Passed     Failed       Error Name
TrigReport     0    0         10         10          0           0 hi

TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 0.004000 Real = 0.002411

Art has completed and will exit with status 0.
```

Every time you run *art*, the first thing to check is the last line in your output or log file. It should be `Art has completed and will exit with status 0.` If the status is not 0, or if this line is missing, it is an error; please contact the *art* team as described in Section 2.4.

A future version of these instructions will specify how much disk space is needed, including space for all ouptut files.

8.7 Understanding the Configuration File `hello.fcl`

The file `hello.fcl` gives *art* its run-time configuration. This file 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*. By convention, files written in FHiCL end in `.fcl`. As you work through the Workbook, the features of FHiCL that are relevant for each exericse will be explained.

The full details of the FHiCL language, plus the details of how it is used by *art*, are given in the Users Guide, Chapter 23. Most people will find it much easier to follow the discussion in the Workbook documentation than to digest the full documentation up front.

8.7.1 Some Bookkeeping Syntax

In a FHiCL file, the start of a comment is marked by the hash sign character (#); a comment may begin in any column.

The hash sign has one other use, however. If the first eight characters of a line are exactly `#include`, followed by whitespace and a quoted list of file paths, then the line will be interpreted as an *include directive* and the line containing it will be replaced by the contents of the file named in the include directive.

The basic element of FHiCL is the *definition*, which has the form

```
name : value
```

A group of FHiCL definitions delimited by braces {} is called a *table*(γ). Within *art*, a FHiCL table gets turned into a C++ object called a *parameter set*(γ); this document set will often refer to a FHiCL table as a parameter set.

The fragment of `hello.fcl` shown in Listing 8.2 contains the FHiCL table that configures the *source*(γ) of events that *art* will read in and operate on.

Listing 8.2: The source parameter set from hello.fcl

```
source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root" ]
}
```

The name `source` is a *keyword* in *art*; i.e., the name `source` has no special meaning to FHiCL but it does have a special meaning to *art*. To be precise, it only has a special meaning to *art* if it is at the outermost *scope*(γ) of a FHiCL file; i.e., not inside any braces {} within the file. The notion of *scope* in FHiCL is discussed further in Chapter 11. When *art* sees a parameter set named `source` at the outermost scope, then *art* will interpret that parameter set to be the description of the source of events for this run of *art*.

In the `source` parameter set, the identifier `module_type` is a keyword in *art* that tells *art* the name of a module that it should load and run, `RootInput` in this case. `RootInput` is one of the standard source modules provided by *art* and it reads disk files containing event-data written in an *art*-defined ROOT-based format. The default behaviour of the `RootInput` module is to start at the first event in the first file and read to the end of the last event in the last file.[1]

The identifier `fileNames` is again a keyword, but this time defined in the `RootInput` module, that gives the module a list of filenames from which to read events. The list is delimited by square brackets and contains a comma-separated list of filenames. This example shows only one filename, but the square brackets are still required. The proper FHiCL name for a comma-separated list delimited by square brackets is a *sequence*(γ).

In most cases the filenames in the sequence must be enclosed in quotes. FHiCL, like many other languages has the following rule: if a string contains white space or any special characters, then quoting it is required, otherwise quotes are optional.

FHiCL has its own set of special characters; these include anything *except* all upper and lower case letters, the numbers 0 through 9 and the underscore character. *art* restricts the use of the underscore character in some circumstances; these will be discussed as they arise.

It is implied in the foregoing discussion that a FHiCL *value* need not be a simple thing, such as a number or a quoted string. For example, in Listing 8.2,

[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` .

the `source` value is a parameter set (of two parameters) and the value of `fileNames` is a (single-item) sequence.

8.7.2 Some Physics Processing Syntax

The identifier *physics*(γ), when found at the outermost scope, is a keyword in *art*. The `physics` parameter set is so named because it contains most of the information needed to describe the physics workflow of an *art* job.

The fragment of `hello.fcl` shown in Listing 8.3 shows a rather long-winded way of telling *art* to find a module named `HelloWorld` and execute it.

Listing 8.3: The physics parameter set from hello.fcl

```
physics :{
  analyzers: {
    hi : {
      module_type : HelloWorld
    }
  }
  e1         : [ hi ]
  end_paths : [ e1 ]
}
```

Why so long-winded? *art* has very powerful features that enable execution of multiple complex chains of modules; the price is that specifying something simple takes a lot of keystrokes.

Within the `physics` parameter set, notice the identifier `analyzers`. When found as a top-level identifier within the `physics` scope, it is recognized as a keyword in *art*. The `analyzers` parameter set defines the run-time configuration for all of the analyzer modules that are part of the job – only `HelloWorld` in this case.

For our current purposes, the module `HelloWorld` does only one thing of interest, namely for every event it prints one line:

```
Hello World! This event has the id: run: <RR> subRun: <SS> event: <EE>
```

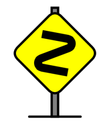
where RR, SS and EE are substituted with the actual run, subRun and event number of each event.

If you look back at Listing 8.1, you will see that this line appears ten times, once each for events 1 through 10 of run 1, subRun 0 (as expected, according to Table 8.1). The remainder of the listing is standard output generated by *art*.

Listing 8.4 shows the remainder of the lines in `hello.fcl`. The line starting with *process_name*(γ) tells *art* that this job has a name and that the name is "hello"; it has no real significance in these simple exericses. It becomes important when an *art* job creates new data products (described in User Guide

Chapter 24) 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 8.4: The remainder of hello.fcl

```
#include "fcl/minimalMessageService.fcl"

process_name : hello

services : {
  message : @local::default_message
}
```

Listing 8.4 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`.

8.7.3 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 interacive use and the long form makes scripts more readable.

8.7.4 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:

```
$ art -c hello.fcl -n 5
$ art -c hello.fcl --nevts 4
```

Run each of these commands and observe their output.

The second way is within the FHiCL file. Start by making a copy of `hello.fcl`:

```
$ cp hello.fcl hi.fcl
```

Edit `hi.fcl` and add the following line anywhere in the source parameter set:

```
maxEvents   : 3
```

By convention this is added after the fileNames definition but it can go anywhere inside the source parameter set because the order of parameters within a FHiCL table is not important. Run *art* again, using `hi.fcl`:

```
$ art -c hi.fcl
```

You should see output from the `HelloWorld` module for only the first three events.

To configure the file for *art* to process all the events, i.e., to run until *art* reaches the end of the input files, either leave off the maxEvents parameter or give it a value of -1.

If the maximum number of events is specified both on the command line and in the FHiCL file, then the command line takes precedence. Compare the outputs of the following commands:

```
$ art -c hi.fcl
$ art -c hi.fcl -n 5
$ art -c hi.fcl -n -1
```

8.7.5 Changing the Input Files

For historical reasons, there are multiple ways to specify the input event-data file (or the list of input files) to an *art* job:

- within the FHiCL file's `source` parameter set
- on the *art* command line via the `-s` option (you may specify one input file only)
- on the *art* command line via the `-S` option (you may specify a text file that lists multiple input files)
- on the *art* command line, after the last recognized option (you may specify one or more input files)

If input file names are provided both in the FHiCL file and on the command line, the command line takes precedence.

Let's run a few examples.

We'll start with the `-s` command line option (second bullet). Run *art* without it (again), for comparison (or recall its output from Table 8.1):

```
$ art -c hello.fcl
```

To see what you should expect given the following input file, check Table 8.1, then run:

```
$ art -c hello.fcl -s inputFiles/input02_data.root
```

Notice that the 10 events in this output are from run 2 subRun 0, in contrast to the previous printout which showed events from run 1. Notice also that the command line specification overrode that in the FHiCL file. The `-s` (lower case) command line syntax will only permit you to specify a single filename.

This time, edit the source parameter set inside the `hi.fcl` file (first bullet); change it to:

```
  source : {
    module_type : RootInput
    fileNames   : [ "inputFiles/input01_data.root",
                    "inputFiles/input02_data.root" ]
    maxEvents   : -1
  }
```

(Notice that you also added `maxEvents : -1`.) The names of the two input files could have been written on a single line but this example shows that newlines are treated simply as white space.

Check Table 8.1 to see what you should expect, then rerun *art* as follows:

```
$ art -c hi.fcl
```

You will see 20 lines from the `HelloWorld` module; you will also see messages from *art* at the open and close operations on each input file.

Back to the `-s` command-line option, run:

```
$ art -c hi.fcl -s inputFiles/input03_data.root
```

This will read only `inputFiles/input03_data.root` and will ignore the two files specified in the `hi.fcl`. The output from the `HelloWorld` module will be the 15 events from the 3 subRuns of run 3.

There are several ways to specify multiple files at the command line. One choice is to use the `-S` (upper case) [`--source-list`] command line option (third bullet) which takes as its argument the name of a text file containing the ROOT input filename(s), e.g., `inputs.txt`.

```
%$ ls inputFiles/*.root | head -3 > inputs.txt
$ cat inputs.txt
$ art -c hi.fcl -S inputs.txt
```

The first command shows you the filenameslisted in the input file. After the *art* command, you should see the `HelloWorld` output from 35 events in the three files.

Finally, you can list the files at the end of the command (fourth bullet), either file-by-file or via a text-file listing of them. .

```
$ art -c hi.fcl inputs.txt
```

When *art* processes its command line options, any strings that follow the last recognized option are presumed to be the names of input files. *art* will form an input file list from these filenames. For example

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

will make the `HelloWorld` printout for input files 02 and 03.

It is recommended that, within a single *art* job, you pick one way of specifying multiple files. It is possible, but needlessly confusing and error-prone, to simultaneously use all of the command line methods (any of which will trump the FHiCL file contents).

8.7.6 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:

```
  firstEvent  : 5
  maxEvents   : 3
```

art will process events 5, 6, and 7 of run 1, subRun 0. Try it:

```
$ 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:

```
$ art -c hello.fcl -e  5 -n 3
$ art -c hello.fcl --nskip 4 -n 3
```

You can also specify the intial event to process relative to a given event ID (which, recall, contains the run, subRun and event number). Edit `hi.fcl` and edit the `source` parameter set as follows:

```
source : {
  module_type : RootInput
  fileNames   : [ ``inputFiles/input03_data.root'' ]
  firstRun    : 3
  firstSubRun : 1
  firstEvent  : 6
}
```

When you run this job, *art* will process events starting from run 3, subRun 2, event 1, – because there are only 5 events in subRun 1.

```
$ art -c hi.fcl
```

8.7.7 Identifying the User Code to Execute

Recall from Section 8.7.2 that the `physics` parameter set contains the physics content for the *art* job. Within this parameter set, *art* must be able to determine which (user code) modules to process. These must be referenced via *module labels*(γ), which as you will see, represent the pairing of a module name and a run-time configuration.

Look back at Listing 8.3, which contains the `physics` parameter set from `hello.fcl`. The `analyzer` parameter set, nested inside the `physics` parameter set, contains the definition:

```
hi : {
  module_type : HelloWorld
}
```

The identifier `hi` is a module label (defined by the user, not by FHiCL or *art*) whose value must be a parameter set that *art* will use to configure a module. The parameter set for a module label must contain (at least) a FHiCL definition of the form:

```
module_type : <module-name>
```

Here `module_type` is a keyword in *art* and `<module-name>` tells *art* the name of the module to load and execute. (Since it is within the `analyzer` parameter set, the module must be of type EDAnalyzer; i.e. the *base type* of `<module-name>` must be EDAnalyzer.)

Module labels are fully described in Section 23.5.

In this example *art* will look for a module named `HelloWorld`, which it will find as part of the toyExperiment UPS product. Section 8.9 describes how *art* uses `<module-name>` to find the shareable library that contains code for the

HelloWorld module. A parameter set that is used to configure a module may contain additional lines; if present, the meaning of those lines is understood by the module itself; those lines have no meaning either to *art* or to FHiCL.

Now look at the FHiCL fragment in Listing 8.5. We will use it to reinforce some of the ideas discussed in the previous paragraph.

art allows you to write a FHiCL file that uses a given module more than once. For example you may want to run an analysis twice, once with a loose mass cut on some intermediate state and once with a tight mass cut on the same intermediate state. In *art* you can do this by writing one module and making the cuts "run-time configurable." This idea will be developed further in Chapter 12.

Listing 8.5: A FHiCL fragment illustrating module labels

```
analyzers : {
  loose : {
    module_type : MyAnalysis
    mass_cut    : 20.
  }
  tight : {
    module_type : MyAnalysis
    mass_cut    : 15.
  }
}
```

When *art* processes this fragment it will look for a module named MyAnalysis and instantiate it twice, once using the parameter set labeled (i.e. with module label) tight and once using the parameter set labeled loose. The two instances of the module MyAnalysis are distinguished by the module labels tight and loose.

art requires that module labels be unique within a FHiCL file. Module label may contain only upper- and lower-case letters and the numerals 0 to 9.

In the FHiCL files in this exercise, all of the modules are analyzer modules. Since analyzers do not make data products, these module labels are nothing more than identifiers inside the FHiCL file. For producer modules, however, which *do* make data products, the module label becomes part of the data product identifier and as such has a real signficance. All module labels must conform to the same naming rules.

Within *art* there is no notion of reserved names or special names for module labels; however your experiment will almost certainly have established some naming conventions.

8.7.8 Paths

In the physics parameter set for hello.fcl there are two parameters that represent paths (discussed in Section 3.6 :

```
  e1        : [ hi ]
  end_paths : [ e1 ]
```

The path defined by the parameter `e1` takes a value that is a FHiCL sequence of module labels. The name of a path is an arbitrary identifier that must be unique within a FHiCL file; it has no persistent signficance and can be any legal FHiCL name.

Sometimes this documentation uses the word *path* in the sense of an *art path*(γ) (a sequence of module labels), other times *path* is used as a path in a file system and in yet other situations, it is used as a colon-delimited set of directory names. The use should be clear from the context.

The name `end_paths`, in contrast to `e1`, is a keyword in *art*. Its value must be a FHiCL sequence of paths – here it is a sequence of one path, `e1`. *reference the rules when available* When *art* processes the `end_paths` definition it combines all of the path definitions and forms the set of unique module labels from all paths defined in the parameter set . In other words, it is legal in *art* for a module label to appear in more than one path; if it does, *art* will recognize this and will ensure that the module is executed only once per event.

If you put the name of a module label into the definition of `end_paths`, *art* will issue an error and stop processing.

The paths listed in `end_paths` may only contain module labels for analyzer and/or output modules; they may *not* contain module labels for producer or filter modules. The reason for this restriction will be discussed in Section .

What about the order of module labels in a path? Since analyzer and output modules may neither add new information to the event nor communicate with each other except via the event, the processing order is not important for the event. By definition, then, *art* may run analyzer and output modules in any order. In a simple *art* job with a single path, *art* will, in fact, run the modules in the order of appearance in the path, but do not write code that depends on execution order because *art* is free to change it.

It may seem that `end_paths` could more simply have been defined as a set of module labels, eliminating the layer of the path altogether, but there is a reason. We will defer this discussion to Section .

If the `end_paths` parameter is absent or defined as:

```
  end_paths : [ ]
```

art will understand that this job has no analyzer modules and no filter modules to execute. It is legal to define a path as an empty FHiCL sequence.

As is standard in FHiCL, if the definition of `end_paths` appears more than once, the last definition takes precendence.

8.7.9 Writing an Output File

The file writeFile.fcl gives an example of writing an output file. This file introduces the parameter set named outputs:

```
outputs : {
  output1 : {
    module_type : RootOutput
    fileName    : "output/writeFile_data.root"
  }
}
```

When it appears at the outermost scope of a FHiCL file, the identifier outputs is a keyword reserved to *art*. In this case the value of outputs must be a parameter set (e.g., output1) of parameter sets (e.g., module_type and fileName); each of the inner parameter sets provides the configuration of one output module.

An *art* job may have zero or more output modules.

The name RootOutput is the name of a standard *art* output module; it writes the events in memory to a disk file in an *art*-defined, ROOT-based format. Files written by the module RootOutput can be read by the module RootInput. The identifier output1 is just another module label that obeys the same rules discussed in Section 8.7.7. The identifier fileName is a keyword known to the RootOutput module; its value is the name of the output file that this instance of RootOutput will write.

There are many more optional parameters that can be used to configure an output module. For example, an output module can be configured to write out only selected events and/or to write out only a subset of the available data products. Optional parameters are described in Chapter .

Notice in writeFile.fcl that the path e1 has been extended to include the module label of the output module:

```
e1  : [ hi, output1 ]
```

Finally, the source parameter set of writeFile.fcl is configured to read only events 4, 5, 6, and 7.

To run writeFile.fcl and check that it worked correctly:

```
$ art -c writeFile.fcl
$ ls -s output/writeFile_data.root
$ art -c hello.fcl -s output/writeFile_data.root
```

The first command will write the ouptut file; the second will check the size of the output file and the last one will read back the output file and print the event IDs for all of the events in the file. You should see the HelloWorld printout for events 4, 5, 6 and 7.

8.8 Understanding the Process for Exercise 1

Section 8.4.2 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 7.3.
2. Follow the site-specific setup procedure; see Chapter 4
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.

8.8.1 Follow the Site-Specific Setup Procedure (Details)

The site-specific startup procedure, described in Chapter 4, 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.

8.8.2 Make a Working Directory (Details)

On the Fermilab computers the home disk areas are quite small so most experiments ask that their collaborators work in some other disk space. This is common to sites in general, so we recommend working in a separate space as a best practice. The Workbook is designed to require it.

This step given as:

```
$ mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/pre-built
```

creates a new directory to use as your working directory. It is defined relative to an environment variable described in Section 8.8.1. It only needs to be done the first time that you log in to work on Workbook exercises – once it's there, it's there!

If you follow the rest of the naming scheme, you will guarantee that you have no conflicts with other parts of the Workbook.

As discussed in Section 8.4.2.2, you may of course choose your own working directory on any disk that has adequate disk space.

8.8.3 Setup the toyExperiment UPS Product (Details)

This step is the main event in the eight-step process.

```
$ setup toyExperiment v0_00_14 -q$ART_WORKBOOK_QUAL:prof
```

This command tells `UPS` to find a product named toyExperiment, with the specified version and qualifiers, and to *setup* that product, as described in Section 6.3.

The required qualifiers may change from one experiment to another and even from one site to another within the same experiment. To deal with this, the site specific setup procedure defines the environment variable `ART_WORKBOOK_QUAL`, whose value is the qualifier string that is correct for that site.

The complete ups qualifier for toyExperiment has two components, separated by a colon: the string defined by `ART_WORKBOOK_QUAL` plus a qualifier describing the compiler optimization level with which the product was built, in this case "prof"; see Section 2.6.7 for information about the optimization levels.

Each version of the toyExperiment product knows that it requires a particular version and qualifier of the *art* product. In turn, *art* knows that it depends on particular versions of ROOT, CLHEP, boost and so on. When this recursive setup has completed, over 20 products will have been setup. All of these products define environment variables and about two-thirds of them add new elements to the environment variables `PATH` and `LD_LIBRARY_PATH`.

If you are interested, you can inspect your environment before and after doing this setup. To do this, log out and log in again. Before doing the setup, run the following commands:

```
$ printenv > env.before
$ printenv PATH  | tr : \\n > path.before
$ printenv LD_LIBRARY_PATH  | tr : \\n > ldpath.before
```

Then `setup` toyExperiment and capture the environment afterwards (`env.after`). Compare the before and after files: the after files will have many, many additions to the environment.

