DEEPAK SAMUEL

LECTURE ON C++ AND ROOT FOR PHYSICISTS

INDIA-BASED NEUTRINO OBSERVATORY

PUBLISHED BY INDIA-BASED NEUTRINO OBSERVATORY

SAMUEL@TIFR.RES.IN

First printing, May 2012

Contents

Introduction

This guide was prepared as an aid for the graduate students of the India-based Neutrino Observatory programme at the Tata Institute of Fundamental Research, Mumbai. I will begin by giving an overview of what happens inside a computer when a program is run and then making you acquainted with the tools like required to write and run a program and then the ways to use the tools. Therefore, I might introduce some concepts in the first few classes which you might not understand. These concepts will become clearer as we progress but they are necessary to explain how the tools work.

I feel that, swimming and programming languages should not be taught using the black board alone. The audience will understand everything that is taught in that way but when pushed into real waters, they are very likely to drown. This comes from my personal experience of learning C++. Therefore we will "jump" right away into the real waters, holding the side walls of the pool we shall learn first how to float and then how to swim. This will make the classes interesting right from the start and less time consuming as well. We will learn from our mistakes by deliberately introducing bugs in our program. This way you will be ready for handling such things when you start programming for your research work. No program is perfect-Even the operating systems that we use give us "security updates" everyday. Therefore do not worry if your program crashes but strive to make it better with every "update". I would appreciate if you come along with your laptop (i.e., the swimming pool) to my classes. You will enjoy the classes as you start understanding the power of the C++ language.

Languages, Operating Systems and Compilers

High Level Languages and Low Level Languages

Though most of this part is not completely right, I hope that it will give a inside view of what happens when a program runs. Read this only to aid your brains to get a picture. You will find a lot of terms within quotes to suggest that they are not to be taken literally. You need to have these concepts in your brains before we jump into C++.

At the core of the computer is its "brain" called the Processor. It is made up of millions of logic gates in addition to other electronics.

Look at how the addition of two numbers is made using a gate.The top one is an XOR gate and the bottom one is an NAND gate. This circuit gives a "SUM" and a "CARRY" output as shown in the truth table.

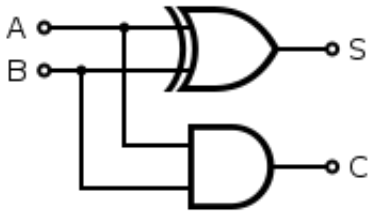

Figure 1: A Half Adder Circuit.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Remember that the Adder circuit above is a simple picture of logical operation using gates. In a processor, the inputs of the NAND gate are not tied directly to the inputs A and B as shown. Rather, there are "Digital Switches" in between those connections. Why? If they are connected directly, then this combination can be used ONLY as an Adder. On the other hand if there are "Switches", then one can connect/disconnect to form a logic circuitry of any kind. In the figure shown, if we have a switch in the inputs so that both the inputs of the NAND gate are connected to B, then the NAND gate can also operate as a logical inverter.

In a typical programming of a microprocessor this is what happens: Any processor comes with a set of instruction set (A set of binary numbers which will do switching and mathematical operations plus other stuff in the processor), which is provided by the manufacturer. Have a look at some of the instructions in the instruction set of

8085 microprocessor:

Mnemonic	Binary Code (in Hexadecimal)	Operation
ADD	87	ADD
CALL	CD	Call unconditional
CC	DC	Call on Carry
CMP	BF	Compare with Memory

For example to add two numbers, you have give the hexadecimal code 87 to the processor input which will "suitably switch and connect the gates" so that an adder is formed and then the address of the locations of the two numbers to be added are given. So A and B are "connected" to these memory locations. The output is also "connected" to a memory location but we need not get into details of that for now. [1]

Programming using the instruction set is usually called "Assembly Level Programming". This is also called Low level language/programming. Programs written in low level languages are faster in some sense. However, there are inherent disadvantages in it. One writes a "mnemonic code" and then replaces them by the binary codes and then checks if the codes are right and then checking if the program runs properly. This is not an efficient way to develop a code. Programmers nowadays use "high level languages" which remove (in fact, hide) all the complexities involved in putting together the pieces in the assembly language. High level languages give an advantage to a developer to write a code by using natural and understandable expressions.[2]

Examples of High Level Languages: C, C++, Java etc., High level languages classified into two types[3]:

- Compiled Languages: Here there is an intermediate "converter" called a compiler which converts the statements in your program into binary instructions that the processor understands, apart from checking your code for possible syntax errors. Examples are C, C++ etc.,

  Advantage: Speed of execution

  Disadvantage: Portability-You need to recompile your code on each machine it runs.

- Interpreted Languages: Each statement in the code and then the corresponding instruction is converted (using a java virtual machine (JVM) for Java) and executed one after the other. Examples are Java, Python etc.,

  Advantage: Portability-The same Java code will run in any machine and in any platform where JVM is installed.

[1] In the instruction set shown you see words like ADD, CALL etc., They are called mnemonics and are there only to aid programmers. What will be "fed" into the processor is not "ADD", "CALL" but the binary number 87, CD etc., here given in hexadecimal for simplicity. Before a programmer writes a program, he writes it using the mnemonics on a piece of paper and then converts them to the binary code and feeds them.

[2] Two types of languages exist in high level programming: compiled languages and interpreted languages.

[3] Compiled languages (C,C++ etc.,) are faster but are not easily portable across platforms. The contrary is true for interpreted languages (Java etc.,).

Disadvantage: Speed-at least 10-50 time slower than a compiled code.

Operating Systems

The operating system is the manager of your computer. It is also another program written in C (well, at least Linux). It manages and controls the use of various resources in your computer. Have a look at the Linux kernel in the figure given below. The bottom block is your hardware and most of the upper blocks are software pieces which build the operating system.

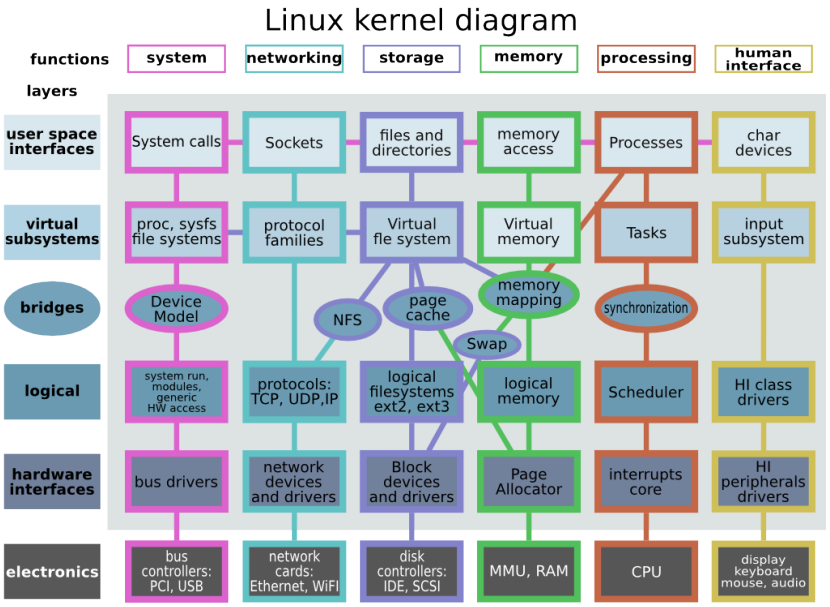

Figure 2: The pieces of codes that make a typical operating system. The most famous OSs are Windows, Linux and Apple. Linux is available for free (you can download the source code of Linux and see what is inside) and Ubuntu is a flavor of Linux.

So what do you need to write a C++ program?

- An Operating System: Although you can use any OS you wish but I prefer to teach the examples in Ubuntu/Scientific Linux as I myself am comfortable with that.
- An Editor: To write your code (gedit, emacs etc.,); You should be comfortable using it. I was never comfortable with these editors and feel that you should start using professional tools like eclipse or Qt Creator soon but only for the first few classes I recommend you to use gedit or emacs.
- A Compiler: As earlier mentioned, it is an intermediate program

called which converts the statements in your program into binary instructions that the processor understands, apart from checking your code for possible syntax errors and linking to other libraries.[4]

[4] For windows there is Visual Studio and for Linux gcc is widely used. You can check if gcc is installed in your computer by typing gcc -v in the terminal. If that is not installed, then do it using your package manager.

First Program: Hello World

So let us start writing our program. The steps that we will are:

- Step 1: Writing the code using an editor
  Open the text editor and type the following lines in it:[5]

main.cpp: Hello World

```
1 #include <iostream>
2 using namespace std;
3 int main ()
4 {
5   cout << "Hello World!";
6   return 0;
7 }
```

[5] Line 1 is the preprocessor directive which will include the contents of the file iostream to the current file. Line 3, the main function, is the entry point for all C/C++ exectuables. Line 5 will print out the statement "Hello World" using the function cout defined in iostream (included in line 1). Line 6 makes the program return from the main function.

- Step 2: Compile the code. This is a process which checks syntax errors, links with libraries and creates executable/objects etc., [6] Typical command for compiling a simple program is: "g++ filename", where, filename is the name of the file in which the code is saved. For example, "g++ Hello.cpp". on successful compilation, this command creates an output file named "a.out"

[6] In a Linux machine, the compile commands are typed in a so called "terminal".

- Step 3: Execute the output file:[7]
  Run the code using: ./a.out

[7] a.out is called an executable. The name of the executable can be changed by slightly modifying the compile command to "g++ Hello.cpp -o output", where output is the name of the output file.

Do it yourself:

Check what happens if the preprocessor directive in the first line is removed and the program is compiled.

Check what happens if you remove the line "return 0" and compile.

How do you comment out a line?

How do you comment out a block of statements?

Complier flags for creating objects and libraries

Functions

In this chapter, we will see how programs are organized and the role of compilers in such a process. You will have to probably memorize some of the commands that you will learn now. I will use the concept of functions to demonstrate some of the compiling options which will be useful later on. You will learn how to create objects and link to them and how to create libraries using these objects.

Operations that are repeated often are usually written as functions. This will not only reduce the size of the program but the program will look more organized. Let us see how to add two numbers, first without using functions and next using functions.

Simple addition. Compile using "g++ filename"and run using ./a.out

```
#include <iostream>
using namespace std;
int main ()
{
int a=99;
int b=100;
cout<<a+b<<endl;// returns the sum of a and b
 return 0;
}
```

main1.cpp: Adding two numbers

Adding using a function. Compile using "g++ filename". Run it using "/a.out"

```
#include <iostream>
using namespace std;
int Add(int x, int y) // The Add function definition
{
return (x+y);
}
int main ()
{
int a=99;
```

```
10 int b=100;
11 cout<<Add(a,b)<<endl;// returns the sum of a and b
12 return 0;
13 }
```

main2.cpp: Adding using a function

The use of functions will not be obvious in simple cases like these but you will see the use of functions in almost all the C/C++ programs. Though both the programs yield the same answer, functions give you more flexibility as they stand out as a separate entity.

Function prototyping and forward declarations

Now try compiling the program using functions after the modification as shown below: Move the function definition after the end of the main function. You will find the compiler returns an error. This is because, the compiler converts the program by reading it line by line. At line 7, the compiler does not know what "Add" means as it has not seen it before.

Moving the Add function definition below the main function returns a compiler error.

```
1  #include <iostream>
2  using namespace std;
3  int main ()
4  {
5  int a=99;
6  int b=100;
7  cout<<Add(a,b)<<endl;// returns the sum of a and b
8  return 0;
9  }
10 int Add(int x, int y) // The Add function definition
11 {
12 return (x+y);
13 }
```

main3.cpp: Function prototyping and forward declaration

To make sure that the compiler knows what "Add" means, the function should be ideally defined before the main function but this method has its own disadvantage. Imagine two functions that call each other-you will obviously fail in trying to decide which function to be placed first. The method of function prototyping is used in such cases to declare a function so that the compiler knows what the function looks like but does not know what the function does. For example, the prototype of the "Add" function is done in a single line like:

```
1 int Add(int a, int b); \\ a function prototype declaration
```

Declaring a function before the main function in order that the compiler is aware of a function is called "Forward Declaration". Using forward declarations, you now have the freedom of writing your function definitions anywhere you want. An example is shown below.

Example of forward declaration. Compile using "g++ filename"and run using "./a.out"

```
1  #include <iostream>
2  using namespace std;
3  int Add(int a, int b);\\forward declaration
4  int main ()
5  {
6  int a=99;
7  int b=100;
8  cout<<Add(a,b)<<endl;\\ returns the sum of a and b
9  return 0;
10 }
11 int Add(int x, int y) \\ The Add function definition
12 {
13 return (x+y);
14 }
```

In large software developments, the usual process is to move the function declarations in a header file and the definitions in a source file. Let us try it out with the "Add" function, you will make 3 files: Add.h, Add.cpp and main.cpp

```
1 \\Add.h
2 int Add(int a, int b); \\function declaration
```

Add.h: Addition function header file

```
1 //Add.cpp
2 #include ''Add.h'' //include contents of Add.h
3 int Add(int a, int b) // function definition
4 {
5 return (a+b);
6 }
```

Add.cpp: Addition function source file

The "Add" function definition is in Add.cpp and is declared in Add.h. In this case the forward declaration is done by including the file Add.h in the main program as shown in line 3.

```
1 //main.cpp
2 #include <iostream>
```

```
#include ‘‘Add.h’’ //include contents of Add.h
using namespace std;
int main ()
{
int a=99;
int b=100;
cout<<Add(a,b)<<endl;// returns the sum of a and b
return 0;
}
```

main3.3.cpp: Compiling multiple files

The compile command is slightly different if your code includes files other than your main source file. Compile this code using:

g++ main.cpp Add.cpp

Run the program using “./a.out”.

Creating objects and linking

Imagine a function (like the “Add” function) that is used by most of your programs. You obviously do not want to repeat writing the “Add” function either in the main program or in a different source file as described earlier. Doing so is a wastage of time, especially in big programs and it also increases the size of the code.[8] There is an alternative to overcome such things. You can compile the functions that are needed often and create so called objects. Objects are binary versions of the functions which can be called by any program by a process called linking. Creating objects is done by using a special flag while compiling. For example to create an object for the “Add” function saved in Add.cpp, the compile command is:

[8] Remember compiling a huge program also is a time consuming process. For example compiling ROOT, a plotting and data analysis program from CERN, takes about 2 hours on a laptop.

g++ -fPIC -c Add.cpp

Now, if the compilation is successful, you will find a file name Add.o in the directory. This is called an object file.[9] Now to call the function “Add” in your program, instead of doing:

[9] You will soon encounter another concept called objects in C++. C++ is called an object oriented language where the term object refers to another concept altogether but here we refer to an “object file” which has nothing to do with it. Do not get confused between these two.

g++ main.cpp Add.cpp

as we did earlier, we do:

g++ main.cpp Add.o

The difference between these two commands is that the first one compiles main.cpp and then Add.cpp and then makes an executable. The second one compiles main.cpp only and the object file Add.o is “inserted at the right place” which reduces the compilation time.

Creating libraries and linking

A collection of objects is called a library. For example, a math library will contain functions for addition, subtraction, multiplication etc., Let us create a simple math library for addition and subtraction. Since we already made an object for addition, we will now make the object for subtraction and then create a simple "math library". Create the files called Sub.cpp and Sub.h and enter the following in those files.

```
\\Sub.h
int Sub(int a, int b); \\ function declaration
```

Sub.h: Subtraction function header file

```
\\Sub.cpp
#include ''Sub.h'' \\ include contents of Sub.h
int Sub(int a, int b) \\ function definition
{
return (a-b);
}
```

Sub.cpp: Subtraction function source file

Now we shall create an object (Sub.o) for this file using the command described earlier:

g++ -fPIC -c Sub.cpp

Now we will create a math library called "myMath" which will contain the basic addition and subtraction functions in it. The command to do it is:

g++ -shared -o libmyMath.so Add.o Sub.o

Of course, this command will be successful only if Add.o and Sub.o already exist.

The prefix "lib" and the extension .so is a standard convention that has to be followed. The flag "-shared" is used to create shared libraries.

Linking to a library in your program

Now we have a library of functions, compiled and ready to be called by any programs. You do not have to write separate files and compile them and then link them. All you have to do is to include the requisite header files to forward declare the functions and then link to the libraries in the compiler.

