

REPUBLIC OF RWANDA
UNIVERSITY OF KIGALI

P.O Box 2611

Website://www.uok.ac.rw



MODULE OF Programming in C++
Computer Science

MODULE DESCRIPTION

1. Module Code : CSC201 Faculty: Science and Technology

2. Module Title: computer programming with C++

3. Level: 1 Semester: 3 Credits: 15

4. First year of presentation: 2015 Administering Faculty: Science and
technology

5. Pre-requisite or co-requisite modules, excluded combinations

CSC103

6. Allocation of study and teaching hours

Total student hours :	52	Student hours
Lectures		52
Seminars/workshops		
Practical classes/laboratory		20
Structured exercises		6
Set reading		2
Self-directed study		20
Assignments — preparation and writing		2
Examination — revision and attendance		2
Other: (Invigilation and Marking)		

7. **Brief description of aims and content**

In this module, students will learn basics of programming with c++, structured and Object Oriented Programming Concepts. They will learn how data abstraction, reusability, inheritance and modularity of code can be enhanced using C++.

8. **Learning Outcomes**

Knowledge and Understanding

Having successfully completed the module; students should be able to demonstrate knowledge and understanding of:

- i. Theoretical and practical programming concepts
- ii. Object Oriented concepts and their implementation in C++

- iii. Basics data structures using C++.

Communication/ICT/Numeracy/Analytic Techniques/Practical Skills

Having successfully completed the module, students should be able to:

- iv. Present their ideas to a general audience using reasonable written and oral communication skills.
- v. Use numerical and statistical methods to solve Computational problems.

General transferable skills

Having successfully completed the module, students should be able to:

- vi. Demonstrate managerial skills
- vii. Justify the importance of team spirit to solve a common problem.
- viii. Be self contained to perform assigned tasks independently or with little guidance.
- ix. Complete their assigned task under tough time constraints.

9. Indicative Content

C++ definition, c and c++, first program in c++, Variables, Inputs and Outputs, structure of controls, Arrays, pointers and structures, Functions, Introduction to OOP and Files.

10. Learning and Teaching Strategy

- Students do hands-on-practice on computers.
- Students will do structured exercises and solve numerical problems.
- Students will make self-study from classroom notes, text books and from websites.
- Students will participate in short-term coursework.
- Seminar/workshop, quiz, debate competitions will be organized and students will be encouraged to participate subject to resource availability.

- Students will discuss the concepts in peer groups during informal conversation and share knowledge.

11. **Assessment Strategies**

- As a self assessment tool, exercises will be solved by students to check their understanding of particular concepts and terminology.
- As formative assessment tools, students may be asked to make oral presentations, write essay/short notes etc.
- Teacher may ask students to make their own research about particular topic using library resources or internet and then produce computerized report. These reports can be assessed formatively or can be awarded as summative assessment.
- There will be comprehensive final examination (Paper-Pencil test) to give score to each student and record them for administrative purpose.

13. **Strategy for feedback and student support during module**

Students will be given feedback for formative assessments to tell them how they are progressing in terms of knowledge, skills and understanding of a subject. Feedback will be given to the students either during or after the assessment.

Teaching team will re-evaluate teaching methods for problematic area and work toward improving student learning. Some additional support like Off-class student counseling, remedial classes for weak students may also be provided if required.

14. **Indicative Resources**

- Britton, Carol., [2005], *A student guide to object-oriented development*, Library Call No: 005.12 BRI 2005

- E. BalaguruSamy, [2004], *Object Oriented Programming with C++*, Second Edition, Tata McGraw Hill, New Delhi, ISBN 0-07-040211-6
- [James Rumbaugh](#), Michael Blaha and others, [2002], *Object Oriented Modeling and Design*, PHI Pvt. Ltd, New Delhi, ISBN 81-203-1046-2
- Mark Allen Weiss., [2006], *Data Structures and Algorithm Analysis in C++*, Third Edition, Pearson International Edition, ISBN 0-321-39733-9
- Walter Savitch, [2007], *Problem Solving with C++*, Sixth Edition, Pearson International Edition, ISBN 0-321-44263-6

