C++ Programming for Physicists

W. H. Bell

The C++ language is introduced through a series of example programs relevant to high energy physicists. The course introduces basic syntax, object orientated programming, the Standard Template Library, interfacing with FORTRAN and high energy packages HepMC, HepPDT, and ROOT. Programming skills and design proccesses are introduced within the discussion of the example programs. The understanding of C++ programming concepts is tested though set problems, for which associated solutions are provided.

Contents

1 Introduction **3**

2 C++ Syntax **3**
2.1 A First Program 3
2.2 Loops, Conditional Statements and Functions 4
2.3 Pointers and Arrays 6
2.4 Basic File Streams 7
2.5 Problems 10

3 Object Orientated Programming **12**
3.1 Implementing Objects 12
3.2 Object-Object Communication 14
3.3 Operator Overloading 15
3.4 Inheritance 16
3.5 Polymorphism 16
3.6 Interfaces 19
3.7 Templates 20
3.8 Problems 21

4 The Standard Template Library **23**
4.1 Complex Numbers 23
4.2 Vectors 24
4.3 Iterators 24
4.4 Alogrithms 25
4.5 Strings 25
4.6 String Streams 26
4.7 Stream Formatting 26
4.8 Problems 26

5 Particle Physics Applications **26**
5.1 C++ and FORTRAN 77 26
5.2 HepMC 29
5.3 ROOT 30
5.4 Problems 33

1 Introduction

C++ is used for a large number of applications within industry and Particle Physics research. The language provides a large amount of functionality and is still being extended. This course focuses on core aspects of C++ and expects the reader with consult reading materials to extend this introduction. The recommended reference materials for this course are:

- "Ivor Horton's Beginning C++", Ivor Horton, Apress, ISBN 1590592271
- "The C++ Programming Language", Bjarne Stroustrup, Addison-Wesley, 1997

The course is largely based on the ANSI standard and should therefore compile with any standard C++ compiler. Since most Particle Physics applications are build on LINUX or OSX, instructions to compile on LINUX or OSX are provided.

2 C++ Syntax

This section discusses basic data types and simple C++ syntax. At the end of this section students should be able to write simple programs, but will not have been introduced to objects.

2.1 A First Program

Programming languages are commonly introduced by writing a program to print a string to the standard output. The standard output is the terminal window or screen. Therefore information printed to the standard output will appear on the terminal window or screen. Using just C++ syntax this is simply demonstrated by example 1:

```
/* W. H. Bell
** A very simple C++ program to print one line to the standard out
*/

#include <iostream>

using namespace std;

int main() {
  cout << "In the beginning..." << endl;
  return 0;
}
```

The execution of every program starts from a `main()` function. From this function other functions can be called. The return type of `main()` is given by the `int` prefix. Within the LINUX/UNIX environment the operating system expects a program to return an exit condition. The value of the return statement from the `main()` function is collected by the operating system and is available to a user to query. For example at a LINUX shell prompt '`]$`' one could type:

```
]$ ./InTheBeginning.exe
In the beginning...

]$ echo $?
0
```

where `$?` contains the return value of the last command.

The contents of the first example's `main()` function are delimited by the brackets { }, which represent a compound statement. Inside this compound statement there may be several statements each terminated by a '`;`' character, together with other compound statements. In this example the `main()` function only contains two statements: one to print a string to the standard output and one to return the exit value to the operating system. The first of these statements prints a string to the screen by calling the standard output stream function `cout` with the insertion operators `<<`. This operator can be used to concatenate strings. In the given example the end of line character `endl`, is appended to the string `"In the beginning..."`. At the top of the example, the declaration of the `cout` function is included by including the header file `iostream`. When this program is compiled the compiler reads the pre-declaration of `cout` from the header file and leaves a call in the machine code to be resolved at link time.

Finally in this first program one of the two comment styles is introduced. Comments in C++ can be entered in two ways: `/* */` can be used to surround the comment area or `//` can be used at the start of each comment line.

2.2 Loops, Conditional Statements and Functions

Functions The syntax of loops, conditional statements and functions are demonstrated by example 2. This example contains an `int main()` function: the same as seen within the previous example. Within this `main()` function three functions are called: `numFingers`, `pickColour`, and `quitTime`. Each function is pre-declared before the `main()` function. Each pre-declaration is a statement where the return type, and input parameter types must be given. The `void` type simply means that no input parameter or return value is expected. All functions must be either predeclared or declared before they are used. There are three pre-declaration statements before the `main()` function:

```
void numFingers(void);
```

```
void pickColour(void);
bool quitTime(void);
```

where the implementation of these functions is given after the `main()` function. Following the same syntax as the `main()` function the implementation of each of these three functions has a return type, a series of input types, and a compound statement enclosing the function contents. If any input parameters are present then their names must be given in functions implementation. When a function is called the input variables are allocated in memory and are initialised with the values passed into the function. If the implementation is not given in either the code to be compiled, or libraries to be linked against, a linker error will result.

Conditional Statements Conditional logic is essential for the control of both loops and selection statements. Most common of the the selection statements are: `if`, `if else`, `else` and `switch` statements.[1]

Example 2 provides a demonstration of the syntax of `if`, `if else`, `else` constructs: `if`, `if else`, `else` syntax is implemented within the `numFingers` and `quitTime` functions of this example. The evaluation of an `if` statement follows simply: if the logic within in the `()` brackets is true then the following compound statement is executed. `if`, `if else`, `else` statements operate sequentially such that each piece of logic is tested in turn. If all the logic tests fail then the statement following `else` is executed.

In some cases where simple sorting is needed a `switch` statement is a better choice than a long `if`, `if else`... `else` statement: it provides a simple construct which executes quickly. An example `switch` statement is given in the `pickColour` function of example 2. While faster than an `if`, `if else`, `else` statement in some cases a `switch` statement is limited to the usage of simple cases and therefore the logic allowed can be somewhat restrictive.

Loops Several types of loops are available to C++ programmers: there are `while`, `do while` and `for` loops. Each of these loops continue to loop while a condition is true. All the logic available within an `if` conditional statement is also available within the conditional test of a loop statement. Instances of these loop types can be found within many of the examples provided in this course. To begin with a simple example of a `do while` loop can be found in the `main()` function of example 2. This loop continues while the boolean evaluated within the `while( );` is true. This remains true until the function `quitTime` returns a true, where: `while` loop tests on NOT `quitTime` return value.