Using this library, let us see how to write a program which returns the sum and the difference of these two numbers. First we create the main file:

```
\\main.cpp
#include <iostream>
#include ‘‘Add.h’’ \\include contents of Add.h
#include ‘‘Sub.h’’ \\include contents of Sub.h
using namespace std;
int main ()
{
int a=99;
int b=100;
cout<<Add(a,b)<<endl;\\sum of a and b
cout<<Sub(a,b)<<endl;\\diff of a and b
return 0;
}
```

main5.cpp: usage of libraries

The program above is compiled using the command:

g++ main.cpp -L. -lmyMath

The flag -l before myMath tells the compiler to link to the library myMath. Notice that the prefix “lib” and the extension “.so” have been dropped. The flag -L. tells that the library is existing in the current directory. If the library is in another directory replace the “.” by the directory name. Now run the code using “./a.out” to see the output.[10]

So far, you have seen ways of compiling a simple program. Now it is time to jump into real programming. We shall use these concepts while writing the programs that follow. You should now be knowing how to compile a program and how to link to a library. The commands will not be explicitly mentioned as often it was in this section.

Do it yourself:

What are environment variables?

List the common environment variables in Linux.

How do you set an environment variable in Linux?

[10] In some systems, the execution might fail with a statement like “error while loading shared libraries” in which case you should add the directory where the library exists, to the environment variable LD_LIBRARY_PATH using the command “export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:dirname”, where dirname is the name of the directory.

Data types and operators

In any programming language you will encounter the term called variable. For a computer, a variable is nothing but a memory slot. In the previous programs, you might have seen the following statement:

int a;

What internally happens is a memory location is allocated for storing an integer.[11]

[11] The main memory of the computer is called as the RAM.

int a=99;

The above statement not only allocated memory but also set the bits in that allocated memory space to the corresponding value. So when the term memory comes, bring the picture shown here to your mind. Though the value of the variable can be changed, the address cannot be altered. You can only read the address of a variable using the "&" operator. The following code prints the address of the variable a and the date stored in that address i.e., the value of a.

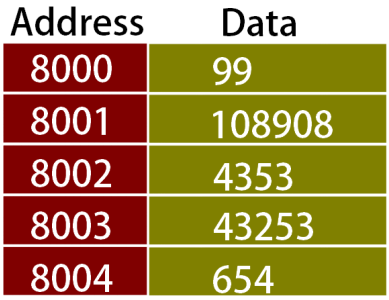

Figure 3: Representation of the physical memory in the computer. For every variable there is an address field and a data field.

```
#include <iostream>
using namespace std;
int main ()
{
int a=99;
int b=100;
cout<<&a<<endl;\\ print the address of a
cout<<a<<endl; \\print the value of a
return 0;
}
```

main6.cpp: Address of a variable

Fundamental data types

The fundamental data types in C/C+ are listed in Table 1. In addition to the fundamental types there is an option to use data types

Name	Description
char	character (1 byte)
short	short integer (2 bytes)
int	integer (4 bytes)
long	long integer (4 bytes)
bool	boolean value-true/false (1 byte)
float	floating point (4 bytes)
double	double precision float (8 bytes)
long double	long double precision float (8 bytes)

Table 1: Fundamental data types in C/C++.

like int as an unsigned type where the numbers are treated as having a positive value. Therefore if the range of an int type is from -2147483648 to 2147483647, the range for the unsigned int will be from 0 to 4294967295. The statements for declaring a unsigned types look like:

```
unsigned int a;
unsigned long b;
unsigned float c;
```

Do it yourself:

Try subtracting two unsigned integers a and b. First try a-b and then b-a. What is the problem here?

Working with hexadecimal notation

You will have to work with the hexadecimal notation at some point of time in programming, especially if your program is interfaced with some hardware. A hexadecimal number is denoted with a prefix "0x". Thus the statement

```
unsigned int a=0xFF;
```

denotes the integer value of 255.

Escape characters

There are some special characters that are used in C/C++ which have a special meaning especially in a cout statement. They are usually not printed. They are listed in Table 2.

Lets write a simple program that takes an input from the user and gives the mean as the output using the concepts that we learnt so far and in a interactive manner. The "cout" command is used to print out a statement while the "cin" command accepts a user input. These commands are useful for an interactive program and are illustrated in the example below.

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)

Table 2: Some of the escape codes in C/C++.

```
#include <iostream>
using namespace std;
float mean(float x, float y) \\ mean function
{
return (x+y)/2;
}
int main ()
{
float a;
float b;
cout<<''Enter number 1''<<endl;
cin>>a;
cout<<''Enter number 2''<<endl;
cin>>b
cout<<''The mean is''<<''\t''<<mean(a,b)<<endl;
return 0;
```

main7.cpp: scope of a variable

Scope of a variable

In the above code, the variables a and b are "seen" only by the main function and therefore they have a local scope. If instead for some reason, you want the function mean to also "see" these variables, you can declare outside main before the mean function. Then these variables are said to have global scope.[12]

[12] Local variables are can be used only within the function in which they are declared. Global variables can be used by any function that can "see" it.

```
#include <iostream>
using namespace std;
float a;
float b;
float mean(float x, float y) \\ mean function
{
```

```
7  cout<<a;
8  return (x+y)/2;
9  }
10 int main ()
11 {
12 cout<<''Enter number 1''<<endl;
13 cin>>a;
14 cout<<''Enter number 2''<<endl;
15 cin>>b
16 cout<<''The mean is''<<''\t''<<mean(a,b)<<endl;
17 return 0;
```

main8.cpp: scope of a variable

Do it yourself:

Check what happens if the float function in the above example takes "int" parameters instead of "float".

What happens if in the above code, a and b are once again declared inside the main function along with those outside?

Assignment Operator

The most basic operator is the assignment operator "=". This assigns a particular value to the variable. For example, the expression:

int a=5;

assigns a value of 5 to the variable a.

Mathematical Operators

Mathematical operators are the most frequent ones that you will use in a program, especially if you are a physicist. They are listed in Table 3

Operator	Description
+	Addition
-	Subtraction
⋆	Multiplication
/	Division
%	Modulo

Table 3: Mathematical operators in C/C++.

Compound assignment operators

Compound assignment operators operate on the variable and store the output in the same variable. They are followed by an "=" sign after a mathematical or a logical operator as shown below.

`+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=`

For example, the expression:

a -= 5;

is similar to the expression:

a = a-5;

i.e., if a has a initial value of 0, then after this expression its value is changed to -5.

Increment and decrement operators

To increase the value of a variable by 1 the increment operator "++" is used and to decrement by 1 the operator "–" is used. For example, after the following expressions:

```
int a=100;
a++;\\a=101
```

the value of a becomes 101 and after the following expressions:

```
int a=100;
a--;\\ a contains 99
```

the value of a becomes 99. The increment and decrement operators can be used either as a suffix to a variable as shown in the previous expressions or as a prefix to a variable. Consider the case:

```
int a;
int b=99;
a=b++;a contains 99, b contains 100
```

Here, after the last expression, the value of a is set to 99 and the value of b is set to 100. That is, the value of b is incremented after the assignment operation. If the increment operator is used as a prefix:

```
int a;
int b=99;
a=++b;\\ a contains 100, b contains 100
```

the value of a and b are set 100. That is, the value of b is incremented before the assignment operation.

Relational operators

These operators compare the values of two expressions and return a boolean value (true or false). The result of various relational operations for the variables a and b after the following expression is tabulated in Table 4:

```
int a=100;
int b=99;
```

Relational Operator	Description	Expression	Return
==	equal to	(a==b)	false
!=	not equal to	(a!=b)	true
>	greater than	(a>b)	true
<	lesser than	(a<b)	false
>=	greater than or equal to	(a>=b)	false
<=	lesser than or equal to	(a<=b)	false

Table 4: Relational operators in C/C++.

Warning: The relational operator "==" should not be confused with the assignment operator "=". This is a frequent mistake made by beginners.

Logical operators

These operators perform logical operations on expressions that return a boolean value. The result of logical operations for the variables a and b after the previous expression is tabulated in Table 5:

Logical Operator	Description	Expression	Return
!	logical inversion	!(a>b)	false
&&	logical AND	(a>b)&&(b<1000)	true
||	logical OR	(a<b)||(b<50)	false

Table 5: Logical operators in C/C++.

Conditional Operator

The conditional operator "?" evaluates an expression and returns one value if it is true and returns another value if it is false. The synatax is:

condition ? result1 : result2

The use of this operator is illustrator is explained in the following code.

```
#include <iostream>
using namespace std;
```

```
int main ()
{
  int a,b,c;
  a=100;
  b=99;
  c = (a>b) ? a : b;
  cout << c<<endl;
  c = (a<b) ? a : b;
  cout << c<<endl;
  return 0;
}
```

main9.cpp: Conditional operator

The first "cout" statement will print out 100 since a is greater than b. The second one will print out 99.

Bitwise operators

These operators modify the bit patterns of a variable. They are listed in the following Table 6.

Bitwise Operator	Description	Expression	Return
&	AND	0x01&0x01	0x01
|	OR	0x01 | 0x01	0x01
^	XOR	0x01 | 0x01	0x00
~	NOT	0xFFFFFFFF	0x00
»	Shift Right	0x01»1	0x00
«	shift Left	0x01«1	0x02

Table 6: Bitwise operators in C/C++.

An alternative to "cout" is the printf statement which can be useful for printing hexadecimal numbers. To print the value of the variable "a" in hexadecimal format, use printf("%x",a);

Typecasting

There are some data types which can be converted into other types using the typecasting operator. A typical example is given below:

```
float pi=3.14;
int i = (int)pi;
```

The value of i is now 3 and the numbers after the decimal point is lost. This is a trivial example and as you can also check that the value "pi" can be directly assigned to "i" without typecasting and it still works the same way. This is because the compiler takes the freedom of typecasting is some cases. Typecasting will become more obvious when you start dealing with pointers.

Do it yourself:

What is the result of the expression int a= 100 + 2 * 5 ?

Is it 110 or 510?
Learn more about operator precedence.

Control Structures

Often in your program, you may have to taken decisions and route your program to a particular segment or repeat a particular segment of code. These are done by control structures which are discussed in this chapter.

Checking if a condition is met: if and else

The "if" and "else" keywords are used for checking if a condition is satisfied and then executing a block of statements which are put inside the curly braces which follow. An example is given below:[13]

```
#include <iostream>
using namespace std;
int main ()
{
int a=99;
int b=100;
if (a<b)
{ \\ begin braces
  cout << "a is less than b";
} \\ end braces
else
{ \\ begin braces
  cout << "a is greater than b";
} \\end braces
return 0;
}
```

main10.cpp: if else structure

The above code prints out "a is less than b" as the condition "a<b" satisfied. But this code will fail if the values of a and b are the same. First it checks if "a<"b and since it fails and therefore it goes into the "else" structure to print out "a is greater than b which is wrong. In such cases, you can use a combination of "if' and "else" using the "else if" keyword.

[13] Notice the begin and end braces which enclose the group of statements that have to be executed if the condition is satisfied. These braces are not needed if there is only one statement. Though in the example there is only one statement, the braces are put for purpose of illustration but it is a good practice to put them during the learning phase. We shall call the set of statements between a begin brace and an end brace as a "block".

```
if (a<b)
{ \\ begin braces
  cout << "a is less than b";
} \\ end braces
else if (a>b)
{ \\ begin braces
  cout << "a is greater than b";
} \\end braces
else
{ \\ begin braces
  cout << "a is equal to b";
} \\end braces
```

main11.cpp: else if structure

Repeating a set of instructions: Loops

Loops repeat a set of expressions until a condition is met. There are different types loops each having a specific purpose in C++.

while loop

The while loop is repeated until the condition checked by the while statement is satisfied as shown below:

```
#include <iostream>
using namespace std;
int main ()
{
  int a=0;
  while (a<100)
  { \\ begin
    cout << a << ", ";
    a++;
  } \\end
  cout << ‘‘a exceeds 100’’;
  return 0;
```

main12.cpp: while loop

In the above code the value of a is printed and then incremented until condition checked by the while statement (a<100) is satisfied (when a exceeds 100). There is a variant of the while loop called the do while loop which has a slightly different functionality:

```
#include <iostream>
using namespace std;
int main ()
{
  int a=0;
  do
  { \\ begin
    cout << a << ", ";
    a++;
  } \\end
  while(a<100);
  cout << ‘‘a exceeds 100’’;
  return 0;
```

main13.cpp: do while loop

Unlike the earlier version, the condition in the while statement is checked at the end of the execution of the statements. There are no rules to state which type to be used. You can choose the one which is most appropriate for you code.

for loop

The for loop is also similar to the while loop but with extra functionalities. Unlike the simple while statement, there is an initialization part, conditional part and a statement which is executed in the for statement.

```
#include <iostream>
using namespace std;
int main ()
{
  for (int a=0; a<101; a++)
  { \\begin
    cout << a << ", ";
  }\\end
  cout << "a exceeds 100";
  return 0;
}
```

main14.cpp: for loop

The code above does the same operation that was earlier done by the while loop. The for statement has three parts separated by a semicolon:

1. int a=0; This initializes a to zero

2. a<101; This checks if a is less than 101 and allows the block to be executed only if it is true

3. a++; This increments the value of a before the end braces

Breaking and skipping in a loop structure

In a loop block you might want to break the loop if some condition is met or you might want to skip executing a part of the block. These are done using the "break" and "continue" keywords as shown in the codes below.

```
#include <iostream>
using namespace std;
int main ()
{
   for (int a=0; a<100; a++)
  {
    cout << a ;
    if (a==50)
    {
      cout << "a is 50";
      break; \\ come out of the loop
    }
  }
  return 0;
}
```

main15.cpp: Breaking a loop

In the above code, the for loop is broken when a reaches 50 and the loop is not more executed.

```
#include <iostream>
using namespace std;
int main ()
{
  int a;
  for (a=0; a<100; a++)
  {

    if (a==50) continue; \\skip the following lines
    cout << a;
  }
  return 0;
```

main16.cpp: Skipping an iteration

In the above code, when a reaches 50, due to the continue keyword, the following lines of code in the block ("cout") are not executed but the unlike the break statement the loop is not broken, the execution continues with the next iteration.

switch-case structure

The switch case structure is similar to a multiple "if else" structure. An example is given below:

```
#include <iostream>
using namespace std;
int main ()
{
  int a =1;
  switch (a) {
  case 1:
    cout << "a is 1";
    break;
  case 2:
    cout << "a is 2";
    break;
  default:
    cout << "value of a unknown";
  }
  return 0;
}
```

mian17.cpp: Switch Case

In the above code, the switch statement checks the value of "a" with the possible cases. In case a is 1 then that segment is executed. It might happen that a is neither 1 nor 2 in which case the default section is executed. Notice, the use of break after the end of each case except the default case.

Exiting and returning from a code

The exit and the return functions are used to exit from a running program or to return from a function. The return function additionally takes an argument depending on the return type of the function. Examples are:

return -1;
return 1.5;

Do it yourself:

Implement the switch case code above using if else structures.

Get an integer input from the user and calculate the factorial using loops. The code should output an error if the user enters a negative integer.

More about functions

So far, you have been using functions to for performing simple operations like adding two numbers. The purpose of introducing functions then was to illustrate the use of libraries and related flags in a compiler. We will now venture a bit into it and come back.

The void type

Consider the following function:

```
1 int add(int x, int y){
2 return (x+y);
3 }
```

This function "takes" integers and "returns" an integer. Sometimes you want to write a function that does not return anything in such cases you can use the "void" return type as shown below:

```
1 void Error(int ErrNum){
2 cout<<''An Error !!!''<<''\t''<<ErrNum<<endl;
3 }
```

In fact the void type can be used as an input to functions but that is an advanced concept which you can learn when you understand pointers.