Background Texts

- Mark A. Weiss. , [2005], *Data Structures and Problem Solving Using C++*, Third Edition, Addison-Wesley, ISBN 0-321-40992-2
- Paul J. Deitel, Harvey M. Deitel, [2005], *Simply C++ - An Application-Driven Tutorial Approach*, Prentice Hall, ISBN 0-13-127768-5
- Sara Baase, Allen Van Gelder , [2000], *Computer Algorithms- Introduction to Design and Analysis*, Third Edition, Addison Wesley, ISBN 0-201-61244-5

Teaching/Technical Assistance

- One Tutorial Assistant
- One Lab Attendant

Laboratory space and equipment

- Students require weekly 5 hours Hands-on-practice in Computer Lab (80 computers minimum required for individual access)
- LCD Projector for routine lectures and seminars, whiteboard, marker pens.
- Weekly 5 Internet hours are required for staff as well as students to access online study material.

Computer requirements

Turbo C++ / Visual C++ compiler together with multiple licenses for educational purposes. Ms-Windows, Ms-Word and Ms-Power point are required for preparation of lecture notes, assignments and presentations.

15. Others

Contents

CHAPTER 1: INTRODUCTORY CONCEPTS	9
1.1. WHAT PROGRAMMING LANGUAGE IS?	9
1.2. REQUIREMENTS FOR CREATING A PROGRAM	9
1.3. C++ LANGUAGE PRESENTATION.....	10
1.4. FIRST PROGRAM IN C++	10
1.5. EXERCISES.....	12
CHAPTER 2: VARIABLES, INPUTS AND OUTPUTS.....	13
2.1. INTRODUCTION	13
2.2. VARIABLES DECLARATION AND MANIPULATION.....	13
2.3. BASIC INPUT STATEMENTS.....	18
2.4. EXERCISES.....	19
CHAPTER 3: CONTROL STRUCTURES	20

3.1. INTRODUCTION	20
3.2. IF, ELSE STATEMENTS.....	20
3.3. SWITCH CASE STATEMENTS.....	22
3.4. LOOP	25
3.5. THE “break” AND “continue” STATEMENTS	30
3.6. EXERCISES.....	30
CHAPTER 4: POINTERS, ARRAYS AND STRUCTURES	31
4.1. POINTERS.....	31
4.2. NUMERIC ARRAYS AND STRINGS.....	34
4.3. STRUCTURES.....	38
4.4. ARRAYS AND POINTERS	41
4.5. POINTERS AND STRUCTURES	44
4.6. EXERCISES.....	46
CHAPTER 5: FUNCTIONS	48
5.1. INTRODUCTION	48
5.2. FUNCTION DECLARATION, DEFINITION AND CALL	49
5.3. PARAMETERS, ARGUMENTS AND SCOPE OF VARIABLES	53
5.4. FUNCTION OVERLOADING.....	61
5.5. INLINE FUNCTIONS.....	62
5.6. EXERCISES.....	63
CHAPTER 6: INTRODUCTION TO OOP	63
6.1. A CLASS	64
6.2. CLASS CONSTRUCTOR AND DESTRUCTOR FUNCTIONS	66
6.3. CLASSES INHERITANCE.....	71
6.4. POLYMORPHISM IN C++.....	76
6.5. DATA ABSTRACTION.....	79
6.6. DATA ENCAPSULATION.....	82
CHAPTER 7: FILE PROCESSING	84
7.1. NOTIONS OF FILES IN PROGRAMMING	84
7.2. FILE MANIPULATION STREAMS	84

7.3. THE USE OF STRUCTURES AND ARRAYS IN FILES PROCESSING.....	93
7.4. EXERCISES.....	96

CHAPTER 1: INTRODUCTORY CONCEPTS

1.1. WHAT PROGRAMMING LANGUAGE IS?

To let you understand well what programming language is, it is better to present separately the key words which are **Programming** and **Language**.

Programming is writing codes constituting instructions that have to be executed into computer's memory in perspective of executing a specific task. Programming mostly refers to computer programming, coding or scripting when a computer program or software has to be built. However, programming involves many other aspects like analysis, conception and design.

A language can be defined as a system of communication consisting of sounds, words and grammar, or the system of communication used by the people of a particular country or profession. As programming involves instruction and instructing requires communication, the creation of a program requires programming language like C, C++, java, php etc.

