Contents

Chapter 1

A Little C++

This chapter introduces you to some useful insights into C++, to allow you to use of the most advanced features in ROOT. It is in no case a full course in C++.

1.1 Classes, Methods and Constructors

C++ extends C with the notion of class. If you're used to structures in C, a class is a `struct` that is a group of related variables, which is extended with functions and routines specific to this structure (class). What is the interest? Consider a `struct` that is defined this way:

```
struct Line {
   float x1;
   float y1;
   float x2;
   float y2;
}
```

This structure represents a line to be drawn in a graphical window. `(x1,y1)` are the coordinates of the first point, `(x2,y2)` the coordinates of the second point. In the standard C, if you want to draw effectively such a line, you first have to define a structure and initialize the points (you can try this):

```
Line firstline;
firstline.x1 = 0.2;
firstline.y1 = 0.2;
firstline.x2 = 0.8;
firstline.y2 = 0.9;
```

This defines a line going from the point `(0.2,0.2)` to the point `(0.8,0.9)`. To draw this line, you will have to write a function, say `LineDraw(Line l)` and call it with your object as argument:

```
LineDraw(firstline);
```

In C++, we would not do that. We would instead define a class like this:

```
class TLine {
   Double_t x1;
   Double_t y1;
   Double_t x2;
   Double_t y2;
   TLine(int x1, int y1, int x2, int y2);
   void Draw();
}
```

Here we added two functions, that we will call methods or member functions, to the **`TLine`** class. The first method is used for initializing the line objects we would build. It is called a constructor. The second one is the `Draw` method itself. Therefore, to build and draw a line, we have to do:

```
TLine l(0.2,0.2,0.8,0.9);
l.Draw();
```

The first line builds the object `l`by calling its constructor. The second line calls the `TLine::Draw()` method of this object. You don't need to pass any parameters to this method since it applies to the object l, which knows the coordinates of the line. These are internal variables `x1`, `y1`, `x2`, `y2` that were initialized by the constructor.

1.2 Inheritance and Data Encapsulation

We have defined a **TLine** class that contains everything necessary to draw a line. If we want to draw an arrow, is it so different from drawing a line? We just have to draw a triangle at one end. It would be very inefficient to define the class **TArrow** from scratch. Fortunately, inheritance allows a class to be defined from an existing class. We would write something like:

```
class TArrow : public TLine {
   int ArrowHeadSize;
   void Draw();
   void SetArrowSize(int arrowsize);
}
```

The keyword "`public`" will be explained later. The class **TArrow** now contains everything that the class **TLine** does, and a couple of things more, the size of the arrowhead and a function that can change it. The Draw method of **TArrow** will draw the head and call the draw method of **TLine**. We just have to write the code for drawing the head!

1.2.1 Method Overriding

Giving the same name to a method (remember: method = member function of a class) in the child class (**TArrow**) as in the parent (**TLine**) does not give any problem. This is called **overriding** a method. Draw in **TArrow** overrides Draw in **TLine**. There is no possible ambiguity since, when one calls the `Draw()` method; this applies to an object which type is known. Suppose we have an object `l` of type **TLine** and an object `a` of type **TArrow**. When you want to draw the line, you do:

```
l.Draw();
```

`Draw()` from **TLine** is called. If you do:

```
a.Draw();
```

`Draw()` from **TArrow** is called and the arrow `a` is drawn.

1.2.2 Data Encapsulation

We have seen previously the keyword "`public`". This keyword means that every name declared public is seen by the outside world. This is opposed to "`private`" that means only the class where the name was declared private could see this name. For example, suppose we declare in **TArrow** the variable `ArrowHeadSize` private.

```
private:
   int ArrowHeadSize;
```

Then, only the methods (i.e. member functions) of **TArrow** will be able to access this variable. Nobody else will see it. Even the classes that we could derive from **TArrow** will not see it. On the other hand, if we declare the method `Draw()` as public, everybody will be able to see it and use it. You see that the character public or private does not depend of the type of argument. It can be a data member, a member function, or even a class. For example, in the case of **TArrow**, the base class **TLine** is declared as public:

```
class TArrow : public TLine { ...
```

This means that all methods of **TArrow** will be able to access all methods of **TLine**, but this will be also true for anybody in the outside world. Of course, this is true if **TLine** accepts the outside world to see its methods/data members. If something is declared private in **TLine**, nobody will see it, not even **TArrow** members, even if **TLine** is declared as a public base class.