[1] There are other ways of constructing conditional statements but they are not covered in this course.

2.3 Pointers and Arrays

Many languages e.g. Fortran and Java use pointers implicitly. In C++ pointers are used explicitly. This section introduces the concept of a pointer and demonstrates two basic implementations.

Pointers are called pointers because they point to a memory address. A pointer can be used to access the memory address to which it points or the value contained within the memory address. Example 3 introduces pointers. Looking at example 3 there are two distinct parts to the program: the call to the function `fun` and the indexing of array `v[]`.

Functions and Pointers The function `fun` is declared as

```
void fun(int, int *);
```

with input parameter types `int` and `int *`. The second input parameter is a pointer. When the function is called the memory address of `p`, `&p` is assigned to the pointer declared in `fun`. The importance of using a pointer in this fashion can be seen from running the program. After calling `fun` the value of `np` is the value which it contained before the function call, while the value contained in `p` is the value assigned via the pointer in `fun`. Stepping through the program this can be explained. Both `np` and `p` are initialised with the value one.

```
int np = 1, p = 1;
```

At the point of initialisation an `int` sized block of memory is allocated to `np` and `p`. Then the function `fun` is called with the value of `np` (default in C++) and the address of `p`. Within the function `fun` a new block of memory is allocated for the local variable `np` distinct from the variable contained in the `main()` function. This memory is given the value from the parameter `np` contained within `main()` . The value 2 is assigned to the local variable `np` and as the function exits, the memory of the local variable `np` is deleted. Therefore the value is never set within the `main()` function.

Unlike `np` the value of the variable `p` declared within the `main()` is set by using a pointer. The pointer is initialised with the memory address of the variable `p` contained within the `main()` function. Then the memory address pointed to by the pointer `*p` is assigned the value 2. Therefore when returning to the `main()` function the value contained in the memory of `p` is still 2.

Arrays and Pointers An array of type t is a series of memory blocks of which the size is fixed by the type t. Each element of the array behaves as a variable of type t.

In example 3 an array `v` is declared with four elements:

```
int v[] = {1,2,3,4};
```

This code is equivalent in function to:

```
int v[4];
for(int i=0;i<4;i++) v[i]=i;
```

The size of the array within example 3 is determined by the number of elements within the brackets {}. After the array has been declared the address of the first element is assigned to the pointer `*pv`. It is important to note that the assignment of an address to a pointer at declaration has different syntax to any following assignment or operation on the pointers address in general. The declaration of `*pv` and its assignment of the address of the first element of the array `v` could alternatively be written as:

```
int *pv;
pv=&v[0];
```

Once the pointer has been assigned the memory address of the first element of the array `v` it can be used to access the elements as demonstrated in the example.

2.4 Basic File Streams

File stream functionality is included within a program by including the `fstream` header file. Example 5 demonstrates some input and output stream functionality. The program reads some text from the command line. Then this text is used to determine the file name and if the file is to be written or read. Depending on the command line input the `main()` function calls one of two functions `fileWrite` or `fileRead`. The file `main.c` does not contain the implementation of `fileWrite` and `fileRead`, but includes the header file `FileIO.hh`: containing their pre-declaration. This program cannot be linked into an executable without the implementation of the `fileWrite` and `fileRead` functions being made available at link time. `fileWrite` and `fileRead` are in fact implemented in the file `FileIO.cc`, which is compiled and then linked with `main.o`.

The function `fileWrite` opens an output stream, using the file name supplied.

```
ofstream file(filename);
```

Then this file output stream is used in exactly the same way as the standard output stream in the previous examples. Finally at the end of the file operation the stream is closed.

```
file.close();
```

Closing the file output stream is essential to flush the data stored in the output memory to the file. (Flushing is not always implicit to a file stream being closed.)

The function `fileRead` uses an input file stream to read the data from the specified file name. It is initialised in the similar way to the file output stream.

```
ifstream file(filename);
```

If for some reason the file opening operation fails the file input stream variable `file` will be assigned 0. Any integer number that is not 0 is considered logically as true. 0 is considered logically as false and hence the usage of the `if` statement.

```
if(!file) {
  cerr << "Error: could not open " << filename << endl;
}
```

The design of example 5 is illustrated in figure 1 and 2.

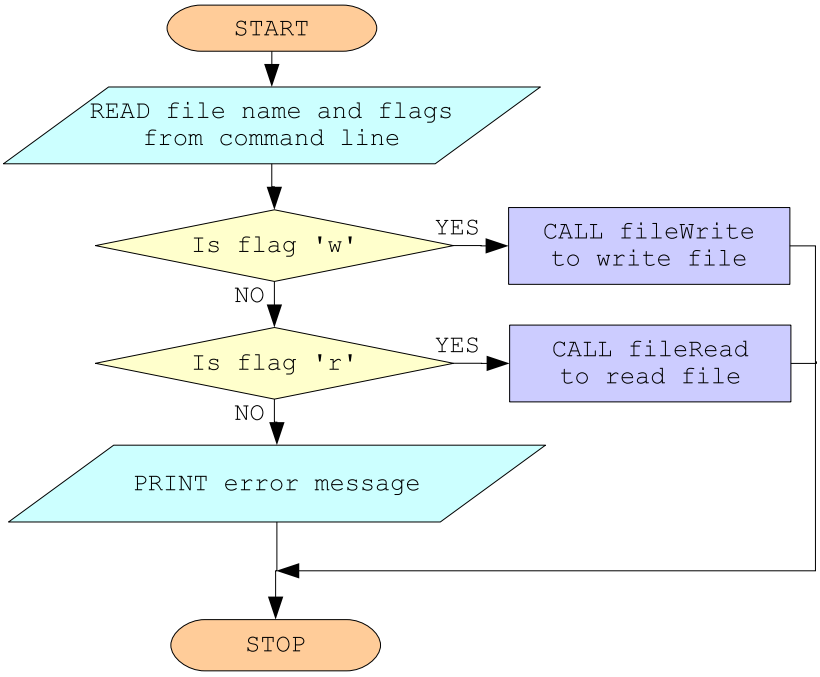

Figure 1: A flowchart describing example 5 in general terms.

```
main()
  Get command line arguments:
    Require 2 or 3 arguments:
      First additional argument - file name
      Second additional argument - read/write flag.
  Use the value of the read/write flag to call either
  fileRead() or fileWrite().

fileRead()
  Open an input stream.
  Read the contents of the file until the EOF.
    Stream each character number into a integer.
    Print these integers in the format of the file.
  Close the input stream.

fileWrite()
  Open an output stream.
  Loop from the numbers 1 to 20.
    Write out each number followed by a space to the
    active file stream.
    After each fifth number is written append a new line.
  Close the output stream.
```

Figure 2: Example 5 in pseudocode.