Passing reference to a variable

The add function defined earlier can be called in the main function using the following syntax:

```
1 int add(int x, int y){
2 return (x+y);
3 }
```

```
int main ()
{
  int a = 101;
  int b = 100;
  int c = add(a,b);
  cout<<''The value of c is ''<<c<<endl;
  return 0;
}
```

main18.cpp: Passing values and references

When we called the function "add" with the inputs "a" and "b" what happens actually is that the value of a and b are "copied" into the local variables "x" and "y" and the addition is done using them. The result of the addition is then stored in "c". Notice that you cannot change the value of "a" or "b" inside the add function or for that matter any variable declared inside the main function. Of course, one can make these global variables so that the add function "sees" them but, there are disadvantages in doing so. The way to do it in C++ is by passing the reference instead of the value itself. Have a look the example below:

```
int add(int x, int y, int& z){
z = x+y;
return (x+y);
}
int main ()
{
  int a = 101;
  int b = 100;
  int c=0;
  cout<<''c before addition is ''<<c<<endl;\\c=0
  add(a,b,c);
  cout<<''c after addition is ''<<c<<endl;\\c=201
  return 0;
}
```

main19.cpp: Passing values and references

Here the add function is slightly modified-now it takes three parameters instead of two. Also notice the & sign before the third parameter. Using this syntax (&), the value of a parameter which is not "seen" by the function can be changed. Here we are storing the result of the sum of the numbers in the third parameter "c" which is not actually seen by the function. We call this method as referencing a variable in a function. This method is also useful when you want a function to return more than one output. For example, you want a function to

return both the sum and the difference, you could do this way:

```
void add_sub(int x, int y, int& sum, int &diff){
sum = x+y;
diff = x-y;
}
int main ()
{
  int a = 101;
  int b = 100;
  int s=0;
  int d=0;
  add_sub(a,b,s,d);
  cout<<''The sum is ''<<s<<endl; \\s=201
  cout<<''The difference is ''<<d<<endl;\\d=1
  return 0;
}
```

What happens when a variable is passed by referencing (i.e., using &) is that instead of the value of the variable, the address of the variable is given as an input. The function manipulates the value at that address. Remember, the address of any variable is given by the OS and you have no way of changing it (although you can change the value at that address). That is why though the call add_sub(101,100,s,d) is a perfect alternate, the call add_sub(101,100,1000,2000) is wrong and the compiler will complain as the last two are not "proper addresses" as they should have been.

main20.cpp: Passing values and references

Default parameters in functions

By giving default values to some parameters in a function, those parameters can be skipped while calling the function in which case the default value is taken and the operation is performed.

```
int add(int x, int y=0){
return(x+y);
}
int main ()
{
  int a = 101;
  int b = 100;
  cout<<add(a); \\ prints 101
  return 0;
}
```

main21.cpp: Default Parameters

In the above example, the second parameter is given a default value of 0, therefore if the function is called without the second parameter, then the default value for that parameter is used. However, not all parameters can have default values; either all the parameters or the last parameter(s) only can have default values. The legal statements are illustrated in the example below:

function	legal
operation(int x=0, int y=0, int z=0, int a=0, int b=0)	yes
operation(int x, int y=0, int z=0, int a=0, int b=0)	yes
operation(int x, int y, int z, int a=0, int b=0)	yes
operation(int x, int y, int z, int a, int b=0)	yes
operation(int x=0, int y, int z, int a, int b)	no
operation(int x=0, int y, int z, int a, int b=0)	no
operation(int x=0, int y=0, int z, int a, int b)	no
operation(int x=0, int y=0, int z, int a, int b=0)	no

Function overloading

Sometimes you need to write functions which have almost similar functionality but take different input types and return different return types. You can do this using the concept of function overloading as illustrated below:

```
1  int add(int x, int y){
2  return(x+y);
3  }
4  float add(float x, float y){
5  return(x+y);
6  }
7  int main ()
8  {
9    int a = 101;
10   int b = 100;
11   float c = 101.2;
12   float d = 102.5;
13   cout<<add(a,b); \\ prints 201
14   cout<<add(c,d); \\ prints 203.7
15   return 0;
16 }
```

main22.cpp: Function overloading

In the above example, there are two functions defined with the same name “add”. These two have almost similar functionality but one operates on integers and the other operates on float. This is an example of function overloading. You could have as well given a different name for the function that adds floats but the benefit of using function overloading is that the user need not worry which function to use for adding integers, the function overloading takes care of it by checking the input parameter types and uses the appropriate function.

Storing collection of data: Arrays and Structures

Arrays

In the previous sections we stored single values in variables using instructions like:

int a;

We now move a step ahead to see how to store a collection of values. For example, if you want to store the age of 5 people, you would do:

```
1 int age1=50;
2 int age2=45;
3 int age3=13;
4 int age4=15;
5 int age5=76;
```

There is a much more efficient way of doing it in C/C++. For such cases where you want to store a collection of values belonging to a data type, you can use arrays. The above example can be done in an efficient way using array as illustrated:

```
1 int age[5]; \\ array declaration
2 age[0]=50;\\ array initialization 0
3 age[1]=45;\\ array initialization 1
4 age[2]=13;\\ array initialization 2
5 age[3]=15;\\ array initialization 3
6 age[4]=76;\\ array initialization 4
```

main23.cpp: 1d Array

The first line declares an one dimensional integer array of size 5. [14] The following lines initialize the values at the respective index. Another way of initializing the same array is as follows:

```
1 int age[5]={50,45,13,15,76}; // array declaration
```

[14] Notice the index of the array starts from 0 and ends at 4. This has to be remembered by beginners who are usually tempted into starting from 1 and ending at 5.

To access the value at a particular index for example 2 is as simple as:

```
int  age2 = age[2];
```

Do it yourself:

Try to see what happens if you try to access an element outside the array limit, say 6 using age[6]

What is the output you get when the arrays are not initialized?

Multidimensional arrays

Multidimensional arrays are another variation of one dimensional arrays. For example, you want to store the age of students in 2 classes class 1 and class 2, each having 5 students then you could do it with a two dimensional array like:

```
int age[5][2]; \\ array declaration
\\ class 1
int age[0][0]=6;\\ array initialization 0
age[1][0]=7;\\ array initialization 1
age[2][0]=7;\\ array initialization 2
age[3][0]=6;\\ array initialization 3
age[4][0]=8;\\ array initialization 4
\\class 2
age[0][1]=7;\\ array initialization 0
age[1][1]=8;\\ array initialization 1
age[2][1]=6;\\ array initialization 2
age[3][1]=6;\\ array initialization 3
age[4][1]=7;\\ array initialization 4
```

main2.cpp: 2d Array

The first line declares a two dimensional array of size 10 (5*2).[15] Of course, we could have interchanged the indexing (2*5), and changed the initializing part as well to obtained similar results. We could have as well called an one dimensional array of size 10 and stored the values but using a multidimensional array has made the code look structured. For example, If we want to loop on the age of the class 2 students, it is now as simple as keeping the second index as 1 and looping over the first index from 0 to 4.

Do it yourself:

Try to implement a 10 dimensional array with each index of size 100 and see if you can compile the code. If the compilation fails try to reduce the dimension one by one and then run the code.

[15] An array can have any number of dimensions but it has practical limitations imposed by the size of the physical memory in the computer

Using array as input parameters to functions

Like any other data type, arrays can also be passed as a parameter to function. However, the syntax is a bit different:

```
\\ function taking a one dimensional array as a parameter
int addArray(int arr[], int size){
int c=0;
for(int ii=0;ii<size;ii++{
c += arr[ii];
}
return c;
}
\\ function taking a two dimensional array as a parameter
int addArray(int arr[][2], int size){
int c=0;
for(int jj=0;jj<2;jj++{
for(int ii=0;ii<size;ii++{
c += arr[ii][jj];
}
}
return c;
}
```

main25.cpp: Arrays as parameters

The above example shows two functions one takes a one dimensional array as its input. As you have noticed, the syntax is that the size of the array is not given (i.e int arr[]). For a two dimensional array, the size of the second dimension is given but not the size of the first one.[16]

[16] The general syntax for giving arrays as input parameters to functions is that the index all but the first dimension should be given. The first dimension is optional.

Structures

Arrays are useful for storing a collection of values which are of similar data type. Structures are a collection of a different data types. For example you could use structures to store the age, height and weight of a student using structures:

```
 struct student {
  int age;
  float height;
  float weight;
};
```

We have used multiple data types like int and float in the structure student. The structure student is now a data type by itself and can be used now the way "int" is used to denote a integer. An example of declaring a variable of type student and initializing the values is shown in the example below.

```
 struct student {
  int age;
  float height;
  float weight;
};
student a;
student b;
\\ initialize for student a
a.age=10;
a.height=100;
a.weight=30;
\\ initialize for student b
b.age=13;
a.height=127;
a.weight=36;
```

main26.cpp: Structures

The variables inside the structures are accessed by the ".". a and b are variables of type student or we call them objects of type student and their respective values like age, height and weight are accessed by suffixing a "." and the variable name. Another way of declaring structures is:

```
 struct student {
  int age;
  float height;
  float weight;
}a,b;
```

Structures used in conjunction with arrays is a neat way to organize a database. For example, to have the database of a class having 100 students you just have to call an one dimensional array of structures:

```
student class1[100];
```

main27.cpp: Structures and Arrays

Pointers

Address	Data
8000	99
8001	108908
8002	4353
8003	43253
8004	654

Figure 4: Representation of the physical memory in the computer. For every variable there is an address field and a data field.

What is a pointer?

Earlier, we have seen how to visualize the physical memory in the computer and how a variable is stored in it.

Any variable is stored in a memory location and every memory location has an associated address to it. A pointer is a special variable which stores the address of a memory location. A pointer to an integer is declared like:

```
int *p;
```

There is a star after the data type int to denote that this variable is only allowed to store address of memory locations. There can be pointers to all the data types we have studied so far (float, double etc.,). Pointers are not so easy for beginners to understand but once you get a pictorial representation, it is easier to follow. Let us assume that we are declaring a variable:

```
int a;
a=100;
```

When the first statement is executed, a memory location is allotted but that memory location is likely to contain a random value as this variable is not yet initialized. The pictures below show the memory location of a before initialization and after initialization to 100.

address	data	variable
5000	56739	a

Figure 5: The memory location of the variable a. The variable is not yet initialized (int a) and therefore contains a random value 56739.

Figure 6: The memory location of the variable a. The variable is now initialized to 100 by the second statement (a=100).

Now let us store the address of the variable a in a pointer variable.

```
int a;
int *p;
```

In the above statements, the first line declares a variable of type int but the second line declares a pointer that points to an integer. At this stage since we have not initialized the variables, the memory locations will look like:

Figure 7: The memory location of the variables a and p before initialization.

After the statements below:

```
int a = 100;
int *p = &a;
```

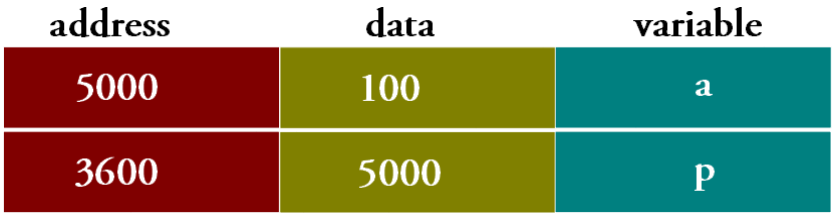

Figure 8: The memory location of the variables a and p after initialization.

As shown above, the way to get the address of a is to put a "&" before it. Now pa contains the address of a as shown in the picture. This is the correct way to initialize a pointer, that is only addresses can be passed to a pointer. You cannot initialize a pointer to a random value by yourself:[17]

```
int a = 100;
int *p = 5000;
```

The compiler will complain if you try to do it.

[17] An exception is that you can set a pointer to 0 in which case it is called as a "Null pointer". A Null pointer should not be used for any operations, a concept which will become clearer soon.

Setting the value of a variable using pointers

Now since we know the address of a variable, we can manipulate the value at that address using the pointers itself. To set the value of "a" to 99 using its pointer "pa", we do:

```
*pa = 500;
```

main28.cpp: Pointers

We use the "*" to denote that the value at the memory location stored in pa is 500. Obviously there is a confusion with the "*" here. Previously we used it to denote a pointer variable now it has different definition. An easier way to understand is that if there is data type like int, float etc., before a "*", then the star denotes that the variable next to it is a pointer. If not, as in the case above, then the star should be read as "the value at memory location pa is ..."

The & is called as the reference operator and * is called as the dereference operator.

Incrementing and decrementing pointers

Like other data types, pointers can also be incremented and decremented (of course, other mathematical operations are permissible but the increment and decrement operations are the most often used). But there is a difference in the way pointers are incremented. For example, if you increment an integer using the increment operator "++", the value is increased by 1 but a pointer of an integer is incremented not by 1 but by the size of an integer. This is a point that should be understood clearly. Memory locations have a fixed width (8 bits usually). We know that the size of a integer is 32 bits which obviously cannot be written in a memory location. Therefore the integer is split into 4 parts and stored sequentially in the memory location. Thus when a pointer to an integer is incremented, it jumps 4 locations.

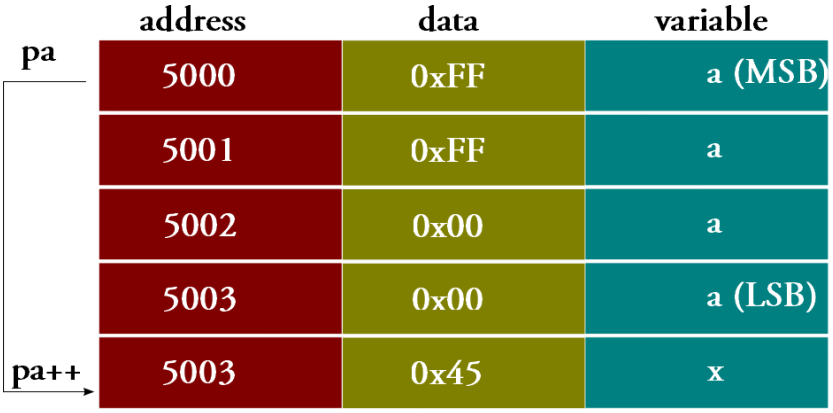

Figure 9: Incrementing the pointer pa (having the address of the integer a) does not increment it by one but by the size of the integer divided by the width of the memory field. In the example shown here the width of the memory field is 8 bits and the size of the integer is 32 bits.

main29.cpp: Pointer arithmetic

Pointers and Arrays

There is a close connection between pointers and arrays. An array when declared, is allocated a contiguous region of memory, therefore if the address of the first element is stored in a pointer then the memory location of the next element can be pointed by just incrementing the pointer. The address of the first element can be done by:

```
int a[5];
int *pa=a;
```

Now pa contains the address of the first element (i.e., a[0]). To get the address of, for example, the third element in the array we do:

```
int a[5];
int *pa=&a[2];
```

main30.cpp: Pointers and Arrays

Do it yourself:

In the above example, what happens when you make a statement like *pa++? Is the pointer incremented or the value pointed by the pointer incremented? Check it out by printing out the values of pa and *pa after the statement

Pointer to pointers

As mentioned earlier, pointers contain the address of a variable. In the similar way, the address of the pointer can also be stored in another pointer. This is illustrated in the example below:

```
1 int a;
2 int *pa=&a; // pa contains the address of a
3 int **ppa=&pa;//ppa contains the address of pa
4 int ***pppa=&ppa;//pppa contains the address of ppa
```

You may add as many “stars” as you want but you are most likely to use only pointers and not anything more than that and therefore we shall not dwell much into it.

Other types of pointers

So far, in the illustrations we have seen only pointers to int and float but there are types of pointers apart from double, long etc., which you will often encounter. The first type is the void pointer. The void pointer has no type as is declared as:

```
1 void *p;// a void pointer
```

A void pointer cannot be incremented or decremented for obvious reasons. The computer does not know how many memory locations to skip, as it does not know the type. But a void pointer has the flexibility that any type of pointers can be stored in it (int, float, double etc.,).

main31.cpp: void pointer

Do it yourself:

How do you increment a void pointer? Learn more about the function sizeof() and try to use it to increment a pointer.

There can be pointers to structures as well but due to the similarities with pointers to classes, we shall do it later on. Pointers to functions are also allowed in C++ but I hope that you will not need it at least in the near future and therefore we shall skip it.

File I/O

In the next chapter, we will jumping into the concept of C++ but before that I will give a short glimpse of file I/O. We will learn how to do various operations like opening, reading, writing and closing a file. This chapter will be brief so that you will just know how to do simple things using files. There are other ways to manipulate files but I will just cover only one.

The first thing to note is that we will include a header file called "fstream" whenever file related operation are done[18]. The general sequence to do any file related operation is:

- Open the file
- Read or Write from/to the file
- Close the file

[18] There are other headers like "ifstream" and "ofstream" specifically used for reading and writing respectively. "fstream" can be used for both reading and writing.

File objects

Prior to doing any file operations, we should first define a file object and that is done like:

```
fstream outFile;
```

fstream is a class in C++ which is used for basic file reading and writing. Since we will get into classes in the next chapter, we will just treat it like we treat int, float etc., We call outFile is an object of type fstream.

Opening a file

Opening a file is done using the function "open" as shown below:

```
outFile.open("out.txt", ios::out | ios::app );
```

The first argument is the file name and the second argument is a combination of some flags as described below:

flag	description
ios::out	open file for writing
ios::in	open file for reading
ios::app	append file for appending
ios::trunc	delete file (if it already exists) and open

These flags can be combined using the "Bitwise OR" operator as shown in the previous example which would open a file for writing and appends the data written in it. However not all combinations are possible. For example, you cannot open a file both for reading and writing (ios::out | ios::in) neither can you open a file with a option like (ios::app | ios::trunc) as these are conflicting each other.