Briefly, programming means telling the computer what to do, while programming language constitutes a set of key words and rules that have to be followed to create instructions (statements) of a program.

1.2. REQUIREMENTS FOR CREATING A PROGRAM

There are basic requirements of creating a program. Some of them are found on what a program is, others are based on the tools and techniques to be used when a program has to be created.

A program is a set of organized instructions that have to be executed in order to perform a specific task or tasks. From this definition we can deduct two important requirements which are: **organized instructions** and **specific task**.

The organization of instructions is done through the **algorithm**, while the specification of the task is done through the **problem analysis**.

After the problem is analyzed and the algorithm is defined, then through **creation of source codes** the program is **edited** in a specific language. After source codes are defined the **compilation** and **execution** follow. The compilation constitutes the check of source codes in order to verify the compliance of source codes with the used language library while execution makes the source codes file an executable file which means the translation of high level language into machine language or low level language.

1.3. C++ LANGUAGE PRESENTATION

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The name is a pun - "++" is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e. converted into a series of low-level instructions that the computer can execute directly) using a C++ compiler. As c language, C++ is also **case sensitive** programming language. This means that capital letters are understood differently from small letters.

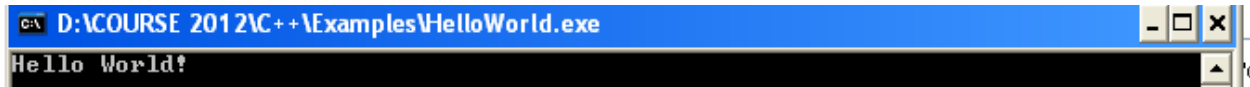
1.4. FIRST PROGRAM IN C++

For any learning of programming language, the first program to perform is to display the message "Hello World". In C++ the source codes are presented as followed:

```
/* My first program in c++*/  
#include<iostream>  
#include<conio.h>  
using namespace std;  
int main()  
{  
    cout<<"Hello World!"<<endl;  
    getch();  
    return 0; }
```

Execution

The execution looks like follow:



Comments about the source codes of the program

The source codes are grouped into five components:

Comments: In the source codes, the first line (`/* My first program in c++*/`) constitutes a comment. A comment is any line of code or a statement that has to be ignored as instruction by the compiler. When comments are extended one or more than one line, they are introduced by `/*` and ended by `*/` while when they are extended on one line only, they are introduced `//`.

Header files declaration: All lines introduced by the keyword `#include` they constitute the declaration of files that are predefined and located in the standard library of the language.

Location of files of standard library: The location of standard library is represented by the statement `using namespace std;`.

Declaration and definition of the function main: The statement `int main()` stands for function declaration. The details about functions will be seen in chapter 5. The statements `cout<<"Hello World!"<<endl;`, `getch();` and `return 0;` are representing the body of the function main. The body of a function is introduced by the symbol `{` and is ended by the `}`.

NB: The `endl` is a constant meaning "end of line" when the program will be displaying something. In other word the output will look differently from the source codes. For example if the statement `[cout<<"Hello World!"<<endl;]` is transformed like this: `[cout<<"Hello"<<endl<<"World!"<<endl;]` then the output will look like followed:



1.5. EXERCISES

- A) Using C++, write a program that with only one output statement will display vertically the words composing the sentence “WORK HARD TO GAIN WORLD”.
- B) Translate the program produced on A into C language.
- C) Why to compile a program? What Standard Library of C++ has to do with compilation of a program?
- D) What are the features and aspects that make C++ different from C?

CHAPTER 2: VARIABLES, INPUTS AND OUTPUTS

2.1. INTRODUCTION

Variable constitutes the **space of memory** reserved in order to keep data of specific **type**. The data to be kept by a variable can be acquired through a process but basically through the input instructions. The purpose of output is to print or display on the output device (Monitor) the texts and the content of variables.

This chapter aims to make learner able to include into C++ program variable manipulation using input and output C++ streams or assigning value to a declared variable.