8.8.4 Copy Files to your Current Working Directory (Details)

The step:

```
$ cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .
```

only needs to be done only the first time that you log in to work on the Workbook.

In this step you copied the files that you will use for the exercises into your current working directory. You should see these files:

```
hello.fcl  makeLinks.sh  skipEvents.fcl  writeFile.fcl
```

8.8.5 Source makeLinks.sh (Details)

This step:

```
$ source makeLinks.sh
```

only needs to be done only the first time that you log in to work on the Workbook. It created some symbolic links that *art* will use.

The FHiCL files used in the Workbook exercises look for their input files in the subdirectory `inputFiles`. This script made a symbolic link, named `inputFiles`, that points to:

```
$TOYEXPERIMENT_DIR/inputFiles
```

in which the necessary input files are found.

This script also ensures that there is an output directory that you can write into when you run the exercises and adds a symbolic link from the current working directory to this output directory. The output directory is made under the directory `$ART_WORKBOOK_OUTPUT_BASE`; this environment variable was set by the site-specific setup procedure and it points to disk space that will have enough room to hold the output of the exercises.

8.8.6 Run *art* (Details)

Issuing the command:

```
$ art -c hello.fcl
```

runs the *art* main program, which is found in `$ART_FQ_DIR/bin`. This directory was added to your `PATH` when you setup toyExperiment. You can inspect your `PATH` to see that this directory is indeed there.

8.9 How does *art* find Modules?

When you ran `hello.fcl`, how did *art* find the module `HelloWorld`?

It looked at the environment variable `LD_LIBRARY_PATH`, which is a colon-delimited set of directory names defined when you setup the toyExperiments product. We saw the value of `LD_LIBRARY_PATH` in Section 8.8.3; to see it again, type the following:

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

(The fragment `| tr : \\n` tells the bash shell to take the output of printenv and replace every occurrence of the colon character with the newline character; this makes the output much easier to read.) The output should look similar to that shown in Listing 8.6.

Listing 8.6: Example of the value of LD_LIBRARY_PATH

```
/ds50/app/products/tbb/v4_1_2/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/sqlite/v3_07_16_00/Linux64bit+2.6-2.12-prof/lib
/ds50/app/products/libsigcpp/v2_2_10/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/cppunit/v1_12_1/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/clhep/v2_1_3_1/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/python/v2_7_3/Linux64bit+2.6-2.12-gcc47/lib
/ds50/app/products/libxml2/v2_8_0/Linux64bit+2.6-2.12-gcc47-prof/lib
/ds50/app/products/fftw/v3_3_2/Linux64bit+2.6-2.12-gcc47-prof/lib
/ds50/app/products/root/v5_34_05/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/boost/v1_53_0/Linux64bit+2.6-2.12-e2-prof/lib
/ds50/app/products/cpp0x/v1_03_15/slf6.x86_64.e2.prof/lib
/ds50/app/products/cetlib/v1_03_15/slf6.x86_64.e2.prof/lib2
/ds50/app/products/fhiclcpp/v2_17_02/slf6.x86_64.e2.prof/lib
/ds50/app/products/messagefacility/v1_10_16/slf6.x86_64.e2.prof/lib
/ds50/app/products/art/v1_06_00/slf6.x86_64.e2.prof/lib
/ds50/app/products/toyExperiment/v0_00_14/slf6.x86_64.e2.prof/lib
/grid/fermiapp/products/common/prd/git/v1_8_0_1/Linux64bit-2/lib
```

Of course the leading element of each directory name, `/ds50/app` will be replaced by whatever is correct for your experiment. The last element in LD_LIBRARY_PATH is not relevant for running *art* and it may or may not be present on your machine, depending on details of what is done inside your site-specific setup procedure.

If you compare the names of the directories listed in `LD_LIBRARY_PATH` to the names of the directories listed in the `PRODUCTS` environment variable, you will see that all of these directories are part of the UPS products system. Moreover, for each product, the version, flavor and qualifiers are embedded in the directory name. In particular, both *art* and toyExperiment are found in the list.

If you `ls` the directories in `LD_LIBRARY_PATH` you will find that each directory contains many shareable object libraries (`.so` files).

When *art* looks for a module named `HelloWorld`, it looks through the directeories defined in LD_LIBRARY_PATH and looks for a file whose name matches the pattern,

```
lib*HelloWorld_module.so
```

where the asterisk matches (zero or) any combination of characters. *art* finds that, in all of the directories, there is exactly one file that matches the pattern, and it is found in the directory:

```
/ds50/app/products/toyExperiment/v0_00_14/slf6.x86_64.e2.prof/lib/
```

The name of the file is:

```
libtoyExperiment_Analyzers_HelloWorld_module.so
```

If *art* had found no files that matched the pattern, it would have printed an error message and tried to shutdown as gracefully as possible. If *art* had found more than one file that matched the pattern, it would have printed a different error message and tried to shut down as gracefully as possible.

One of the important features of *art* is that, whenever it detects an error condition that is serious enough to stop execution, it always attempts to shut down as gracefully as possible. Among other things this means that it tries to properly close all output files. This feature is not so important when an error occurs at the start of a job but it ensures that, when an error occurs after hours of execution, your results up to the error are correct and available.

8.10 The *art* Run-time Environment

This discussion is aimed to help you understand the process described in this chapter as a whole and how the pieces fit together in the *art* run-time environment. This evironment is summarized in Figure 8.1. In this figure the boxes refer either to locations in memory or to files on a disk.

At the center of the figure is a box labelled "*art* executable;" this represents the *art* main program resident in memory after being loaded. When the *art* executable starts up, it reads its run-time configuration (FHiCL) file, represented by the box to its left. Following instructions from the configuration file, *art* will load shared libraries from toyExperiment, from *art*, from ROOT,

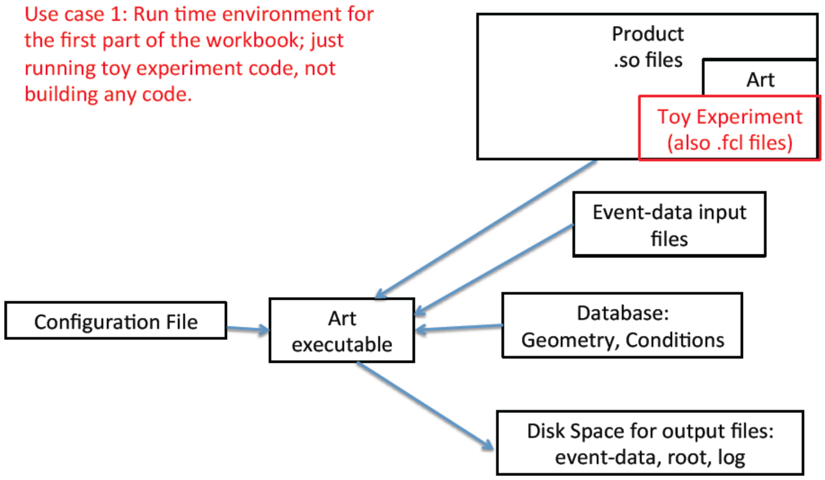


Figure 8.1: Elements of the *art* run-time environment for the first Workbook exercise

from CLHEP and from other UPS products. All of these shared librares (`.so` files) will be found in the appropriate UPS products in `LD_LIBRARY_PATH`, which points to directories in the UPS products area (box at upper right). Also following instructions from the FHiCL file, *art* will look for input files (box labeled "Event-data input files" at right). The FHiCL file will tell *art* to write its event-data and histogram output files to a particular directory (box at lower right).

One remaining box in the figure (at right, second from bottom) is not encountered in the first Workbook exercise but has been provided for completeness. In most *art* jobs it is necessary to access experiment-related geometry and conditions information; in a mature experiment, these are usually stored in a database that stands apart from the other elements in the picture.

The arrows in Figure 8.1 show the direction in which information flows. Everything but the output flows into the *art* executable.

8.11 Finding FHiCL files: FHICL_FILE_PATH

This section will describe where *art* looks for FHiCL files. There are two cases: looking for the file specified by the command line argument `-c` and looking for files that have been included by a `#include` directive within a FHiCL file.

8.11.1 The `-c` command line argument

When you issued the command

```
$ art -c hello.fcl
```

art looked for a file named `hello.fcl` in the current working directory and found it. You may specify any absolute or relative path as the argument of the `-c` option. If *art* had not found `hello.fcl` in this directory it would have looked for it relative to the path defined by the environment variable `FHICL_FILE_PATH`. This is just another path-type environment variable, like `PATH` or `LD_LIBRARY_PATH`. You can inspect the value of `FHICL_FILE_PATH` by:

```
$ printenv FHICL_FILE_PATH
  .:$TOYEXPERIMENT_DIR
```

Acutally the output will show the translated value of the environment variable `TOYEXPERIMENT_DIR`. The presence of the current working directory (dot) in the path is redundant when processing the command line argument but it is significant in the case discussed in the next section.

Some experiments have chosen to configure their version of the *art* main program so that it will not look for the command line argument FHiCL file in `FHICL_FILE_PATH`. It is also possible to configure *art* so that only relative paths, not absolute paths, are legal values of the `-c` argument. This last option can be used to help ensure that only version-controlled files are used when running production jobs. Experiments may enable or disable either of these options when their main program is built.

8.11.2 `#include` Files

Section 8.7 discussed Listing 8.4, which contains the fragments of `hello.fcl` that are related to configuring the message service. The first line in that listing is an include directive. *art* will look for the file named by the include directive relative to `FHICL_FILE_PATH` and it will find it in:

```
$TOYEXPERIMENT_DIR/fcl/minimalMessageService.fcl
```

This is part of the toyExperiment `UPS` product.

The version of *art* used in the Workbook does not consider the argument of the include directive as an absolute path or as a path relative to the current working directory; it only looks for files relative to `FHICL_FILE_PATH`. This is in contrast to the choice made when processing the `-c` command line option.

When building *art*, one may configure *art* to first consider the argument of the include directive as a path and to consider `FHICL_FILE_PATH` only if that fails.

Add a section called Review that looks at trigger paths, end paths, etc and works backwards

9 Exercise 2: Build and Run Your First Module

9.1 Introduction

In this exercise you will build and run a simple *art* module. Section 2.6.7 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 to the run-time environment (discussed in Section 8.10)

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 9.4.1, and skip Section 9.5.

If you are an advanced user and wish to manage your working directory yourself, skip Section 9.4.1, and follow the steps in Section 9.5, then go back to Section 9.4.2 and 9.4.4 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.

9.2 Prerequisites

Before running this exercise, you need to be familiar with the material in Part I (Introduction) of this documentation set and Chapter 8 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 *defintion* 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.

9.3 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

9.4 Setting up to Run Exercises: Standard Procedure

9.4.1 "Source Window" Setup

In your source window do the following:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Table 4.1
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 $ART_WORKBOOK_WORKING_BASE/<username>/workbook`
5. Set up the source code management system `git`; check the output for each step in Section 9.4.2.1:
   (a) `$ setup git`
   (b) `$ git clone http://cdcvs.fnal.gov/projects/art-workbook`

   (c) `$ cd art-workbook`
   This will be referred to as your source directory.

   (d) `$ git checkout -b v0_00_13  v0_00_13`

6. `$ source ups/setup_deps -p $ART_WORKBOOOK_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.

9.4.2 Examine Source Window Setup

9.4.2.1 About `git` and What it Did

`git` is a source code management system[2] that is used to hold the source code for the Workbook exercises. A source code managment 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 `PATH`[3].

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_13 '
```

If you do not see the expected output, contact the *art* team as described in Section 2.4. 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).

[2]Other source code management systems with which you may be familar are `cvs` and `svn`.
[3]No version needs to be supplied because the `git` UPS product has a current version declared; see Section 6.4.

9.4.2.2 Contents of the Source Directory

At the end of the setup procedure, see what your source directory contains:

```
$ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook
$ ls
admin  art-workbook  CMakeLists.txt  ups
```

(Yes, it contains a subdirectory of the same name as its parent, art-workbook.)

- The admin directory contains some scripts used by **cetbuildtools** to customize the configuration of the development environment.
- The art-worbook directory contains the main body of the source code.
- The file CMakeLists.txt is the file that the build system reads to learn what steps it should do.
- The ups directory contains information about what UPS products this product depends on; it contains additional information used to configure the development environment.

Look inside the art-workbook ("junior") directory (via ls) and see that it contains several files and subdirectories. The file CMakeLists.txt contains more instructions for the build system. Actually every directory contains a CMakeLists.txt; each contains additional instructions for the build system. The subdirectory FirstModule contains the files that will be used in this exericse; the remaining subdirectories contain files that will be used in subsequent Workbook exercises.

If you look inside the FirstModule directory, you will see

```
CMakeLists.txt      FirstAnswer01_module.cc  First_module.cc
firstAnswer01.fcl   first.fcl
```

The file CMakeLists.txt in here contains yet more instructions for the build system and will be discussed later. The file First_module.cc is the first module that you will look at and first.fcl is the FHiCL file that runs it. This exercise will suggest that you try to write some code on your own; the answer is provided in FirstAnswer01_module.cc and the file firstAnswer01.fcl runs it. These files will be discussed at length during these exercises.

9.4.3 "Build Window" Setup

Now go to your build window and do the following:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Chapter 4
3. $ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook

4. `$ mkdir build-prof`
5. `$ cd build-prof`
   This new directory will be your *build* directory.
6. `$ source ../art-workbook/ups/setup_for_development \`
   `-p $ART_WORKBOOK_QUAL`
   The output from this command will tell you to take some additional steps; do not do those steps.
7. `buildtool`
   This step may take a few minutes.

9.4.4 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 2.6.7)

Step 6 sources a script called `setup_for_development` found in the `ups` subdirectory 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 8.8.3. The expected output is shown in Listing 9.1.

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 9.1: Example of output created by `setup_for_development`

```

The working build directory is /ds50/app/user/kutschke/workbook/build-prof
The source code directory is /ds50/app/user/kutschke/workbook/art-workbook
----------- check this block for errors -----------------------
----------------------------------------------------------------
/ds50/app/user/kutschke/workbook/build-prof/lib has been added to LD_LIBRARY_PATH
/ds50/app/user/kutschke/workbook/build-prof/bin has been added to PATH

CETPKG_SOURCE=/ds50/app/user/kutschke/workbook/art-workbook
CETPKG_BUILD=/ds50/app/user/kutschke/workbook/build-prof
CETPKG_NAME=art_workbook
CETPKG_VERSION=v0_00_13
CETPKG_QUAL=e2:prof
CETPKG_TYPE=Prof

Please use this cmake command:
```

```
cmake -DCMAKE_INSTALL_PREFIX=/install/path -DCMAKE_BUILD_TYPE=$CETPKG_TYPE $CETPKG_SOURCE
```

This script sets up all of the UPS products on which the Workbook depends; this is analogous to the actions taken by Step 5 in the first exercise (Section 8.4.2.1) when you were working in the *art* run-time environment. This script also creates several files and directories in your `build-prof` directory; these comprise the working space used by **cetbuildtools**.

After sourcing this script, the contents of `build-prof` will be

```
art_workbook-v0_00_13  bin  cetpkg_variable_report  diag_report  lib
```

At this time the two subdirectories `bin` and `lib` will be empty. The other files are used by the build system to keep track of its configuration.

Step 7 (`buildtool`) tells **cetbuildtools** to build everything found in the source directory; this includes all of the Workbook exercises, not just the first one. The build process will take two or three mintues on an unloaded (not undergoing heavy usage) machine. Its output should end with the lines:

```
------------------------------------
INFO: Stage build successful.
------------------------------------
```

After the build has completed do an `ls` on the directroy `lib`; you will see that it contains a large number of shared library (`.so`) files; for `v0_00_13` there will be 29 `.so` files; these are the files that *art* will load as you work through the exercises.

Also do an `ls` on the directory `bin`; these are scripts that are used by **cetbuildtools** to maintain its environment; if the Workbook contained instructions to build any executable programs, they would have been written to this directory.

After runing `buildtool`, the build directory will contain:

```
admin                   CMakeFiles               fcl
art-workbook            cmake_install.cmake      inputFiles
art_workbook-v0_00_13   CPackConfig.cmake        lib
bin                     CPackSourceConfig.cmake  Makefile
cetpkg_variable_report  CTestTestfile.cmake      output
CMakeCache.txt          diag_report              ups
```

Most of these files are standard files that are explained in the **cetbuildtools** documentation. . However, three of these items need special attention here because they are customized for the Workbook.

An `ls -l` on the files `fcl`, `inputFiles` and `output` will reveal that they are symbolic links to

```
inputFiles -> ${TOYEXPERIMENT_DIR}/inputFiles
    output -> ${ART_WORKBOOK_OUTPUT_BASE}/<username>/art_workbook_output
```

