Object-Oriented Programming, Writing Classes, and Creating Libraries and Applications

Brent A. Perdue

ROOT @ TUNL

July 15, 2009
Outline of the Talk

1. Example of a Real Object
2. Pros and Cons of Writing Classes
3. Components of a C++ Class
4. Inheritance
5. Shared Object Libraries
6. Stand-alone Applications
class Amplifier : public NIM

- **Parent Class: NIM**
  Attributes: width, height, length, rear connector specs
  Interface Points: rear connector power pins

- **Child Class: Amplifier**
  Attributes: power voltage, fine gain, coarse gain, shaping time, input polarity, internal ICs
  Interface Points: fine/coarse gain knobs, shaping time knob, input polarity switch, input connector, unipolar/bipolar output connectors
Why is this Amplifier object useful?

1. Easy to move from one NIM bin to another \(\rightarrow\) Amplifier \textit{inherits} NIM attributes.

2. Easy to change operational variables (e.g., fine gain) \(\rightarrow\) Amplifier provides \textit{methods} of changing variables via knobs, switches, and pots.

3. Easy to pass information in and out \(\rightarrow\) Amplifier provides \textit{methods} of signal passing via input and output connectors.

4. Easy to achieve desired shaping and amplification \(\rightarrow\) Amplifier conceals the internal details from the user (\textit{encapsulation}).

5. Easy to use inside a more complicated circuit \(\rightarrow\) Amplifier allows one to think at higher level of \textit{abstraction}.
Pros and Cons of Writing Classes

- **Pros**
  - modular programming
  - encapsulation
  - abstraction
  - inheritance
  - reuse of code

- **Cons**
  - more time to design and develop
  - more lines of code
  - more complexity

- **C++ software packages that use the OOP framework**
  - ROOT
  - Geant4
Definition: Class - A user-defined type. It defines the characteristics of an object, including its attributes (data members) and behaviors (methods or member functions). It can be thought of as a blueprint that describes the nature of an object.
The Class Specification

// Using preprocessor directives to avoid multiple inclusion
// of the header files in a program.
#ifndef ClassName_h
#define ClassName_h 1

// Class definition
class ClassName
{
  public:
    // Prototypes of member methods that are public and can be
    // used outside the class definition and implementation.
  private:
    // Data member declarations that are private and can be used
    // only inside the class definition and implementation.
};

#endif

Figure: ClassName-1.h
Constructors

class ClassName
{
  public:
    // Demonstrates polymorphism
    ClassName(); // Default constructor
    ClassName(Int_t i); // Alternate constructor
    ClassName(Double_t d); // Alternate constructor

  private:
    // Private data members and methods
};

Figure: ClassName-2.h
Destructors

```cpp
class ClassName
{
    public:
        ClassName(); // Default constructor
        ~ClassName(); // Destructor

        // Destructors are used to clean up and release resources. The most
        // common use of destructor is to release memory acquired in a
        // constructor. Destructors do not need to be called explicitly.

    private:
        // Private data members and methods
};
```

Figure: ClassName-3.h
Components of a C++ Class

The Class Specification

Example From ROOT: TH1D

public:
    TH1D();
    TH1D(const TVectorD &v);
    TH1D(const TH1D &h1d);
    TH1D(const char *name,const char *title,Int_t nbinsx,const Float_t *xbins);
    TH1D(const char *name,const char *title,Int_t nbinsx,const Double_t *xbins);
    TH1D(const char *name,const char *title,Int_t nbinsx,Double_t xlow,Double_t xup);
    virtual ~TH1D();