What if **TLine** is declared "`private`" instead of "`public`" ? Well, it will behave as any other name declared private in **TArrow**: only the data members and methods of **TArrow** will be able to access **TLine**, its methods and data members, nobody else. This may seem a little bit confusing and readers should read a good C++ book if they want more details. Especially since, besides public and private, a member can be protected. Usually, one puts private the methods that the class uses internally, like some utilities classes, and that the programmer does not want to be seen in the outside world.

With "good" C++ practice (which we have tried to use in ROOT), all data members of a class are private. This is called data encapsulation and is one of the strongest advantages of Object Oriented Programming (OOP). Private data members of a class are not visible, except to the class itself. So, from the outside world, if one wants to access those data members, one should use so called "getters" and "setters" methods, which are special methods used only to get or set the data members. The advantage is that if the programmers want to modify the inner workings of their classes, they can do so without changing what the user sees. The user does not even have to know that something has changed (for the better, hopefully). For example, in our **TArrow** class, we would have set the data member `ArrowHeadSize` private. The setter method is `SetArrowSize()`, we do not need a getter method:

```
class TArrow : public TLine {
   private:
      int ArrowHeadSize;
   public:
      void Draw();
      void SetArrowSize(int arrowsize);
}
```

To define an arrow object you call the constructor. This will also call the constructor of **TLine**, which is the parent class of **TArrow**, automatically. Then we can call any of the line or arrow public methods:

```
root[] TArrow *myarrow = new TArrow(1,5,89,124);
root[] myarrow->SetArrowSize(10);
root[] myarrow->Draw();
```

1.3 Creating Objects on the Stack and Heap

To explain how objects are created on the stack and on the heap we will use the `Quad` class. You can find the definition in `$ROOTSYS/tutorials/quadp/Quad.h` and `Quad.cxx`. The `Quad` class has four methods. The constructor and destructor, `Evaluate` that evaluates `ax**2 + bx +c`, and `Solve` which solves the quadratic equation `ax**2 + bx +c = 0`.

`Quad.h` :

```
class Quad {
   public:
      Quad(Float_t a, Float_t b, Float_t c);
      ~Quad();
      Float_t Evaluate(Float_t x) const;
      void Solve() const;
   private:
      Float_t fA;
      Float_t fB;
      Float_t fC;
};
```

`Quad.cxx`:

```
#include <iostream.h>
#include <math.h>
#include "Quad.h"

Quad::Quad(Float_t a, Float_t b, Float_t c) {
   fA = a;
   fB = b;
   fC = c;
}
Quad::~Quad() {
   Cout <<"deleting object with coeffts: "<< fA << "," << fB << ","
                                          << fC << endl;
}
Float_t Quad::Evaluate(Float_t x) const {
   return fA*x*x + fB*x + fC;
}
void Quad::Solve() const {
   Float_t temp = fB*fB - 4.*fA*fC;
```

```
      if ( temp > 0. ) {
         temp = sqrt( temp );
         cout << "There are two roots: " << ( -fB - temp ) / (2.*fA)
         << " and " << ( -fB + temp ) / (2.*fA) << endl;
      } else {
         if ( temp == 0. ) {
            cout << "There are two equal roots: " << -fB / (2.*fA)
                                                  << endl;
         } else {
            cout << "There are no roots" << endl;
         }
      }
   }
```

Let us first look how we create an object. When we create an object by:

```
root[] Quad my_object(1.,2.,-3.);
```

We are creating an object on the stack. A FORTRAN programmer may be familiar with the idea; it is not unlike a local variable in a function or subroutine. Although there are still a few old timers who do not know it, FORTRAN is under no obligation to save local variables once the function or subroutine returns unless the SAVE statement is used. If not then it is likely that FORTRAN will place them on the stack and they will "pop off" when the RETURN statement is reached. To give an object more permanence it has to be placed on the heap.

```
root[] .L Quad.cxx
root[] Quad *my_objptr = new Quad(1.,2.,-3.);
```

The second line declares a pointer to `Quad` called `my_objptr`. From the syntax point of view, this is just like all the other declarations we have seen so far, i.e. this is a stack variable. The value of the pointer is set equal to

```
new Quad(1.,2.,-3.);
```

`new`, despite its looks, is an operator and creates an object or variable of the type that comes next, `Quad` in this case, on the heap. Just as with stack objects it has to be initialized by calling its constructor. The syntax requires that the argument list follow the type. This one statement has brought two items into existence, one on the heap and one on the stack. The heap object will live until the delete operator is applied to it.