```
fcl -> <your source directory>/art-workbook
```

These links are present so that the FHiCL files for the Workbook exercises can be machine-independent.

- The link `inputFiles` points to the directory inputFiles present in the toyExperiment UPS product; this directory contains the input files that *art* will read when you run the first exercise. These are the same files used in the first exercise; if you need a reminder of the contents of these files, see Table 8.1. These input files will also be used in many of the subsequent exercises.
- The link `outputFiles` points to a directory that was created to hold your output files; the environment variable `ART_WORKBOOK_OUTPUT_BASE` was defined by your site-specific setup procedure.
- The link `fcl` points into your source directory hierarchy; it allows you to access the FHiCL files that are found in that hierarchy with the convenience of tab completions.

9.5 Setting up to Run Exercises: Self-managed Working Directory

If you have worked through Section 9.4, skip this section and proceed to Section 9.6.

The explanation for the steps in these procedures that are the same as for the “standard” procedures are found in Section 9.4.2 (for the source window) and Section 9.4.4 (for the build window).

In your source window do the following:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Chapter 4
3. Make a working directory to hold the checked out source
4. cd to the directory made in the previous step
5. Ensure that `git` is in your `PATH`
6. `$ git clone http://cdcvs.fnal.gov/projects/art-workbook`
7. `$ cd art-workbook`
   In the following, this will be refered to as your source directory.
8. `$ git checkout -b v0_00_13  v0_00_13`

Now go to your build window and do the following:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Chapter 4
3. Make a directory to hold the code that you will build; this is your build directory in your build window. In the following, this will be refered to as your build directory.
4. cd to your build directory
5. Make a directory, *outside of the heirarchy rooted at your build directory*, to hold output files created by the workbook exercises.
6. $ ln -s <directory-made-in-previous-step> output
7. $ source <your-source-directory>/ups/setup_for_development -p $ART_WORKBOOK_QUAL
   The output from this command will tell you to take some additional steps; do not do those steps.
8. $ buildtool

9.6 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:

In your source window:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Table 4.1
3. cd to your source directory
   $ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook
4. source ups/setup_deps -p

In your build window:

1. Log in to the computer you chose in Section 7.3.
2. Follow the site-specific setup procedure; see Chapter 4
3. cd to your build directory
   $ cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/build-prof
4. $ source ../art-workbook/ups/setup_for_development -p $ART_WORKBOOK_QUAL

If you chose to manage your own directory names (ie you followed Section 9.5), then the names of your source and build directories will be different than those shown.

Compare these steps with those given in Sections 9.4.1 and Section 9.4.3. You will see that five steps and are missing from the source window instructions and three steps are missing from the build window instructions. The missing steps only needed to be executed the first time.

9.7 The *art* Development Environment

In the preceeding sections of this chapter you established what is known as the *art development environment*; this is a superset of the *art* run-time environment, which was described in Section 8.10. This section summarizes the new elements that are part of the development environment but not part of the run-time environment.

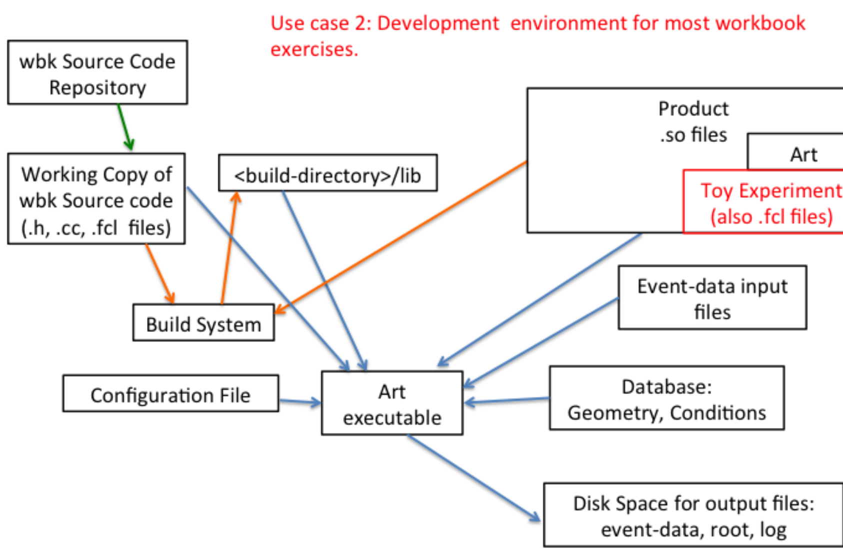


Figure 9.1: Elements of the *art* development environment as used in most of the Workbook exercises; the arrows denote information flow, as described in the text.

In Section 9.4.1, step 5b (`git clone ...`) was to contact the source code repository and make a clone of the repository in your disk space; step 5d `git checkout ...`) was to check out the correct version of the code from the clone and to put it into your source directory. The repository is hosted on a central Fermilab server and is accessed via the network. The upper left box in Figure 9.1 denotes the repository and the box below it denotes your working copy of the Workbook code. The flow of information during the clone and checkout processes is indicated by the green arrow in the figure.

In step 7 of Section 9.4.3, you ran `buildtool`, which read the source code files from your working copy of the Workbook code and turned them into shared libraries. The script `buildtool` is part of the build system, which is denoted as the box in the center left section of the figure. When you ran `buildtool`, it wrote shared library files to the `lib` subdirectory of your build directory; this directory is denoted in the figure as the box in the top center labeled `<build-directory>/lib`. The orange arrows in the figure denote the information flow at build-time. In order to perform this task, `buildtool` also needed to read header files and shared libraries found in the UPS products area, hence the orange arrow leading from the UPS Products box to the build system box.

In the figure, information flow at run-time is denoted by the blue lines. When you ran the *art* executable, it looked for shared libraries in the directories defined by `LD_LIBRARY_PATH`. In the *art* development environment, `LD_LIBRARY_PATH` contains

1. the `lib` subdirectory of your build directory.
2. all of the directories previously described in Section 8.9

In all environments, the *art* executable looks for FHiCL files in

1. in the file specified in the `-c` command line argument
2. in the directories specified in `FHICL_FILE_PATH`

The first of these is denoted in the figure by the box labeled "Configuration File." In the *art* development environment, `FHICL_FILE_PATH` contains

1. some directories found in your checked out copy of the source
2. all of the directories previously described in Section 8.11

The remaining elements in Figure 9.1 are the same as described for Figure 8.1.

9.8 Running the Exercise

9.8.1 Run *art* on `first.fcl`

In your build window, make sure that your current working directory is your build directory. From here, run the first part of this exercise by typing the following:

```
$ art -c fcl/FirstModule/first.fcl > output/first.log
```

(We suggest you get in the habit of routing your output to the output directory.) The output of this step will look much like that in Listing 8.1, but with two signficant differences. The first difference is that the output from `first.fcl` contains an additional line

```
Hello from First::constructor.
```

The second difference is that the words printed out for each event are a little different; the printout from `first.fcl` looks like

```
Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
```

while that from `hello.fcl` looked like

```
Hello World! This event has the id: run: 1 subRun: 0 event: 1
```

The reason for changing this printout is so that you can identify, from the printout, which module was run.

9.8.2 The FHiCL File `first.fcl`

Compare the FHiCL file used in this exercise, `fcl/FirstModule/first.fcl`, with `hello.fcl` from the first exercise (i.e., run `cat` or `diff` on them). Other than comments, the only difference is that the `module_type` has changed from `HelloWorld` to `First`:

```
$ diff $TOYEXPERIMENT_DIR/HelloWorldScripts/hello.fcl \
 fcl/FirstModule/first.fcl
...
<        module_type : HelloWorld
---
>        module_type : First
```

The file `first.fcl` tells *art* to run a module named `First`. As described in Section 8.9, *art* looks through the directories defined in LD_LIBRARY_PATH and looks for a file whose name matches the pattern `lib*First_module.so`. This module happens to be found at this location, relative to your build directory:

```
lib/libart-workbook_FirstModule_First_module.so
```

This shared library file was created when you ran `buildtool`.

9.8.3 The Source Code File `First_module.cc`

This section will describe the source code for the module `First` and will use it as a model to describe modules in general. The source code for this module is found in the following file, relative to your source directory (go to your source window!):

```
art-workbook/FirstModule/First_module.cc
```

For convenience, the contents of the file is also shown in Listing 9.2. When you ran `buildtool`, it compiled and linked this source file into the following shared library (relative to your your build directory):

`lib/libart-workbook_FirstModule_First_module.so`

This is the shared library that was loaded by *art* when you ran code for this exercise, in Section 9.8.2.

In broad strokes, the file `First_module.cc`:

- declares a class named `First`
- provides the implementation for the class
- contains a call to the C-Preprocessor macro named `DEFINE_ART_MODULE`, discussed in Section 9.8.3.7

All module files that you will see in the Workbook share these "broad strokes." Some experiments that use *art* have chosen to split the source code for one module into three separate files; the *art* team does not recommend this practice, but it is in use and it will be discussed in Section 9.11.2.

Listing 9.2: The contents of `First_module.cc`

```

#include ``art/Framework/Core/EDAnalyzer.h''
#include ``art/Framework/Core/ModuleMacros.h''
#include ``art/Framework/Principal/Event.h''

#include <iostream>

namespace tex {

  class First : public art::EDAnalyzer {

  public:

    explicit First(fhicl::ParameterSet const& );

    void analyze(art::Event const& event) override;

  };

}

tex::First::First(fhicl::ParameterSet const& ){
  std::cout << ``Hello from First::constructor.'' << std::endl;
}

void tex::First::analyze(art::Event const& event){
  std::cout << ``Hello from First::analyze. Event id: ``
            << event.id()
            << std::endl;
}

DEFINE_ART_MODULE(tex::First)
```

Those of you with some C++ experience will have noticed that there is no file named `First_module.h` in the directory `art-workbook/FirstModule`. The explanation for this will be given in Section 9.11.1.

9.8.3.1 The `#include` Files

The first three non-blank lines in Listing 9.2 are three *include directives* that include header files. All three of these files are included from the *art* UPS product (where to find included header files is discussed in Section 6.6).

If you are a C++ beginner you will likely find these files difficult to understand; you do not need to understand them at this time but you do need to know where to find them for future reference.

The next non-blank line, `#include <iostream>`, includes the C++ header that enables this code to write output to the screen; for details, see any standard C++ documentation.

9.8.3.2 The Declaration of the Class `First`

The next portion of Listing 9.2 starts with the line "`namespace tex {`" which opens the namespace `tex` (the namespace is closed with a "}" about half way down the listing). If you are not familiar with namespaces, consult the standard C++ documentation.

All of the code in the toyExperiment UPS product was written in a namespace named `tex`; the name `tex` is an acronym-like shorthand for the toyExperiment (ToyEXperiment) UPS product. In order to keep things simple, all of the classes in the Workbook are also declared in the namespace `tex`. For more information about this choice, see Section 6.6.4.

The namespace contains the declaration of a class named `First`, which has only two members:

1. a constructor, described in Section 9.8.3.3
2. a member function, named `analyze`, described in Section 9.8.3.5

art will call the constructor once at the start of each job and it will call `analyze` once for each event.

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` is a (public[4]) *derived class* that *inherits* from a *base class* named `art::EDAnalyzer`. At this time it is not necessary to understand

[4] The members of this class can be accessed by member and nonmember functions.

C++ inheritance, base classes or derived classes; just follow the pattern when you write you own modules.

Section 2.6.3 discussed the idea of *module types*: analyzer, producer, filter and so on. If a class inherits from art::EDAnalyzer then the class is an analyzer module and it will have the properties of an analyzer module that were discussed in Section 2.6.3.

For a class to be a valid *art* analyzer module, it must follow a set of rules defined by *art*:

1. It must inherit from art::EDAnalyzer.
2. It must provide a constructor with the argument list:
   fhicl::ParameterSet const&
3. It must provide a member function named analyze, with the signature[5]:
   analyze(art::Event const&)
4. If the name of a module class is <ClassName> then the source code for the module must be in a file named <ClassName>_module.cc and this file must contain the lines:
   ```
   #include ``art/Framework/Core/ModuleMacros.h''
   DEFINE_ART_MODULE(<namespace>::<ClassName>)
   ```
5. It may optionally provide other member functions with signatures prescribed by *art*; if these member functions are present in a module class, then *art* will call them at the appropriate times. Some examples are provided in Chapter 10.

You can see from Listing 9.2 that the class First follows all of these rules and that it does not contain any of the optional member functions.

A module may also contain any other member data and any other member functions that are needed to do its job.

The next line of the class declaration is:

```
public:
```

which tells the compiler that *art* is permitted to call the constructor First and the member function analyze[6].

The next line of the class declaration declares a constructor with the argument list prescribed by *art*:

```
First(fhicl::ParameterSet const& );
```

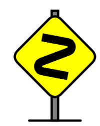
[5] In C++ the *signature* of a member function is the name of the class of which the function is a member, the name of the function, the number, types and order of the arguments, and whether the member function is marked as const or volatile. The signature does not include the return type.

[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.

The requirement that the class name match the filename (minus the `_module.cc` portion) is enforced by *art*'s system for dynamically loading shared libraries. The requirement that the class provide the prescribed constructor is enforced by the macro `DEFINE_ART_MODULE`, which will be described in Section 9.8.3.7. And the last line of the class declaration declares the member function, `analyze` with the argument list required by *art*:

```
  analyze( art::Event const &) override;
```

The *override* contextual keyword is a feature that is new in C++ 11 so older references will not discuss it. It is a new safety feature that we recommend you use; we cannot give a proper explanation until we have had a chance to discuss inheritance further. For now, just consider it a rule that, in all analyzer modules, you should provide this keyword as part of the declaration of `analyze`.



For those who are knowledgeable about C++, the base class `art::EDAnalyzer` declares the member function `analyze` to be pure virtual; so it must be provided by the derived class. The optional member functions of the base class are declared virtual but not pure virutal; do-nothing versions of these member functions are provided by the base class.

In a future version of this documentation suite, more information will be available in the Users Guide in Chapter .

9.8.3.3 The Constructor for the Class `First`

In Listing 9.2, following the class declaration and the closing brace of the namespace, is the *definition* of the constructor:

```
tex::First::First(fhicl::ParameterSet const& ){
  std::cout << ``Hello from First::constructor.'' << std::endl;
}
```

It has the argument required by *art* (`fhicl::ParameterSet const&`). This constructor simply prints some information (via `std::cout`) to let the user know that it has been called.

The fragment `tex::First::First` should be parsed as follows: the part `First::First` says that this definition is for a constructor of the class `First`. In principle there might be many classes named `First`, each in a different namespace; the leading `tex::` says that this is the constructor for the class named `First` that is found in the namespace `tex`.

The argument to the constructor is of type `fhicl::ParameterSet const&`; the class `ParameterSet`, found in the namespace `fhicl`, is a C++ representation of a FHiCL parameter set (aka FHiCL *table*). This argument is not used in this exercise; you will see how it is used in Chapter 11.

You will also notice that the argument to the constructor is passed by `const` reference, `const&`. This is a requirement specified by *art*; if you write a constructor that does not have exactly the correct argument type, then the compiler will issue a diagnostic and will stop compilation. Because the argument is `const`, your code may not modify it; because it is passed by reference, it is efficient to pass a large parameter set. If you are not familiar with `const`'ness or with passing arguments by reference, consult the standard C++ documentation.

9.8.3.4 Aside: Unused Formal Parameters

You have probably noticed that neither the declaration of the constructor nor the definition of the constructor provided a name for the argument of the constructor; both only provided the type. This section describes why the name was omitted.

Each argument of a function (remember that a constructor is just a special kind of function) has a type and a *formal parameter*; in casual use most of us refer to the *formal parameter* as the name of the argument.

In a function definition, if a formal parameter is unused in the body of the function (i.e., between the braces {}) then the C++ standard says that the formal parameter is optional; it is common to provide formal parameters in function declarations as a form of documentation but the compiler always ignores these formal parameters. Even when the formal parameter is omitted, the type is still required because the full name of the function includes the number, type and order of its arguments.

In the case of the Workbook, however, **cetbuildtools** has been configured to go one step further. It enforces the following rule:

- If a function has a formal parameter that is not used by the defintion of the function, and if you *intend* that it not be used, then you must omit that formal parameter when writing the argument list in the definition.

Consequently, if the compiler sees a formal parameter that is not used by the definition of the function, it will presume that this is an error and it will issue a diagnostic that stops compilation.

cetbuildtools is configured this way because an unused formal parameter is frequently an indication of an error and the authors of the Workbook recommend that we make full use of all safety features provided by the compiler. It is easy enough to indicate to the compiler what your intention is; so we say "Just do it!"

Your experiment's build system might or might not be configured to follow this rule. It might permit unused formal parameters in function definitions or it might consider this situation to warrant a warning level diagnostic, not an error level diagnostic.

9.8.3.5 The Member Function `analyze` and `art::Event`

In Listing 9.2, following the definition of the constructor, you will find the definition of the member function `analyze`:

```
void tex::First::analyze(art::Event const& event){
  std::cout << ``Hello from First::analyze. Event id: ``
            << event.id()
            << std::endl;
}
```

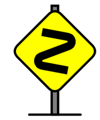
The `override` contextual keyword that was present in the declaration of this member function is not present in its definition; this is standard C++ usage.

This function has the argument list required by *art* (`art::Event const& event`). If the type of the argument is not exactly correct, including the the `const&`, the compiler will issue a diagnostic and stop compilation. The compiler is able to do this because of one of the features of *inheritance*: it requires that the member function named `analyze` have exactly the signature specified by the base class (the details of how this works is beyond the scope of this discussion).

Section 2.6.1 discussed the HEP idea of an event and the *art* idea of a three-part event identifier. The class `art::Event` is the representation within *art* of the HEP notion of an event. For the present discussion it is safe to consider the following over-simplified view of an event: it contains an event identifier plus a collection of data products (see Section 2.6.4). The name of the formal parameter `event` has no meaning either to *art* or to the compiler – it is just an identifier – but your code will be easier to read if you choose a meaningful name.

At any given time in a running *art* program there is only ever one `art::Event` object; in the rest of this paragraph we will call this object *the event.* It is owned and managed by *art*, but *art* lets analyzer modules see the contents of the event; it does so by passing the event by `const` reference when it calls the `analyze` member function of analyzer modules. Because the event is passed by reference (indicated by the `&`), the member function `analyze` does not get a copy of the event; instead it is told where to find the event. This makes it efficient to pass an event object even if the event contains a lot of information. Because the argument is a `const` reference, if your code tries to change the contents of the event, the compiler will issue a diagnostic and stop compilation.

As described in Section 2.6.3, analyzer modules may only inspect data in `event`, not modify it. This section has shown how *art* institutes this policy as a hard



rule that will be enforced rigorously by the compiler:

1. The compiler will issue an error if an analyzer module does not contain an member function named `analyze` with exactly the correct signature.
2. In the correct signature, the formal parameter `event` is a `const` reference.
3. Because `event` is const, the compiler will issue an error if the module tries to call any member function of `art::Event` that will modify the event.

You can find the header file for `art::Event` by following the guidelines described in Section 6.6.2. A future version of this documentation will contain a chapter in the Users Guide that provides a complete explanation of `art::Event`. Here, and in the rest of the Workbook, the features of `art::Event` will explained as needed.

The body of the function is almost trivial: it prints some information to let the user know that it has been called. In Section 9.8.1, when you ran *art* using `first.fcl`, the printout from the first event was

```
Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
```

If you compare this to the source code you can see that the fragment

```
 << event.id()
```

creates the following printout

```
  run: 1 subRun: 0 event: 1
```

This fragment tells the compiler to do the following:

1. In the class `art::Event`, find the member function named `id()` and call this member function on the object `event`.
2. Whatever is returned by this function call, find its stream insertion operator and call it.

From this description you can probably guess that the member function `art::Event::id()` returns an object that represents the three part event identifier. In Section 9.8.3.6 you will learn that this guess is correct.

9.8.3.6 `art::EventID`

Before you work through this section, you may wish to review Section 6.6 which discusses how to find header files.

Section 2.6.1 discussed the idea of an event identifier, which has three components, a run number, a subRun number and event number. In this section you will learn where to find the class that *art* uses to represent an event identifier.

Rather than simply telling you the answer, this section will guide you through the process of discovering the answer for yourself.

In the previous section you looked at some code and the printout that it made; this strongly suggested that the member function `art::Event::id()` returns an object that represents the event identifier. To follow up on this suggestion, look at the header file for `art::Event`:

```
$ less  $ART_INC/art/Framework/Principal/Event.h
```

Or use one of the code browsers discussed in 6.6.2. In this file you will find the definition of the member function `id()`:[7]

```
  EventID
  id() const {return aux_.id();}
```

The important thing to look at here is the return type, `EventID`, which looks like a good candidate to be the class that holds the event identifier; you do not need to (or want to) know anything about the data member `aux_`. If you look near the beginning of `Event.h` you will see that it has the line:

```
#include "art/Persistency/Provenance/EventID.h"
```

which looks like a good candidate to be the header file for `EventID`. Look at this header file,

```
 $ less $ART_INC/art/Persistency/Provenance/EventID.h
```

In this file you will discover that it is indeed the header file for `EventID`; you will also see that the class `EventID` is within the namespace `art`, making its full name `art::EventID`. Near the top of the ifle you will also see the comments:

```
// An EventID labels an unique readout of the data acquisition system,
// which we call an ``event''.
```

This is another clue that `art::EventID` is the class we are looking for. Look again at `EventID.h`; you will see that it has accessor methods that permit you see the three components of the an event identfier:

```
  RunNumber_t    run()    const;
  SubRunNumber_t subRun() const;
  EventNumber_t  event()  const;
```

Earlier in `EventID.h` the C++ *type*[8] `EventNumber_t` was defined as:

```
namespace art {
  typedef std::uint32_t EventNumber_t;
```

[7] In 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.

[8] In 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.