Writing to a file

Once you have opened a file for writing, the file object can be used the way "cout" was used to write on the screen as shown in the example below:

main32.cpp: Writing to a file

```
#include <iostream>
#include <fstream> // required for file related operations
using namespace std;
int main () {
   fstream outFile;
   outFile.open("test.txt",ios::out|ios::trunc);
   outFile<<"I am here"<<endl; // write this into the file
   cout<<''Data written...''<<endl;
   outFile.close(); //close the file
   return 0;
}
```

Reading from a file

For reading out data from a file, the flag "ios::in" should be used first to open the file. The reading syntax is similar to "cin" which takes inputs from a user and stores it in a variable. You should therefore know what the data type is before reading the data.

main33.cpp: Reading from a file

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
   fstream inFile;
```

```
    inFile.open("test.txt",ios::in);
    int a;
    inFile>>a;
    cout<<a<<endl;
    inFile>>a;
    cout<<a<<endl;
    inFile>>a;
    cout<<a<<endl;
    inFile>>a;
    cout<<a<<endl;
    inFile.close();
    return 0;
}
```

Reading out line by line using string

For reading out lines of data from a file, you can use the "getline" function. You need to use the data type string for that and declare the string header file as shown below:

main34.cpp: Reading lines of data from a file

```
#include <iostream>
#include <fstream>
#include <string> //header file for strings
using namespace std;
int main () {
    fstream in;
    in.open("test1.txt",ios::in);

    string s;
    cout<<"Line 1:"<<endl;
    getline(in,s); // read first line
    cout<<s<<endl;

    cout<<"Line 2:"<<endl;
    getline(in,s);// read second line
    cout<<s<<endl;
    in.close();
    return 0;
```

Closing a file

Closing a file is done using the function "close()" as illustrated in the examples above. If the file is not closed, the memory might not be freed for other uses or the file might get corrupted.

Checking the end of file (EOF)

You might want to read a file which contains a long list of integers. So in that case you implement a loop to read the data but the problem is most of the times you do not know how many numbers (or data elements) are stored in it. To overcome this, you can use the "eof" function which will return "true" if the end of the file has been reached. The use of this function is shown in the example below:

main35.cpp: Checking end of files

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
   fstream inFile;
   inFile.open("test.txt",ios::in);
   int a;
   while(!inFile.eof()){ // loop until end of file
       inFile>>a;
       cout<<a<<endl;
   }
   return 0;
}
```

The function "eof" will return false until the end of the file is reached and therefore the reading will continue until then.

Do it yourself:

Check what happens if you continue reading even after the end of file in the above example.

Reading binary files

Binary files are special files which contain only 0s and 1s.[19]. The reading and writing of binary files is somewhat different from the writing shown in the previous examples. You also need to know the manipulation of pointers. You might very rarely be asked to use a binary file and therefore we need not get into it but one thing you should remember is to use the "ios::binary" while opening it.

[19] Of course all files are basically a series of 0s and 1s but they are encoded in some format (ASCII for example) but binary files have no encoding they are in some sense just a series of 0s and 1s and therefore very compact. This was useful in the olden days when memory was very costly

As far as possible you try to avoid binary files. You will be using ROOT files for storing most of the data in particle physics which you will learn soon. If at all you need to read or write simple data, follow the simple examples given in this chapter.

Do it yourself:

TIFR wants you to write a program for that stores the details like name, age, department and salary of an employee in a database. Implement it with the use of structures and make the program interactive. The program should asked the user to input the values and should allow the user to decide whether to exit the program.

Classes

From C to C++

The concepts and features we have learned so far are more or less common to C and C++. We shall now learn about classes, the main ingredient of C++. C++ can be called as a higher version of C in that whatever we could do with C++ can be done with C but not in a straight forward way. Data abstraction and object oriented programming are the main features of C++. For a beginner it might seem that the use of C++ is a bit of work around and that most of the things could be achieved with C itself. This is true but the question here is not the achievement but the way it is achieved and the time it takes to achieve. It will take sometime to convince yourself about the power of C++ and probably you might not be convinced at all until you write a program that deserves the power of C++. You should be aware that a language is used only to express an idea and how powerful a language is depends on how easily we can express not only simple ideas but also abstract ideas. That is to say that just because the expressions are easier in a particular language does not necessarily imply that the program will perform better in that language. It depends on the algorithm and other things also. A program written in C++ can be slower than a program written Java if it is not well written. We shall move on with these things in mind.

Built-in and user-defined types

We have so far used the built-in data types like int, float etc., By built-in, we mean that when we do not have to give a description about them anywhere. We also saw how to group data of similar data types using arrays and finally try to mix multiple data types using structures. In structures, we had to describe our data type explicitly, i.e., the structure student was described using built-in data types int and float. Now the structure student is an abstract data type which internally contains the attributes like age, height, weight of a student. Here the structure student is an user defined data type. The

user has defined it according to his convenience (and therefore omits some attributes that he might deem unnecessary in his case).

```
 struct student {
  int age;
  float height;
  float weight;
};
```

We could have easily done away with structures by just having the age, height and weight variables as simple variables and still would have got the program to run successfully.

```
  int age;
  float height;
  float weight;
```

What then is the advantage of structures? The answer is that the program is more structured and organized when the details belonging to an abstract data type are a sort of "tied together" so that the access and manipulation becomes simpler. Expressing the attributes becomes more elegant by using structures. When you want to store details of students of a class room without the use of structures, you would declare some variables like

```
  int age1, age2, age3, age4, age5 ....;
  float height1, height2, height3, height4, height5 ....;
  float weight1, weight2, weight3, weight4, weight5 ....;
```

for the entire class which is not an elegant way.[20]. So far, I have just tried to convince you that using structures makes your program look more concise by tying the elements of an abstract data type like student. Let us look at the other features that structures offer.

[20] Of course, you can put them in arrays to make it look elegant but there is a reason for arranging things in a particular manner. For example, putting apples in one basket and oranges in another one is a good strategy when you want to pack them for selling but when you want to test the quality of fruits coming from a particular country, the best strategy is to make baskets for each country and then fill them with the fruits from that country and do the testing. (Each countries basket may have smaller baskets for each fruit.)

Member functions in structures

We shall now use a structure to define a three vector and see how functions can be implemented in structures:

```
struct Vector{
    int x,y,z; // member variables
    void Initialize(int a, int b, int c); //member function
```

```
};
void Vector::Initialize(int a, int b, int c){ // definition
    x=a;
    y=b;
    z=c;
}
int main () {
    Vector v;
    v.Initialize(1,2,3);
    cout<<v.x<<endl;
    cout<<v.y<<endl;
    cout<<v.z<<endl;
    return 0;
}
```

structureFunction.cpp: Function in a structure

In the above example, "Initialize" is a member function of the structure Vector which sets the values of the three components of a vector. The function is declared inside the structured but defined outside of it. Notice the use of scope operator "::". This operator tells that this function belongs to the structure Vector. This is needed because there could be other structures which might have an "Initialize"function. The scope operator tells us to which structure the function belongs to. The above example is a simple revelation of how the data abstraction, that is, to organize data types that build the abstract structure and implement useful methods to manipulate them and to extract useful information using the the fundamental data types in the structure, brings in simplicity into code development.

From structures to classes

Structure are a subset of so called classes. In the last example we see that the member variables of the structure vector could be accessed using directly by using "v.x" etc., . However for some reasons you want these variables to be hidden from the user or to be manipulated directly by user. Such a thing cannot be done using structures as all the members are "public" i.e visible to everyone. This can be implemented using classes[21] Lets start our first example of classes by implementing the vector structure using a class:

For example, you want to avoid the user from accidently forgetting to initializing one of the values.

[21] and this is not the only thing that can be done using classes.

```
class Vector{
    int x,y,z; // member variables
                public:
    void Initialize(int a, int b, int c); //member function
```

```
};
void Vector::Initialize(int a, int b, int c){ // definition
    x=a;
    y=b;
    z=c;
}
int main () {
    Vector v;
    v.Initialize(1,2,3);
    cout<<v.x<<endl; // error: x is private
    cout<<v.y<<endl; // error: y is private
    cout<<v.z<<endl; // error: z is private
    return 0;
}
```

firstClass.cpp: Example of a simple vector class

As you see, there is not much of difference in implementation of classes and structures. The keyword struct has been replaced by class. There is however another keyword called "public" in the class definition. This keyword tells that the function Initialize is "open to all". There are another couple of keywords which define the access level of the members of a class- private and protected. The access definitions are given below:

Friends and derived classes will be discussed in the next chapter.

keyword	accessible by
private	members of the same class or friends
protected	members of the same class, friends and derived classes
public	all

By default, if any access specifier is not given, the member is given only private access. In the example above, the members x,y,z are given the default access level (private) and therefore are not accessible by the usual method (i.e., using v.x etc.,) and hence the compilation will fail. However they could still be accessed by the function Initialize because it is a member of the same class.

Do it yourself:

Implement some extra methods pertaining to a vector object.

Now let us implement some useful methods for this class vector like a method to get the length of the vector and the dot product between two vector objects. This can be implemented in the following way:

You might have to include the math.h header file for some math operations

```
class Vector{
    int x,y,z;
public:
    void Initialize(int a, int b, int c);
```

```
    float Length();
    float dotProduct(Vector a);
};
void Vector::Initialize(int a, int b, int c){
    x=a;
    y=b;
    z=c;
}
float Vector::Length()
{
return (sqrt((x*x)+(y*y)+(z*z)));
}
float Vector::dotProduct(Vector a)
{
return(x*a.x + y*a.y + z*a.z);
}
```

classVectorMethods1.cpp: The vector class with additional methods

The implementation of the Length method is straight forward. For taking the "dot product"we define a method which takes another vector object and operates on components of that vector. In that method notice how the variable x belonging to the calling object and that belonging to input object are being accessed. In the first case the variables are used directly but in the latter case they are called using the "dot" operator. This distinction should be clearly understood.

Constructors and Destructors

In the previous example we had an explicit method called Initialize to set the values of the vectors. Calling the other methods without calling the Initialize method would give us junk values. Therefore it is good that all classes have such a method which initializes the variables properly and it is better if such crucial methods are not avoided accidentally. This feature is built in C++ using constructors. A constructor can be called as an initialization method of a class. The constructor method has the same name of the class and takes in variables as parameters just like any other function. However a constructor can be called only at the time of construction of an object but not anywhere else. Have a look at the constructor in the example below. The constructor has replaced the Initalize function.

```
class Vector{
    int x,y,z;
public:
```

```
4       Vector(int a, int b, int c); // ctor 1
5       Vector(); // ctor 2
6                   Vector(); // ctor 2
7       ~Vector(); // dtor
8       float Length();
9       float dotProduct(Vector a);
10  };
11  Vector::Vector(int a, int b, int c){
12      x=a;
13      y=b;
14      z=c;
15  }
16  Vector::Vector(){
17      x=0;  y=0;  z=0;
18  }
19  Vector::~Vector()
20  {
21  cout<<"Deleting object..."<<endl;
22  }
23  float Vector::Length()
24  {
25  return (sqrt((x*x)+(y*y)+(z*z)));
26  }
27  float Vector::dotProduct(Vector a)
28  {
29  return(x*a.x + y*a.y + z*a.z);
30  }
31  int main () {
32      Vector v1(1,2,3);// using ctor 1
33      Vector v2;// using ctor 2 (no brackets!!!)
34      cout<<"Length of V1: "<<v1.Length()<<endl;
35      cout<<"Length of V2: "<<v2.Length()<<endl;
36      return 0;
37  }
```

constDestr.cpp: Constructors and Destructors

The main difference is, unlike the Initialize function which can be called anywhere and any number of times, the constructor is called only when the objects are created.

Constructor overloading

A class can have many constructors each taking different arguments, a concept similar to overloading of functions which we saw earlier. In the above example we see the explicit declaration of two construc-

tors one that takes three parameters and another one that take no parameters. To create an object of this class you can use either one of these but notice the way how the object is created using the first constructor and the second constructor. There is no bracket used when created using the second constructor though that constructor definition has them. This is an important point to remember that if the constructor has no parameters you need not use the brackets when you create object.

You will have to use brackets if you create a pointer to the object instead but that will be treated later on.

Default Constructor

What happens if the constructor for a class is not explicitly declared?. The compiler takes over and writes a dummy constructor- one that does nothing. This is called as the default constructor. A default constructor will be made only if there are no existing constructors defined.

Destructor

A destructor is the opposite of the constructor. It is called when an object is deleted. The syntax for defining a constructor is a ~(tilde) followed by the class name as shown in the above example. As you will notice, the destructor is called before the end of the program automatically even without an explicit call to it[22]. Destructor overloading is not allowed in C++.

[22] explicit calls to the destructor can be made using the delete method which will be introduced when we deal with pointers to objects

Pointers to classes

Just like we had pointers to other data types, we can also have pointers to classes. Before that let us see how a pointer to a structure is created and how we can access the variables of a structure using pointers.

```
1  struct Vector{
2      int x,y,z;
3      void Initialize(int a, int b, int c);
4  };
5
6  void Vector::Initialize(int a, int b, int c){
7      x=a;
8      y=b;
9      z=c;
10 }
11 int main () {
```

```
12      Vector v;
13      Vector *pv = &v; // pointer init
14      pv->Initialize(1,2,3);
15      cout<<"Values of x,y,z"<<endl;
16      cout<<v.x<<endl;
17      cout<<v.y<<endl;
18      cout<<v.z<<endl;
19      return 0;
20  }
```

structurePointer.cpp: Pointer to a structure

As demonstrated in the example above, we create a pointer to a structure "pv" as usual using the "*". This pointer is initialised with the address of v using the address operator "&". We call v as a object and pv as the pointer to the object. To access a member function using the object we use the "." (dot) operator. For example to call the "initialize" function using the object, we would have called "v.Initialize(1,2,3)". In the above example we call this function using the pointer instead. Notice the "arrow" operator in place of the "dot". This is the operator to be used to access a member using a pointer.

The same procedure applies for classes as well. However creation of pointer is made in a different way for classes as shown below.

```
Vector *v1 = new Vector(1,2,3);
Vector *v2 = new Vector();
```

Pointer to arrays can also be created using the new keyword. An example: int *parr = new int [5] ;

The new keyword returns a pointer to the class. Notice how the constructors are used here. The second constructor which does not take input parameters has brackets here (unlike when just the object was created). To access a member of the class using the pointer we use the arrow operator $\rightarrow$ instead of the dot operator:

```
v1→Length();
```

An object can and should be deleted so that the memory is freed for other purposes by calling the destructor using the delete keyword:

```
delete v1;
```

When a pointer to a class is created instead of an object, the destructor should be called explicitly failing so will result in what we call as "memory leaks". The delete function should be called when you think that the object will be of no use anymore.

```
1   class Vector{
2       int x,y,z;
```

```
public:
    Vector(int a, int b, int c); // ctor 1
    Vector(); // ctor 2
    ~Vector(); // dtor 1
    float Length();
    float dotProduct(Vector a);
};
Vector::Vector(int a, int b, int c){
    x=a;
    y=b;
    z=c;
}
Vector::Vector(){
    x=0;  y=0;  z=0;
}
Vector::~Vector()
{
    cout<<"Deleting object..."<<endl;
}
float Vector::Length()
{
    return (sqrt((x*x)+(y*y)+(z*z)));
}


float Vector::dotProduct(Vector a)
{
    return(x*a.x + y*a.y + z*a.z);
}
int main () {
    Vector v1(1,2,3);// using ctor 1
    Vector v2;// using ctor 2
    cout<<"Length of V1: "<<v1.Length()<<endl;
    cout<<"Length of V2: "<<v2.Length()<<endl;
    // using pointers
    Vector *v3 = new Vector(1,2,3); // using ctor 1
    Vector *v4 = new Vector();// using ctor 2
    cout<<"Length of V3: "<<v3->Length()<<endl;
    cout<<"Length of V4: "<<v4->Length()<<endl;
    delete v3; // try commenting this out
    delete v4; // try commenting this out
    return 0;
}
```

classPointer.cpp: Pointer to a class

The example above demonstrates the use of pointers to classes. The main things to remember when you work with pointers are:

1. new: to create a pointer
2. $\rightarrow$: to access members
3. delete: to delete and freed the memory used by the object

Do it yourself:
Create a class for a simple two by two matrix and implement all methods pertaining to it like determinant, addition, subtraction etc.,

Operator Overloading

Suppose you make a class for a 2 x 2 matrix, you will include all relevant methods in the class like the determinant. Now, if you want to add two matrices, you cannot just use the "+"operator and get the answer. The "+" operator works directly for integers, floats etc., but when using with classes, it is not so because it is not directly relevant which is to be added to what. However, you make it relevant by the mechanism of function overloading.

```
class twoBytwoMatrix{
    public:
    float a11,a12,a21,a22;
    twoBytwoMatrix(float a, float b, float c, float d);
    twoBytwoMatrix(); // ctor 2
    ~twoBytwoMatrix(); // dtor 1
    float determinant();
    twoBytwoMatrix operator - (twoBytwoMatrix);
    twoBytwoMatrix operator + (twoBytwoMatrix);
};

twoBytwoMatrix twoBytwoMatrix::operator+ (twoBytwoMatrix m)
{
    twoBytwoMatrix out;
    out.a11 = a11 + m.a11;
    out.a12 = a12 + m.a12;
    out.a21 = a21 + m.a21;
    out.a22 = a22 + m.a22;
    return out;
}
twoBytwoMatrix twoBytwoMatrix::operator- (twoBytwoMatrix m)
{
    twoBytwoMatrix out;
```

```
    out.a11 = a11 - m.a11;
    out.a12 = a12 - m.a12;
    out.a21 = a21 - m.a21;
    out.a22 = a22 - m.a22;
    return out;
}
twoBytwoMatrix::twoBytwoMatrix(float a, float b, float c,
                               float d){
    a11=a;
    a12=b;
    a21=c;
    a22=d;
}
twoBytwoMatrix::twoBytwoMatrix()
{
    a11=0;  a12=0;  a21=0; a22=0;
}
twoBytwoMatrix::~twoBytwoMatrix()
{
    // cout<<"Deleting object..."<<endl;
}
float twoBytwoMatrix::determinant()
{
    return ((a11*a22)-(a21*a12));
}
int main () {
    twoBytwoMatrix m1(1,2,3,4);
    twoBytwoMatrix m2(2,3,4,5);
    twoBytwoMatrix m3;
    m3=m1+m2;
    cout<<"Determinant: "<<m3.determinant()<<endl;
    return 0;
}
```

operatorOverloading.cpp: Overloading operators

In the above example, we have overloaded the addition and subtraction operators. The syntax is as follows:

return type operator + (input parameter type)

The "+" operator can be replaced by most of the operators we have learnt so far like -,*,/ etc.,. When the operator is a binary operator (an operator which has operands before and after it) the left hand operand is an object of the class. The type of the right hand operand is given in the input parameter type and the output type is given in the return type.