Figure: ROOTExample-1.h
Members of a Class

```cpp
#include <ctime>
#include "TRandom3.h"

#ifndef Coin_h
#define Coin_h 1

class Coin
{
  public:
    Coin();
    ~Coin();

    // Public member methods
    void Flip(void);
    Int_t GetFace(void);  // Public "get" method
    void SetFace(Int_t face); // Public "set" method

    static Int_t HEADS(void){ return 0; }; // static = only one copy for all Coin objects
    static Int_t TAILS(void){ return 1; });

  private:
    // Private data members
    Int_t fFace;
    TRandom3 fRandGen;

    Int_t ZeroOrOne(void); // Private "helper" method

};
#endif
```

Figure: Coin.h
#include "Coin.h"
Coin::Coin()
{
    fRandGen.SetSeed(time(0));
    Flip(); // Initializes the state of the coin.
} // END of constructor

Coin::~Coin(){} // Destructor does nothing in this case.

void Coin::Flip(void)
{
    SetFace( ZeroOrOne() ); // Set method calls helper method.
} // END of Flip()

Int_t Coin::ZeroOrOne(void)
{
    return (Int_t)(fRandGen.Rndm() * 2);
} // END of ZeroOrOne()

void Coin::SetFace(Int_t face)
{
    fFace = face;
} // END of SetFace()

Int_t Coin::GetFace(void)
{
    return fFace;
} // END of GetFace()
#include "Riostream.h"
#include "Coin.cxx"

void countflips(void)
{
  Int_t numHeads = 0, numTails = 0;

  Coin aCoin; // Create an instance of the Coin class

  for(Int_t i=0; i<1000; i++)
  {
    aCoin.Flip();

    if( aCoin.GetFace() == Coin::HEADS() ) numHeads++;
    else numTails++;
  }

  cout << "The number of flips: " << numHeads + numTails << endl;
  cout << "The number of heads: " << numHeads << endl;
  cout << "The number of tails: " << numTails << endl;
}

Figure: countflips.C
Components of a C++ Class

Running the Program

[perdue@riesling: simple]$ root countflips.C
root [0]
Processing countflips.C...
The number of flips: 1000
The number of heads: 512
The number of tails: 488
root [1] .q
[perdue@riesling: simple]$ ▶

[perdue@riesling: simple]$ root
root [0] .L countflips.C
root [1] countflips()
The number of flips: 1000
The number of heads: 501
The number of tails: 499
root [2] .q
[perdue@riesling: simple]$ ▶

[perdue@riesling: simple]$ root
root [0] .L countflips.C.so
root [1] countflips()
The number of flips: 1000
The number of heads: 518
The number of tails: 482
root [2] .q
[perdue@riesling: simple]$ ▶
Summary: Components of a Class

- **Class Specification**
  - **public:** the interface to the world
    - constructors, destructors: how the objects are created and destroyed
    - set and get methods: how to pass information to and from the objects
      - Flip(), SetFace(), GetFace() in Coin
      - knobs, switches, and connectors on Amplifier
  - **private:** the internal structure (encapsulation and information hiding)
    - data members and helper methods
      - fFace, fRandGen, ZeroOrOne() in Coin
      - fine gain, shaping time, internal ICs, etc in Amplifier

- **Class Implementation**
  - definitions of constructors, destructors, and methods, and how they interact
    - Flip() calls SetFace(), and SetFace() calls ZeroOrOne()
    - wires and solder in Amplifier
**Definition:** Inheritance - A consequence of deriving a new class from an existing one. The new class automatically *inherits* some or all of the data members and methods defined in the original class. Inheritance is a defining characteristic of object-oriented programming.
Purposes of Inheritance

- Reuse existing software - derived classes can be created via inheritance faster and easier than by writing them from scratch.
- Classification hierarchies can be developed that exploit natural commonalities between objects.
Descendants of TObject in ROOT
class Coin
{
    public:
        // Constructor and destructor
        // Public member methods
        // Public data members

    private:
        // Private data members

    protected:
        // Private inside Coin, but is inherited by derived (child) classes.
        // It also remains private inside the derived classes.
        void SetSeed(UInt_t seed);

};