2.2. VARIABLES DECLARATION AND MANIPULATION

Before a variable is recognized by the compiler it has to be declared. The declaration of variable is an instruction of reservation of space of memory. A variable in C++ is declared in the following syntax:

Type NameOfVariable;

Three component of this syntax are to be clarified and detailed:

Type

A type of a variable constitute a keyword of the language or a defined structure (see chapter 4), that describes the **format of data** (character, logical or number) and the **size** (number of bytes) of the space of memory the data will be stored in. A type constituting a keyword of the language is called also a **predefined type**.

C++ offer the programmer a rich assortment of built-in as well as user defined data types. Following table list down seven basic C++ data types:

Type	Keyword
Boolean	Bool
Character	Char
Integer	Int
Floating point	Float
Double floating point	Double
Valueless	Void

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value memory and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
Char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
Int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767

unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
Float	4bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

The following is the example which will produce correct size of various data type on your computer.

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;
    getch();
    return 0;
}
```

Manipulation of variables

Once a variable is declared then it is subject to a usage in a program. That usage is what we call manipulation of the variable. Specifically the manipulation of a variable concerns assigning a value to the variable. The symbol used in arithmetic to express equality “=” is used in C++ to assign a value to a declared variable.

E.g.: `int a=10;`

The statement (`int a=10;`) can be read “*The variable a receives or gets 10*”. It is possible that the value is replaced by another variable containing a value.

E.g.: `int b=12;`

`int a=b;`

When the value assigned to a variable is given by a combination of values and arithmetic operators, the value is called “**Expression**”. C++ supports the following operators:

ARITHMETIC OPERATORS	
SYMBOL	MEANING
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

COMPARISON OPERATORS	
SYMBOL	MEANING
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Different from

LOGICAL OPERATORS	
	Or
&&	And
!	Not

Arithmetical operators are used to provide calculation expression while comparison and logical operators are used respectively for conditional and combination of conditions into one test expressions.

Precedence of operators

The precedence of operators is the order used to evaluate them during the evaluation of the complete expression. To be compliant with the usual mathematical notations, the evaluation is not left-to-right. For example

$$3 + 4 * 5 + 6 * 7$$

is considered by the compiler as

$$3 + (4 * 5) + (6 * 7)$$

When two operators have same precedence (i.e. when we have the same operator twice), the evaluation is left-to-right. The specifications of C++ do not specify the order of evaluation when operators have the same precedence, except for logical operations.

Assignment operator

A strange thing in C++ is that assignments are also expressions:

$$j = 3 + (i = 5);$$

is legal, and will assign to i the value 5, and to j the value 8. But feel free not to use such weird tricks.

NB: When a variable has the type **char**, the assigned character must be between quotations. Otherwise the compiler will qualify the absence of quotations as an error. If a number is assigned to a variable of type **char** the number must not be less than -127 and not greater than 255. However, the value that will be kept into the variable is an equivalent character of that number in code ASCII. The following program explains more. The functions `int()` and `char()` can be used to translate character into a number or number into character basing on codes ASCII.

```
#include<iostream>
```

```
#include<conio.h>

using namespace std;

main() {

char b=65;

char c=66;

cout<<"In   ASCII   "<<int(b)<<"   is   "<<b<<endl<<"In   ASCII
"<<int(c)<<" is "<<c;

getch();

}
```

The output looks like followed:



```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\CharAndNumbers.exe
In ASCII 65 is A
In ASCII 66 is B_
```

2.3. BASIC INPUT STATEMENTS

Through examples we saw the basic outputs statements, where programmer can instruct computer to display or to print something on screen using C++ stream which is “**cout<<**”. Essentially, a program may need input statements so that data can be introduced into computer’s memory by user. The recorded data may undergo a treatment or process, then after the result is outputted. The stream for input in C++ is “**cin>>**”. To make input possible, there must be a well declared variable that will keep the inputted data. The following example explains more about input.

E.g.: *Using C++, write a program that will print an entered number doubled.*