In the above example we saw that the "+" operator has been overloaded to perform the addition of two matrices. However, note that the same thing can be done by replacing it by the "-" operator. That is you can overload the subtraction operator to perform an addition. This is completely legal and valid but such things are never done. It is always better to overload the operators to perform what their "sign" signifies rather than confusing yourself and other end users eventually.

Static members

In the above example, we created three matrix objects m1, m2 and m3. When each of these objects are created, a memory location is allocated for their members. Now m1 has its set of member data and functions (a11, a12 ... etc.,) and so do m2 and m3. There might be cases where you want to share a variable across all the objects of a class. That is, a variable that has an unique value for all the objects. Such variables are called static variables and they are somewhat like global variables to a class. The way they are declared and initialized is different from the way it is done for other variables. Lets see an example using static variables to keep track of the number of objects created.

```
1
2   class twoBytwoMatrix{
3   public:
4       static int N; // declare inside class
5       float a11,a12,a21,a22;
6       twoBytwoMatrix(float a, float b, float c,
7
8   };
9   twoBytwoMatrix::twoBytwoMatrix(float a, float b,
10                                 float c, float d){
11      a11=a;
12      a12=b;
13      a21=c;
14      a22=d;
15      N++; // incrementing N
16
17  }
18  int twoBytwoMatrix::N=0; // init outside the class
19  int main () {
20
21      twoBytwoMatrix m1(1,2,3,4);
```

```
    cout<<"Creating Matrix Number "<<m1.N<<endl;
    twoBytwoMatrix m2(2,3,4,5);
    cout<<"Creating Matrix Number "<<m2.N<<endl;
    cout<<"The matrix number accessed directly:"<<
            twoBytwoMatrix::N<<endl;
    return 0;
}

```

staticVarcpp: Static Variables

In the above example a static integer is created using the keyword "static" insided the class. A static variable should be initialized only outside the class as shown in the above example. Though the variable was declared as an integer already inside the class, the type (i.e., int)is used while initializing also:

int twoBytwoMatrix::N=0; // init outside class

The variable N is incremented in the constructor and therefore keeping track of the number of objects created. Though there might be many objects created, the value of N will be same for all of them. Static variables can be accessed just like other member variables or using the scope operator "::" as shown in the above example.

The this keyword

Sometimes you need to access the pointer of the object inside the class functions. In such cases you can use the "this" keyword. It returns the pointer of the object that calls that function.

```
twoBytwoMatrix::twoBytwoMatrix(float a, float b,
                               float c, float d){
    a11=a;
    a12=b;
    a21=c;
    a22=d;
   cout<<"This pointer value of this object:"<<this<<endl;
}
int main () {

    twoBytwoMatrix m1(1,2,3,4);
    cout<<"Address of m1: "<<&m1<<endl;
    return 0;
                }
```

this.cpp: The this keyword

More Features in C++

In this section we will review the main features of C++. Although you might not use these features in your research, it is important to have an idea of how these features are used in order to understand codes written by others who would have used these features. The main features are:

- Friendship
- Inheritance
- Polymorphism
- Templates

Friendship

In the last class we saw the use of access specifiers: public, private and protected. The private and protected members can be accessed by friend functions and friend classes. Let us see the use of friend function using examples:

```
class rect{
  float length, breadth;// private mem
public:
  rect(float a, float b){
    length = a;
    breadth = b;
  }
  float GetArea();
  friend float GetLength(rect r);
};
float rect::GetArea()
{
  return length*breadth;
}
```

```
// function outside class
float GetLength(rect r)
{
  return r.length;
}
int main() {
  rect first(2,3);
  cout<<"The area of rect:"<<first.GetArea()<<endl;
  cout<<"The length of rect:"<<GetLength(first)<<endl;
  return 0;
}
```

friendFunction.cpp: Friend function

In the above example, we have made a class for the shape called rectangle which has its length and breadth as its private members. It also has a public method to compute the area of the rectangle. Now the user has implemented a function called Getlength() which returns the length of the rectangle. Since length is a private member it is not accessible by ordinary functions. However, if the function is declared as a friend inside the class, then it can access the private members also. The syntax to declare a function as a friend is as shown in the example above:

friend float GetLength(rect r);

Like friend functions, there can be friend classes too. If a class is declared as a friend, the friend class can access the members of the class that declared as a friend (not vice-versa):

```
class square{
  float length;// private
public:
  square(float a){//ctor
    length = a;
  }
  // give access to rect
  friend class rect;
};
class rect{
  float length, breadth;// private
public:
  rect(float a, float b){//ctor
    length = a;
    breadth =b;
  }
  float GetArea();
  float GetArea(square);
```

```
};
float rect::GetArea()
{
  return length*breadth;
}
float rect::GetArea(square s)
{
  length=s.length;
  breadth=s.length;
  return GetArea();
}
int main() {
  square s(2);
  rect r(3,4);
  cout<<"The area of rect is "<<r.GetArea()<<endl;
  cout<<"The area of square is "<<r.GetArea(s)<<endl;
}
```

friendClass.cpp: Friend class

In the above example, we have implemented a method to calculate the area of a square. This method takes a square object has its input, and calculates the area by setting the length and breadth variables to the length of the square. The GetArea(square s) is able to access the member s.length only because the rect class is declared as friend in square class. If we remove the friend declaration in the square class, the compiler will give out errors complaining that the member length is private to square.

Inheritance

Now we come to an important concept in object oriented programming called inheritance. Inheritance is a method by which classes which share some of the features with other classes can inherit methods and variables common to both of them thus reducing the code size.

```
class shape{
protected:
  float length, breadth;
public:
  shape(){
    cout<<"In Shape ctor 0"<<endl;
    length=0;
    breadth =0;
  }
```

```
10    shape(float l, float b){
11      cout<<"In Shape ctor 1"<<endl;
12      length=l;
13      breadth =b;
14    }
15  };
16  class square:public shape{
17  public:
18    square(float a){//ctor
19      cout<<"In square ctor"<<endl;
20      length = a;
21      breadth=a;
22    }
23    float area(){
24      return  length*breadth;
25    }
26  };
27  class rect:public shape{
28  public:
29    rect(float a, float b):shape(a,b){//ctor
30      cout<<"In rect ctor"<<endl;
31    }
32    float area(){
33      return length*breadth;
34    }
35  };
36  int main() {
37    square s(2);
38    rect r(3,4);
39    cout<<"The area of square is "<<s.area()<<endl;
40    cout<<"The area of rect is "<<r.area()<<endl;
41  }
```

inherit.cpp: Inheritance in classes

In the above example, there is a class called shape which has two variables length and breadth. The shape and rect classes too have these variables and therefore can make use of the shape class using the feature of inheritance. The syntax is as shown in the example:
class rect:public shape

class square:public shape

The first class in the above syntax is called as the derived class and the second class, the base class. The derived class can access the members of the base class as if they if their own member variables. The access specifier public can be replaced by other access specifiers like protected and private.[23] Note that the square and rect classes

[23] If the access level in the base class is less than the one specified while inheriting, then the members are inherited with the inheritance specification. For example, if base class access is public and inheritance specification is private, the members are inherited with private access. If it is the other way round the access specification in the base class in retained.

have no mention of the length and breadth variables in their classes but can access the variables of the shape class. Also note that the members in the shape class have protected access. Private members cannot be accessed by derived classes.

Initializing constructor of a base class using inherited class

How do we call the constructor of the base class using the derived class? In the square class, its constructor is simple has no mention about the constructor of the shape class in which case, the default constructor (ctor 0) of shape is called. In the rect class however, we see that the constructor has a different syntax: rect(float a, float b):shape(a,b)

This constructor takes in two values and calls the constructor of shape (ctor 2) giving these two values as its input.

Multiple inheritance

Classes can inherit multiple classes too. In the example below there is a simple class for printing out the values of an input float variable. The square inherits from the shape and the Print class. As with single inheritance, the square class can access the members of the Print class as if it were is its own member.

```
class Print{
public:
  Print(){
  }
  void PrintValue(float a){
  cout<<"Printing Value:"<<a<<endl;
  }
};
class shape{
protected:
  float length, breadth;
public:
  shape(){
    cout<<"In Shape ctor 0"<<endl;
    length=0;
    breadth =0;
  }
  shape(float l, float b){
    cout<<"In Shape ctor 1"<<endl;
    length=l;
    breadth =b;
```

```
22    }
23  };
24  class square:public shape,public Print{
25  public:
26    square(float a){//ctor
27      cout<<"In square ctor"<<endl;
28      length = a;
29      breadth=a;
30    }
31    float area(){
32      return  length*breadth;
33    }
34  };
35  int main() {
36    square s(2);
37    s.PrintValue(s.area());
38  }
```

multInherit.cpp: Inheritance in classes

Polymorphism

The next feature in C++ is polymorphism. We have seen what a base class is and what a derived class in previous examples. The shape class was a base class and square was a class derived from shape. Let us create a pointer for the derived class square: square *s=new square(2);

Now let us create a pointer for the base class shape: shape *sh=new shape();

Let us proceed to the next logical step. Now since square is a derived class from shape, its pointer is type compatible with shape and therefore we can as well do:

shape *sh=s;

This type compatibility of the pointer of the base class with the derived class is called polymorphism. That is the pointer of the base class can point to the pointer of the derived class. Note however the vice-versa is not true:

square *s=sh;

The above (reverse polymorphism) is not allowed in C++. The following is an example of polymorphism:

```
1   class shape{
2   protected:
3     float length, breadth;
```

```
public:
  shape(){
    cout<<"In Shape ctor 0"<<endl;
    length=0;
    breadth =0;
  }
  shape(float l, float b){
    cout<<"In Shape ctor 1"<<endl;
    length=l;
    breadth =b;
  }
  void PrintValues(){
    cout<<"Length: "<<length<<" Breadth: "<<breadth<<endl;
  }
};
class square:public shape{
public:
  square(float a){//ctor
    cout<<"In square ctor"<<endl;
    length = a;
    breadth=a;
  }
  float area(){
    return  length*breadth;
  }
};
int main() {
  square *s=new square(2);
  shape *sh1= new shape();
  shape *sh2 = s; // type compatible
 // square *s2=sh2;// error
 sh1->PrintValues();
 sh2->PrintValues();
 // sh1->area();// not allowed
}
```

poly.cpp: Polymorphism in classes

Virtual Members

In the above example, thougth there was type compatibility of the base class with the derived class, there was one pitfall in it. The base class pointer thus obtained cannot still access the members of the derived class. square *s=new square(2);

shape *sh2= s;

In the above statements, the base class pointer sh2 points to the derived class pointer s. Though the pointer s can access its member variable area directly:

Allowed

s->area();

the pointer sh2 however cannot access it:

Not allowed

sh2->area();

There is a workaround in such cases: move the area method into the base class. But remember that we will lose the flexibility in doing so for if we implement a another class called triangle where the area is calculated differently, this we lead to a complex code structure. This is where one can use virtual members. If a particular method is common to all the derived classes but have different implementations, a virtual method should be defined in the base class as shown in the example below:

```
using namespace std;
class shape{
protected:
  float length, breadth;
public:
  shape(){
    cout<<"In Shape ctor 0"<<endl;
    length=0;
    breadth =0;
  }
  shape(float l, float b){
    cout<<"In Shape ctor 1"<<endl;
    length=l;
    breadth =b;
  }
  void PrintValues(){
    cout<<"Length: "<<length<<" Breadth: "<<breadth<<endl;
  }
  virtual float area(){
  return -1;
  }
};
class square:public shape{
public:
  square(float a){//ctor
    cout<<"In square ctor"<<endl;
    length = a;
```

```
    breadth=a;
  }
  float area(){
    return  length*breadth;
  }
};
class tri:public shape{
public:
  tri(float a, float b){//ctor
    cout<<"In triangle ctor"<<endl;
    length =   a;
    breadth=  b;
  }
  float area(){
    return  0.5*(length*breadth);
  }
};
int main() {
  square *s=new square(2);
  tri *t=new tri(2,3);
  shape *sh1 =s;
  shape *sh2 =t;

  cout<<"square area: "<<sh1->area()<<endl;
  cout<<"triangle area: "<<sh2->area()<<endl;
}
```

virtual.cpp: Virtual methods in classes

In the above code, the two classes square and tri have a common method called area which are implemented differently. For the base class to access these members, a virtual method for area has been defined in the base class.

```
  virtual int area ()
      { return (0); }
```

The above virtual function is a sort of dummy function which has a simple implementation which returns 0. We can also remove this implementation using a statement like:

virtual int area () =0;

A virtual function which has no implementation is called a pure virtual function and a class that contains a pure virtual function is called a abstract base class.

Templates

Function templates are special functions than can adapt themselves to different data types.

```
template <class T>
T Subtract (T a, T b) {
  T diff= a-b;
  return diff;
}
template <class T, class U>
U Subtract2 (T a, U b) {
  U diff= a-b;
  return diff;
}

int main () {
  float a,b;
  int c,d;
  a=2.2;
  b=5.6;
  c=3;
  d=4;
  float d1=Subtract<float>(a,b);
  int d2= Subtract<int>(c,d);
  float d3=Subtract2<int,float>(c,a);
  cout<<"Diff of floats: "<<d1<<endl;
  cout<<"Diff of int: "<<d2<<endl;
  cout<<"Diff of int and float "<<d3<<endl;
  return 0;
}
```

functionTemplate.cpp: Function Templates

In the above code we have declared a template function called Subtract which takes in two variables of type T and returns the same type T, the difference between the inputs. T is just a place holder and any other letter can be used in its place. The statement: template <class T>

tells that the function following takes a template variable T. The Subtract function can take any of the data types now by replacing the template variable: float d1=Subtract<float>(a,b);

takes float as its inputs and

int d2= Subtract<int>(c,d);

takes int as its inputs. A template function can takes different types as its input also by declaring a statement like:

template <class T, class U>

The above statement tells that the following function will use two template variables for its implementation. The Subtract2 function can take an int and a float as an input and return a float as in the following statement:

float d3=Subtract2<int,float>(c,a);

Just like we had function templates, we can also have class templates which uses template parameters.

```
template <class T>
class square{
  T side;
    public:
  square(T a){//ctor
    side=a;
  }
  T area(){
    return a*a;
  }
};
int main () {
  square<int> s1(2);
  square<float> s2(3.3);
  cout<<"area of s1 "<<s1.area()<<endl;
  cout<<"area of s2 "<<s2.area()<<endl;
  return 0;
}
```

classTemplate.cpp: Class Template

In the above code, a template variable T is defined and the class square makes use of this template variable. The template variable is replaced by the user provided data type in the following statements:

square<int> s1(2);

square<float> s2(3.3);

The first statement replaces the template variable T by int for all instances of its object and the second statement replaces the template variable T by float.