Figure: Coin.h
Creating a Child Class

```cpp
#include <ctime>
#include "Coin.h"

#ifndef Nickel_h
#define Nickel_h 1

class Nickel : public Coin
{
    public:
        Nickel(){ SetSeed(time(0)); };
        ~Nickel(){;};

        Int_t GetValue(void){ return VALUE; };

    private:
        static const Int_t VALUE = 5;

};
#endif
```

Figure: Nickel.h
Creating a Child Class

```cpp
#include <ctime>
#include "Coin.h"

#ifndef Quarter_h
#define Quarter_h 1

class Quarter : public Coin
{
    public:
        Quarter(){ SetSeed(time(0)+12345); }
    ~Quarter(){;};

        Int_t GetValue(void){ return VALUE; ;}
    
    private:
        static const Int_t VALUE = 25;

};
#endif
```

*Figure: Quarter.h*
Using the Child Classes

#include "Riostream.h"
#include "Nickel.h"
#include "Quarter.h"
#include "Coin.cxx"

void getrich(void)
{
    Nickel aNickel;
    Quarter aQuarter;

    Int_t numNickelHeads = 0, numQuarterHeads = 0;

    for(Int_t i=0; i<10; i++)
    {
        aNickel.Flip();
        aQuarter.Flip();

        if( aNickel.GetFace() == Coin::HEADS() ) numNickelHeads++;
        if( aQuarter.GetFace() == Coin::HEADS() ) numQuarterHeads++;
    }  //END of i loop

    Int_t cents = aNickel.GetValue()*numNickelHeads + aQuarter.GetValue()*numQuarterHeads;
    printf("You can keep $%.2f\n", cents/100, cents%100);
}  //END of getrich()

Figure: getrich.C
How do you create a child class from an existing class?

- class ChildName : public ParentName
- class Nickel : public Coin

What gets inherited?

- all public members of the class
  - Flip(), GetFace(), SetFace(), HEADS(), TAILS()
- all protected members of the class
  - SetSeed()

What doesn’t get inherited?

- constructors and destructors
  - Coin(), ~Coin()
- any private members of the class
  - fFace, fRandGen, ZeroOrOne()
"ACLiC will build a CINT dictionary and a shared library from your C++ script using the compiler and the compiler options that were used to compile the ROOT executable. You do not have to write a makefile remembering the correct compiler options, and you do not have to exit ROOT."

- ROOT User Guide, Chapter 7

Commands in Cint to compile the Coin, Nickel, and Quarter classes:

- root [0] .L Coin.cxx+
- root [1] .L Nickel.h+
- root [2] .L Quarter.h+

Resulting shared object libraries:

- Coin.cxx.so
- Nickel.h.so
- Quarter.h.so
Compiling a Shared Object Library Using a Shell Script

#!/bin/sh

# (1) Creating dictionary code from header files using rootcint
echo "Creating dictionary"
rootcint -f dictCoins.cxx -c Coin.h Nickel.h Quarter.h

# (2) Compiling source files
echo "Compiling dictionary"
gcc -c -o dictCoins.o dictCoins.cxx ‘root-config --cflags’

echo "Compiling source code"
gcc -c -o Coin.o Coin.cxx ‘root-config --cflags’

# (3) Creating shared object library
echo "Creating shared object library"
gcc -shared -o libCoins.so dictCoins.o Coin.o

echo "Finished"

Figure: shared.sh
Breakdown of the Shell Script

1. Creating dictionary: `rootcint -f dictCoins.cxx -c Coin.h Nickel.h Quarter.h`
   - The program *rootcint* generates the CINT dictionaries needed in order to get access to one's classes via the interpreter.
   - "-f" = force overwrite of the output dictionary file (dictCoins.cxx)
   - "-c" = write interpreter method interface stubs to the output file

   - `gcc` = GNU C and C++ compiler
   - "-c" = compile, but do not link into an executable
   - "-o" = specify the output file (Object.o in this case)
   - `root-config –cflags` = returns compiler flags needed for code containing ROOT includes