```
#include<iostream>

#include<conio.h>

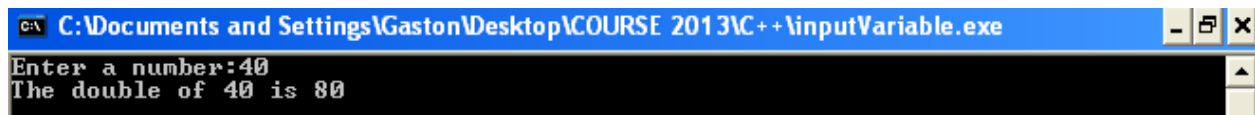
using namespace std;
```

```

main() {
    int num;//declaration of variable named num
    cout<<"Enter a number:";//To print message to user
    cin>>num;//Input statement
    cout<<"The double of "<<num<<" is "<<num*2;//Output of result
    getch();// to let output to be consulted
}

```

Example of output of the program



2.4. EXERCISES

- A) Using C++, write a program that will display the opposite of an entered number.
- B) Use C++, to display the sum and product of two numbers.
- C) Print the sum of equivalent numbers of two characters entered by user. These numbers must be displayed too. Use C++.
- D) Knowing that when a float number is assigned to an integer variable the decimal part is lost, create a program that will ask user a float number, then it displays the decimal part only.
- E) Use C++, and write a program that will display two entered numbers swapped. This is verifiable only into source codes.

CHAPTER 3: CONTROL STRUCTURES

3.1. INTRODUCTION

The control structures constitute all keywords that are used in a program in order to make a set of instructions to be executed after checking a condition. Once the condition is true, the concerned instruction(s) can be executed once or many times depending on the used control structure. In this chapter we will examine the control structures that C++ supports.

3.2. IF, ELSE STATEMENTS

The structure of control **if** allows programmer to specify instructions that have to be executed or not according to the result from condition check is found **true** or **false**.

Syntax: `if(condition) {
 Statements
 }`

The keyword **else** is used to specify the opposite case of **if condition**. All else must be referring to an if condition, but if does not require always an else.

Syntax: `if(condition) {
 Statements
 }
 else{
 Statements
 }`

Example of application

The following program will print the absolute value of an entered integer.

```
#include<iostream>
#include<conio.h>
using namespace std;
main(){
    int n;
    cout<<"Enter a number: ";
    cin>>n;
    if(n<0)
        cout<<"The absolute value of "<<n<<" is "<<-n;
    else
        cout<<"The absolute value of "<<n<<" is "<<n;
    getch();
}
```

The output case if



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\Documents and Settings\Gaston\Desktop\COURSE 2013\VC++\VFElse.exe". The command prompt area is black with white text. It shows the prompt "Enter a number: " followed by the input "-10". Below that, it shows the output "The absolute value of -10 is 10_" with a cursor at the end.

The output case else



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\Documents and Settings\Gaston\Desktop\COURSE 2013\VC++\VFElse.exe". The command prompt area is black with white text. It shows the prompt "Enter a number: " followed by the input "13". Below that, it shows the output "The absolute value of 13 is 13_" with a cursor at the end.

Meaning of else according to the given if condition

Each else in a program has an equivalent if condition. The following table explains more.

if(a<5)	else = if(a>=5)
if(a>5)	else= if(<=5)
if(a<=5)	else= if(a>5)
if(a<10 a>20)	else= if(a>=10 && a<=20)
if(a>=10 && a<=20)	else= if(a<10 a>20)

3.3. SWITCH CASE STATEMENTS

Switch case statements are a substitute for long if statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a char). The basic format for using switch case is outlined below. The value of the variable given into switch is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point.

Syntax

```
switch ( <variable> ) {  
  case this-value:  
    Code to execute if <variable> == this-value  
    break;  
  case that-value:  
    Code to execute if <variable> == that-value  
    break;  
  ...  
  default:  
    Code to execute if <variable> does not equal the value  
    following any of the cases  
    break;  
}
```

The condition of a switch statement is a value. The case says that if it has the value of whatever is after that case then do whatever follows the colon. The break is used to break out of the case statements. Break is a keyword that breaks out of the code block, usually surrounded by braces,

which it is in. In this case, break prevents the program from falling through and executing the code in all the other case statements. An important thing to note about the switch statement is that the case values may only be constant integral expressions. Sadly, it isn't legal to use case like this:

```
int a = 10;
int b = 10;
int c = 20;

switch ( a ) {
case b:
    // Code
    break;
case c:
    // Code
    break;
default:
    // Code
    break;
}
```

The default case is optional, but it is wise to include it as it handles any unexpected cases. Switch statements serves as a simple way to write long if statements when the requirements are met. Often it can be used to process input from a user.

Example of application

The following program will allow user to record a number and choose one of the following options concerning the output: absolute value, opposite number, double of the number and the square of the number.