```
}
```

meaning that the event number is represented as a 32-bit unsigned integer. If you are not familiar with the C++ concept of *typedef*, or if you are not familiar with the definite-length integral types defined by the <cstdint> header, consult any standard C++ documentation. If you dig deeper into the layers included in the art::EventID header, you will see that the run number and subRun number are also implemented as 32-bit unsigned integers.

At this point you can be sure that art::EventID is the class that *art* uses to represent the three part event identifier: the class has the right functionality. It's also true that the comments agree with this hypothesis but comments are often ill-maintained; be wary of comments and always read the code. This is a fairly typical tour through the layers of software.

The authors of *art* might have chosen an alternate definition of EventNumber_t

```
namespace art {
  typedef unsigned EventNumber_t;
}
```

The difference is the use of unsigned rather than std::uint32_t. This alternate version was not chosen because it runs the risk that some computers might consider this type to have a length of 32 bits while other computers might consider it to have a length of 16 or 64 bits. In the defintion that is used by *art*, an event number is guaranteed to be exactly 32 bits on all computers.

Why did the authors of *art* insert the extra level of indirection and not simply define the following member function inside art::EventID?

```
  std::unit32_t event()  const;
```

The answer is that it makes it easy to change the definition of the type should that be necessary. If, for example, an experiment requires that event numbers be of length 64 bits, only one change is needed, followed by a recompilation.

It is good practice to use typedefs for every concept for which the underlying data type is not absolutely certain.

It is a very common, but not universal, practice within the HEP C++ community that typedefs that are used to give context-specific names to the C++ built-in types (int, float, char etc) end in _t.

9.8.3.7 DEFINE_ART_MACRO: The Module Maker Macros

The final line in First_module.cc invokes a C preprocessor macro:

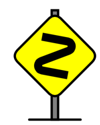
```
DEFINE_ART_MODULE(tex::First)
```

This macro is defined in the header file that was included by:

```
#include ``art/Framework/Core/ModuleMacros.h''
```

If you are not familiar with the C preprocessor, don't worry; you do not need to look under the hood. But if you would like to learn more about the C preprocessor, consult any standard C++ reference.

The `DEFINE_ART_MODULE` macro instructs the compiler to put some additional code into the shared library made by `buildtool`. This additional code provides the glue that allows *art* to create instances of the class `First` without ever seeing the header or the source for the class; it only gets to see the `.so` and nothing else.



The `DEFINE_ART_MODULE` macro adds two pieces of code to the `.so` file. It adds a factory function that, when called, will create an instance of `First` and return a pointer to the base classes `art::EDAnalyzer`. In this way, *art* never sees the derived type of any analyzer module; it sees all analyzer modules via pointer to base. When art calls the factory function, it passes as an argument the parameter set specified in the FHiCL file for this module instance. The factory function passes this parameter set through to the constructor of `First`. The second piece of code put into the `.so` file is a static object that will be instantiated at load time; when this object is constructed, it will contact the *art* module registry and register the factory function under the name `First`. When the FHiCL file says to create a module of type `First`, *art* will simply call the registered factory function, passing it the parameter set defined in the FHiCL file. This is the last step in making the connection between the source code of a module and the *art* instantiation of a module.

9.8.3.8 Some Alternate Styles

C++ allows some flexibility in syntax, which can be seen as either powerful or confusing, depending on your level of expertise. Here we introduce you to a few alternate styles that you will need to recognize and may want to use.

Look at the `std::cout` line in the `analyze` method of Listing 9.2:

```
  std::cout << ``Hello from First::analyze. Event id: ``
            << event.id()
            << std::endl;
}
```

This could have been written:

```
  art::EventID id = event.id();
  std::cout << "Hello from First::analyze. Event id: "
            << id
            << std::endl;
```

This alternate version explicitly creates a temporary object of type `art::EventID`, whereas the original version created an *implicit* temporary object. When you 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.

Another style that you will certainly encounter is to write the first line of the above as:

```
art::EventID id(event.id());
```

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.

You may also see the argument list of the analyze function written a little differently,

```
void  analyze( const art::Event& );
```

instead of

```
void  analyze( art::Event const& );
```

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.

Finally, Listing 9.3 shows the same information as Listing 9.2 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 9.3: An alternate layout for `First_module.cc`

```

#include ``art/Framework/Core/EDAnalyzer.h''
#include ``art/Framework/Core/ModuleMacros.h''
#include ``art/Framework/Principal/Event.h''

#include <iostream>

namespace tex {

  class First : public art::EDAnalyzer {

  public:

    explicit First(fhicl::ParameterSet const& );

    void analyze(art::Event const& event) override;

  };
```

```

  First::First(fhicl::ParameterSet const& ){
    std::cout << ``Hello from First::constructor.'' << std::endl;
  }

  void First::analyze(art::Event const& event){
    std::cout << ``Hello from First::analyze. Event id: ``
              << event.id()
              << std::endl;
  }

}

DEFINE_ART_MODULE(tex::First)
```

9.9 What does the Build System Do?

9.9.1 The Basic Operation

In Section 9.4.3 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 directory. The instructions in CMakeLists.txt tell cetbuildtools to look for more instructions in the subdirectory ups and in the file art-workbook/CMakeLists.txt, which, in turn, tells it to look for more instructions in the CMakeLists.txt files in each subdirectory of art-workbook.

When cetbuildtools has digested these instructions it knows the rules to build everything that it needs to build.

The object file created by the compilation step is a temporary file and, once it has been inserted into the shared library, it is not used any more. Therefore the name of the object file is not important.

On the other hand, the name of the shared library file is very important. *art* requires that for every module source file (ending in _module.cc) the build system must create exactly one shared library file (ending in _module.so). It also requires that the name of each _module.so file conform to a pattern. Consider the example of the file First_module.cc; *art* requires that the shared library for this file match the pattern

```
lib*First_module.so
```

where the * wildcard matches 0 or more characters.

When naming shared libraries, buildtool uses the following algorithm, which satisfies the *art* requirements and adds some addtional features; the algorithm is illustrated using the example of First_module.cc:

1. find the relative path to the source file, starting from the source directory
   art-workbook/FirstModule/First_module.cc
2. replace all slashes with underscores
   art-workbook_FirstModule_First_module.cc
3. change the trailing .cc to .so
   art-workbook_FirstModule_First_module.so
4. add the prefix lib
   libart-workbook_FirstModule_First_module.so
5. put the file into the directory lib, relative to the build directory
   lib/libart-workbook_FirstModule_First_module.so

You can check that this file is there by issuing the following command from your build directory:

```
$ ls -l lib/libart-workbook_FirstModule_First_module.so
```

This algorithm guarantees that every module within art-workbook will have a unique name for its shared library.

The experiments using art have a variety of build systems. Some of these follow the minimal *art*-conforming pattern, in which the wildcard is replaced with zero characters. If the Workbook had used such a build system, the name of the shared library file would have been

```
lib/libFirst_module.so
```

Both names are legal. Some additional features that are enabled by including the full path in the name will be discussed in Section .

9.9.2 Incremental Builds and Complete Rebulds

When you edit a file in your source area you will need to rebuild that file in order for those changes to take effect. If any other files in your source area depend on the file that you edited, they too will need to be rebuilt. To do this, reissue the `buildtool` command:

```
$ buildtool
```

Remember that the `buildtool` command must be executed from your build directory and that, before running `buildtool`, you must have setup the environment in your build window. When you run this command, **cetbuildtools** will automatically determine which files need to be rebuilt and will rebuild them; it will not waste time rebuilding files that do not need to be rebuilt. This is called an *incremental build* and it will usually complete much faster than the initial build.

If you want to clean up everything in your build area and rebuild everything from scratch, use the following command:

```
$ buildtool -c
```

This command will give you five seconds to abort it before it starts removing files; to abort, type `ctrl-C` in your build window. It will take about the same time to execute as did your initial build of the Workbook. The name of the option `-c` is a mnemoic for "clean".

When you do a clean build it will remove all files in your build directory that are not managed by **cetbuildtools**. For example, if you redirected the output of *art* as follows,

```
$ art -c fcl/FirstModule/first.fcl >& first.log
```

then, when you do a clean build, the file `first.log` will be deleted. This is why the instructions earlier in this chapter told you to redirect ouptut to a log file by

```
$ art -c fcl/FirstModule/first.fcl >& output/first.log
```

When you ran `buildtool`, it created a directory to hold your output files and you created a symbolic link, named output, from your build directory to this new directory. Both the other directory and the symbolic link survive clean builds and your output files will be preserved. The Workbook exercises write all of their root and event-data output files to this directory.

If you edit certain files in the ups subdirectory of your source directory, rebuilding requires an extra step. If you edit one of these files, the next time that you run buildtool, it will issue an error message saying that you need to re-source setup_for_development. If you get this message, make sure that you are in your build directory, and

```
$ source ../art-workbook/ups/setup_for_development -p  $ART\_WORKBOOK\_QUAL
$ buildtool
```

9.9.3 Finding Header Files at Compile-time

When setup_for_development establishes the working environment for the build directory, it does a UPS setup on the UPS products that it requires; this triggers a chain of additional UPS setups. As each UPS product is set up, that product defines many enviroment variables, two of which are <PRODUCT-NAME>_INC and <PRODUCT-NAME>_LIB. The first of these points to a directory that is the root of the header file hierarchy for that version of that UPS product. The second of these points to a single directory that holds all of the shared library files for that UPS product.

You can spot-check this by doing, for example,

```
$ ls $TOYEXPERIMENT_INC/*
$ ls $TOYEXPERIMENT_LIB
$ ls $ART_INC/*
$ ls $ART_LIB
```

You will see that the _INC directories have a subdirectory tree underneath them while the _LIB directories do not.

There are a few small perturbations on this pattern. The most visible is that the ROOT product puts most of its header files into a single directory, $ROOT_INC and does not clone the directory heirarchy of its source files. The Geant4 product does the same thing.

When the compiler compiles a .cc file, it needs to know where to find the files specified by the #include directives. The compiler looks for included files by first looking for arguments on the command line of the form

```
-I<absolute-path-to-a-directory>
```

There may be many such arguments on one command line. The compiler assembles the set of all -I arguments and uses it as an include path; that is, it looks for the header files by trying the first directory in the path and if it does not find it there, it tries the second directory in the path, and so on. The choice of -I for the name of the argument is a mnemonic for Include.

When buildtool compiles a .cc file it adds many -I options to the command line; it adds one for each UPS product that was set up when you sourced setup_for_development. When building First_module.cc, buildtool added -I$ART_INC, -I$TOYEXPERIMENT_INC and many more.

A corollary of this discussion is that when you wish to include a header file from a UPS product, the #include directive must contain the relative path to the desired file, starting from the _INC environment variable for that UPS product.

This system illustrates how the Workbook can work the same way on many different computers at many different sites. As the author of some code, you only need to know paths of include files relative to the relevant _INC environment variable. This environment variable may have different values from one computer to another but the setup and build systems will ensure that the site specific information is communicated to the compiler using environment variables and the -I option.

This system has the potential weakness that if two products each have a header file with exactly the same relative path name, the compiler will get confused. Should this happen, the compiler will always choose the file from the earlier of the two -I arguments on the command line, even when the author of the code intended the second choice to be used. To mitgate this problem, the *art* and UPS teams have adopted the convention that, whenever possible, the first element of the relative path in an #include directive will be the UPS package name. It is the implementation of this convention that led to the repeated directory name art-workbook/art-workbook that you saw in your source directory. There are a handful of UPS products for which this pattern is not followed and they will be pointed out as they are encountered.

The convention of having the UPS product name in the relative path of #include directives also tells readers of the code where to look for the included file.

9.9.4 Finding Shared Library Files at Link-time

The module First_module.cc needs to call methods of the class art::Event. Therefore the compiler left a notation in the object file saying "to use this object file you need to tell it where to find art::Event."[9] The technical way to say this is that the object file contains a list of *undefined symbols* or *undefined external references*. When the linker makes the shared library

libart-workbook_FirstModule_First_module.so

it must resolve all of the undefined symbols from all of the object files that go 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.

defines that symbol. When it discovers the answer, it will write the name of that shared library into something called the *dependency list* of

`libart-workbook_FirstModule_First_module.so`

A dependency list is kept inside each shared library. **cetbuildtools** tells the linker that the dependency list should contain only the filename of each shared library, not the full path to it. If, after the linker has finished, there remain unresolved symbols, then the linker will issue an error message and the build will fail.

Dependency lists are not recursive. If library A depends on library B and library B depends on library C, then the dependency list library A needs to contain only library B. Sometimes this is discussed by saying that the dependency list needs to contain only *direct dependencies* or *first order* dependencies.

To learn where to look for symbol definitions, the linker looks at its command line to find something called the *link list*. The link list can be specified in several different ways and the way that **cetbuildtools** uses is simply to write the link list as the absolute path to every `.so` file that the linker needs to know about. The link list can be different for every shared library that the build system builds. However it is very frequently true that if a directory contains several modules, then all of the modules will require the same link list. The bottom line is that the author of a module needs to know the link list that is needed to build the shared library for that module.

For these Workbook exercises, the author of each exercise has determined the link list for each shared library that will be built for that exercise. In the **cetbuildtools** system, the link list for `First_module.cc` is located in the `CMakeLists.txt` file from same directory as `First_module.cc`; the contents of this file are shown in Listing 9.4. This `CMakeLists.txt` file says that all modules found in this directory should be built with the same link list and it gives the link list; the link list is the seven lines that begin with a dollar sign; these lines each contain one `cmake` variable. Recall that **cetbuildtools** is a build system that lives on top of `cmake`, which is another build system. A `cmake` variable is much like an environment variable except that is only defined within the environment of the running build system; you cannot look at it with printenv.

The five `cmake` variables beginning with `ART_` were defined when `buildtool` set up the UPS *art* product. Each of these variables defines an absolute path to a shared library in `$ART_LIB`. For example `${ART_FRAMEWORK_CORE}` resolves to

`$ART_LIB/libart_Framework_Core.so`

Almost all *art* modules will depend on these five libraries. Similarly the other two variables resolve to shared libraries in the **fhiclcpp** and **cetlib** UPS products.

When **cetbuildtools** constructs the command line to run the linker, it copies the link list from the `CMakeLists.txt` file to the command linker line.

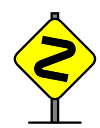
The experiments that use *art* use a variety of build systems. Some of these build systems do not require that all external symbols be resolved at link-time; they allow some external symbols to be resolved at run-time. This is legal but it can lead to certain difficulties. *A future version of this documentation suite will contain a chapter in the Users Guide that discusses linkage loops and how use of closed links can will prevent them. This section will then just reference it.*

Consult the `cmake` and **cetbuildtools** documentation to understand the remaining details of this file.

Listing 9.4: The `file art-workbook/FirstModule/CMakeLists.txt`

```
art_make(MODULE_LIBRARIES
  ${ART_FRAMEWORK_CORE}
  ${ART_FRAMEWORK_PRINCIPAL}
  ${ART_PERSISTENCY_COMMON}
  ${ART_FRAMEWORK_SERVICES_REGISTRY}
  ${ART_FRAMEWORK_SERVICES_OPTIONAL}
  ${FHICLCPP}
  ${CETLIB}
  )
```

9.9.5 Build System Details



This section provides the next layer of details about the build system; *in a future version of this documentation set, the Users Guide will have a chapter with all of the details.* This entire section contains expert material.

If you want to see what `buildtool` is actually doing, you can enable verbose mode by issuing the command:

```
$ buildtool VERBOSE=TRUE
```

For example, if you really want to know the name of the object file, you can find it in the verbose output. For this exercise, the object file is

```
./art-workbook/FirstModule/CMakeFiles/
art-workbook_FirstModule_First_module.dir/First_module.cc.o
```

where the above is really just one line.

Also, you can read the verbose listing to discover the flags given to the compiler and linker. The compiler and linker flags, valid at time of writing are given in Table 9.1; actually a few of them are not present in the table because they take a lot of space but don't provide critical functionality. The C++ 11 features are selected by the presence of the `-std=c++11` flag and a high level of error checking is specified. The linker flag,

Table 9.1: 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

```
-Wl,--no-undefined
```

tells the linker that it must resolve all external references at link time. This is sometime referred to as a *closed link*.

9.10 Suggested Activities

This section contains some suggested exercises in which you will make your own modules and in which you will learn more about how to use the class `art::EventID`.

9.10.1 Create Your Second Module

In this exercise you will create a new module by copying `First_module.cc` and making the necessary changes; you will build it using buildtool; you make a FHiCL file to run the new module by copying `first.fcl` and making the necessary changes; and you will run the new module using the new FHiCL file.

Go to your source window and cd to your source directory. If you have logged out, out remember to re-establish your working environment; see Section 9.6 Type the following commands:

```
$ cd art-workbook/First
$ cp First_module.cc Second_module.cc
$ cp first.fcl second.fcl
```

Edit the files `Second_module.cc` and `first.fcl`. In both files, change every occurence of the string “First” to “Second”; there are eight places in the source file and two in the FHiCL file, one of which is in a comment.

The new module needs the same link list as did `First_module.cc` so there is no need to edit `CMakeLists.txt`; the instructions in `CMakeLists.txt` tell `buildtool` to build all modules that it finds in this directory and to use the same link list for all modules.

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 9.6. Rebuild the Workbook code:

```
$ buildtool
```

This should complete with the message:

```
------------------------------------
INFO: Stage build successful.
------------------------------------
```

If you get an error message, consult a local expert or consult the *art* team as described in Section 2.4.

When you run `buildtool` it will perform an incremental build (see Section 9.9.2), during which it will detect `Second_module.cc` and build it.

You can verify that `buildtool` created the expected shared library:

```
$ ls lib/*Second*.so
lib/libart-workbook_FirstModule_Second_module.so
```

Stay in your build directory and run the new module:

```
$ art -c fcl/FirstModule/second.fcl >& output/second.log
```

Compare `output/second.log` with `output/first.log`. You should see that the printout from `First_module.cc` has been replaced by that from `Second_module.cc`.

9.10.2 Use `artmod` to Create Your Third Module

This exercise is much like the previous one; the difference is that you will use a tool named `artmod` to create the source file for the module.

Go to your source window and cd to your source directory. If you have logged out, remember to re-establish your working environment; see Section 9.6

The command `artmod` creates a file containing the skeleton of a module. It is supplied by the UPS product **cetpkgsupport**, which was set up when you performed the last step of establishing the environment in the source window, sourcing `setup_deps`. You can verify that the command is in your path by using the bash built-in command `type`:

```
$ type artmod
artmod is hashed (/ds50/app/products/cetpkgsupport/v1_02_00/bin/artmod)
```

In general the leading elements of the directory name will be different on your computer; they will be the leading elements of your UPS products area.

From your source directory, type the following commands:

```
$ cd art-workbook/First
$ artmod analyzer tex::Third
$ cp first.fcl third.fcl
```

The second argument tells `artmod` to create a file named `Third_module.cc` that contains the skeleton for a module named `Third` in the namespace `tex`; the first argument tells `artmod` that it should write the skeleton for an analyzer module.

If you compare `Third_module.cc` to `First_module.cc` you will see a few differences:

1. the layout of the class is a little different but the two layouts are equivalent
2. there are some extra `#include` directives
3. the include for `<iostream>` is missing
4. a formal parameter is supplied in the definition of the constructor
5. in the `analyze` member function, the name of the formal parameter is different.
6. `artmod` supplies the skelton of a destructor

The `#include` directives provided by `artmod` are a best guess, made by the author of `artmod`, about what `#include` directives will be needed in a "typical" module. Other than slowing down the compiler by an amount you won't notice, the extra `#include` directives do no harm; keep them or leave them as you see fit.

Edit `Third_module.cc`

1. add the `#include` directive for `<iostream>`
2. copy the bodies of the constructor and the `analyze` member function from `First_module.cc`; change the string "First" to "Third".
3. delete the formal parameter from the definition of the constructor
4. in the definition of the member function `analyze`, change the name of the formal parameter to `event`.

When you built `First_module.cc`, the compiler wrote a destructor for you that is identical to the destructor written by `artmod`; so you can leave the destructor as `artmod` wrote it. This class has no work to do in the destructor so the one written by `artmod` has an empty body.

Edit `third.fcl` Change every occurence of the string "First" to "Third"; there are two places, one of which is in a comment.

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 9.6. Rebuild the Workbook code:

```
$ buildtool
```

Refer to the previous section to learn how to identify a successful build and how to verify that the expected library was created.

Stay in your build directory and run the third module:

```
$ art -c fcl/FirstModule/third.fcl >& output/third.log
```

Compare `output/third.log` with `output/first.log`. You should see that the printout from `First_module.cc` has been replaced by that from `Third_module.cc`.

`artmod` has many options that you can explore by typing:

```
$ artmod --help
```

9.10.3 Running Many Modules at Once

In this exercise you will run four modules at once, the three made in this exercise plus the `HelloWorld` module from Chapter 8.

Go to your source window and cd to your source directory. Type the following commands:

```
$ cd art-workbook/First
$ cp first.fcl all.fcl
```

Edit the file `all.fcl` and replace the `physics` parameter set with that found in Listing 9.5. This parameter set:

1. defines four module labels
2. puts all four module labels into the `end_paths` sequence.

When you run *art* on this FHiCL file, *art* will first look at the definition of `end_paths` and learn that you would like it to run four module labels. Then it will look in the `analyzers` parameter set to find the definition of each module label; in each definition *art* will find the class name of the module that it should run. Given the class name and the environment variable `LD_LIBRARY_PATH`, *art* can find the right shared library to load. If you need a refresher on module labels and `end_paths`, refer to Sections 8.7.7 and 8.7.8.

Listing 9.5: The `physics` parameter set for `all.fcl`

```
physics :{
  analyzers: {
    hello : {
      module_type : HelloWorld
    }
    first : {
      module_type : First
    }
    second : {
```

```
      module_type : Second
    }
    third : {
      module_type : Third
    }
 }

  e1        : [ hello, first, second, third ]
  end_paths : [ e1 ]

}
```

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 9.6. 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 8.7.8.

9.10.4 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 9.8.3.6 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:

```
Hello from FirstAnswer01::analyze.  run number: 1 sub run number: 0 event number
Hello from FirstAnswer01::analyze.  run number: 1 sub run number: 0 event number
```

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`.

When you think that you have successfully rewritten the module, you can test it by going to your build window and cd'ing to your build directory. Then:

```
$ buildtool
$ art -c fcl/FirstModule/second.fcl >& output/eventid.log
```

Work on this for 15 minutes or so. If you have not figured out how to do it, you can look at the file `FirstAnswer01_module.cc`, in the same directory as `First_module.cc`. This file as one possible answer.

To run the answer module, to verify that it makes the requested output:

```
$ art -c fcl/FirstModule/firstAnswer01.fcl >& output/firstAnswer01.log
```

You did not need to build this module because it was already built the first time that you ran `buildtool`; that run of `buildtool` built all of the modules in the Workbook.

There is second correct answer to this exercise. If you look at the header file for `art::Event`, you will see that this class also has member functions

```
  EventNumber_t   event()  const {return aux_.event();}
  SubRunNumber_t  subRun() const {return aux_.subRun();}
  RunNumber_t     run()    const {return id().run();}
```

So you could have called these directly,

```
  std::cout << ``Hello from FirstAnswer01::analyze. ``
            << `` run number: ``     << event.run()
            << `` sub run number: `` << event.subRun()
            << `` event number: ``   << event.event()
            << std::endl;
```

Instead of

```
  std::cout << ``Hello from FirstAnswer01::analyze. ``
            << `` run number: ``     << event.id().run()
            << `` sub run number: `` << event.id().subRun()
            << `` event number: ``   << event.id().event()
            << std::endl;
```

But the point of this exercise was to learn a little about how to dig down into nested header files to find the information you need.

9.11 Final Remarks

9.11.1 Why is there no `First_module.h` File?

When you performed the exercises in this chapter, you saw, for example, the file `First_module.cc` but there was no corresponding `First_module.h` file.

This section will explain why.

In a typical C++ programming environment there is a header file (.h) for each source file (.cc). For definiteness, consider the examples of Point.h and Point.cc that you saw in Section 5.6.10.

The reason for having Point.h is that the implementation of the class, Point.cc, and the users of the class, need to agree on what the class Point is; in this example the only user of the class is the main program, ptest.cc. The file Point.h serves as the unique, authoritative declaration of what the class is; both Point.cc and ptest.cc rely on on this declaration.

If you think carefully, you are already aware of a very common exception to the pattern of one .h file for each .cc file: there is never header file for a main program. For example, in the examples that exercised the class Point, there was never a header file for the main program ptest.cc. The reason for this is that there is no other piece of user written code that needs to know about the declaration of any classes or functions declared or defined inside ptest.cc.

The reason that there is no First_module.cc file is simply that every entity that needs to see the declaration of the class First is already inside the file First_module.cc. Therefore there is no reason to have a separate header file. There was a dangerous bend paragraph at the end of Section 9.8.3.7 that described how *art* is able to use modules without needing to know about the declaration of the module class.

The architecture of *art* says that only *art* may construct instances of module classes and only *art* may call member functions of module classes. In particular, modules may not construct other modules and may not call member functions of other modules. The absence of a First_module.h, provides a physical barrier that enforces this design.

9.11.2 The Three File Module Style

In this chapter, the source for the module First was written in a single file. You may also write it using three files, First.h, First.cc and First_module.cc. The authors of *art* do not recommend this style because it exposes the declaration of First in a way that permits it to be misused (as was discussed in Section 9.11.1).

However some experiments do use this style. Therefore this section has been provided.

In this style, First.h contains the class declaration plus any necessary #include directives; it also now requires code guards (see Section 30.8); this is shown in Listing 9.6. The file First.cc contains the definitions of the constructor and the analyze member function, plus the necessary #include directives; this is shown in Listing 9.7. And First_module.cc is now stripped down to the invocation of the DEFINE_ART_MODULE macro plus the necessary #include directives; this is shown in Listing 9.8.

The build system distributed with the Workbook has not been configured to build modules written in this style.

Listing 9.6: The contents of First.h in the 3 file model

```
#ifndef art-workbook_FirstModule_First_h
#define art-workbook_FirstModule_First_h

#include ``art/Framework/Core/EDAnalyzer.h''
#include ``art/Framework/Principal/Event.h''

namespace tex {

  class First : public art::EDAnalyzer {

  public:

    explicit First(fhicl::ParameterSet const& );

    void analyze(art::Event const& event) override;

  };

}
#endif
```

Listing 9.7: The contents of First.cc in the 3 file model

```
#include ``art-workbook/FirstModule/First.h''

#include <iostream>

tex::First::First(fhicl::ParameterSet const& ){
  std::cout << ``Hello from First::constructor.'' << std::endl;
}

void tex::First::analyze(art::Event const& event){
  std::cout << ``Hello from First::analyze. Event id: ``
            << event.id()
            << std::endl;
}
```

Listing 9.8: The contents of First_module.cc in the 3 file model

```
#include ``art-workbook/FirstModule/First.h''
#include ``art/Framework/Core/ModuleMacros.h''

DEFINE_ART_MODULE(tex::First)
```

9.12 Review

9.12.1 What Makes a class an Analyzer Module

This section reviews the properties that a class must have in order to be an analyer module. These were first given in Section 9.8.3.2 but are repeated here for easy reference:

1. it must inherit from art::EDAnalyzer
2. it must provide a constructor with the argument list
   fhicl::ParameterSet const&
3. it must provide a member function named analyze, with the signature:
   analyze(art::Event const&)
4. if the name of a module class is <ClassName> then the source code for the module must be in a file named <ClassName>_module.cc and this file must contain the lines:
   #include ``art/Framework/Core/ModuleMacros.h''
   DEFINE_ART_MODULE(tex::<ClassName>)
5. it may, optionally, provide other member functions with signatures prescribed by *art*; if these member functions are present in a module class, then *art* will call them at the appropriate times. Some examples are provided in Chapter 10

The *art* team recommends that you write modules using the one file style, not the three file style; the above list is written presuming that you use the one file style.

9.12.2 Flow from source to .fcl

This section reviews how the source code found in First_module.cc is executed by *art*:

1. the script setup_for_development defines many environment variables that are used by buildtool and by *art* and toyExperiment.
2. one of the important environment variables is LD_LIBRARY_PATH. This contains the directory lib in your build area plus the lib directories from many UPS products, including *art*.
3. buildtool compiles First_module.cc to a temporary object file.
4. buildtool links the temporary object file to create a shared library in the lib subdirectory of your build area:
   lib/libart-workbook_FirstModule_First_module.so

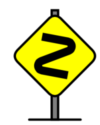
5. when you run *art* using file `first.fcl`, the FHiCL file tells art to find and load a module with the `module_type` `First`.

6. in response to this request, *art* will search the directories in `LD_LIBRARY_PATH` to find a shared library file whose name matches the pattern:
`lib*First_module.so`

7. if *art* finds zero matches to this pattern, or more than one match to this pattern, it will issue and error message and stop

8. if *art* finds exactly one match to this pattern, it will load the shared library.

9. after *art* has loaded the shared library, it has access to a function that can, on demand, create instances of the class `First`.



The last bullet really means that the shared library contains a factory function that can construct instances of `First` and return a pointer to the base class, `art::EDANalyzer`. The shared library also contains a static object that, at load-time, will contact the *art* module registry and register the factory function under the `module_type` `First`.

10 Exercise 3: The Optional Member Functions of art Modules

10.1 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.

10.2 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 2.6.2.

10.3 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&)`

4. `endJob()`
5. `endRun( art::Run const&)`
6. `endRun( art::SubRun const&)`

You will also learn about the classes,

1. `art::RunID`
2. `art::Run`
3. `art::SubRunID`
4. `art::SubRun`

10.4 Setting up to Run this Exercise

To run this exercise, you need to be logged in to the computer on which you ran Exercise 2 (in Chapter 9). If you are continuing on from the previous exercise, you need to keep both your source and build windows open.

If you are logging back in, follow the instructions in Section 9.6 to reestablish your source and build windows.

In your source window, `cd` to your source directory. Then `cd` to the directory for this exercise and look at its contents

```
$ cd art-workbook/OptionalMethods
\begin{samepage}$ ls
CMakeLists.txt        OptionalAnswer01_module.cc  Optional_module.cc
optionalAnswer01.fcl  optional.fcl
\end{samepage}
```

The source code for the module you will run is `Optional_module.cc` and the FHiCL file to run it is `first.fcl`. The file `CMakeLists.txt` is identical that used by the previous exericse; this is because the new features introduced by this module do not require any modifications to the link list. The other two files are the answers to the exercise you will be asked to do in Section 10.9.

In your build window, make sure that you are in your build directory. In this exercise you do not need to build any code becaue all code for the Workbook was built the first time that your ran `buildtool`.

10.5 The Source File `Optional_module.cc`

In your source window, look at the source file, `Optional_module.cc` and compare it to `First_module.cc`. The differences are

1. the name of the class has changed from `First` to `Optional`
2. it has two new include directives, for `Run.h` and `SubRun.h`
3. the class declaration declares three new member functions
   ```
   void beginJob () override;
   void beginRun ( art::Run const& run ) override;
   void beginSubRun( art::SubRun const& subRun ) override;
   ```
4. the text printed by the constructor and the analyze member functions has changed
5. the file contains the definitions of the three new member functions, each of which simply makes some identifying printout

Each of the new member functions must have exactly the argument list prescribed by *art*. The `override` keyword instructs the compiler to do the following: if a member function has a name that is spelled incorrectly or it if has an incorrect argument list, the compiler will issue an error message and stop.

This is a very handy feature. If the `override` keyword were absent, and if the function were spelled incorectly or the argument list were incorrect, then the compiler would assume that it was your intention to define a new member function that is unrelated to one of the optional *art* defined member functions. The result would be a difficult to diagnose run-time error: *art* would simply not recognize your member function and would never call it.

Always provide the `override` keyword when your class provides one of the optional *art* defined member functions.

For those with some C++ background, the three member functions `beginJob`, `beginRun` and `beginSubRun` are declared as virtual in the base class, `art::EDAnalyzer`. The `override` keyword is new in C++-11 and will not be described in older text books. It instructs the compiler that this member function is intended to override a virtual function from the base class; if the compiler cannot find such a function in the base class, it will issue an error.

As described in Section 2.6.2, *art* will call the `beginJob` method of each module once at the start of the job; it will call the `beginRun` method of each module at the start of each run and it will call the `beginSubRun` method of each module at the start of each sunRun.

10.6 The classsss `art::Run`, `art::RunID`, `art::SubRun` and `art::SubRunID`

In Section 9.8.3.6 you learned about the class `art::EventID`, which describes the three-part event identifier. *art* also provides two related classes:

1. `art::RunID`, which is an one-part identifier for a run number
2. `art::SubRunID`, which is a two-part identifier for a subRun

You can find the header files for these classes at:

```
$ less $ART_INC/art/Persistency/Provenance/RunID.h
$ less $ART_INC/art/Persistency/Provenance/SubRunID.h
```

Remember that you type a lower case letter "q" to exit `less`. Or you can look at these header files using one of the code browsers described in Section-ssec:ups:setup:headers:art.

The argument to the `beginRun` method is a `const` reference to an object of type `art::Run`. This object is similar to an `art::Event`: a simplified picture is that it holds an `art::RunID` plus a collection of data products. The purpose of the `art::Run` object is to hold information that describes an entire run; some of that information might be available at the start of the run but some of it might only be added at the end of the run.

You can find the header file for `art::Run` at:

```
$ less $ART_INC/art/Framework/Principal/Run.h
```

If you take a snapshot of a running *art* job you will see that there is exactly one object of type `art::Run`. This object is owned by *art* and *art* gives modules access to it when it calls their `beginRun` and `endRun` methods. Because it is passed by reference, the `beginRun` member function does not get a copy of the `art::Run` object; instead it has access to the unique `art::Run` object that is owned by *art*. Because it is passed by `const` reference, your analyzer module may look at information in the run object but it may not add information to the run object.

In your `analyze` member function, if you have an `art::Event`, named `event`, you can access the associated run information by:

```
art::Run const& run = event.getRun();
```

You may sometimes see this written as:

```
auto const& run = event.getRun();
```

Both version mean exactly the same thing. When a type is long and awkward to write, the auto keyword is very useful; however it is likely to be very confusing to beginners. When you encounter it, check the header files for the classes on right hand side of the assignment; from there you can learn the return type of the member function that returned the information.

In both cases the `const&` is very important. If you omit the reference part, the `&`, then the variable `run` will contain a *copy* of the run object that is owned by *art*. This is a waste of both CPU time and memory and in some circumstances it can be a significanct waste.

If you omit the `const`, but remember the `&`, then you will get a compiler error because the `getRun()` method only permits you `const` access to the run object.

There is a very important habit that you need to develop as a user of *art*. Many methods in *art*, in the Workbook code and very likely in your experiment's code, will return information by `&` or by `const&`. If you receive these by value, not by reference, then you will make copies that waste both CPU and memory; in some cases these can be significant wastes. Unfortunately there is no way to tell the compiler to catch this mistake. The only solution is your own vigilance.

In the call to `beginSubRun` the argument is of type `art::SubRun const&`. A simplified description of this object is that it contains an `art::SubRunID` plus a collection of data products that describe the subRun. All of the comments about the class `art::Run` in the preceding few paragraphs apply to `art::SubRun`. You can find the header file for `art::SubRun` at:

```
$ less $ART_INC/art/Framework/Principal/SubRun.h
```

If you have an `art::Event`, named `event`, you can access the associated subRun object by,

```
art::SubRun const& subRun = event.getSubRun();
```

If you have an `art::SubRun` object, named `subRun`, you can access the associated `art::Run` object:

```
art::Run const& run = event.getRun();
```

10.7 Running this Exercise

Look at the file `optional.fcl`. This FHiCL file runs the module `Optional` on the the input file `inputFiles/input03_data.root`. Consult Table 8.1 and you will see that this file contains 15 events, all from run 3. It contains events 1 through 5 from each of subRuns 0, 1 and 2. With this knowledge, and the knowledge of the source file `Optional_module.cc`, you should have a clear idea of what this module will print out.

In your build directory, run the following command

```
$ art -c fcl/OptionalMethods/optional.fcl  >& output/optional.log
```

The part of the printed output that comes from the module `Optional` is given in Listing 10.1. Is this what you expected to see? If not, understand why this module made the printout that it did.

Listing 10.1: The output produced by `Optional_module.cc` when run using `optional.fcl`

```
Hello from Optional::constructor.
Hello from Optional::beginJob.
Hello from Optional::beginRun: run: 3
Hello from Optional::beginSubRun: run: 3 subRun: 0
Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 1
Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 2
Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 3
Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 4
Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 5
Hello from Optional::beginSubRun: run: 3 subRun: 1
Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 1
Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 2
Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 3
Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 4
Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 5
Hello from Optional::beginSubRun: run: 3 subRun: 2
Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 1
Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 2
Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 3
Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 4
Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 5
```

10.8 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 the `begin` methods, perhaps `beginJob` or `beginRun`. Other constraints may enter into this decision. If booking a histogram requires geometry information to set the limits correctly, that information may be run dependent and you will need to book these histograms in `beginRun`, not `beginJob`.

10.9 Suggested Activities

10.9.1 Add the Matching `end` Member functions

art defines the following three member functions:

```
    void endJob     () override;
    void endRun     ( art::Run const&    run    ) override;
    void endSubRun  ( art::SubRun const& subRun ) override;
```

In the file `Optional_module.cc`, add these methods to the declaration of the class `Optional` and provide an implementation for each. In your implementation, just copy the printout created in the corresponding `begin` function and, in that printout, change the string "begin" to "end".

Then rebuild this module and run it:

```
$ buildtool
$ art -c fcl/OptionalMethods/optional.fcl >& output/optional2.log
```

Consult Chapter 9 if you need to remember how to indentify that the build completed successfully. Compare the output from this run of *art* with that of the previous run: do you see the addtional printout from the methods that you added?

The solution to this activity is provided as the file `OptionalAnswer01_module.cc`. It is already built. You can run it with:

```
$ art -c fcl/OptionalMethods/optionalAnswer01.fcl  >& output/optionalAnswer01.lo
```

Does the output of your code match the output from this code?

10.9.2 Run on Multiple Input Files

In a single run of *art*, run your modified version of the module `Optional` on all of the three of the following input files:

```
inputFiles/input01_data.root
inputFiles/input02_data.root
inputFiles/input03_data.root
```

If you need a reminder about how to tell art to run on three input files in one job, consult Section 8.7.5.

11 Parameter Sets

11.1 Introduction

In the previous few chapters you used FHiCL files to configure *art*. In particular you learned how to define a module label

```
moduleLabel : {
   module_type : ClassName
}
```

where `module_type` is a keyword 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"
   primes           : [ 1, 3, 5, 7, 11]
   nestedPSet       : {
                        a : 1
                        b : 2
                      }
}
```

The only constraints are that each additional piece of information must be a legal FHiCL definition.