There is no FORTRAN parallel to a heap object; variables either come or go as control passes in and out of a function or subroutine, or, like a COMMON block variables, live for the lifetime of the program. However, most people in HEP who use FORTRAN will have experience of a memory manager and the act of creating a bank is a good equivalent of a heap object. For those who know systems like ZEBRA, it will come as a relief to learn that objects do not move, C++ does not garbage collect, so there is never a danger that a pointer to an object becomes invalid for that reason. However, having created an object, it is the user's responsibility to ensure that it is deleted when no longer needed, or to pass that responsibility onto to some other object. Failing to do that will result in a memory leak, one of the most common and most hard-to-find C++ bugs.

To send a message to an object via a pointer to it, you need to use the "`->`" operator e.g.:

```
root[] my_objptr->Solve();
```

Although we chose to call our pointer `my_objptr`, to emphasize that it is a pointer, heap objects are so common in an object-oriented program that pointer names rarely reflect the fact - you have to be careful that you know if you are dealing with an object or its pointer! Fortunately, the compiler won't tolerate an attempt to do something like:

```
root[] my_objptr.Solve();
```

As we have seen, heap objects have to be accessed via pointers, whereas stack objects can be accessed directly. They can also be accessed via pointers:

```
root[] Quad stack_quad(1.,2.,-3.);
root[] Quad *stack_ptr = &stack_quad;
root[] stack_ptr->Solve();
```

Here we have a `Quad` pointer that has been initialized with the address of a stack object. Be very careful if you take the address of stack objects. As we shall see soon, they are deleted automatically, which could leave you with an illegal pointer. Using it will corrupt and may well crash the program!

It is time to look at the destruction of objects. A destructor is a special C++ function that releases resources for (or destroy) an object of a class. It is opposite of a constructor that create the object of a class when is called. The compiler will provide a destructor that does nothing if none is provided. We will add one to our Quad class so that we can see when it is called. The class names the destructor but with a prefix ~ which is the C++ one's complement

i.e. bit wise complement, and hence has destruction overtones! We declare it in the .h file and define it in the `.cxx` file. It does not do much except print out that it has been called (still a useful debug technique despite today's powerful debuggers!).

Now run root, load the Quad class and create a heap object:

```
root[] .L Quad.cxx
root[] Quad *my_objptr = new Quad(1.,2.,-3.);
```

To delete the object:

```
root[] delete my_objptr;
root[] my_objptr = 0;
```

You should see the print out from its destructor. Setting the pointer to zero afterwards is not strictly necessary (and Cling does it automatically), but the object is no more accessible, and any attempt to use the pointer again will, as has already been stated, cause grief. So much for heap objects, but how are stack objects deleted? In C++, a stack object is deleted as soon as control leaves the innermost compound statement that encloses it. Therefore, it is singularly futile to do something like:

```
root[] {  Quad my_object(1.,2.,-3.); }
```

Cling does not follow this rule; if you type in the above line, you will not see the destructor message. As explained in the Script lesson, you can load in compound statements, which would be a bit pointless if everything disappeared as soon as it was loaded! Instead, to reset the stack you have to type:

```
root[] gROOT->Reset();
```

This sends the Reset message via the global pointer to the ROOT object, which, amongst its many roles, acts as a resource manager. Start ROOT again and type in the following:

```
root[] .L Quad.cxx
root[] Quad my_object(1.,2.,-3.);
root[] Quad *my_objptr = new Quad(4.,5.,-6.);
root[] gROOT->Reset();
```

You will see that this deletes the first object but not the second. We have also painted ourselves into a corner, as `my_objptr` was also on the stack. This command will fail.

```
root[] my_objptr->Solve();
```

Cling no longer knows what `my_objptr` is. This is a great example of a memory leak; the heap object exists but we have lost our way to access it. In general, this is not a problem. If any object will outlive the compound statement in which it was created then a more permanent pointer will point to it, which frequently is part of another heap object. See Resetting the Interpreter Environment in the chapter "Cling the C++ Interpreter".