```
#include<iostream>
#include<conio.h>
using namespace std;
main(){
    int n;
```

```

int choice;
cout<<"Enter a number:";
cin>>n;
cout<<endl<<"1 to display absolute value";
cout<<endl<<"2 to display opposite number";
cout<<endl<<"3 to display the double of the number";
cout<<endl<<"4 to display the square of the number";
cout<<endl<<endl<<"Your choice please:";
cin>>choice;
switch(choice){
    case 1:
        if(n>=0)
            cout<<"The absolute value of "<<n<<" is "<<n;
        else
            cout<<"The absolute value of "<<n<<" is "<<n*-1;
        break;
    case 2:
        cout<<"The opposite value of "<<n<<" is "<<n*-1;
        break;
    case 3:
        cout<<"The double of "<<n<<" is "<<n*2;
        break;
    case 4:
        cout<<"The square of "<<n<<" is "<<n*n;
        break;
    default:
        cout<<"Wrong choice";
}

```

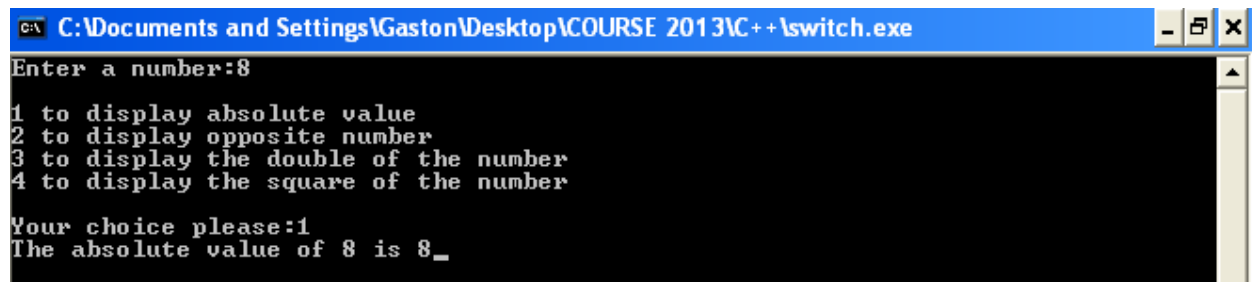


```
        break;

    }

    getch();}
```

Sample of output



3.4. LOOP

In programming, it happens that certain tasks require that some instructions are executed repeatedly. The illustrative case is to imagine a program that will print the sum and average of certain numbers. If it is two or three numbers, it is possible to declare two or three variables that will keep these numbers. However, can you imagine when you will have to calculate the summation and average of 10 numbers! Will you declare 10 different variables? Assume that the declaration of 10 variables is possible! What about 100 numbers? Let us believe that you can declare 100 variables! How will you solve the problem in case it requires that you calculate summation and average of N numbers? It is impossible to declare unknown number of variables. Loops are inescapable structure of control that will let you to handle the kind of problems.

To set a loop you need firstly to identify the process of the program needing to be repeated in order to perform correctly the task, secondly to conceive the condition of starting and ending the loop.

The language C++ supports the following loops:

Loop “for”

This loop is mostly used when number of iterations is known in advance by the programmer. Its syntax look like followed:

```
for(Initial value; starting or ending condition; Increment or  
decrement statement)
```

```
    statement;
```

Or

```
for(Initial value; starting or ending condition; Increment or  
decrement statement){
```