2.5 Problems

Reading a Configuration File Write a program to read in the table of particle data contained in the file `problem_01/particle.dat`. Read each column of data into a separate two dimensional `char` array. Then use these arrays to print each column of `particle.dat` to the screen.

1. Start by opening a file as demonstrated in example 5. Then use the input stream function `getline`[2] to read lines from the file.

```
char str[MAX_LINE_LENGTH];
int lineLength;
...
...
  while(!file.eof()) {
    file.getline(str,MAX_LINE_LENGTH);
    lineLength = strlen(str);
```

2. Skip any lines that begin with a comment character:

```
    if(str[0]!="#") { // If character isn't #
    ...
    }
```

3. And parse each column of the file, skipping spaces as necessary.

The function `strcpy` from the `cstring` header file should be helpful for copying a section of a `C` string. The string terminator '\0' can be used to terminate a `C` string: where the maximum length is determined by the size of the character array but the minimum length is set by the position of the string terminator within the array.

File Encryption Write a program to 'encrypt' text files by using a binary mask e.g.:

```
int mask = 0xA3;  // A number less than 255
char c;           // The character to be read and encrypted/decrypted.
```

- Read one byte from the standard input, encrypt it and print the result to the standard output, e.g.:

```
while((c=std::cin.get())!=EOF) { // Get character until end of file.
  //Replace this line with a bitwise operation.
  std::cout << c; //output character.
}
```

[2]The first argument is a character array, the second argument is the length of the array and the return value is the length of the string

- Use one bitwise exclusive `OR` operation to encrypt and another to decrypt e.g.:
  `a = a ^ mask`
  (Two exclusive `OR` operations cancel.)
- Having written the encryption program, check the executable works by using the command line syntax:

  ```
  encrypt.exe < inputfile > outputfile
  ```

EPS File Extraction Many postscript files contain embedded eps files which can be extracted and saved in a separate file. Write a program to parse the file `problem_03/document.ps`, saving any instances of

```
%%BeginDocument:
...
%%EndDocument
```

to files of appropriate names.

1. Open an input file stream:

   ```
   std::ifstream file(argv[1]);

     if(!file) {
       std::cerr << "Error: could not open " << argv[1] << std::endl;
     }
     else {
      ...
     }
   ```

2. Read single lines from `problem_03/document.ps`:

   ```
     while(!file.eof()) { // While not end of file
       file.getline(line,MAX_LINE_LENGTH);  // Read one line.
       ...
     }
   ```

3. Use the functions `strstr` and `strlen` from `cstring` to catch included documents:

   ```
   char beginDocument[]="%%BeginDocument: "; // declare C string
   char endDocument[]="%%EndDocument"; // declare C string
   ...
   ...
       if((filename=strstr(line,beginDocument))!=NULL) { // If begin document
         filename += strlen(beginDocument);  // File name follows %%BeginDocument:
         outputFile =  new std::ofstream(filename); // Open output file
       }
   ```

where `filename` is of type `char*`.

4. Save a copy of the information between and including the `%%BeginDocument` and `%%EndDocument` statements to a separate file.

3 Object Orientated Programming

For many years developers have worked with languages such as FORTRAN and `C`. These languages allow developers to write functions and complicated data blocks. While suitable for many applications large programs written with these languages quickly become un-wieldy. Unlike `C`, C++ allows object orientated programming, offering two improvements: conceptual building blocks, and code re-use. Carefully designed C++ programs should therefore be easier to understand and contain fewer lines than an equivalent `C` program.

Before an object can be created a definition of its contents needs to be written. The definition of an object is called a `class` and can be thought of as an extended type. For example, a variable of type `int` can be defined by:

```
int i;
```

where as a object of `ClassName class` is instantiated by

```
ClassName obj;
```

Each `class` definition can contain data types and function methods.

3.1 Implementing Objects

The principle of implementing a basic object is outlined in example 7. This example is composed of three source files: `main.cc`, `BasicParticle.hh` and `BasicParticle.cc`. The example contains a single class definition of a class called `BasicParticle`, which is written in the `BasicParticle.hh` header file:

```
class BasicParticle {
...
};
```

Inside this class declaration there are `public` member functions, `private` member functions and `private` data members. The member functions are implemented in `BasicParticle.cc`.

Constructors Example 7 has a `main()` function contained within the `main.cc` file. Within the `main()` function two instances of the `BasicParticle` class are instantiated:

```
BasicParticle particle1(fourvector1);
BasicParticle particle2(fourvector2);
```

where each one of these lines calls a constructor of the `BasicParticle` class,

```
class BasicParticle {
public:
  BasicParticle(double *fourvector);
```

which is implemented in the `BasicParticle.cc` source file:

```
BasicParticle::BasicParticle(double *fourvector)
{
  assignFourVector(fourvector);
}
```

When an object is instantiated the constructor is called: defining an object or instance of the class in memory. Once an object has been created, member functions of the object can be called. (It is not possible to call the member functions of a `class`, except in the case where the member functions are `static`[3].)

Member Functions and Data Members Following the instantiation of an object of the `BasicParticle` class within example 7, two methods of the class are called:

```
cout << "Mass of particle 1=" << particle1.getMass() << endl;
cout << "pt of particle 1=" << particle1.getPt() << endl << endl;
```

These methods can be called in this way because they are declared as `public` in the header file. Looking in the `BasicParticle.hh` header file there are two `private` member functions:

```
private:
  void calculatePt();
  void calculateMass();
```

These member functions can only be called by member functions of objects which are instantiated from this class definition. Within the given example these member functions

[3]The use of Static will be covered briefly in later examples.

are called by the `assignFourVector` function when a new four vector is assigned to an object of `BasicParticle` type. The object then calculates the p_t and mass and assigns these values to `m_pt` and `m_mass` respectively. `m_pt` and `m_mass` are `private` data members of the class `BasicParticle` the rules governing access to these data members are the same as those affecting private member functions.