You saw in all of the previous exercises that the constructor of module takes a parameter of type `fhicl::ParameterSet`. Until now the modules that you have seen have ignored this parameter.

In this chapter you will learn about this parameter and how to use it.

The values of your new parameters can be arbitrarily complex

These additional parameters will be passed to the module class in its constructor.

In this way the module

11.2 What You Will Learn

In the previous few chapters you used FHiCL files to configure *art*.

11.3 Prerequisites

11.4 Running the Exercise

11.5 Discussion

11.6 Suggested Activities

12 Multiple Instances of a Module within one art Process

12.1 Prerequisites

12.2 What You Will Learn

12.3 Running the Exercise

12.4 Discussion

12.5 Suggested Activities

13 Accessing Data Products

13.1 Prerequisites

13.2 What You Will Learn

13.3 Running the Exercise

13.4 Discussion

13.5 Suggested Activities

14 Making Histograms and TFileService

14.1 Prerequisites

14.2 What You Will Learn

14.3 Running the Exercise

14.4 Discussion

14.5 Suggested Activities

15 Looping Over Collections

15.1 Prerequisites

15.2 What You Will Learn

15.3 Running the Exercise

15.4 Discussion

15.5 Suggested Activities

16 The Geometry Service

16.1 Prerequisites

16.2 What You Will Learn

16.3 Running the Exercise

16.4 Discussion

16.5 Suggested Activities

17 The Particle Data Table

17.1 Prerequisites

17.2 What You Will Learn

17.3 Running the Exercise

17.4 Discussion

17.5 Suggested Activities

18 GenParticle: Properties of Generated Particles

18.1 Prerequisites

18.2 What You Will Learn

18.3 Running the Exercise

18.4 Discussion

18.5 Suggested Activities

Part III

Users Guide

19 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.

19.1 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 confir
```

[1] The FERMI.WIN.FNAL.GOV realm is available for PCs.

```
Kerberos password changed.
```

Your Kerberos password will remain valid for 400 days.

19.2 Fermilab Services Account

The Services Account enables you to access a number of important applications at Fermilab with a single username/password (distinct from your Kerberos username/password). Applications available via the Services Account include SharePoint, Redmine, Service Desk, VPN and others.

To get your initial Services Account password, a user must first contact the Service Desk to get issued a first time default password. Once a default password is issued, users can access http://password-reset.fnal.gov/ to change it.

If you are not on-site or connected to the Fermi VPN, call the Service Desk at 630-840-2345. You will be given a one-time password and a link to change it.

20 Using git

The source code for the exercises in the *art* workbook is stored in a source code management system called *git* and maintained in a repository managed by Fermilab. Think of *git* as an enhanced *svn* or (a VERY enhanced) *cvs* system. The repository is located at . You will be shown how to access it with *git*.

If you want some background on *git*, we suggest the Git Reference.

You will need to know how to install *git*, download the workbook exercise files initially to your system and how to download updates. You will not be checking in any code.

To install *git* on a Mac:

```
$ http://git-scm.com/download/mac
```

This will automatically download a disk image. Open the disk image and click on the .pkg file.

In your home directory, edit the file .bash_profile and add the line:

```
$ export PATH=/usr/local/git/bin/:${PATH}
```

```
$ git clone ssh://p-art-workbook@cdcvs.fnal.gov/cvs/projects/art-workbook
```

and how to download updates as the developers make them:

```
$ git pull
```

21 art Run-time and Development Environments

21.1 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 21.1, 21.2 and 21.3 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

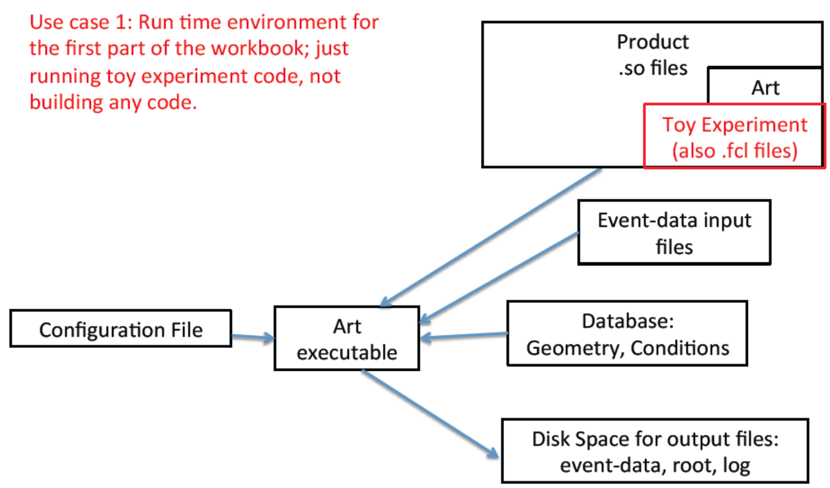


Figure 21.1: Elements of the *art* run-time environment, just for running the Toy Experiment code for the Workbook exercises

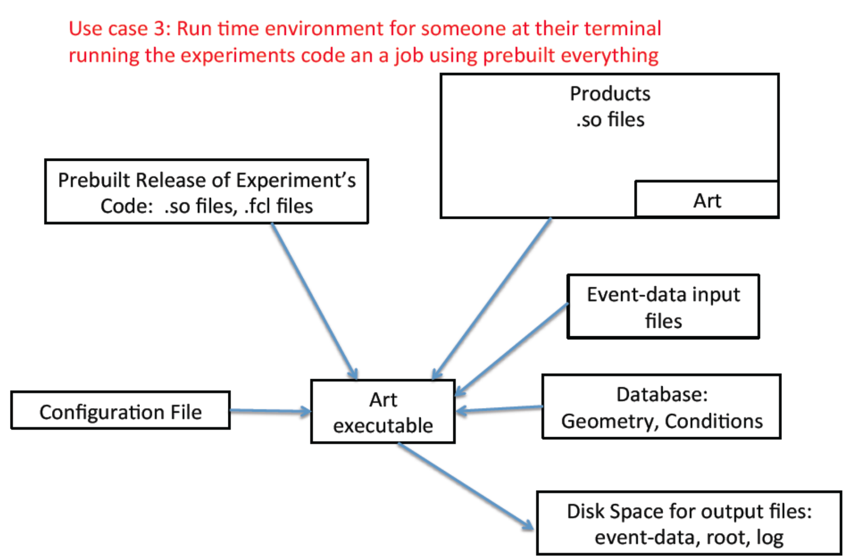


Figure 21.2: Elements of the *art* run-time environment for running an experiment's code (everything pre-built)

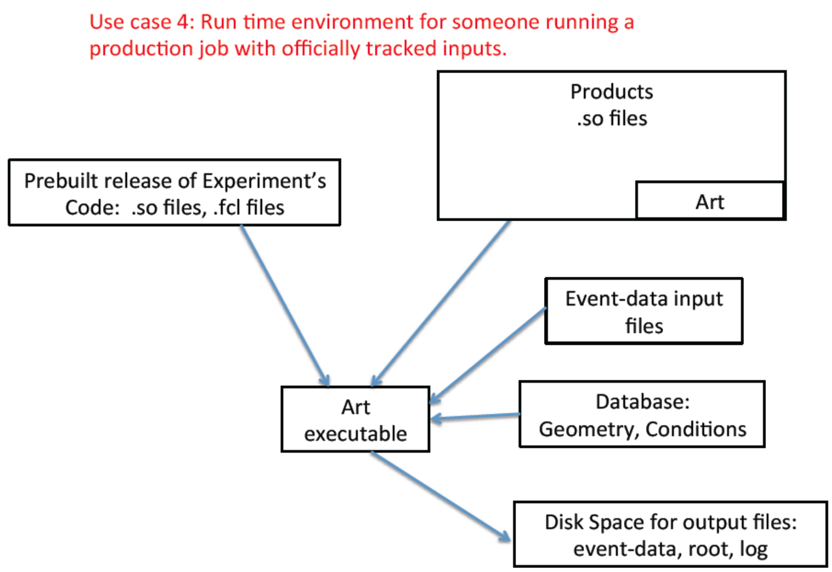


Figure 21.3: Elements of the *art* run-time environment for a production job with officially tracked inputs

2. secondly, it looks for shell keywords in your environment with the command name you provide
3. thirdly, it looks for shell functions in your environment with that name
4. then it looks for shell built-ins in your environment with that name
5. finally, it looks in the first directory defined in `PATH` and looks for a file with that name; if it does not find a match, it continues with the next directory, and so on, followed by the paths defined in the other two variables.

Some parts of the run-time environment will be established at login time by your login scripts. This is highly site-dependent. We will describe what happens at Fermilab - consult your site experts to find out if anything is provided for you at your remote site.

When running the workbook, the interesting parts of your environment are established in two steps:

- source a site-specific setup script
- source a project-specific setup script

The Workbook, and the software suites for most IF experiments, are designed so that all site dependence is encoded in the site-specific setup script; that script adds information to your environment so that the project-specific scripts can be written to work properly on any site.

21.2 The *art* Development Environment

The development environment includes the run-time environment in Section 21.1 plus the following.

- the source code repository
- the build tools (these are the tools that know how to turn `.h` and `.cc` files in to `.so` files)
- additional environment variables and `PATH` elements that simplify the use of the above

Figures 21.4, 21.5 and 21.6 illustrate the development environment for various scenarios.

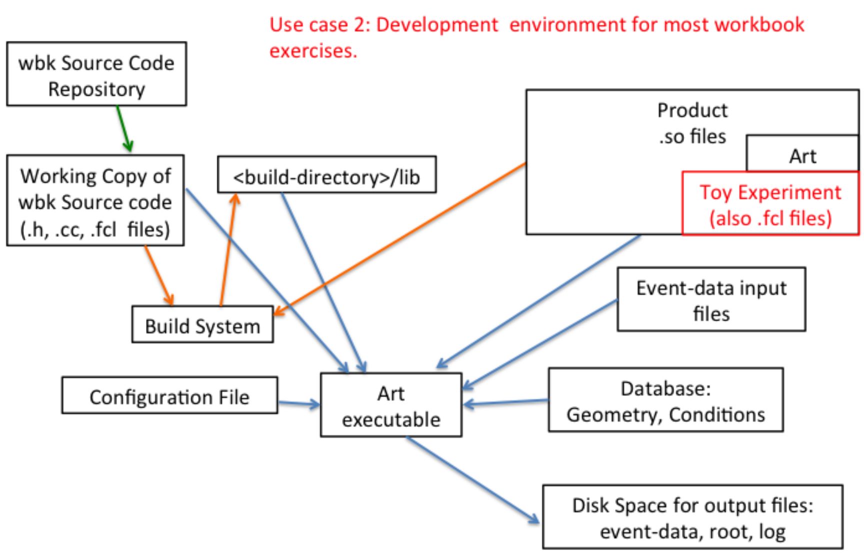


Figure 21.4: Elements of the *art* development environment as used in most of the Workbook exercises

In some experiments the run-time and development environments are identical.

It turns out that there is no perfect solution for the job that build tools do. As a result, several different tools are widely used. Every tool has some pain associated with it. You never get to avoid pain entirely but you do get to pick where you will take your pain.

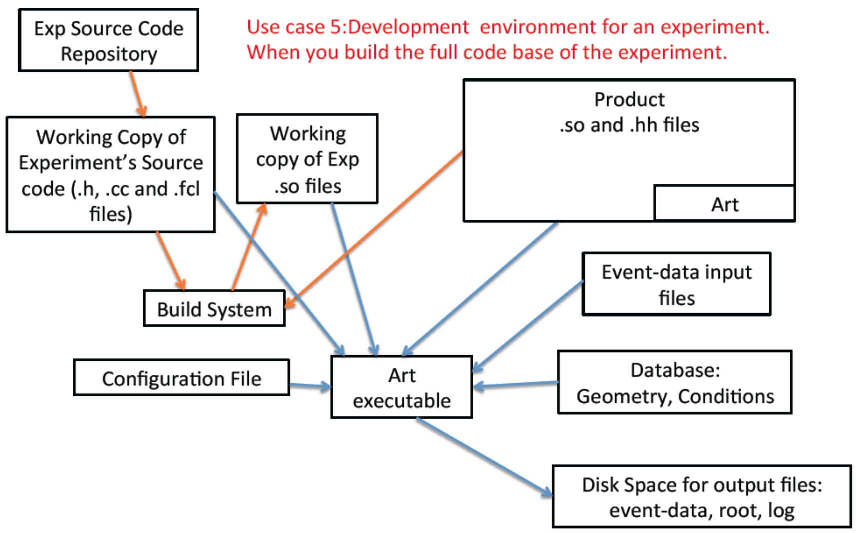


Figure 21.5: Elements of the *art* development environment for building the full code base of an experiment

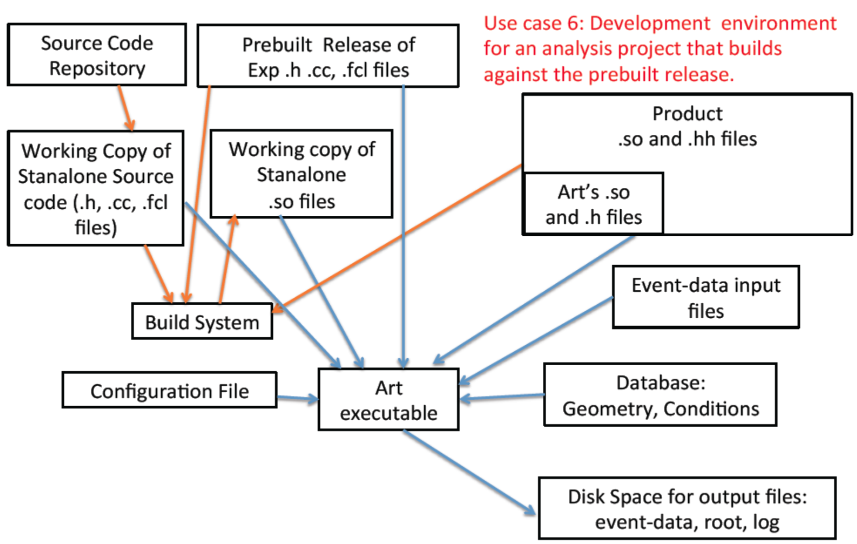


Figure 21.6: Elements of the *art* development environment for an analysis project that builds against prebuilt release

The workbook uses a build tool named **cetbuildtools**. Other projects have chosen `make`, `cmake`, `scons` and Software Release Tools (SRT). Here is something to watch out for: “build tools” written as two words refers generically to the above set of tools; but “buildtools” written as one word is the name of the executable that runs the build for **cetbuildtools**.

22 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.

22.1 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"
sneakystring1: "nil"
sneakystring2: "true"
sneakystring3: "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
```

22.2 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.

```
# Prolog (as many as desired, but they must all be contiguous with only
# whitespace or comments inbetween.
BEGIN_PROLOG
pset:
{
  nested_pset:
  {
    v1: [ a, b, "c-d" ]
    b1: false
    c1: 29
  }
}
END_PROLOG

# Defaulted if missing: you should define it in most cases.
process_name: PNAME

# Descriptions of service and general configuration.
services:
{
  # Parameter sets for known, built-in services here.
  # ...

  # Parameter sets for user-provided services here.
  user:
  {
  }

  # General configuration options here.
  scheduler:
  {
  }
}

# Define what you actually want to do here.
physics:
{
  # Parameter sets for modules inheriting from EDProducer.
  producers:
```

```
  {
    myProducer:
    {
      module_type: MyProducer
      nested_pset: @local::pset.nested_pset
    }
  }

  # Parameter sets for modules inheriting from EDFilter.
  filters:
  {
    myFilter: { module_type: SomeFilter }
  }

  # Parameter sets for modules inheriting from EDAnalyzer.
  analyzers:
  {
  }

  # Define parameters which are lists of names of module sets for
  # inclusion in end_paths and trigger_paths.

  p1: [ myProdroducer, myFilter ]
  e1: [ myAnalyzer, myOutput ]

  # Compulsory for now: will be computed automatically in a future
  # version of ART.

  trigger_paths: [ p1 ]
  end_paths: [ e1 ]
}

# The primary source of data: expects one and only one input source
parameter set.
source:
{
}

# Parameter sets for output modules should go here.
outputs:
{

}
```

22.3 Services

22.3.1 System Services

These services are always loaded regardless of whether a configuration is specified.

22.3.2 FloatingPointControl

These parameters control the behavior of floating point exceptions in different modules.

Table 22.1: *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	BOOL	false	
	moduleNames	SEQUENCE	[]	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	

22.3.3 Message Parameters

These parameters configure the behavior of the message logger (this is a pseudo-service – not accessible via ServiceHandle).

Table 22.2: *art* Message Parameters

Enclosing Table Name	**Parameter Name**	**Type**	**Default**	**Notes**
services	message	TABLE		Top-level parameter set for the service
message				

22.3.4 Optional Services

These services are only loaded if a configuration is specified (although it may be empty).

22.3.5 Sources

22.3.6 Modules

Output modules

23 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 21.1 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.

23.1 Basics of FHiCL Syntax

23.1.1 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 be a reference to a previously defined value. The white space on either side of the colon is optional. However, to include whitespace within a string, the string must be quoted (single or double quotes are equivalent in this case).

The fragment below will be used to illustrate some of the basics of FHiCL syntax:

```
# A comment.
// Also a comment.


 name0   : 123             # A numeric value. Trailing comments
                           # work, too.
 _name0   : 123             # Names can begin with underscores

 name00 : "A quoted comment prefix, # or //, is just part of a
                                    # quoted string, not a comment"

 name1:456.                # Another numeric value; whitespace is
                           # not important within a definition
 name2 : -1.e-6
 name3 : true              # A boolean value
 NAME3 : false             # Other boolean value; names are case-
                           # sensitive.
 name4 : red               # Simple strings need not be quoted
 name5 : "a quoted string"
 name6 : 'another quoted string'

 name7 : 1 name8 : 2       # Two definitions on a line, separated by
                           # whitespace.
 name9                     # Same as name9:3 ; newlines are just
 :                         # whitespace,  which is not important.
 3


 namea : [ abc, def, ghi, 123 ]  # A sequence of atomic values.
                                 # FHiCL allows heterogeneous
                                 # sequences, which are not,
                                 # however, usable via the C++ API.

 nameb :                   # A table of definitions; tables may nest.
 {
    name0: 456
    name1: [7, 8, 9, 10 ]
    name2:
    {
      name0: 789
```

```
    }
 }

 namec : [ name0:{ a:1 b:2 } name1:{ a:3 c:4 } ]
                        # A sequence of tables.

 named : []             # An empty sequence
 namee : {}             # An empty table

 namef : nil            # An atomic value that is undefined.

abc : 1                 # If a definition is repeated twice within
abc : 2                 # the same scope, the second definition
def : [ 1, 2, 3 ]       #  will win (e.g., "abc" will be 2 and
def : [ 4, 5, 6 ]       # "def" will be [4,5,6])
name : {
  abc : 1
  abc : 2
}

cont1:{x: 1.0 y: 2.0 z: 3.0}  # Hierarchical (compound) names denote
cont1.x : 5                   # levels of scope; here set x in cont1 to 5.
OR
cont2:[1, 2, 3]
cont2[0] : 1                  # Here, redefine the first (atomic) value
                              # for cont2, assign it the value 1. I.e., here,
                              # no action.  Indices of PHiCL sequences
                              # begin with 0.  \fixme{right?}

name0:{ a:1 b:2 }
x : @local::name0.a   # Using reference notation "@local," this assigns
                      #  to xthe value of a in table name0, in the
                      # line above, this value is 1.
```

23.1.2 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 to FHiCL:

```
, : :: @ [ ] { } ( )
```

The following keywords have special meaning to FHiCL. They can be used as parameter values to pass to classes, e.g., to initialize a variable within a program, but their uses will not be fully described here because of subtleties and variations. As you work with C++ and FHiCL, the way to use them will become clearer.

true, false These values convert to a boolean

nil This value is associated with no data type. E.g. if `a : nil`, then `a` can't be converted to any data type, and it must be redefined before use

infinity, +infinity, -infinity These values initialize a variable to positive (the first two) or negative (the third) infinity

BEGIN_PROLOG, END_PROLOG ()

The first six keywords (three lines) above are only keywords when entered as lower case and unquoted; the last two keywords (the last line) are only keywords when they are in upper case, unquoted and at the start of a line. Otherwise these are just strings. You may include any of the above reserved characters and keywords in a “quoted” string to prevent them from being recognized as keywords.