ROOT: Introduction and Installation

This is the beginning of the second part of this lecture notes where we will discuss about ROOT. ROOT is a famous open source data analysis and graphing library developed by CERN. If you are a high energy physicists you will probably be spending a good amount time in writing codes using ROOT. ROOT has almost all the features that you will require in your research work. Examples are histograms, graphs, profile plots, 2d plots, fitting, image manipulation, random number generation, FFT, signal analysis, PCA etc.,.[24] ROOT is not an application as such like MS Excel where you just open your data files and start plotting. ROOT offers only the libraries that is to say the header and source files required for plotting and data analysis. It is your job to compile the source code of ROOT and then link it to your program. I shall not be able to cover all the features of ROOT owing to the limited time of this course. Once you get acquainted with the way ROOT works, using others features will not be a big problem.

[24] For a complete list, please refer to the ROOT website.

If you have forgotten what a library is and how they are done, please go to the second chapter and review it.

Installing ROOT

The installation of ROOT (in Linux) consists of three steps which will be explained briefly in the following subsections. The sections will not describe advanced features as it is mainly aimed at beginners trying to learn ROOT. If you need more details about it you can refer to the ROOT website.

Downloading ROOT

The source code of ROOT is available for free download at:
http://root.cern.ch/drupal/content/downloading-root

ROOT regularly releases new updates and patches for bugs found in their released source code. In the link above, you will find a huge list of ROOT versions available for download. It is usually recommended to use the latest version to make sure that you up to date with the features. However, it might be the case that the codes written for your experiment depend on a specific version of ROOT in

which case you should download the appropriate version.[25] Once you have clicked on the appropriate version, you will be taken to a page that gives you options to either download the complete source tree or the binaries. To have a stable version on ROOT and to prevent any library conflicts, it is always better to download the source tree. The source tree contains all the header and source file of ROOT in a compressed format (.tar.gz)

[25] The developers will remove/add new features depending on the reviews and requests they get on their forum. Should you find something that does not work as expected in ROOT you can contact the developers by writing in their forum.

Compiling ROOT

Once you have downloaded the ROOT source tree, follow the steps here:

1. Un zip/tar in a convenient location and the contents will be stored in a folder called root, by default.
2. cd to the root folder
3. type ./configure
4. make -j 4

As described in Chapter 2, one can create objects of functions (with declarations in the header file and definitions in the source file) so that another program can easily link to these objects without having to re-compile the sources and headers[26]. The rules for compiling ROOT will depend on the Operating System and the libraries that are installed in your system. These rules are generated automatically by the ./configure command which will generate a "makefile". The makefile contains the commands for compiling the sources and headers for your system. The number 4 in the make command is the number of cores that will be used for the compiling process. You can change it to the number of cores in your machine. Once you type the make command, the compilation process will start and you see lines of compile commands starting with g++. This is similar to what you did for compiling the codes in the examples. ROOT contains thousands of files and each has to compiled and therefore it takes sometime to complete. Sometimes, the compilation process might stop with an error. In that case, it might be that you have not installed some libraries on which the compilation of ROOT depends. Check: what the error is from the compilation output and try to check where the error first started by scrolling up. That line will show you some information like for example "-lGl not found" in which case you should install the GL library. Once you have installed it, you can restart the compilation by typing the "make -j 4" command. The compilation will restart from the place where it last stopped.

[26] Compilation of ROOT may take more than 30 minutes

Setting the environment variables

Once the compilation is successfully done, you have to set the environment variables for proper operation of ROOT. Your ROOT programs will use these environment variables to locate the proper location of ROOT libraries and other configuration files. The main environment variables is:

ROOTSYS

which should point to the folder where the root folder is present. For example, if you have extracted the source tree in the folder "/home/myprograms/root" then you should set the ROOTSYS variable like this:

export ROOTSYS=/home/myprograms/root/ [27]

in a terminal.The system environment variables like:

LD_LIBRARY_PATH and

PATH

have to be modified by the appending the library folder location and the bin folder location of ROOT respectively. If you have set your ROOTSYS variable using the method above, then proceed to do the following in the same terminal where you set the ROOTSYS variable:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib

export PATH=$PATH:$ROOTSYS/bin

There is however a shortcut to do all the environment settings of ROOT. The following file

$ROOTSYS/bin/hisroot.sh (csh)[28]

(where $ROOTSYS is the folder where ROOT was installed) contains all the commands required to set the environment variables for ROOT. You should execute this file using the commands:

export ROOTSYS=/home/myprograms/root/

source $ROOTSYS/bin/thisroot.sh

in a terminal. You should remember that this should be done every time you open a new terminal. In case you want it to be set permanently so that you do not need to set the variables every time you open a new terminal, you can do so by opening your .bashrc file adding the above command in it. Follow the commands to do that:

cd

gedit .bashrc

The .bashrc file will be opened and in it you should add the following lines:

export ROOTSYS=/home/myprograms/root/

source $ROOTSYS/bin/thisroot.sh

and save the file.

In case you have multiple versions of ROOT in your system then you should avoid using this method and try to invoke the right script

[27] In some machines, you would have to use the command setenv instead of export and the syntax for setenv is a bit different from export. This depends on the shell you use. If you use the Bourne shell then you should use export and if you use the C shell then you should use the setenv command. More about setting environment variables in different shells can be found here

[28] If your shell is a Bourne shell use the .sh file and if yours is a C shell, use the .csh file. These are script files which contain commands for various operations for proper operation of ROOT

You can ignore the export command and set the ROOTSYS variable in the source command to the actual location of your ROOT installation.

yourself in the terminal using the previous method. Now you are all set to run ROOT. In the terminal, type root and you will enter into the ROOT terminal. Type: TCanvas t and press Enter and you will see a canvas popping up. Enter .q in the terminal and you will be out of the ROOT terminal.

Working with ROOT

There are two ways to work with ROOT,

1. Using CINT interface which will not be covered in this lecture
2. Using the traditional C++ way which we shall discuss in detail

Using the CINT interface can be useful while developing the code where you will trying out all options until your code works properly. You need not compile the program after it has been changed. However CINT interface is an interpreter module which is way slower that a compiled program and therefore do not use it for running huge programs. I will present one example here to show how it works:

```
#include <iostream.h>
int rootTest()
{
   cout<<"Hi welcome to ROOT CINT interface"<<endl;
   return 0;
}
```

rootTest.C: A macro for testing ROOT CINT interface

Let us see how the code above can be run using the ROOT CINT interface. In a terminal, cd to the location where the above file is present and start the ROOT CINT interface by typing “root” and then type:

.x rootTest.C

The program will be executed and you should see the output. You can exit the root terminal by typing:

.q

You can use any of the ROOT classes in these files which will be executed by the ROOT CINT interface. More ways to ron a ROOT program can be found in this wiki link.

Programming with ROOT: Starting with Histograms

Lets start our first example of ROOT by opening a canvas. A canvas is a sort of drawing board for ROOT plots on which all the graphs, histograms etc., will be drawn. The TCanvas class is responsible for all canvas related operations like opening, dividing it into subpads etc.,.

```
1  #include <iostream>
2  #include <TApplication.h>
3  #include <TCanvas.h>
4  using namespace::std;
5  int main()
6  {
7    TApplication *t = new TApplication("myapp",0,0);
8    TCanvas *C = new TCanvas;
9
10   t->Run();
11   delete C;
12   return 0;
13 }
```

canvas.cpp: A simple program to instantiate a ROOT canvas

If you try to compile the above program with the usual compile command:

g++ canvas.cpp

The compiler will complain that it could not find the header files (TApplication, TCanvas etc.,). These file reside in the include folder present in the location where ROOT is installed ($ROOTSYS/include). You will have to inform the compiler about this location so that it can search for these file there. The flag used for the search directory for include files is "-I". The compile command now looks like:

g++ canvas.cpp -I$ROOTSYS/include

Executing the above command gives errors like "undefined reference to TApplication::TApplication" etc.,. This is because though the header files where the functions were declared are given, the pro-

gram cannot find the definitions for those functions. We have already compiled ROOT and have made the objects files for all the files so we need to link to those object files (or libraries) using the flag -l. The flag used for the search directory for library files is -L. The compile command looks like:

g++ rootTest.C -I$ROOTSYS/include -L$ROOTSYS/lib -lCore -lGpad

where $ROOTSYS/lib denotes the location where the ROOT libraries are located and libCore and libGpad are the libraries where the definitions for the TApplication and TCanvas classes are found.If you use other classes, you should link to the appropriate libaries Now you will see the program compiles successfully and you can run it using:

./a.out

When the -l flag is used the "lib" prefix of the libraries is dropped as explained in Chapter 2.

Getting the correct compiler flags for program using ROOT libraries

As you saw earlier, you need to know the correct flags for the include directory and for linking to the correct libraries. There is easier and faster way of doing this. ROOT provides a program called "root-config" to output these flags for your system. This program is found in the folder:

$ROOTSYS/bin

If you just execute "root-config" on a terminal, it will shown the options for using the program. FOr example using the option "–glibs" will give you the libraries required for linking and "–cflags" will give you other options requires for compiling like the include folder path etc.,.Type:

root-config –glibs

and you will see an output containing the libraries to be linked to for a root program:

"-L/home/myprogram/root/lib -lGui -lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -lPhysics -lMathCore -lThread -pthread -lm -ldl -rdynamic"

You can use execute the root-config program directly in your compile command and automatically append these flags in this manner:

g++ canvas.cpp ‘root-config --glibs --cflags‘

The ‘ symbol is not the symbol of single quote but of the accent found close the 1 key (at least in my computer) and there is no space in between the two dashes.

Plotting a histogram

Let us move to a real world example where we will plot a distribution of a variable using a histogram. Let us see how to generate a gaussian distribution and poisson distribution of random number and plot it in a canvas using ROOT.

```
#include <iostream>
#include <TApplication.h>
#include <TCanvas.h>
#include <TH1F.h>
#include <TRandom.h>
using namespace std;
int main()
{
TApplication app("my_app",0,0);
TCanvas *c = new TCanvas("myCanvas","My Canvas");
TRandom *ran = new TRandom(450);
TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
TH1F *h2 = new TH1F("my_hist2","Poisson",50,-100,100);
h1->SetFillColor(4);
h2->SetFillColor(6);
for(int ii=0;ii<10000;ii++){
h1->Fill(ran->Gaus(0,100));
h2->Fill(ran->Poisson(40));
}
c->Divide(2,1);
c->cd(1);
h1->Draw();
c->cd(2);
h2->Draw();
gPad->Update();
app.Run();
delete ran;
delete c;
return 0;
}
```

histo1.cpp: Histograms of a gaussian and a possion distributed random numbers.

For the above job, we will use the following three classes in ROOT:

1. TApplication: creates the ROOT application environment
2. TCanvas: Opens a canvas
3. TRandom: Generates random numbers
4. TH1F: Histogram class

In the above code, we create an object of TApplication which should be done only once in your program. Without the TApplication class, your canvases will not be shown. The Run method should be called

before you return from your code. Next we create a pointer to a TCanvas object using its constructor which takes a alias name as the first parameter and the canvas title as the second parameter. Next we create a pointer to the random number generator class TRandom. This class gives you useful methods to generate different types of distributions. The constructor takes an input value as its seed. Next we create two pointers for creating histograms, one for the gaussian and another for the poisson. The constructors syntax is:

All root classes derive from the base class called TObject and most of ROOT objects will have an alias name which can be used as an handle to access the object.

alias name, title, number of bins, lower limit and upper limit

Of course there are other constructors as well about which you can learn from the ROOT documentation. Next, we fill the histgrams with 10000 entries in a loop. In each loop the following functions:

```
h1->Fill(ran->Gaus(0,100));
h2->Fill(ran->Poisson(40));
```

return a random number obeying the appropriate functional distribution. The first one returns a random number from a gaussian distribution with mean 0 and sigma 100. The second one returns a random number for a poission distribution with mean 40. The next thing is to plot these two histograms. Before that we need to subdivide the canvas which we do using:

```
c->Divide(2,1);
```

The canvas is now split into sub-pads with 2 columns and 1 row. We will plot the gaussian in the first pad and therefore we need to do:

```
c->cd(1);
```

which makes the first pad active and then:

```
h1->Draw();
```

The poission function will be plotted in the next sub-pad so we need to set the second sub-pad as active by doing:

```
c->cd(2);
```

and then we draw the second histogram:

```
h2->Draw();
```

The fill color of the histograms are set by the functions:

```
h1->SetFillColor(4);
h2->SetFillColor(6);
```

The Draw function of the histogram can take arguments which will change the plotting style.

Saving a canvas

Once you have plotted anything on a canvas, you can save them in a familiar format like pdf, png, jpg etc.,.You can use the Print function in the TCanvas class like this:

```
c->Print("canvas.pdf");
```

to generate a pdf file. The canvas file menu also has a Print or Save option which also helps you to do it.

Histograms and Graphs

Plotting histograms

In this chapter we shall see what we options exists for plotting histograms and graphs. These things are the most often used objects and therefore it is necessary to have an idea of what can be done with them.

Weighing your histogram

When you fill a histograms with values, the bin in which that value falls in is incremented by one. Sometimes, you may want to give more importance to some values and less importance to some values. This can done by giving weights to the values.

```
int main()
{
  //App instance
  TApplication app("my_app",0,0);
  // create canvas and divide
  TCanvas *c = new TCanvas("myCanvas","My Canvas");
  c->Divide(2,2);
  // create histograms
  TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
  TH1F *h2 = new TH1F("my_hist2","Poisson",50,-100,100);
  TH1F *h3 = new TH1F("my_hist3","Weighted Gauss",50,-500,500);
  TH1F *h4 = new TH1F("my_hist4","Weighted Poisson",50,-100,100);
  // set styles
  h1->SetFillColor(4);
  h2->SetFillColor(6);
  h3->SetFillColor(5);
  h4->SetFillColor(7);
  // create random number and fill histogram
  TRandom *ran = new TRandom(450);
  for(int ii=0;ii<10000;ii++){
```

```
    h1->Fill(ran->Gaus(0,100));
    h2->Fill(ran->Poisson(40));
    h3->Fill(ran->Gaus(0,100),2);// weighted by 2
    h4->Fill(ran->Poisson(40),2);// weighted by 2
  }
  // draw histograms
  c->cd(1);
  h1->Draw();
  c->cd(2);
  h2->Draw();
  c->cd(3);
  h3->Draw();
  c->cd(4);
  h4->Draw();
  // setting the axis title for h1
  h1->GetXaxis()->SetTitle("Gauss Values");
  h1->GetYaxis()->SetTitle("Counts");
  gPad->Update();
  app.Run();
  delete ran;
  delete c;
  return 0;
```

histo2.cpp: Adding weights in a histogram.

The above example illustrates the use of weights. As you can see, the second argument of the Fill function gives the weight. By default, it is set to one. Histograms h1 and h2 are filled with no weights and in h3 and h4 all values are filled with a weight of 2. The number of counts (not the entries, see the y axis) in the histograms h3 and h4 (Refer Fig.10) are therefore double the counts in the histograms h1 and h2.

The axis title of an histogram can be set using the functions GetXaxis or GetYaxis and then SetTitle("title"), as shown in the example.

Adding histograms

Just like you add ordinary numbers, you can also add histograms too. In histograms, the number of counts in a particular bin in the first histogram is added to the number of counts in the same bin in the second histogram. One has to be careful here while setting the bin size and the number of bins in the histograms. They should be similar for both the histograms or you will end up getting wrong results.

```
int main()
{
  // app instance
```

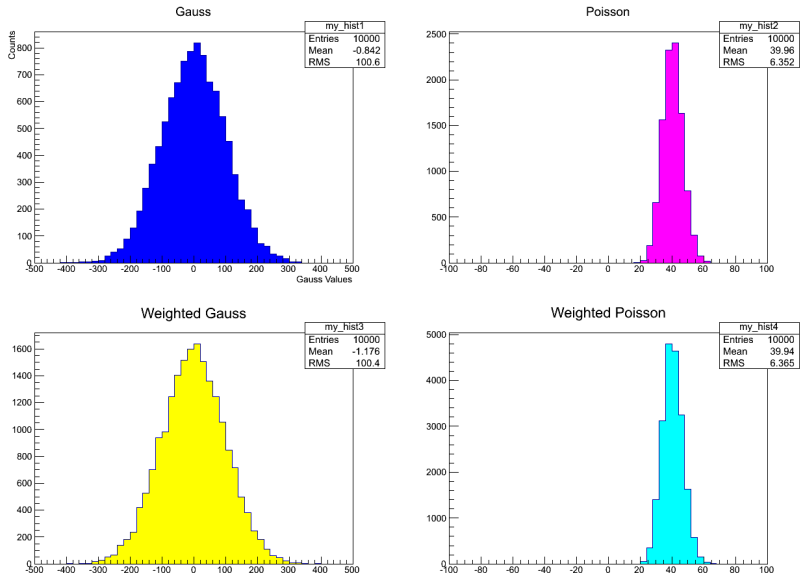

Figure 10: Example of weighted histograms.

```
4   TApplication app("my_app",0,0);
5   //canvas creation
6   TCanvas *c = new TCanvas("myCanvas","My Canvas");
7   c->Divide(1,3);
8   // histogram creation
9   TH1F *h1 = new TH1F("my_hist1","Gauss1",50,-500,500);
10  TH1F *h2 = new TH1F("my_hist2","Gauss2",50,-500,500);
11  TH1F *h3 = new TH1F("my_hist3","Gauss1+Gauss2",50,-500,500);
12  // styles
13  h1->SetFillColor(4);
14  h2->SetFillColor(6);
15  h3->SetFillColor(5);
16  //fill random values
17  TRandom *ran = new TRandom(450);
18  for(int ii=0;ii<10000;ii++){
19    h1->Fill(ran->Gaus(0,100));
20    h2->Fill(ran->Gaus(0,100));
21  }
22  c->cd(1);
23  h1->Draw();
24  c->cd(2);
25  h2->Draw();
26  // add histograms
27  h3->Add(h1,h2);
28  c->cd(3);
```

```
29    h3->Draw();
30    gPad->Update();
31    app.Run();
32    delete ran;
33    delete c;
34    return 0;
35  }
```

histo3.cpp: Adding two histograms.