Data members of a given class can be accessed by all member functions within the class in the same manner as a global variable would be. `public` data members can also be accessed by any other function outside the class in a similar manner to a `public` method.

Compilation and Header Files When each source file is compiled classes must not be included more than once. When including a header file containing a class definition it may already be included via including another header file. To prevent a double declaration, causing compilation errors, precompiler case clauses should be used.

```
#ifndef CLASS_NAME_HH
#define CLASS_NAME_HH

... class declaration ...

#endif
```

3.2 Object-Object Communication

In a program several objects may have to interact with each other: each object calling the member functions of another. To access the data stored within an object a call should be made to the particular instantiation of the class containing the data. Different objects may contain different data. Communication between objects therefore needs to be handled carefully.

There are two common situations where an object needs to communicate with and other object:

1. An object creates another object and then needs to access data within the created object.
2. An object is created by another object and then needs to access data within the object that created it.

The second example situation can cause trouble. The problem is that the created object needs to access the parent object and not just another object of the parents class. Example 8 demonstrates both of these basic communication actions.

Following the execution of example 8, the `main()` function creates a `Parent` object and assigns the address to the pointer `*parent`. The `run` method of the object of `Parent`

type is then called.

```
Parent *parent = new Parent(id, mass);
parent->run();
```

The `run` method creates a new `Child` object and calls its `run` method. The `Child` class constructor takes one argument: a pointer of `Parent` type. To allow the `Child` object to call the methods of this instantiation of the class `Parent` a pointer to this class must be given to the `Child` object. This is done by using the `this` pointer as shown. The `this` pointer points to this instantiation of the object.

Within the class `Child` the pointer to the object of `Parent` type is stored in a `private` data member and then is used in the `run()` method to call the parent objects methods.

```
void Child::run() {
  cout << "parent mass = " << m_parent->getMass() << endl;
  cout << "parent id = " << m_parent->getId() << endl;
}
```

3.3 Operator Overloading

With basic types such as `int` and `float` arithmetic can be carried out with operators such as `*` and `+`:

```
float x=4, y=5, z;
z=x*y;
```

Arithmetic and other operators can be defined within a class. This is called operator overloading. An implementation of operator overloading is given in example 9. This example uses the same `BasicParticle` class from example 7, but includes implementation for the `+` operator.

```
BasicParticle BasicParticle::operator+(BasicParticle particle) {
...
}
```

This operator is defined so that when adding objects of `BasicParticle` type another object is created and returned which contains the fourvector resultant of the two input `BasicParticle` objects. The addition of the two objects is called from the `main()` function.

```
BasicParticle *particle1 = new BasicParticle(fourvector1);
BasicParticle *particle2 = new BasicParticle(fourvector2);
BasicParticle particle3 = *particle1 + (*particle2);
```

Note that the brackets round the second pointer argument are necessary to separate the pointer syntax from the arithmetic operator.

3.4 Inheritance

One class can inherit data members and member functions from another. In example 10 there are set of three classes `Bag`, `ColouredBag`, and `BeanBag`. Each class is made slightly more complex than the last by inheriting features from the previous one. `Bag` is the simplest of these classes and contains only volume information. `ColouredBag` inherits from it as stated in the header file `ColouredBag.hh`

```
class ColouredBag: public Bag {
...
}
```

By inheritance any `ColouredBag` object has access to the methods contained in the `Bag` class as demonstrated within the `main()` function.

```
ColouredBag colouredBag;
colouredBag.setVolume(40.0);
```

Data members are inherited in the same way. The behaviour of the members depends on the type of the base class: `public` , `private` or `protected` . If the base class is `public` then the `protected` and `public` members become `protected` and `public` members of the derived class. The class `BeanBag` takes advantage of this property to directly set the value of `m_bagColour` directly:

```
BeanBag::BeanBag(char colour) {
  m_bagColour = colour;
}
```

where `m_bagColour` is a `protected` member of the class `ColouredBag`. If the base class is `protected` or `private` then the `public` and `protected` members become both `protected` or `private` according to the base class type. All `private` methods of the base class are not accessible by the derived class for all base class types.

3.5 Polymorphism

Polymorphism is only possible through inheritance. Consider the case of figure 3: a base class which has two member functions, one calling the other. If another class is created that inherits from the base class then it could call one of the `public` or `protected` member functions of the base class. This could in turn call another member

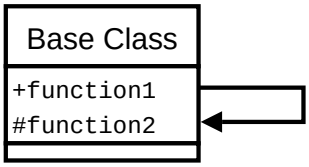

Figure 3: A simple UML diagram of a base class with two member functions, where one calls the other.

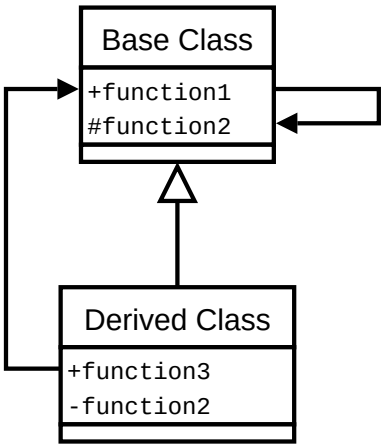

Figure 4: A simple UML diagram of a derived class with two member functions, one of which calls a function in the base class which in turn calls another member function.

function illustrated in figure 4. Now consider that the derived class contains two member functions, where one of which has the same name and parameter types as the member function called by the base class. If the derived class calls the base class then without polymorphism the member function of the base class calls the function within the base class. Unsurprisingly the base class method does not call the member function within the derived class.

By introducing polymorphism it is possible to select which of the two member functions is called: the one within the base class or the one within the derived class. It is therefore possible to write a program that functions illustrated in figure 5. Implementations of programs illustrated in figures 4 and 5 are given in example 11.

Within example 11 a base class `BasicParticle` and a derived class `SmearedParticle` are implemented. There are two forms of the example code given: with polymorphism and without polymorphism. There is only one difference between the two directories: in the `with/` directory the method `calculateMass` is declared as virtual void, where as in the `without/` directory the method is simply declared as `void`.