3. Creating shared object: `gcc -shared -o libCoins.so dictCoins.o Coin.o`
   - "-shared" = Produce a shared object which can then be linked with other objects to form an executable
   - "-o" = specify the output file (libCoins.so in this case)
Creating dictionary
Compiling dictionary
Compiling source code
Creating shared object library

root [0] / libCoins.so
root [1] Coin aCoin
(root int)
(root int)
root [5] aCoin.HEADS()
(root int)
root [6] aCoin.TAILS()
(root int)
root [7] Nickel aNickel
root [8] aNickel.GetValue()
(root int)
root [9] aNickel.GetFace()
(root int)
root [10] aNickel.Flip()
(root int)
root [12] aNickel.Flip()
(root int)
root [14] Quarter aQuarter
root [15] aQuarter.GetValue()
(root int)

rem: cannot remove "getrich": No such file or directory

The main function is the entry point for every C++ program. It is required to build a stand-alone application.

- **Signature** - `Int_t main(Int_t argc, Char_t *argv[])`
  - returns an `Int_t`
  - `argc` = the number of command-line arguments
  - `argv[]` = an array of the command-line arguments
    - `argv[0]` is usually the name of the program
- **main()** returns 0 at the end
The Structure of main()

```c
#include "Riostream.h"
#include "Nickel.h"
#include "Quarter.h"
#include "Coin.h"

Int_t main(Int_t argc, Char_t *argv[]) // The body of main is identical to getrich.C
{
    Nickel aNickel;
    Quarter aQuarter;

    Int_t numNickelHeads = 0, numQuarterHeads = 0;

    for(Int_t i=0; i<10; i++)
    {
        aNickel.Flip();
        aQuarter.Flip();

        if( aNickel.GetFace() == Coin::HEADS() ) numNickelHeads++;
        if( aQuarter.GetFace() == Coin::HEADS() ) numQuarterHeads++;
    }

    Int_t cents = aNickel.GetValue()*numNickelHeads + aQuarter.GetValue()*numQuarterHeads;
    printf("You can keep $%i.%i\n",cents/100,cents%100);

    return 0;
}
```

Figure: main.cxx
Compiling a Stand-alone Application Using a Shell Script

```bash
#!/bin/sh

# (1) Creating dictionary code from header files using rootcint
echo "Creating dictionary"
rootcint -f dictCoins.cxx -c Coin.h Nickel.h Quarter.h

# (2) Compiling source files
echo "Compiling dictionary files"
gcc -c -o dictCoins.o dictCoins.cxx 'root-config --cflags'
echo "Compiling source code"
gcc -c -o Coin.o Coin.cxx 'root-config --cflags'
gcc -c -o main.o main.cxx 'root-config --cflags'

# (3) Creating shared object library
echo "Creating shared object library"
gcc -shared -o libCoins.so dictCoins.o Coin.o

# (4) Linking object files into an application
gcc -o getrich main.o libCoins.so 'root-config --glibs'

echo "Finished"
```

Figure: compile.sh
Compiling and Running the Program Using a Shell Script

[perdue@riesling: application]$ ./compile.sh
Creating dictionary
Compiling dictionary
Compiling source code
Creating shared object library
Finished
[perdue@riesling: application]$ ./getrich
You can keep $1.85
[perdue@riesling: application]$ ./getrich
You can keep $1.40
[perdue@riesling: application]$ ./getrich
You can keep $1.65
[perdue@riesling: application]$ ./cleanup.sh
removed 'Coin.o'
removed 'dictCoins.o'
removed 'main.o'
removed 'dictCoins.h'
removed 'dictCoins.cxx'
removed 'libCoins.so'
removed 'getrich'
[perdue@riesling: application]$  ■
Definition: Makefile - Text files formatted in a way that tell the *make* utility how to compile one or multiple source files. The Makefile is essentially a list of rules for automating the compilation process. The advantage of using a Makefile over a shell script is that the Makefile sets up dependencies for when a file should be recompiled.
Components of a Makefile

- A Makefile rule:
  
  target: dependencies
  
  [tab] system command
  
  - Rule execution is based on timestamps. If any dependency is newer than the target, then the command is executed.