23.2 FHiCL Keywords Reserved to *art*

FHiCL supports run-time configuration for several projects, not only for *art*. *art* reserves certain FHiCL names as keywords that it uses in well-defined ways. (Other projects may use FHiCL names differently.) Within FHiCL files used by *art*, these FHiCL names obey scoping rules similar to C++. These keywords appear in the FHiCL file with a scope, i.e.,

```
keyword : {
...
}
```

if they define a list of modules or a processing block, or with square brackets

```
keyword :[
...
]
```

if they define a list of paths.

The following is a list of the keywords reserved to *art* and their meanings. In the outermost scope within a FHiCL file, the following keywords can appear:

process_name A user-given name to identify the configuration defined by the FHiCL file (it is recommended to make it similar to the FHiCL file name). This must appear at the top of the file. It may not contain the underscore character (_).

source Identifies the data source, e.g., a file in ROOT format containing HEP events.

services Identifies ...

physics Identifies the block of code that configures the scientific work to be done on every event (as contrasted with the "bookkeeping" portions).

outputs List of output modules.

The following may appear within the `physics` scope:

producers Sets the list and order of producer modules; see Chapter 25

analyzers Sets the list and order of analyzer modules; see Chapter 26

filters Sets the list and order of filter modules; see Chapter 27

trigger_paths List of producer and/or filter module paths; for each event, *art* executes all these module paths. The paths may only contain the module labels of producer and filter modules that are in the list of defined module labels. *art* can identify module labels that are common to several trigger_paths and will execute them only once per event. The various paths within the trigger_paths may be executed in any order.

end_paths List of analyzer and/or output module paths; for each event, *art* executes all these module paths exactly once. The various paths within the end_paths may be executed in any order.

The keyword `process_name` is really only reserved to *art* within the outermost scope (but it would seem to be needlessly confusing to use `process_name` as the name of a parameter within some other scope). The names `trigger_paths` and `end_paths` are artifacts of the first use of the CMS framework, to simulate the several hundred parallel paths within the CMS trigger; their meaning should be come clear after reading the remainder of this page.

23.3 Structure of a FHiCL Run-time Configuration File for *art*

Here is a sample FHiCL file called `ex01.fcl` that will do a physics analysis using the code in the *art* module `Ex01_module.so` (the object file of the C++ source file `Ex01_module.cc`). In this configuration, *art* will operate sequentially on the first three events contained in the source file `inputFiles/input01_data.root`.

```
#include "fcl/minimalMessageService.fcl"

process_name : ex01

source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root" ]
  maxEvents   : 3
}

services : {
  message : @local::default_message
}

physics :{
  analyzers: {
    hello : {
      module_type : Ex01
    }
  }

  e1        : [ hello ]
  end_paths : [ e1 ]
}
```

Let's look at it step-by-step.

```
#include "fcl/minimalMessageService.fcl"
```

Similar to C++ syntax, this effectively replaces the '#include' line with the contents of the named file. This particular file sets up a messaging service.

```
process_name : ex01
```

The value of the parameter `process_name` (`ex01`, here, the same as the FHiCL file name) identifies this *art* job. It is used as part of the identifier for data products produced in this job. For this reason, the value that you assign may not contain underscore (_) characters. If the `process_name` is absent, *art* substitutes a default value of "DUMMY."

```
source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root" ]
  maxEvents   : 3
}
```

This `source` parameter describes where events come from. There may be at most one source module declared in an *art* configuration. At present there are two options for choosing a source module:

module_type : RootInput art::Events will be read from an input file or from a list of input files; files are specified by giving their pathname within the file system.

module_type : EmptyEvent Internally *art* will start the processing of each event by incrementing the event number and creating an empty art::Event. Subsequent modules then populate the art::Event. This is the normal procedure for generating simulated events.

Here RootInput is used; the data input file, in ROOT format, is assigned to the variable `fileNames`. The `maxEvents` parameter says: Look at only the first three events in this file. (A value of -1 here would mean "read them all.")

Note that if no source parameter set is present, *art* substitutes a default parameter set of:

```
source : {
  module_type : EmptyEvent
  maxEvents : 1
}
```

See the web page about configuring input and output modules for details about what other parameters may be supplied to these parameter sets.

```
services : {
  message : @local::default_message
}
```

Before starting processing, this puts the message logger in the recommended configuration.

```
physics :{
  analyzers: {
    hello : {
      module_type : Ex01
    }
  }
```

In *art*, `physics` is the label for a portion of the run-time configuration of a job. It contains the "meat" of the configuration, i.e., the scientific processing instructions, in contrast to the more administrative or bookkeeping information. The `physics` block of code may contain up to five sections, each labeled with a reserved keyword (that together form a parameter set within the FHiCL language); the keywords are *analyzers*, *producers*, *filters*, *trigger_paths* and *end_paths*. In our example it's set to `analyzers`.

The `analyzers` keyword takes values that are FHiCL tables of parameter sets (this is true also for `filters` and `producers`). Here it takes the value `hello`, which is defined as a table with one parameter, namely `module_type`, set to the value `Ex01`. The setup defined a variable called LD_LIBRARY_PATH; *art* knows to match the value defined by the name `module_type` to a C++ object file with the name `Ex01_module.so` somewhere in the path defined by LD_LIBRARY_PATH.

We will expand on the `physics` portion of the FHiCL configuration in Section 23.5.

```
e1        : [ hello ]
end_paths : [ e1 ]
```

23.4 Order of Elements in a FHiCL Run-time Configuration File for *art*

In FHiCL files there are very, very few places in which order is important. Here are the places where it matters:

- A `#include` must come before lines that use names found inside the `#include`.
- A later definition of a name overrides an earlier definition of the same name.
- The definition of a name resolved using `@local` needs to be earlier in the file than the place(s) where it is used.
- Within a trigger path, the order of module labels is important.

Here is a list of *a few places* (of many) where order does not matter. This list is by no means exhaustive.

- Inside the physics scope, the order in which modules are defined does NOT matter for filters and analyzers blocks. These blocks define the run-time configurations of *instances* of modules.
- The five *art*-reserved words that appear in the outermost scope of a FHiCL file can be in any order. You could put outputs first and process_name last, as far as FHiCL cares. It may be more difficult for humans to follow, however.
- Within the services block, the services may appear in any order.

Regarding trigger_paths and end_paths, the following is a conceptual description of how *art* processes the FHiCL file:

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 interupt 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!

23.5 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 `selectMode1` 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
    }
    selectMode1: {
      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 built from MakeA_module.cc. Since it falls within the scope producers, it must be a module of type EDProducer.

Let's take this a step farther, and assume that this EDProducer-type module MakeA accepts four arguments that we want to provide to *art*. The configuration may look like this:

```
moduleLabel : {
   module_type : MakeA
   pname0 : 1234.
   pname1 :  [ abc, def]
   pname2 : {
        name0: {}
        }
}
```

This list under module_type : MakeA represents parameters that will be formed into a fhicl::ParameterSet object and passed to the module MakeA as an argument in its constructor. pname0 is a double, pname1 is a sequence of two atomic character values, pname2 consists of a single table named name0 with undefined contents.

Note that *paths* are lists of module labels, while the two reserved names, trigger_paths and end_paths are lists of paths.

23.6 Choosing and Using Module Labels and Path Names

For a module label or a path name, you may choose any name so long as it is unique within a job, contains no underscore (_) characters and is not one of the names reserved to *art* (see Section 23.2.

Any name that is a top-level name inside of the physics parameter set is either a reserved name or the name of a path.

It is important to recognize which identifiers are module labels and which are path names in a FHiCL file. It is also important to distinguish between a class that is a module and instances of that module class, each uniquely identified by a module label.

art has several rules that were recommended practices in the old framework but which were not strictly enforced by that framework. *art* enforces some of these rules and will, soon, enforce all of them:

- A path may go into either the trigger_paths list or into the end_paths list, but not both.

- A path that is in the `trigger_paths` list may only contain the module labels of producer modules and filter modules.
- A path that is in the `end_paths` list may only contain the module labels of analyzer modules and output modules.

Analyzer modules and the output modules may be separated into different paths; that might be convenient at some times but it is not necessary. On the other hand, keeping trigger paths separate has real meaning.

23.7 Scheduling Strategy in *art*

A set of scheduling rules is enforced in *art*. (Some of the details are remnants of compromises and conflicting interests with CMS.) One of the top-level rules in the scheduler is that all producers and filters must be run first, using the ordering rules specified below. After that, all analyzer and output modules will be run. Recall that analyzer modules and output modules may not modify the event, nor may they produce side effects that influence the behavior of other analyzer or output modules. Therefore, *art* is free to run analyzer and output modules in any order.

The full description of the scheduler strategy is given below:

- If a module name appears in the definition of a path name but it is not found among the the list of defined module labels, FHiCL will issue an error.
- One each event, before executing any of the paths, execute the source module.
- On each event, execute all of the paths listed in the trigger_paths.
  - Within one path, the order of modules listed in the path is followed strictly; at present there is one exception to this: see the discussion about the remaining issues
  - *art* can identify module labels that are in common to several trigger_paths and will execute them only once per event. In the above example, aProducer and bProducer are executed only once per event.
  - The various paths within the trigger_paths may be executed in any order, subject to the above constraints.
  - If a path contains a filter, and if the filter return false, then the remainder of the path is skipped.
  - The module name of a filter can be negated in path using, !moduleLabel; in this case the path will continue if the filter returns false and will be aborted if the filter returns true.

  - If the module label of a filter appears in two paths, negated in one path and not negated in the other, art will only run the instance of filter module once and will use the result in both places.
  - If a module in a trigger path throws, the default behaviour of art is to stop all processing and to shut down the job as gracefully as possible. Art can be configured, at run time, so that, for selected exceptions, it behaves differently. For example it can be configured to continue with the current trigger path, skip to the next trigger path, skip to the next event, and so on.
- On each event, execute all of the paths listed in the end_paths.
  - The module labels listed in end_paths are executed exactly once per event, regardless of how many paths there are in the trigger_paths and regardless of any filters that failed.
  - If a module label appears multiple times among the end paths, it is executed only once. No warning message is given.
  - Even if all trigger_paths have filters that fail, all module labels in the end path will be run.
  - End_path is free to execute the modules in the end_path in any order.
  - If a module in the end_path throws, the default response of art is to make a best effort to complete all other modules in the end path and then to shutdown the job in an orderly fashion. This behaviour can be changed at run-time by adding the appropriate parameter set to the top level .fcl file.
- One can ask that an output module be run only for events that pass a given trigger_path; this is done using the SelectEvents parameter set,
- At present there is no syntax to ask that an analyzer module be run only for events that pass or fail some of the trigger paths. A planned improvement to *art* is to give analyzer modules a SelectEvents parameter that behaves as it does for output modules.
- If a path appears in neither the trigger_paths nor the end_paths, there is no warning given.
- If a module label appears in no path, a warning will be given.

In the above there is a lot of focus on which groups of modules are free to be run in an arbitrary order. This is laying the groundwork for module-parallel execution: *art* is capable of identifying which modules may be run in parallel and, on a multi-core machine, art could start separate threads for each module. At present both ROOT and G4 are not thread-safe so this is not of immediate interest. But there are efforts underway to make both of these thread-safe and we may one day care about module-parallel execution; our interest in this will depend a great deal on the future evolution of the relative costs of memory and CPU.

For simple cases, in which there is one trigger path with only a few modules in the path, and one end path with only a few modules in the path, the extra level of bookkeeping is just extra typing with no obvious benefit. The benefit comes when many work groups wish to run their modules on the same events during one art job; perhaps this is a job skimming off many different calibration samples or perhaps it is a job selecting many different streams of interesting Monte Carlo events. In such a case, each work group needs only to define their own trigger path and their own end path, without regard for the requirements of other work groups; each work group also needs to ensure that their paths are added to the end_paths and trigger_paths variables. Art will then automatically, and correctly, schedule the work without redoing any work twice and without skipping work that must be done. This feature came for free with art and, while it imposes a small burden for novice users doing simple jobs, it provides an enormously powerful feature for advanced users. Therefore it was retained in art when some other features were removed.

23.8 Scheduled Reconstruction using Trigger Paths

Consider the following problem. You wish to run a job that has:

- Two producers MakeA_module.cc and MakeB_module.cc. You want to run both producers on all events.
- One analyzer module that you want to run on all events, CheckAll_module.cc.
- You have a filter module, Filter1_module.cc that has two modes, 0 and 1; the mode can be selected at run time via the parameter set.
- You wish to write all events that pass mode 0 of the filter to the file file0.root and you wish to write all events that pass mode 1 of the filter to file1.root

Here is code that would accomplish this:

```
process_name: filter1

source: {
   # Configure some source here.
}

physics: {

 producers : {
    aProducer: { module_type: MakeA }
```

```
    bProducer: { module_type: MakeB }
  }

 analyzers : {
    checkAll: { module_type: CheckAll }
  }

 filter : {
    selectMode0: {
      module_type: Filter1
      mode: 0
    }
    selectMode1: {
      module_type: Filter1
      mode: 1
    }
  }

  mode0: [ aProducer, bProducer, selectMode0 ]
  mode1: [ aProducer, bProducer, selectMode1 ]
  analyzermods: [ checkAll  ]
  outputFiles:  [ out0, out1 ]

  trigger_paths : [ mode0, mode1 ]
  end_paths : [ analyzermods, outputFiles ]
}

outputs: {
  out0: {
   module_type: RootOutput
   fileName: "file0.root"
   SelectEvents: { SelectEvents: [ mode0 ] }
  }

  out1: {
   module_type: RootOutput
   fileName: "file1.root"
   SelectEvents: { SelectEvents: [ mode1 ] }
  }

}
```

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, selectMode1, out0 and out1 are module labels, and these are names of paths: mode0, mode1, outputFiles, analyzermods.

23.9 Reconstruction On-Demand

23.10 Bits and Pieces

What variables are known to art? physics (which has the five reserved keywords fi

lters, analyzers, producers, trigger paths and end paths), what else? input file type RootInput

I know that trigger path are // different from end paths, they can contain different types of modules; // event gets frozen after trigger path.

art knows to match the value defi

ned by the name 'module_name" to a C++ object fi

le with the name module_name_module.so" somewhere in the path defi

ned by LD LIBRARY PATH.

Further information on the FHiCL language and usage can be found at the mu2e FHiCL page.

24 Data Products

24.1 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.

24.2 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.

ModuleLabel identifies the module that created the product; this is the module label , which distinguishes multiple instances of the same module within a produces . It is *not* the class name of the module.

InstanceName is a label for the data product that distinguishes two or more data products of the same type that were produced by the same module, in the same process . If a data product is already unique within this scope, it is legal to leave this field blank . The instance label is the optional argument of the call to "produces" in the constructor of the module (xxxx below) :

```
produces<T>("xxxx");
```

ProcessName is the name of the process that created this product. It is specified in the FHiCL file that specifies the run-time configuration for the job (shown as *ReadBack02* below):

```
process_name : ReadBack02
```

Because the full name of the product uses the underscore character to delimit fields, it is forbidden to use underscores in any of the names of the fields. Therefore, none of the following items may contain underscores:

- the class name of a class that is a data product; the exception is the cet::map_vector template; when creating the friendly name, *art* internally recognizes this case and protects against it
- the namespace in which a data product class lives
- module labels
- data product instance names
- process names

It is important to know which names need to match each other; see Section 30.1.