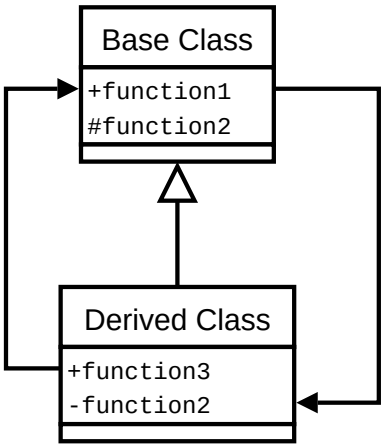

Figure 5: A simple UML diagram of a derived class with two member functions, one of which calls a function in the base class which by polymorphism in turn calls a member function within the derived class.

Without Polymorphism In the `without/` directory the `SmearedParticle` class inherits from the `BasicParticle` base class. In the `main()` function an instance of the `SmearedParticle` class is created and the `calculateMass` function is called by the `assignFourVector` member function. The value of the mass is then printed.

```
int main() {
  double fourvector1[4] = {3.0, 4.0, 5.0, 7.35};
  ...
  SmearedParticle *smearedParticle = new SmearedParticle(fourvector1);
  ...
  cout << "smearedParticle mass = " << smearedParticle->getMass() <<  endl;
```

The value returned from the `getMass` member function of the `SmearedParticle` object is the same as that returned from the one called from the `BasicParticle` object. What is happening is that the member function `assignFourVector` is calling the `calculateMass` function within the `BasicParticle` class and not the `SmearedParticle` class as required. To get around this problem polymorphism can be used.

With Polymorphism In the `with/` directory the member function `calculateMass` is defined as virtual within the header file `BasicParticle.hh`.

```
class BasicParticle {
...
protected:
```

```
...
  virtual void calculateMass();
};
```

The effect of this is that when the `assignFourVector` member function is called from the `BasicParticle` object then the method `calculateMass` within the class `BasicParticle` is called, but when the `assignFourVector` method is called via the `SmearedParticle` class the method `calculateMass` within the class `SmearedParticle` is called.

For polymorphism to work the member function in the base class must be `virtual` and identical in its parameter types to the member function declared in the derived class. (It is good pratice to declare the methods within the derived class that use polymorphism as virtual too.)

3.6 Interfaces

Interfaces are abstract classes that contain only pure virtual member functions. A pure virtual member function is defined by declaring a virtual member function to be equal to zero. For instance, in example 12 an interface containing a single pure virtual member function is defined in the header file `IDataRecord.hh`:

```
class IDataRecord {
 public:
  virtual int appendRow(int *rowData) = 0;
};
```

where `IDataRecord` is the class name of this interface. Any class that contains one or more pure virtual functions is defined as being abstract and cannot be instantiated. Furthermore any pure virtual member function defined within a class should not implemented within the class but rather must be implemented by a non-abstract derived class. Any class that is derived from an abstract class but does not implement the inherited pure virtual functions will also be an abstract class. Notice therefore an interface can be used to require a set of member functions to be present in a class that is derived from it.

In example 12 classes `AsciiRecord` and `BinaryRecord` inherit from the interface `IDataRecord`. Both classes `AsciiRecord` and `BinaryRecord` implement the pure virtual member function of `IDataRecord` and are therefore not abstract. Again within this example the convention is used that the derived classes implementation of the virtual function is declared as virtual. While this is not necessary for functionality it allows anyone reading the code to quickly see which member functions are affected by polymorphism.

When example 12 executes a pointer of `IDataRecord` type is assigned the address of either a `AsciiRecord` or `BinaryRecord` object. As within the previous example the type

of object the pointer points to implies which virtual member function is called. Having created a pointer of `IDataRecord` type the pointer is then passed to a function:

```
void fillRecord(IDataRecord *record) {
  int arr[] = {1,2,3,4,5,6,7,8,9,10};
  record->appendRow(arr);
}
```

This function calls the member function `appendRow` defined within the `IDataRecord` class. Following polymorphism this method call will actually call the `appendRow` method of `AsciiRecord` or `BinaryRecord` depending on the type of object `IDataRecord` points to. The example therefore illustrates that functions and classes can be written to perform operations on interfaces rather than on every given class type within an inheritance structure.

3.7 Templates

Templates can be used in conjunction with classes or functions. Within this course only class templates will be considered. A class template is a class where one or more parameters have a template type. In the case where similar functionality is needed to operate over different types a templated class can be used to generate the needed code. Instead of writing several classes with different types within the constructors and the member functions, one template can be written. For each usage of the class template the compiler generated the needed extra class definitions.

Example 13 contains a simple class template called `Array`. This class template contains an array of type `T`. The type `T` is determined when the template is instantiated and must be one of those listed at the bottom of the `Array.hh` header file.

```
template class Array<char>;
template class Array<int>;
template class Array<float>;
template class Array<double>;
```

Within the `main()` function there are two instantiations of the `Array` templated class.

```
int main() {
  ...
  Array<int> arrayInt(N);
  Array<double> arrayDouble(N);
  ...
```

where the type `T` is given inside the `<>` brackets. Then in the code that follows the two methods `setElement` and `getElement` are called from both template generated

objects. The parameter types of these member functions depends on the type specified in the instantiation. The files `Array.hh` and `Array.cc` demonstrate how to declare and implement templated constructors, member functions and data members of a class template.

3.8 Problems

Histogramming Starting from the files provided in the `problem_04` directory, complete the program by writing a class called `Histogram`. Then follow the instructions in the `README` file to build and test the final program.

- The `Histogram` class should include a constructor:

  ```
  Histogram(char *filename, int nbins, float min, float max);
  ```

  and two member functions:

  ```
  void saveHisto(void); // Save the histogram to file
  book(float value, float weight);  // Book an entry into the histogram.
  ```

  where `filename` is the output filename, `nbins` is the number of bins, `min` and `max` are the limits, `value` is the value to be booked and `weight` is the weight to give the entry.

- Create `private` data members to store the number of bins, the entries in each bin, the limits of the histogram and the file name.

- The `saveHisto` member function should write the contents of the histogram to a text file, where each row of the file contains a bin centre and then a number of entries.

Extended Array Starting from the files provided in the `problem_05` directory, complete the program by creating a class called `DataContainer`.