- An example rule:
  
  Coin.o: Coin.cxx Coin.h
  
  @gcc -c -o Coin.o Coin.cxx ‘root-config –cflags‘

- The Makefile executes the same commands as the shell script compile.sh, but it only executes the commands if they are needed.
Compiling and Running the Program Using a Makefile

```
[perdue@riesling: application]$ make
  Compiling: main.cxx ---> main.o
  Compiling: Coin.cxx ---> Coin.o
  Making Dictionary: Coin.h Nickel.h Quarter.h ---> dictCoins.cxx
  Compiling: dictCoins.cxx ---> dictCoins.o
  Creating Shared Lib: Coin.o dictCoins.o ---> libCoins.so
  Linking: main.o libCoins.so ---> getrich
[perdue@riesling: application]$ ./getrich
You can keep $0.70
[perdue@riesling: application]$ ./getrich
You can keep $1.75
[perdue@riesling: application]$ ./getrich
You can keep $0.95
[perdue@riesling: application]$ make clean
removed `Coin.o`
removed `dictCoins.o`
removed `main.o`
removed `dictCoins.h`
removed `dictCoins.cxx`
removed `libCoins.so`
removed `getrich`
[perdue@riesling: application]$ 
```
Advantage of Using a Makefile

```bash
[perdue@riesling: application]$ make
  Compiling: main.cxx --> main.o
  Compiling: Coin.cxx --> Coin.o
  Making Dictionary: Coin.h Nickel.h Quarter.h --> dictCoins.cxx
  Compiling: dictCoins.cxx --> dictCoins.o
  Creating Shared Lib: Coin.o dictCoins.o --> libCoins.so
  Linking: main.o libCoins.so --> getrich
[perdue@riesling: application]$ touch Coin.cxx
[perdue@riesling: application]$ make
  Compiling: Coin.cxx --> Coin.o
  Creating Shared Lib: Coin.o dictCoins.o --> libCoins.so
  Linking: main.o libCoins.so --> getrich
[perdue@riesling: application]$ touch main.cxx
[perdue@riesling: application]$ make
  Compiling: main.cxx --> main.o
  Linking: main.o libCoins.so --> getrich
[perdue@riesling: application]$ touch Nickel.h
[perdue@riesling: application]$ make
  Making Dictionary: Coin.h Nickel.h Quarter.h --> dictCoins.cxx
  Compiling: dictCoins.cxx --> dictCoins.o
  Creating Shared Lib: Coin.o dictCoins.o --> libCoins.so
  Linking: main.o libCoins.so --> getrich
[perdue@riesling: application]$ make clean
removed 'Coin.o'
removed 'dictCoins.o'
removed 'main.o'
removed 'dictCoins.h'
removed 'dictCoins.cxx'
removed 'libCoins.so'
removed 'getrich'
[perdue@riesling: application]$ 
```
To be able to see any ROOT graphics windows created in an application an instance of the class TApplication must be created.

TApplication - “creates the ROOT Application Environment that interfaces to the windowing system event loop and event handlers. This class must be instantiated exactly once in any given application.”


```c++
#include "TApplication.h"

Int_t main(Int_t argc, Char_t *argv[])
{
  TApplication theApp("theApp", &argc, argv);

  // The body of main goes here.
  theApp.Run();
  return 0;
}
```

Figure: tapplication.cxx
Recap of the Topics Covered

1. Example of a Real Object
2. Pros and Cons of Writing Classes
3. Components of a C++ Class
4. Inheritance
5. Shared Object Libraries
6. Stand-alone Applications