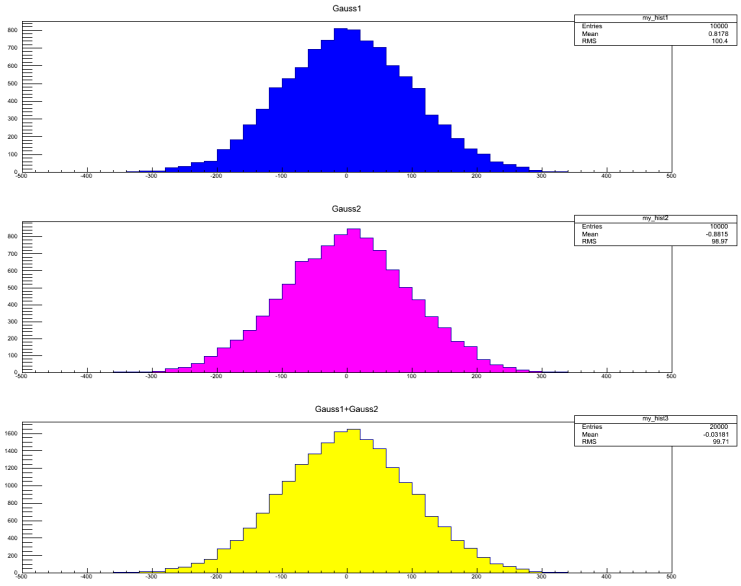

Figure 11: Adding two histograms

In figure 11, the top one is the first histogram and the middle one is the second histogram, having the same bin size and range as the first one. The bottom one is the result of these two histograms. Notice that the counts in the third for each bin is equal to the counts in the first plus the counts in the second.

Plotting multiple histograms on a single canvas

Sometimes you want to plot multiple histograms one on top of each other. This can be achieved in two ways: one is using the "SAME" option and second is using the THStack Class. Examples are given below:

```
1   TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
2   TH1F *h2 = new TH1F("my_hist2","Gauss",50,-500,500);
3      // styles
4      h1->SetFillColor(4);
```

```
    h2->SetFillColor(6);
    // fill histograms with random numbers
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<10000;ii++){
        h1->Fill(ran->Gaus(0,50));
        h2->Fill(ran->Gaus(0,100));
    }
    //draw  first
    h1->Draw();
    // draw over the last
    h2->Draw("SAME");
}
```

histo4.cpp: Plotting multiple histograms using the “SAME” options.

The option “SAME”is given as an argument to the Draw function of the histogram. The first histogram drawn on a pad should not use this option. The same option should be used carefully and only if a histogram is already plotted on the canvas otherwise you are likely to end up in a crash. The “SAME”option is not always useful

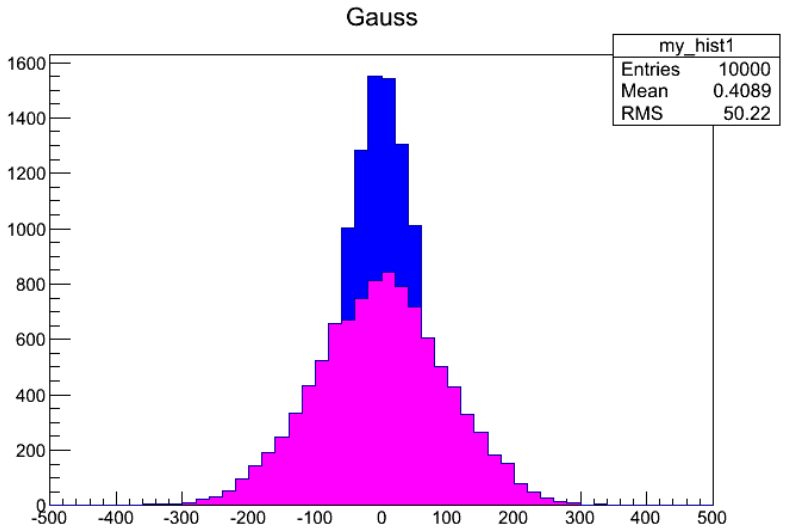

Figure 12: The "SAME" option for drawing mulitple histograms.

especially in the management of the axis. The axis scale is managed by the one that was plotted first and therefore any histograms plotted thereafter will have to be within that range. To overcome this, you can use the THStack class.

```
  TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
    TH1F *h2 = new TH1F("my_hist2","Gauss",50,-500,500);
    // styles
```

```
    h1->SetFillColor(4);
    h2->SetFillColor(6);
    // fill histograms with random numbers
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<10000;ii++){
        h1->Fill(ran->Gaus(0,50));
        h2->Fill(ran->Gaus(0,100));
    }

    THStack hs("Stack","Stack of my histograms");
    hs.Add(h1);
    hs.Add(h2);
    hs.Draw(); //try "nostack" option
}
```

histo5.cpp: Plotting multiple histograms using the THStack class.

For using the THStack class, you should have your histograms created and call the Add function giving the pointers as the argument as shown in the above example.

Two dimensional histograms

The histogram class TH1F is for one dimensional binning. To plot a 2D histogram, you should use the TH2F class. It is almost similar to the 1D class except for the fact that the Fill function take two arguments x and y. Of course, the weight can be given as the third argument. The constructor here takes the binning information for the yaxis also.

```
   TH2F *h1 = new TH2F("my_hist1","Gauss",50,-100,100,50,-100,100);
    // fill histograms with random numbers
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<50000;ii++){
        h1->Fill(ran->Gaus(0,50),ran->Gaus(0,50));
    }
    // draw histogram
    h1->Draw("SURF2");
}
```

histo6.cpp: Plotting a 2D histogram.

The Draw function takes the drawing options which are listed in the ROOT documentation.

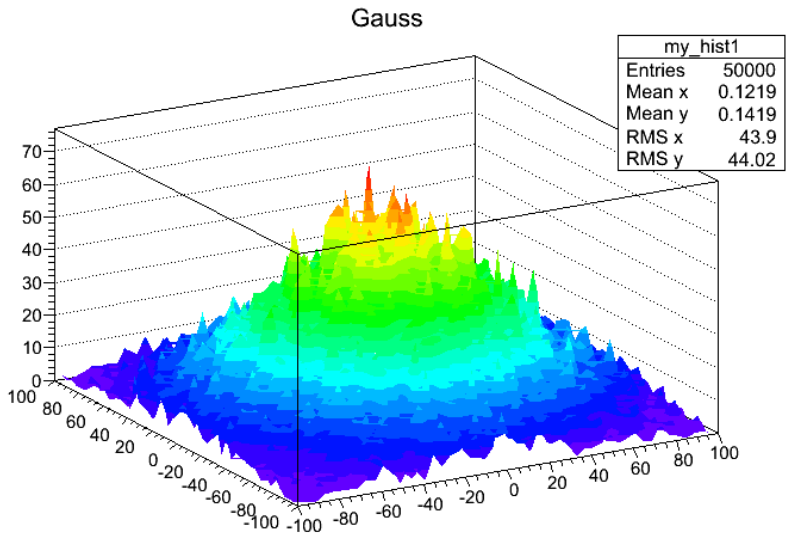

Figure 13: A 2D histogram using the TH2F class.

Profile histograms

Profile histograms are a replacement for 2d histograms and are plotted using the TProfile class. The x axis contains the x bins and the y axis shows, the mean of y and the corresponding error as against the y bins in a 2d histogram.

```
    TProfile *hprof  = new TProfile("hprof","Profile of pz versus px",100,-4,4,0,20);
  // fill histograms with random numbers
    TRandom *ran = new TRandom(450);
    float px,py,pz;
    for(int ii=0;ii<50000;ii++){
        //Return 2 numbers distributed following a gaussian with mean=0 and sigma=1.
        ran->Rannor(px,py);
        pz = px*px + py*py;
        hprof->Fill(px,pz,1);
    }
    // draw histogram
    hprof->Draw();
}
```

histo7.cpp: Plotting a profile histogram.

Filling is similar to a 2D histogram but the constructor is slightly different from the TH2 class in that it does not take the binning information for the y axis as it is not required. The drawing options are listed in the source code.

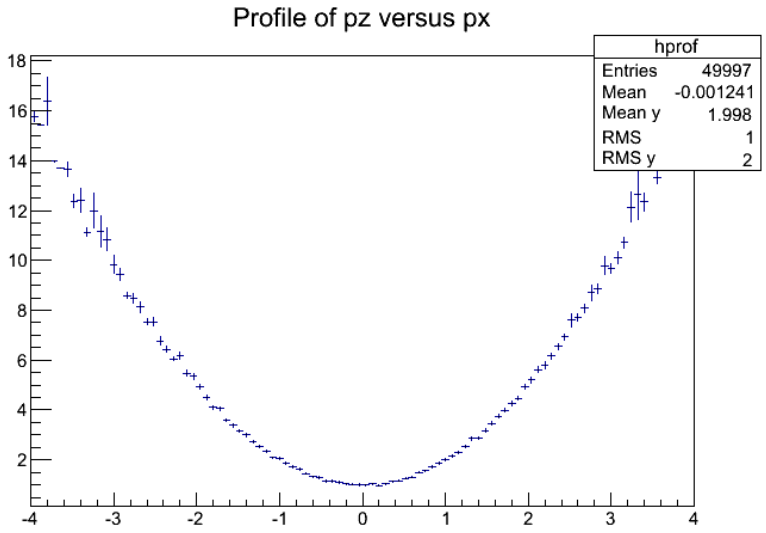

Figure 14: A profile histogram.

Fitting a histogram

The next part of histogram is the fitting. The fit panel can be accessed by right clicking on the histogram data points in the canvas. To do it in C++, you should first create the function to fit using the TF1 class. There are some pre-defined functions like gauss, polo, pol1, pol2, expo, landau etc., which do not require any initialization. However if you user your own function for fitting, you should initialize the parameters.

```
1  TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
2    // styles
3    h1->SetFillColor(4);
4    // fill histograms with random numbers
5    TRandom *ran = new TRandom(450);
6    for(int ii=0;ii<10000;ii++){
7        h1->Fill(ran->Gaus(0,50));
8    }
9    //draw  first
10   h1->Draw();
11   // change style to show fit parameters
12   gStyle->SetOptFit(1011);
13   // Fitting using pre-defined function
14   //h1->Fit("gaus");
15   // Fitting using user-defined functions
16   TF1 *func = new TF1("myGauss","[0]*exp(-0.5*((x-[1])/[2])**2)",-100,100);
```

```
17      // parameter names
18      func->SetParNames("Factor","Mean","Sigma");
19      //Set Parameters:
20      func->SetParameter(0,1);
21      func->SetParameter(1,0);
22      func->SetParameter(2,50);
23      h1->Fit(func);// use option "R" to restrict to a certain region for fitting
24      // fixing parameters: FixParamter, SetParamterLimits
25     //To get the paramter values programmatically:
26      cout<<"Parameter Factor: "<<func->GetParameter(0)<<endl;
27      cout<<"Parameter Mean: "<<func->GetParameter(1)<<endl;
28      cout<<"Parameter Sigma: "<<func->GetParameter(2)<<endl;
29      // update
30  }
```

histo9.cpp: Fitting a histogram.

In the above example, the user has defined his own gaussian function using the TF1 class and initializes the parameters using the SetParameters function. After this, the Fit function of the histogram class is called with the TF1 pointer as its argument. The option "R" can be used to restrict the fitting range to the one set in the TF1 pointer. The fitting results can be accessed by the GetParameter function.

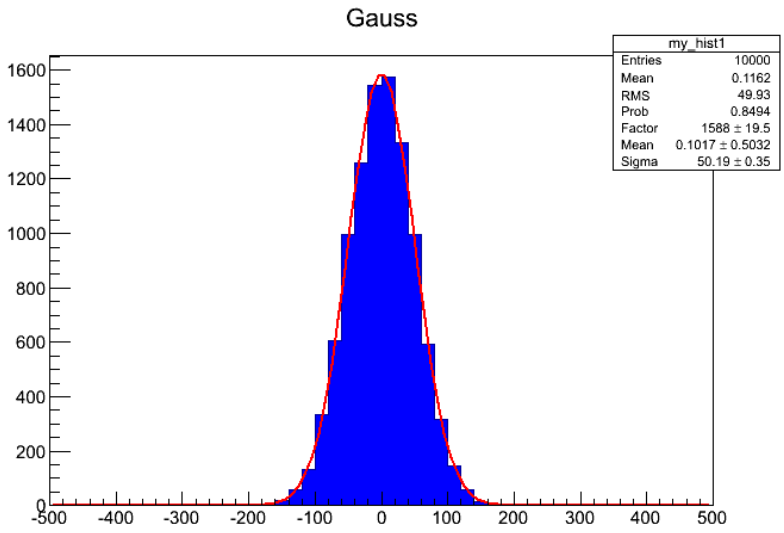

Figure 15: Fitting a histogram.

Graphs

Graphs are plotted using the TGraph class. Unlike histograms, there is no binning here. There are manys to fill a graph one of them is

to fill the x and y arrays and give them as a input to the TGraph constructor.

```
int n =5000;
double x[n], y[n];
// create random number and fill graph
TRandom *ran = new TRandom(450);
for(int ii=0;ii<n;ii++){
    x[ii]= ran->Gaus(0,50);
    y[ii]=x[ii]*x[ii];
}

TGraph *graph  = new TGraph(n,x,y);
            // another way of filling individually
            //graph->SetPoint(ii,xval,yval);
// draw histogram
        graph->Draw("A*"); //options L,A,F,C,*,P,B
```

histo9.cpp:Plotting a graph.

Like the histogram classes, the TGraph class also takes drawing options.

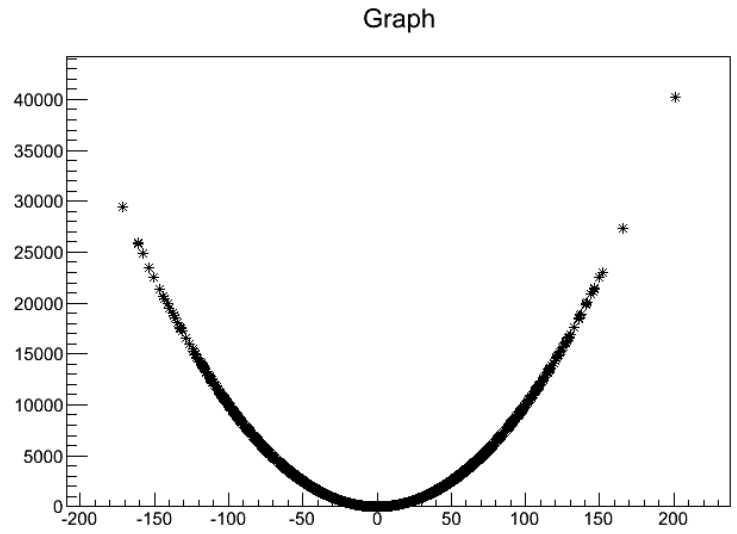

Figure 16: Plotting a graph.

Plotting multiple graphs on a single canvas

Just like we plotted multiple histograms using the “SAME”option, we can plot multiple graphs too using it. When doing so, the option “a” should be added along with a plotting style (”a*”, “al” etc.,) to the

first graph that is drawn and the "SAME" option should be added for the following graphs ("lsame","*same" etc.,). The "a" option should be used only for the first graph.