- The `DataContainer` class should store an array and its size as `private` data members:

  ```
  DATA_TYPE *m_array;
  int m_size;
  ```

  where `DATA_TYPE` should be set via a `#define` statement in the class' header file:

  ```
  #define DATA_TYPE float
  ```

- Provide a constructor of the form:

  ```
  DataContainer::DataContainer(DATA_TYPE *array, int size)
  {
  ```

```
  m_size = size;
  m_array = new DATA_TYPE[_size];
...
```

- Write member functions to perform arithmetic operators for `*`, `+`, and `/`, which create a new class containing array elements of the form:

$$z_i = x_i * y_i \ , \ z_i = x_i + y_i \ , \ z_i = x_i / y_i$$

- Write a `printArray` member function to print the contents of the `m_array` private data member.

Inheritance Starting from the files provided in the `problem_06` directory, complete the program by writing three classes: `Particle`, `DetParticle` and `CalParticle`.

- Using inheritance, provide constructors of the form:

```
Particle (int id, double *p3vec);
DetParticle(int id, double *p3vec, int trackId);
CalParticle(int id, double mass, double *p3vec, double eCal);
```

  where `p3vec` is an array of three elements.

- Store the data passed into each class in `private` data members.
- Provide `public` member functions to access the data members of each class.

Transformation Interface Starting from the files provided in the `problem_07` directory, complete the program by writing: (i) an interface class that has member functions to perform a transformation and a reverse transformation, (ii) a class implementation of the Transformation interface to perform rotations in two dimensions.

- Create an interface class `ITranslation` that contains the virtual member functions from `OffsetTranslation` as pure virtual functions.
- Create a class called `RotationTranslation` that inherits from an interface class `ITranslation`.
  - Refer to OffsetTranslation for hints on how `RotationTranslation` might be implemented.
  - A rotation in two dimensions can be calculated via:

$$x' = xcos\theta - ysin\theta$$

$$y' = ycos\theta + xsin\theta$$

    where θ is the angle of rotation, x is the original x and x' is the rotated one.

Track Container Starting from the files provided in the `problem_08` directory, complete the program by writing a class template called `TrackContainer`. `TrackContainer` should allow `float` and `double` template types to be instantiated. The class should also contain public data members: p_t, $\cos(\theta)$, phi_0, d_0 and z_0, where their type is set by the template's instantiation.

4 The Standard Template Library

The Standard Template Library (STL) contains templates which are: containers, and algorithms to operate on these containers. Within the ISO/ANSI standard there are serveral containers:

Sequential Containers	<deque> <list> <queue> <stack> <vector>
Associative Containers	<map> <set>

In addition to these containers STL includes `numeric` algorithms, generic `algorithms` and `complex` class templates. The following subsections introduce `complex`, `vector`, `list`, `iterators` and `algorithms`. More information on STL in general and areas not covered in this course can be found in the recommended text books.

4.1 Complex Numbers

Complex numbers are implemented in a `complex` class. Example 14 introduces some of the `complex` operations available. At the top of the example 14 the `complex` header is included, making the `complex` classes and global methods accessible. The allowed templated types for a complex number are: `float`, `double`, and `long double`. In the example two of the three `complex` class types are implemented:

```
std::complex<float> complexFloat(3,4);
std::complex<double> complexDouble(-1,0);
```

The `complex` class is part of the `std` namespace. Therefore the class name must be used either with the `std::` specifier explicitly or by quoting `using namespace std`. The constructor comes in three forms: including real and imaginary parts as in the example, the copy constructor and the implicit default constructor. The type of the variables passed into the constructor must match the type given in the template instantiation. For example `std::complex<float>` would mean that `complex(float, float)` is the valid constructor.

After the instantiation of the `complex` objects the rest of example 14 demonstrates some of the functionality available via the inclusion of `<complex>`.

4.2 Vectors

For a physicist STL `vectors` are probably the most useful of the STL container classes. Basically a `vector` can be thought of as an array with extra functionality. Unlike an array `vectors` do not have a fixed size and elements can be added and removed as necessary. Beyond this the class provides other advanced functionality. Example 15 demonstrates simple `vector` usage. Within the `main()` function one `vector` of `int` type is created.

```
int main() {
  std::vector<int> intVector;
....
```

Elements are then added to the vector using the `push_back` method. As the vector increases in size the size is printed out. Then the length of the vector is reduced by calling the member function `pop_back`, which pops elements off the end of the vector. Before the elements are popped off the back the value of the element is printed retreived by calling the method `back`.

4.3 Iterators

Iterators provide access to the different elements of the data container classes. Iterators relate in a similar way to the container classes as the pointer did to the array in section 2. For an iterator to step over the correct amount of memory the iterator must be initialised from a data container of the same type as it will be used with. Iterators are introduced in example 16. Contained within this example a `list` of `char` type is initialised with a string. Then an iterator of the corresponding container type is created and assigned with the address of the first element.

```
list<char>::iterator itr;
itr = charList.begin();
```

Once the iterator has been initialised with the starting address it is then used to iterate over all of the elements of the `list`. The value of each element of the list is printed,

```
cout << *itr << " ";
```

and the address stored in the iterator is moved on to the next element.

```
itr++;
```

4.4 Alogrithms

STL provides a number of powerful algorithms which use iterators to operate on data containers. Example 17 introduces the sort algorithm. Algorithm functionality is accessible via the inclusion of the header file `<algorithm>`. To use algorithms interators of type according to the data container should be created. Within example 17 a vector of `int` type initialised with a random jumble of numbers. Two iterators of the same type are then created.

```
std::vector<int>::iterator first;
std::vector<int>::iterator last;
```

These iterators are then assigned the memory addresses of the first and last memory element of the vector.

```
first = numbers.begin();
last = numbers.end();
```

The iterators are then passed to the sort algorithm.

```
std::sort(first,last);
```

Since the algorithm methods are part of the `std` namespace the specifier is used. This is just one of the methods available. Look at one of the text books or in the header file for the other methods.

4.5 Strings

While many useful programs can be written with `cstring`s the `string` class provides extra functionality and more flexibility than a simple `cstring` character array. The `string` class is part of the Standard Template Library (STL) and inherits many useful features from container base classes. In general terms the `string` class is a container which contains an array of characters. Unlike a `cstring` objects of the `string` class have dynamic size and the memory allocated for the storage of characters can be reduced or increased as needed. Where strings need to grow quickly it is also possible to allocate sections of memory such that any appending operation does not necessarily trigger additional memory operations.