25 Producer Modules

26 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)
}
```

27 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)
}
```

28 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:
      {
```

```
        # experiment- or user-defined plugin service
      }
      ...
    }
```

29 art Input and Output

29.1 Input Modules

29.1.1 Configuring Input Modules to Read from Files

When reading from an existing file, *art* allows you to select the input files, the starting event, the number of events to read, etc., either from the command line or from the FHiCL file. If a particular quantity is controlled from both the command line and the FHiCL file, the value on the command line takes precedence.

The following code fragment tells *art* to read event data from the file of type "ROOT," named "file01.root" and to start at the beginning of the file. A value of "-1" for maxEvents tells *art* to read events until the end of file is reached:

To tell *art* to read 100 events, or until the end of file, which ever comes first, change the parameter maxEvents to 100. This also shows how to specify a list of input files:

The number of files in the list of input files is arbitrary. The following fragment tells *art* to skip the first two events (and thus start with the third):

The fragment below shows some other parameters that can be included in the source parameter set:

The parameters whose names start with *first* specify that the first event to be processed will be the first event that has an EventID greater than or equal to

Listing 29.1: Reading in a ROOT data file

```
source :{
  module_type : RootInput
  fileNames   : [ "file01.root" ]
  maxEvents   : -1
}
```

Listing 29.2: Reading in a ROOT data file

```
source : {
  module_type : RootInput
  fileNames   : [ "file01.root", "file02.root",  "file03.root" ]
  maxEvents   : 100
}
```

Listing 29.3: Reading in a ROOT data file

```
source : {
  module_type : RootInput
  fileNames   : [ "file01.root", "file02.root",  "file03.root" ]
  maxEvents   : 100
  skipEvents  : 2
}
```

the specified event. If one of the first* parameters is not specified, it takes a default value of -1 and is excluded from the comparison.

If a file of unsorted events is read in, *art* will, by default, present the events for processing in order of increasing event number. As a corollary to this, the output file will contain the events in sorted order. This sorting occurs one input file at a time; *art* does not sort across file boundaries in a list of input files. If the noEventSort parameter is set to *true*, the sorting is disabled, which will, in most cases, yield a minor performance improvement.

I have not yet learned the precise meaning of the skipBadFiles and the fileMatchMode parameters.

The inputCommands parameter tells *art* to delete certain data products from the copy of the event in memory after reading the input file. In other words, the input file itself is not modified but data products are removed from the copy of the event in memory before any modules are called. The syntax of this language is the same as for outputCommands, described .

In the pre-*art* versions of the framework, there were methods to select ranges of events or ranges of SubRuns. This is not yet working in *art*; the *art* developers will add this feature back once we decided exactly what we mean by ”ranges of

Listing 29.4: Reading in a ROOT data file

```
   firstRun             : 0
   firstSubRun          : 0
   firstEvent           : 0
   noEventSort          : false
   skipBadFiles         : false
   fileMatchMode        : "permissive"
   inputCommands        : ""
```

Listing 29.5: Reading in a ROOT data file

```
source :{
  module_type : EmptyEvent
  maxEvents   : 200
}
```

Listing 29.6: Reading in a ROOT data file

```
source :{
  module_type          : EmptyEvent
  firstRun             : 2
  firstSubRun          : 1
  firstEvent           : 1
  numberEventsInRun    : 1000
  numberEventsInSubRun :  100
  maxEvents            : 200
  resetEventOnSubRun   : true
}
```

events".

Specifying Many Input Files In the pre-art, python based, configuration language, the standard syntax to initialize a list of input files was limited to 255 files, after which an alternate syntax was required. This is no longer necessary; the length of a fhicl list is limited only by available memory.

Empty Source

In many simulation applications one wishes to start with an empty event, run one or more event generators, pass the generated particles through the Geant4, and so on. In art the first step in this chain is accomplished using a source module named EmptySource, as follows:

Instead of reading event-data from a file, the empty source increments the event number and presents an empty event to the modules that will do the work. One may configure EmptySource to specify the EventId of the first event, to specify the maximum number of events in a SubRun or SubRuns in a run.

The last option tells art to reset event numbers to start at 1 whenever art starts a new SubRun begins; this is the default behaviour and is opposite to the behaviour we inherited from CMS.

29.2 Output Filtering

Any output module can be configured to write out only those events passing a given trigger path.

The parameter set that configures the output module uses a parameter SelectEvents to control the output, as shown in the example below:

```
# this is only a fragment of a full configuration ...
 physics:
 {
   pathA: [ ... ]  # producers and filters are put in this path
   pathB: [ ... ]  # other producers, other filters are put in this path

   outA: [ passWriter ] # output modules and analyzers are put in this path
   outB: [ failWriter ] # output modules and analyzers are put in this path
   outC: [ exceptWriter ] # output modules and analyzers are put in this path

   trigger_paths: [ pathA, pathB ] # declare that these are "trigger paths"
   end_paths: [ outA outB outC ]   # declare these are "end paths"
 }

 outputs:
 {
   passWriter:
   {
     module_type: RootOutput
     fileName: "pathA_passes.root"
     # Write all the events for which pathA ended with 'true' from filtering.
     # Events which caused an exception throw will not be written.
     SelectEvents: { SelectEvents: [ "pathA&noexeception" ] }
   }
   failWriter:
   {
     module_type: RootOutput
     fileName: "pathA_failures.root"
     # Write all the events for which pathA ended with 'false' from filtering
     # Events which caused an exception throw will not be written.
     SelectEvents: { SelectEvents: [ "!pathA&noexception" ] }
   }
   exceptWriter:
   {
     module_type: RootOutput
     fileName: "pathA_exceptions.root"
     # Write all the events for which pathA or pathB ended because an excepti
     SelectEvents: { SelectEvents: [ "exception@pathA", "exception@pathB" ] }
   }
 }
```

29.3 Configuring Output Modules

30 art Misc Topics that Will Find Home

30.0.1 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 left to each experiment. Typically, experiments will choose to start new runs or new subRuns when any of the following happen:

- a preset number of events have been acquired
- a preset time interval has expired
- a disk file holding the ouptut has reached a preset size
- certain running conditions change

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

As runs are collections of subRuns, and subRuns are collections of events, events in turn are collections of *data products*. A data product is the smallest unit of data that can be added to or retrieved from a given event. Each experiment defines types (classes and structs) for its own data products. These include types that describe the raw data, and types to define the reconstructed data and the information produced by simulations. *art* knows nothing about the internals of any experiment's data products; for *art*, the data product is a "fundamental particle."

At the outside shell of the Russian doll that is the bookkeeping structure in *art*, runs are collected into the *event-data*, defined as all of the data products in an experiment's files; plus the metadata that accompanies them.

When an experiment takes data, events read from Data Acquisition System (DAQ) are typically written to disk files, with copies made on tape. *art* imposes only weak constraints on the event sequence within a file. The events in a single subRun may be spread over several files; conversely a single file may contain many runs, each of which contains many subRuns.

A critical feature of *art*'s design is that each event must be uniquely identifable by its event ID. This requirement also applies to simulated events.

30.1 Rules for Module Names

Within any experiment's software, sometimes names of files, classes, libraries, etc., must follow certain rules. Other times, conventions are just conventions. This section is concerned with actual rules only.

Consider a class named `MyClass` that you wish to make into an *art* module. First, your class must inherit from one of the module base classes, EDAnalyzer, EDProducer or EDFilter. Secondly, it must obey the following rules, all of which are case-sensitive.

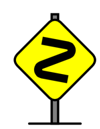
1. it must be in a file named `MyClass_module.cc`
   The build system will make this into a file named `lib/libMyClass_module.so`.

Listing 30.1: Module source sample

```
namespace xxxx {

  class MyClass : public art::EDAnalyzer {

  public:
    explicit MyClass(fhicl::ParameterSet const& pset);
    // Compiler generated destructor is OK.

    void analyze( art::Event const& event );

  };

  MyClass::MyClass(fhicl::ParameterSet const& pset){
   // Body of the constructor. You can access information
       in the parameter set here.
  }

  void MyClass::analyze(art::Event const& event){
    mf::LogVerbatim("test")
      << "Hello,␣world.␣From␣analyze.␣"
      << event.id();
  }

} // end namespace xxxx

using xxxx::MyClass;
DEFINE_ART_MODULE(MyClass);
```

2. the module source file must look like Listing 30.1 (where your experiment's namespace replaces xxxx):

   This example is for an analyzer. To create a producer or a filter module, you must inherit from either art::EDProducer or art::EDFilter, respectively. The last line (`DEFINE_ART_MODULE(MyClass);`) invokes a macro that inserts additional code into the `.so` file.

   For the experts: it inserts a factory method to produce an instance of the class and it inserts and auto-registration object that registers the factory method with *art*'s module registry.

   To declare this module to the framework you need to have a fragment like the following in your FHiCL file:

```

physics :
{
   analyzers:
   {
      looseCuts : { module_type  : MyClass }

      // Other analyzer modules listed here ...
   }
```

```
               }
```

where the string looseCuts is called a *module label* and is is defined below.

3. the previous item was for the case that your module is an analyzer. If it is a producer or filter, then the label *analyzers* needs to be either *producers* or *filters*.
4. When you put a data product into an event, the data provenance system records the module label of the module that did the "put."

30.2 Data Products and the Event Data Model

The part of *art* that deals with the bookkeeping of the data products is called the *Event Data Model*, which concerns itself with the following ideas:

1. what a data product looks like when it is in the memory of a running program
2. what it looks like on disk
3. how it moves between memory and disk
4. how a data product refers to another piece of event-data within the same event
5. how a given piece of experiment code accesses a data product
6. how the experiment code adds a new data product to the event
7. metadata that describes, for each data product,
   - what piece of code was used to create it
   - what is the run-time configuration of that code
   - what data products were read in by this experiment code
8. The mechanism by which the metadata is "married" to the data

One of the core principles of *art* is that experiment code modules may communicate with each other only via the event.

30.3 Basic *art* Rules

art prescribes that your classes (i.e., your *art modules*) always contain a *member function* that has a particular name, takes a particular set of arguments, and operates on every event; *art* will call this member function for every event 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. [1]

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 .

30.4 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 30.1. 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 30.2. 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.)

30.5 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

[1] Actually the loader that loads the shareable library, rather than *art* itself, will figure this out.

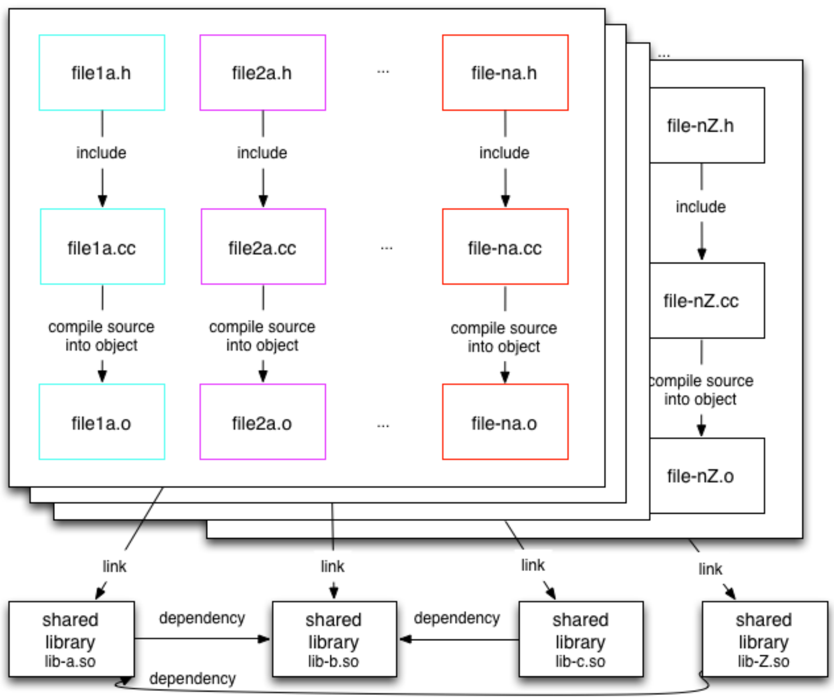


Figure 30.1: Illustration of compiled, linked "regular" C++ classes (not *art* modules) that can be used within the *art* framework. Many classes can be linked into a single shared library.

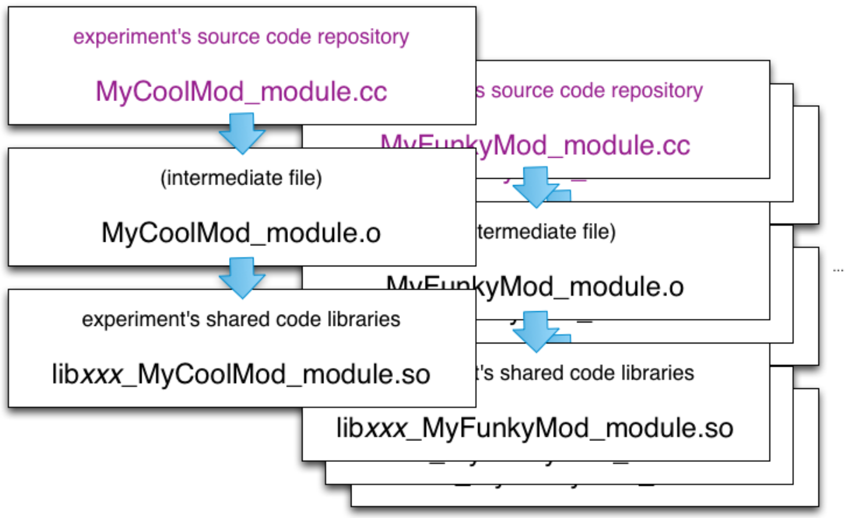


Figure 30.2: Illustration of compiled, linked *art* modules; each module is built into a single shared library for use by *art*

executable in the form of shareable object libraries that *art* loads dynamically at run time; these libraries are also called *dynamic load libraries* or *plugins*.

Your experiment will likely have many "regular" C++ classes (as distinct from the C++ classes that are modules, aka "*art* modules"). These "regular" classes get built into a set of shareable libraries, where each library contains object code for multiple classes.

Your experiment will likely have many modules, too. In fact you will likely be writing some for your own analyses. A module must be compiled into its own shareable object library, i.e., there is a one-to-one correspondance between the `.cc` file and the `.so` file for a given module. When the configuration file tells *art* to run a particular module, *art* finds the corresponding `.so` file, loads it, and calls the appropriate member function at each stage of the event loop.

30.6 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>
```

The notion of *namespace* is related to that of *scope*: Within a C++ source file (`.cc` files) a scope is designated by a set of curly braces ({ ... }). Once a namespace is defined within a given scope, any identifiers within that scope that “belong to” that namespace no longer need to be written with the prefi. E.g., the following fragment uses the `analyze` defined in the namespace `tex` (i.e., `tex :: analyze`):

```
namespace tex {
     class First : public art :: EDAnalyzer {
     public :
         explicit First ( fhicl :: ParameterSet const & );
         void analyze ( art :: Event const & event ) override ;
    };
}
```

Note that `EDAnalyzer` is defined in the namespace `art`, as is `Event`, and `ParameterSet` is in `fhicl`.

Note also that namespaces are often associated with UPS product code, although the product and the namespace names may not always be identical. E.g., code associated with the UPS product *fhiclcpp* is in the namespace `fhicl`.

All of the code in the toyExperiment UPS product was written in a namespace named `tex`; the name `tex` is an acronym-like shorthand for the toyExperiment (ToyEXperiment) UPS product. Because all of the Workbook code builds on top of the toyExperiment code, this code has been placed in the same namespace. The `tex` namespace has no special meaning to *art*, it is just a convenience. (Note that the *art* code itself is in a separate namespace called `art`.)

If you need more information about the C++ notion of *namespaces*, see a standard C++ reference.

30.7 Orphans

A best practice: define ids in the narrowest scope possible to avoid accidental name collisions

During processing, derived information in the event may be changed, added to or deleted; the raw data is not modified. The *event* is the smallest unit of data that art can process at one time.

How bash shell scripts work

If you would like to understand how they work, the following will be useful:

- BASH Programming - Introduction HOW-TO
  http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

- Bash Guide for Beginners
  http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
- Advanced Bash Scripting Guide
  http://www.tldp.org/LDP/abs/html/abs-guide.html

The first of these is a compact introduction and the second is a more comprehensive introduction.

The above guides were all found at the Linux Documentation Project: Workbook:

- http://www.tldp.org/guides.html

30.8 Code Guards

All of the header files that you will see in the Workbook wrap their contents with the following three lines:

```
#ifndef path_to_this_header_file_h
#define path_to_this_header_file_h
    // contents of the header file
#endif /* path_to_this_header_file_h */
```

The three lines beginning with # are macros that will be processed by the C preprocessor at the start of compilation. These lines are called *code guards* and they address the following issue.

Suppose that you have a main program that includes two header files `A.h` and `B.h`; further suppose that both of `A.h` and `B.h` include a third header file `C.h`. When you compile the main program, the C preprocessor will expand all of the include directives to create a temporary `.cc` file on which the compiler will do its work. This temporary file must contain exactly one copy of the header file `C.h`; if it contains either zero copies or more than one copy (as it would in this case), the compiler will issue an error. The C preprocessor, by itself, is not smart enough to skip the second inclusion of `C.h` but it does provide the tools for us to help it do so.

In the first two lines, the text `path_to_this_header_file_h` is the name of a C preprocessor variable; the choice of the variable name will be described later but the important feature is that it must be unique within the compilation unit (the file being compiled). When the C preprocessor encounters the included file `C.h`, the line `#ifndef` tells the preprocessor to check to see if the C preprocessor variable with this name is defined. If the variable is not defined then the lines between the `#ifndef` line and the `#endif` line will be included in the output of the C preprocessor. If it is has already been defined, these lines will be excluded from the output.

The first time that the preprocessor encounters `C.h` within a compilation unit, the variable will not have been defined and the contents of the header will be included in the output of the preprocessor. At the same time the second line of the above fragment will be executed; it is a preprocessor directive that tells the preprocessor to define the variable. In either case, when the preprocessor encounters the second inclusion of `C.h`, the `#ifndef` test will fail and the body of the header will not be copied into the output of the preprocessor. And so on for subsequent inclusions of `C.h`.

If every header file in a code base correctly uses code guards, then every header file can safely include all other header files on which it depends and one need not worry about this causing compiler errors due to multiple declarations of a class or function.

The full syntax of the `#define` directive allows one to specify a value for the variable but that is not important here; the `#ifndef` test only cares that the variable is defined, not what it value is.

For code guards to work, each header file must chose a C preprocessor variable name that is unique within every compilation unit in which it might be included, either directly included or indirectly included. The convention that is used by *art*, by other libraries managed by the *art* team, by the toyExperiment UPS product and by the Workbook is that the name of the variable is the name of the path to the header file, starting from the root of the code base and with the slash and dot characters changed to underscores; the reason for this change is that slash and dot characters are not legal in the name of a C preprocessor variable. This works because all of these products also adopt the convention that the path to their header file starts with the product name. While this is not perfect security it is a very high level of security.

30.9 Inheritance

30.9.1 Introduction

This section introduces a few of the ideas behind *inheritance* and *polymorphism*. There are many, many different ways to use *inheritance* and *polymorphism* but you only need to understand the small subset that are relevant for the Workbook exercises. You can read about inheritance and polymorphism at the following url:

http://www.cplusplus.com/doc/tutorial/inheritance/

Skip the section on Friendship and start at the section on inheritance. When you get to the bottom of the page, continue to the next page by clicking on the arrow for "Polymorphism". You can skip the discussion of protected and private inheritance because you will only need to know about public inheritance.

After you have learned this material, return to this section and work through the following example which serves as a test that you have learned the necessary material. This example is motivated by the Polygon example given in the referenced material. In this example there is a base class named `Shape` and three derived classes, `Circle`, `Triangle` and `Rectangle`. The main program that exercises the these four classes is `itest.cc`.

30.9.2 Homework

To build and run this example:

1. log in and follow the follow the steps in Section **??**
2. cd to the directory for this exercise
```
$ cd Inheritance/v1
$ ls
build Circle.h Rectangle.cc Shape.cc Triangle.cc
Circle.cc itest.cc Rectangle.h Shape.h Triangle.h
```
3. build the exercise
```
$ ../build
```
This will create the executable file `itest`
4. run the exercise
```
$ itest
Area of circle c1 is:  3.14159
Area of circle c2 is:  12.5664
Area of rectangle r1 is:  4
Area of triangle t1 is:  0.5
Area of triangle t2 is:  2
This circle has an area of 3.14159 and a color of undefined
This circle has an area of 12.5664 and a color of red
Unknown shape has color:  blue
This triangle has an area of 0.5 and a color of green
This triangle has an area of 2 and a color of yellow
```

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

Read the code in the example and apply what you learned from the `cplusplus.com` website. Understand why the example prints out what it does.

The next subsection contains some discussion about the example. In particular it will discuss the *explicit* and `override` keywords.

30.9.3 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 dervived classes.
2. the two constructors guarantee that the `color_` data member will be initialized whenever a derived class is instantiated.
3. the class as 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.

30.10 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

you focus on the relevant information if you need to consult C++ references on inheritance.

30.11 Pointers

C++, like many other computer languages, allows you to define variables that are pointers to information held in other variables. The value of a pointer is the memory address of the information held by the given variable. A native C++ pointer is often referred to as a *bare pointer*. While pointers provide great flexibility for producing fast, efficient algorithms, they are also easy to misuse. *art* has been designed so that user code will rarely, if ever, interact with *art* via bare pointers; when pointer-like behaviour is required, *art* will provide that information inside a wrapper that is generically referred to as a *smart pointer* or a *safe pointer*; *art* defines different sorts of smart pointers for use in different circumstances. The job of a smart pointer is to recognize misuse and to protect against it. One commonly used type of smart pointer is called a *handle*.

30.12 RootOutput and table of event IDs

When RootOutput writes a file, it writes the event information to the file and it also writes a table of event Ids that allows it to random access a single event without needing to read all of the events before it. This table is kept in order of increasing event id. When you open a file and read it, RootInput starts reads events in the order found in the table.

30.13 Troubleshooting

(Section 6.3) `setup` returns the error message

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

The simplest solution is to log out and log in again.

Part IV

Index

Index