Similar to the THStack, there is the TMultiGraph class which takes care of all the axis management for plotting multiple graphs.

```
int n =5000;
  double x[n], y[n];
  double x1[n], y1[n];
  // create random number and fill graph
  TRandom *ran = new TRandom(450);
  for(int ii=0;ii<n;ii++){
    x[ii]= ran->Gaus(0,50);
    y[ii]=x[ii]*x[ii];
    x1[ii]= ran->Gaus(0,150);
    y1[ii]=x1[ii];
  }
  TGraph *graph1  = new TGraph(n,x,y);
  TGraph *graph2  = new TGraph(n,x1,y1);
  graph1->SetLineColor(kBlue);
  graph2->SetLineColor(kRed);

  TMultiGraph *m = new TMultiGraph();
  m->Add(graph1,"lp");
  m->Add(graph2,"lp");
  m->Draw("a");
```

The graphs are added to the TMultigraph object with their appropriate drawing options, however without the "a" option. The "a" option has to be given only while drawing the multigraph.

Plotting a 2D graph

If you want to plot a surface or a 2d graph, then the TGraph2D class in used. The filling can be done using an array like the TGraph class or by using the SetPoint function:

```
    TGraph2D *dt = new TGraph2D();
//    // create random number and fill values
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<n;ii++){
        x = 2*6*(ran->Rndm(ii))-6;
        y = 2*6*(ran->Rndm(ii))-6;
        z = (sin(x)/x)*(sin(y)/y)+0.2;
```

```
8           dt->SetPoint(ii,x,y,z);
9
10      }
11  //    gStyle->SetPalette(1);
12      dt->Draw("surf1");
```

histo10.cpp:Plotting a 2D graph.

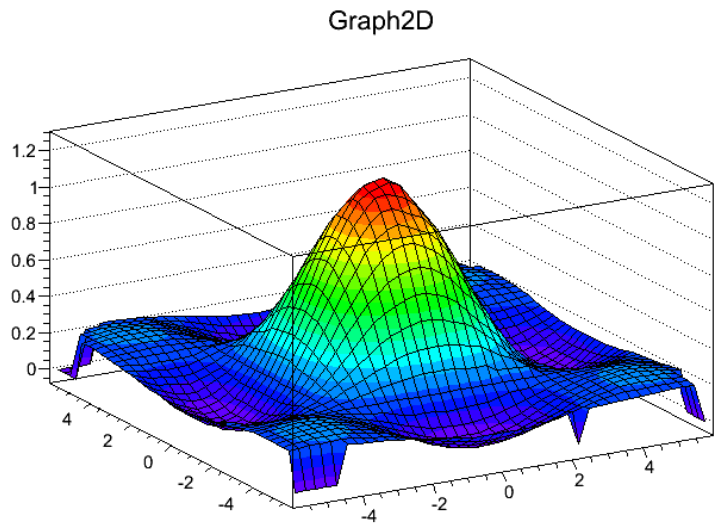

Figure 17: Plotting a 2D graph using the TGraph2D class.

Fitting a graph

Fitting can be done following the same procedure illustrated for histograms.

The TFile class for ROOT objects storage

Like we saved ASCII data in text file, there is a way to save ROOT objects like graphs, histograms and trees. The class used for storing is called TFile. Let us begin with a small example of saving two histograms into a TFile:

```
1   TCanvas *c = new TCanvas("myCanvas","My Canvas");
2   // create file
3   TFile *f = new TFile("rootfile.root","RECREATE","My root file");
4   // create histograms
5   TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
6   TH1F *h2 = new TH1F("my_hist2","Gauss",50,-500,500);
7   // set style
8   h1->SetFillColor(4);
9   h2->SetFillColor(6);
10  // fill histograms
11  TRandom *ran = new TRandom(450);
12  for(int ii=0;ii<10000;ii++){
13      h1->Fill(ran->Gaus(0,50));
14      h2->Fill(ran->Gaus(0,100));
15  }
16  h1->Draw();
17  h2->Draw("SAME");
18  gPad->Update();
19     cout<<"Press anything to quit..."<<endl;
20  cin>>a;
21  // close file
22  f->Write();
23  f->Close();
```

filewrite.cpp: A simple program to write histograms to a root file.

As usual we instantiate the TApplication class and then create the TCanvas for plotting our histograms. Now, before we create the histograms, we should create the TFile object. Whatever ROOT objects created after the creation of TFile object will belong to that file. The first parameter is the file name and the second parameter is an option

which describes the access options and the third parameter is the title of the file. The access option can be one of the following:

NEW or CREATE: create a new file and open for writing. If the file already exists it is not opened.

RECREATE: Same as NEW or CREATE but if the file already exists it is overwritten.

UPDATE: Open an existing file for writing. If no file exists, it is created.

READ: Open file for reading (default).

Other options exist and can be seen in the ROOT documentation.

As shown in the example, after the file is created, the histogram pointers are created and therefore these histograms belong to that file. The histograms are filled with values. The objects are written to the file using the Write() function and then closed using the Close() function. Let us see how to read the contents of a TFile.

```
1  TCanvas *c = new TCanvas("myCanvas","My Canvas");
2   // create file
3   TFile *f = new TFile("rootfile.root","READ","My root file");
4   // read objects by type casting
5   TH1F *h1 = (TH1F *)f->Get("my_hist1");
6   h1->Draw();
7   gPad->Update();
8   cout<<"Press anything to quit..."<<endl;
9   cin>>a;
10  // close file
11  f->Close();
```

fileread.cpp: A simple program to read a histograms from a root file.

To read, you should open the file using the "READ" option. You should also know the object name and the object type that you want to read from. In the above example, we are reading from the file created using the last file write example. The Get function in the TFile takes the object name as the argument and then returns a TObject pointer. This TObject pointer should then be type-casted to the type of the object you are reading like in the example above. The file should be closed at the end of all operations.

Handling multiple files

Working with multiple files is a tricky thing to do in ROOT. To have a simple picture, imagine a TFile to be like a directory. If you have multiple files, you need to use the cd() function in the TFile class to go to the appropriate file, just like you use the cd bash command.

```
    TFile *f1 = new TFile("rootfile1.root","RECREATE","My root file1");
    TFile *f2 = new TFile("rootfile2.root","RECREATE","My root file2");
    // create histograms
    //f1->cd();
    TH1F *h1 = new TH1F("my_hist1","Gauss",50,-500,500);
    //f2->cd();
    TH1F *h2 = new TH1F("my_hist2","Gauss",50,-500,500);
    // set style
    h1->SetFillColor(4);
    h2->SetFillColor(6);
    // fill histograms
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<10000;ii++){
        h1->Fill(ran->Gaus(0,50));
        h2->Fill(ran->Gaus(0,100));
    }
    h1->Draw();
    h2->Draw("SAME");
    gPad->Update();
    cout<<"Press anything to quit..."<<endl;
    cin>>a;
    // close file
    f1->Write();
    f2->Write();
    f1->Close();
    f2->Close();
```

multifilewrite.cpp: Program to explain handling multiple files.

In the above example we have created two files f1 and f2. Let us see what happens for the following cases in the above example:

1. f1->cd() and f2->cd() both commented: The histograms are written in f2 and nothing is written in f1. This is because the file that is created last will be the active one and all the objects will be written into it. (Just like a directory).
2. f1->cd() is used and f2->cd() is commented: The histograms will be written in f1 and nothing in f2. The command cd changes the active file to f1.
3. f1->cd() is commented f2->cd() is used: The histograms will be written to f2 and nothing in f1.
4. Both f1->cd() and f2->cd() are used: The first histogram will be saved in f1 since f1->cd() is called before its creation and then since f2->cd() is called, the second histogram is saved in f2.

Viewing TFile contents in a ROOT terminal

Though you can open a file for reading using C++, there is a simpler way to have an overview of the contents. Open the file using the following command in a terminal:

root filename.root

where filename.root is the file you want view. The ROOT terminal is automatically opened wherein you should type:

TBrowser th

which will open a browser showing a list of all the files, with a special icon next to the file that you opened. Clicking on the file will show you the contents of the file and clicking on objects like histogram, graphs and trees (about which you will learn in the next chapter) will plot them in a canvas.

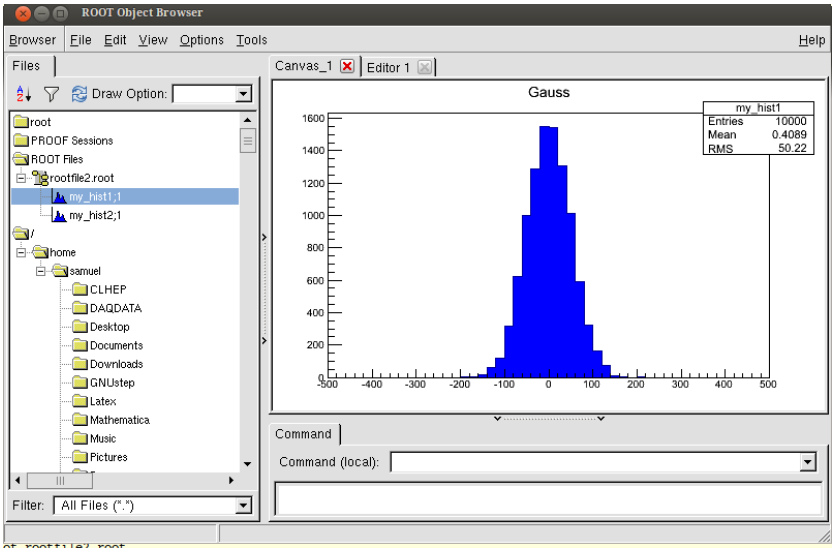

Figure 18: The TBrowser window of ROOT.

The TTree class: The spreadsheet for ROOT

Writing to a TTree

The last thing that is most important and often used ROOT in the TTree class. This is nothing like a spreadsheet like container for ROOT. If you want to analyse your data using ROOT, having them in the form of branches to a TTree object is the way to go. If you go through the functions in this class, you will find that it offers attractive data analysis and plotting features. There are good examples in the ROOT tutorials that you can try out to get a feeling what this class is all about. If you want to have a in-depth analysis of your data, especially in HEP, try using TTree. Beginners easily fall into the trap of writing their data in simple ASCII format and then try thousands of plotting application in the web without being convinced by any of them. Understanding TTree is a bit difficult especially if you are new to ROOT but learning it once will save you from the trouble of hunting for various applications and libraries (that are already existing in ROOT). Let us begin with an example to illustrate how a TTree object is created and how it is used for writing data.

```
1   float E,P;
2    TFile *f1 = new TFile("treefile.root","RECREATE","My root file1");
3     TTree *tr = new TTree("Tree","My Tree");
4     tr->Branch("Energy",&E,"E/F");
5     tr->Branch("Momentum",&P,"P/F");
6     TRandom *ran = new TRandom(450);
7     for(int ii=0;ii<10000;ii++){
8         E = ran->Gaus(0,50);
9         P = E*E;
10        tr->Fill();
11    }
12    tr->Draw("E");
13    //tr->Draw("E","E<10");
14    //tr->Draw("E:P");
15    gPad->Update();
```

```
16      cout<<"Press anything to quit..."<<endl;
17      cin>>a;
18      f1->Write();
19      f1->Close();
```

tree.cpp: How to create branches and write to a TTree

The picture you should have while using a TTree is that of a spreadsheet. In a spreadsheet we have columns, which in TTree is called as a branch, the rows are called as entries. Let us fill our TTree with two parameters E and P and try plot it using the TTree Draw function. We will store our TTree in a TFile to view it offline. The TTree is created using its constructor which takes the name and the title of the tree as arguments. The tree name can be used to retrieve the tree from the file. To create branches, we use the Branch function. This function takes the branch name, the address of the variable from where the data to be written will be taken and the variable name and its type in a special format. We will simply use the variable name E for E and P for P. Since both are floats, the syntax is:

E/F and P/F

The other variable types and the syntax to be used are as follows:

C : a character string
B : an 8 bit signed integer
b : an 8 bit unsigned integer
S : a 16 bit signed integer
s : a 16 bit unsigned integer
I : a 32 bit signed integer
i : a 32 bit unsigned integer
F : a 32 bit floating point
D : a 64 bit floating point
L : a 64 bit signed integer
l : a 64 bit unsigned integer
O : [the letter 'o', not a zero] a boolean

Once the branches are created, the next step is to fill our data using the Fill command. When this command is called, the data that is stored in E and P (the addressess of which were used to create the branches), at that point of time are written as an entry in the tree. Therefore the number of entries stored in the tree will be the number of times the Fill command was called.

Now once the data is stored we can use the tree object to plot them directly. For example, to plot the histogram of E you can call:

tr->Draw("E");

Note here E is the variable name that we gave for the parameter E. If you had given the variable name for E as "energy" then you can call plot using:

tr->Draw("energy");

If you want to plot with some conditions, for example if you do not want to plot the histogram of E when E<10, you would the cut argument of the Draw function:

tr->Draw("E","E<10");

If you want to plot the correlation (graph) between E and P then:

tr->Draw("E:P")

You can also use a condition in the above case.

Reading from a TTree

Reading from a TTree is similar to reading other object like histograms from a TFile. First, you should know the tree name and then you should use the Get function, then typecast the returned pointer to TTree. Using the tree pointer, you can do all other things.

```
1   TCanvas *c = new TCanvas("myCanvas","My Canvas");
2   // create file
3   TFile *f1 = new TFile("treefile.root","READ","My root file1");
4   TTree *tr = (TTree*)f1->Get("Tree1");
5   tr->Draw("E");
6   //tr->Draw("E","E<10");
7   //tr->Draw("E:P");
8   gPad->Update();
9   cout<<"Press anything to quit..."<<endl;
10  cin>>a;
11  f1->Close();
```

treeread.cpp: Reading a TTree from a file.

To view the contents of a tree stored in a TFile, you can follow the method mentioned in the last chapter to view the contents of TFile.

Using a C structure for a branch

Sometimes, you might for the sake of convenience store your data in a structure. In such cases you have to use tree in a slightly different way:

```
1     typedef struct {
2        int LayerNumber;
3        int StripNumber;
4        float noiseRate;
5     } RPCNOISERATE;
6   int main()
7   {
```

```
    // app instance
    TApplication app("my_app",0,0);
    RPCNOISERATE nr;
    int a;
    // create canvas*
    TCanvas *c = new TCanvas("myCanvas","My Canvas");

    // create file
    TFile *f1 = new TFile("treefile.root","RECREATE","My root file1");
    TTree *tr = new TTree("Tree","My Tree");
    tr->Branch("RPCNR",&nr,"L/I:S/I:N/F");
    TRandom *ran = new TRandom(450);
    for(int ii=0;ii<12;ii++){
      for(int jj=0;jj<32;jj++){
        nr.LayerNumber = ii;
        nr.StripNumber = jj;
        nr.noiseRate = ran->Gaus(0,50);
        tr->Fill();
      }
    }
    tr->Draw("L:N");
    //tr->Draw("E","E<10");
    //tr->Draw("E:P");
    gPad->Update();
    cout<<"Press anything to quit..."<<endl;
    cin>>a;
    f1->Write();
    f1->Close();
```

treeStruct.cpp: Creating a tree branch using a C structure.

In the above example, we have created a structure called RPCNOISERATE which contains the variables LayerNumber, StripNumber and the noiseRate. To create a branch this structure we call the Branch function like this:

```
tr->Branch("RPCNR",&nr,"L/I:S/I:N/F");
```

RPCNR is the name of the branch and nr is the address of the structure. The last argument is a string which is a concatenation of the variable names and their respective types. In the structure the first variable in LayerNumber for which we give the variable name L and since its type is int we append /I to it and similarly we give the variable name S to the second variable in the structure and append /I to it as it is also an integer. The last variable is name N and a /F

is appended since its a float. All these string are concatenated with a ":". This is the syntax to be followed if a branch has to be made like a structure. The filling part is similar to the previous example.

Getting the values at a particular entry

Often you want read the value of a particular entry in the tree. For that, you need to first set the branch address appropriately as shown below:

```
float E,P;
  // create file
  TFile *f1 = new TFile("treefile.root","READ","My root file1");
  TTree *tr = (TTree*)f1->Get("Tree");
  tr->SetBranchAddress("Energy",&E);
      for(int ii=0;ii<tr->GetEntries();ii++){
tr->GetEntry(ii);
      cout<<E<<endl;
      }
```

getEntry.cpp: Reading the values at a particular entry.

In the above example, we read a tree named Tree from a file (the file we created in the first example in this chapter) and we know that it has a branch named Energy. To read the value of it, you have to supply the address where the value will be read into. This is done using the SetBranchAddress function wherein we supply the branch name and the address. We now loop over all the entries and use the GetEntry function which takes the entry number as the argument and stores the value of that entry in the address given in the Set-BranchAddress (i.e in E). The GetEntry function does the opposite of Fill.

Skeleton Code Generation

Sometimes it would be tedious to set branch address for all the branches because the structure of the tree is too complicated or you might not know the branch structure of your tree. In that case you should use the MakeClass function which generated a class for your tree. It writes a .C and .h file for your tree which will set the branch addresses for you. All you need to do is to include the .C and .h file in you analysis code and access the tree using the constructor of the class that you generated. This is the most efficient way to analyze a tree.