Example 18 demonstrates some of the functionality of the `string` class. While this example implements many of the `string` class member functions it does not use `iterators`. `iterators` are defined within the `string` class and can be used in conjunction with some of its member functions as well as the STL algorithms.

4.6 String Streams

A string stream is a stream connected to a `string` object. Such a stream allows objects or simple variables to be inserted and extracted from a string using stream syntax. While C functions like `sscanf` are still available within C++ , string streams provide a useful type safe means of converting variables to and from strings. Example 19 demonstrates the type safe nature of the stream operators by using an input string stream to convert a string into both an `int` and a `double`.

4.7 Stream Formatting

When writing numbers into a stream it may be necessary to format the numbers to have a particular precision or width. Example 20 demonstrates some C++ stream formatting options. Again the advantage of using C++ streams over C functions like `printf` is that C++ streams are type safe.

4.8 Problems

Algorithms Create a vector object and fill it with sequential numbers from 0 to 100. Then use the random_shuffle algorithm to shuffle the elements.

```
std::vector<int> numbers;
...
first = numbers.begin();
last = numbers.end();
std::random_shuffle(first, last);
```

Print the elements out before and after the random shuffling. Then find the position of the number 7 within the vector.

5 Particle Physics Applications

5.1 C++ and FORTRAN 77

C++ developers writing physics analysis programs may want to access efficient and thoroughly tested FORTRAN algorithms. Thankfully C++ and FORTRAN 77 can be easily linked togther. This course discusses how to link FORTRAN 77 programs compiled with the `g77` compiler to C++ programs compiled the `g++` compiler. While it may be possible to link to programs compiled with other FORTRAN compilers it should be noted that the symbols produced from other such compilers may well not follow the rules given within this course.

Example 21 demonstrates all of the nuances of connecting FORTRAN 77 and C++ together. This examples fills a FORTRAN common block using a C++ function, prints the contents of the common using a FORTRAN function and then calls a FORTRAN function which in turn calls a C++ function. The `main.cc` file includes `fortran.hh`. This header file contains the declaration of an external struct `forcom_`, together with the declaration of two functions implemented in FORTRAN

```
void commons_(void);
float call_back__(float *,char *, int);
```

and a function implemented in C++

```
float mult_a__(float *);
```

When example 21 executes the `fillCommon` function is called to fill the FORTRAN common block `FORCOM` via the C++ external `struct forcom_`. The syntax for accessing the `struct` data members is exactly the same as that used for accessing public data members of an object. The struct provides access to the FORTRAN common because it has the same name within the compiled object file and is declared to be external. The `extern` prefix causes the compiler not to define an additional memory block for the struct but instead link the FORTRAN definition of the memory block to the C++ one. Without the `extern` prefix the struct would occupy a different memory location and therefore not provide access to the FORTRAN common block. While the mapping of the external struct to the common block is possible via choosing the correct name it will not function properly unless the order, type and size of the variables declared in the struct match those declared in the FORTRAN common block. Notice however that the declaration of the C++ struct is not entirely the same as the FORTRAN common, for example:

```
int arr[2][3][4]
float farr[5][6]
```

has the same memory mapping as

```
INTEGER ARR(4,3,2)
REAL FARR(6,5)
```

Notice that the outer array indices commute. The last feature to common block mapping is that FORTRAN strings do not contain string terminators and therefore can contain a string as long as the number of elements in the character array.

Once the common block has been filled the

```
commons_();
```

function is called to print the values contained within the common block. The function

```
void commons_(void);
```

is a declaration of the Fortan `SUBROUTINE`:

```
SUBROUTINE COMMONS
```

When this `SUBROUTINE` is compiled the `g77` compiler creates an object containing a symbol of the form:

```
void commons_(void);
```

If a function has an underscore in its name the `g77` compiler will add a second underscore at the end of the name. This is the default behaviour of the `g77` compiler which can be altered by using different compiler flags.

The next thing the `main()` function does is to call

```
call_back__(&a,name,size);
```

which is a call to the FORTRAN `FUNCTION`

```
FUNCTION CALL_BACK(A,NAME)
```

which in turn calls

```
C = MULT_A(A)
```

which is in fact a C++ function

```
float mult_a__(float *a) {
  return (*a)*10.;
}
```

FORTRAN 77 uses pointers implicitly but when FORTRAN functions are called from C++ these pointers must be declared explicitly. While the FORTRAN programmer is not aware of it the `g77` compiler compiles in an extra integer for each string in a SUBROUTINE or FUNCTION call. The purpose of this extra integer is to store the length of the FORTRAN string. For example

```
SUBROUTINE STRINGS(A, B, C)
IMPLICIT NONE
CHARACTER*(*) A, B, C
...
```

becomes

```
void strings_(char *a, char *b, char *c,
              int size_a, int size_b, int size_c);
```

5.2 HepMC

π^0 decay generator Example 22 demonstrates the use of the `HepMC` and `Vector` packages. The `HepMC` package contains classes describing a Monte Carlo event format suitable for simulation programs where as the `Vector` package provides classes describing three and four vectors together with associated transformations.

The `main()` function of example 22 creates a new event container, generatates a π^0 particle, decays the π^0 into two photons, and finally prints the event record to the screen. The source code for this program is divided up into: `MonteCarlo` - containing two static methods to generate a π^0 particle and produce a two body generic decay; `LorentzBoost` - a wrapper class providing a static method to lorentz boost a fourvector via the FORTRAN code contained in `lorentz.for`. When the program runs all of the event data are recorded within the event record `GenEvent`. Once the `GenEvent` instance has been created `GenVertex`s can be added. Each `GenVertex` can connect a number of different `GenParticle`s together.

The `main()` function instantiates a `GenEvent` object and then calls a static member function of the `MonteCarlo` class:

```
HepMC::GenParticle pi0 = MonteCarlo::generate();
```

to create a `GenParticle` that describes a π^0 particle. The `main()` function then calls a second static member function of the `MonteCarlo` class:

```
MonteCarlo::twoBody(evt,&pi0,22,22,0.,0.);
```

to produce a decay of the π^0 into two photons. The `twoBody` member function starts by creating a vertex and then adds the π^0 particle to it

```
HepMC::GenVertex *vert = new HepMC::GenVertex();
vert->add_particle_in(parent);
```

Once the two daughter photons have been produced they are also added to this vertex,

```
vert->add_particle_out(new HepMC::GenParticle(fvecDaughter1,
                                              particleId1,1));
vert->add_particle_out(new HepMC::GenParticle(fvecDaughter2,
                                              particleId2,1));
```

and the vertex is then added to the `GenEvent` event container:

```
evt->add_vertex(vert);
```

The complete decay is then printed in the `main()` function.

5.3 ROOT

ROOT [1] is a data analysis package written in C++ and supported at CERN: tutorials and how-tos and other documentation can be downloaded from the ROOT web site [1]. This course focuses on introducing basic features of ROOT data storage.

Histograms When analysing large statistical samples histograms provide an important means of visualising accumulated results. ROOT provides classes for both one and two dimensional histograms. Example 23 uses ROOT to produce a single one dimensional histogram. Within the `main()` function memory associated with root histograms is initialised,

```
TROOT simple("histos","Histogram Examples");
```

the root file that will contain the histogram is opened, writing over any existing file of the same name by quoting `"RECREATE"`,

```
TFile *rfile = new TFile(argv[1],"RECREATE","Histogram Example");
```

a one dimension histogram is then created,

```
TH1F *histo = new TH1F("histo","Sine Wave",nbinsx,xlow,xup);
```

the histogram is filled,

```
histo->Fill(x,w);
```

then the contents of the root memory including the single one dimensional histogram is written to the ROOT file.

```
rfile->Write();
```

Once the example has been compiled and executed a single ROOT file will be produced. Intructions on how to plot the contents of the ROOT file can be found in the `README` file.

TNuples TNtuples are conceptually a table like structure, where each column is filled with the values of a variable. Once each column has been associated with a variable rows of data can be added. A TNtuple is stored as a ROOT TTree containing only TLeaf objects. Example 24 opens a TFile into which the TNtuple will be saved and then instantiates a TNtuple object containing three columns: x, y, and z.

```
TNtuple *ntuple = new TNtuple("random_dat","Random Data","x:y:z");
```

It is important to note that unlike ROOT histograms TNtuple objects require the presence of an open TFile for them to be created. This open file is required for saving the TNtuple data when the TNtuple's memory buffer is full.

Once the TNtuple has been created ten thousand rows of random number data are written to it by looping over:

```
dat[0] = RandGauss::shoot(&randomEngine); // x (Gaussian)
dat[1] = RandExponential::shoot(&randomEngine); // y (Exponential)
dat[2] = dat[0]*dat[1];                   // z = x*y
ntuple->Fill(dat);
```

Following this any remaining data within the TNtuple's memory buffer is flushed to the TFile via:

```
rfile->Write();
```

In addition to the C++ code example 24 contains a number of root macros that can be used to plot the data stored within the output TFile. The usage of these macros is described in the README file.

TTrees TTrees are flexible data containers. Each TTree can have several TBranches. Data are written in rows to all TBranches of a TTree, but can be read back from a single TBranch or all the attached TBranches at once.

Example 25 demonstrates simple TTree instantiation and serialisation. The example offers two choices: either a TTree is written to a TFile by calling writeTree or it is read for the TFile by calling readTree. The function writeTree starts by opening a TFile by calling the static TFile member function Open. The function then creates a new TTree:

```
TTree *tree = new TTree("tree","test tree");
```

where the arguments are the key name and the label respectively. In a similar manner to a TNtuple the key is added to the open TFile such that the TTree is saved to this file.

Once a `TTree` has been instantiated five branches are added

```
tree->Branch("Run",&run,"Run/I");
tree->Branch("Event",&event,"Event/I");
tree->Branch("x",&x,"x/F");
tree->Branch("y",&y,"y/F");
tree->Branch("z",&z,"z/F");
```

to the `TTree` object. Each of these member function calls causes a new `TBranch` to be instantiated and connected to the parent `TTree` object. The arguments of the `Branch` member function are: the key, the address of an associated variable and the label. When the `TTree` member function

```
tree->Fill();
```

is called the value in the memory at the address assigned to each `TBranch` is saved into the `TTree`. Finally when all the data have been entered into the `TTree` the `TFile` is saved,

```
root_file->Write();
```

flushing any remaining data to disk.

The `readTree` function reads a `TTree` from a `TFile` and prints out each entry from all the `TBranch`es. The `TFile` is opened and then a pointer to the `TTree` is collected by supplying the key name of the `TTree`

```
Tree *tree = (TTree*)root_file->Get("tree");
```

where a cast to a `TTree*` is necessary because the `Get` member function of `TFile`, (inherited from `TDirectory`), returns type `TObject*`. After collecting a pointer to the `TTree`, a pointer to each `TBranch` is collected

```
TBranch *run_branch = tree->GetBranch("Run");
```

and the branches addresses are set to be addresses of variables defined within the scope of the `readTree` function

```
run_branch->SetAddress(&run);
```

Instead of reading each `TBranch` individually data from the `TTree` are copied on mass into the `TBranch` addresses by calling

```
tree->GetEvent(i);
```

If it is necessary to read data from just one `TBranch` the `GetEvent` member function of that branch can be called.

5.4 Problems

Other random number distributions Write a program to histogram the numbers generated with `RandGauss::shoot` and `RandExponential::shoot`.

B mesons generated using PYTHIA Starting from the files provided in the `problem2` directory, complete program by: (i) for each B meson store the contents of the `hepevt_.phep` and `hepevt_.idhep` arrays in a `TNtuple` and (ii) histogram the transverse momentum:

$$p_t = \sqrt{p_x^2 + p_y^2}$$

in the range 0 to 10GeV/c for all B mesons.

Compile and run the completed program to generate an output ROOT file. Write a root macro to plot the p_t distribution between 0 and 15GeV/c for the B_s and $\overline{B_s}$ meson: `idhep` values 531 and -531 respectively. The types of the B meson are given by the identifier code stored in the `hepevt_.idhep` array. The possible values of this identifier code can be found in `problem2/generator.cc`. Information on the contents of the `HEPEVT` common block is given in the `problem2/README.hepevt` file.

References

[1] An Object-Oriented Data Analysis Framework. http://root.cern.ch/.