```
    statements
```

```
}
```

NB: The increment or decrement expressed here in the syntax stands for update of the variable that has been initialized to the initial value in perspective of making the condition of the loop false. It constitutes also the last statement executed before the condition is rechecked in order to decide whether the next iteration has to be done or not.

Application Example

Take the example of displaying the summation and average of 10 numbers that has been entered by user.

The process to be executed repeatedly is composed of the following actions:

- Display instructive message telling user to enter a number;
- Read the entered number; and
- Calculate cumulative sum.

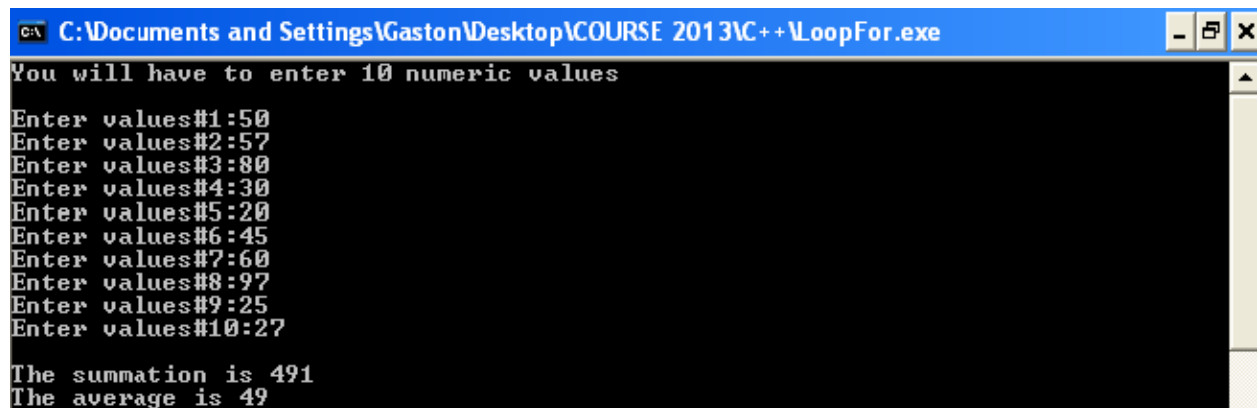
```
#include<iostream>  
#include<conio.h>  
using namespace std;  
main(){  
    int i,n,sum;
```

```

sum=0;
cout<<"You will have to enter 10 numeric values"<<endl<<endl;
for(i=1;i<=10;i++)
{
    cout<<"Enter values#"<<i<<":";
    cin>>n;
    sum=sum+n;
}
cout<<endl<<"The  summation  is  "<<sum<<endl<<"The  average  is
"<<sum/10;
getch();
}

```

Sample of output



```

C:\Documents and Settings\Gaston\Desktop\COURSE 2013\VC++\LoopFor.exe
You will have to enter 10 numeric values
Enter values#1:50
Enter values#2:57
Enter values#3:80
Enter values#4:30
Enter values#5:20
Enter values#6:45
Enter values#7:60
Enter values#8:97
Enter values#9:25
Enter values#10:27
The summation is 491
The average is 49

```

The loop “while”

This loop is mostly used when programmer ignores the number of iterations. The syntax looks like followed.

```
while (condition)
```

```
statement;
```

Or

```
while (condition)

{

    statements

}
```

Application Example

Let us come back to the example illustrating the loop for which example provides a program that will display the summation and average of 10 different entered numbers.

```
#include<iostream>
#include<conio.h>
using namespace std;
main(){
int i,n,sum;
sum=0;
cout<<"You will have to enter 10 numeric values"<<endl<<endl;
i=1;
while(i<=10)
{
    cout<<"Enter values#"<<i<<":";
    cin>>n;
    sum=sum+n;
    i++;
}
```

```

}

cout<<endl<<"The  summation  is  "<<sum<<endl<<"The  average  is
"<<sum/10;

getch();

}

```

Comments: Remark that initialization is done out from the “loop while” while the increment is done inside the loop while. This means that the increment or decrement of the variable the condition of the loop is built on is the part.

Output

```

C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\LoopFor.exe
You will have to enter 10 numeric values
Enter values#1:50
Enter values#2:57
Enter values#3:80
Enter values#4:30
Enter values#5:20
Enter values#6:45
Enter values#7:60
Enter values#8:97
Enter values#9:25
Enter values#10:27
The summation is 491
The average is 49

```

Loop “do while”

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

The syntax of a do...while loop in C++ is:

```

do
{
    statement(s);
}while( condition );

```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

3.5. THE “break” AND “continue” STATEMENTS

The keywords **continue** and **break** are respectively used to interrupt iteration and loop.

The **break** statement may appear inside a loop or a switch statement. Its effect is to terminate the loop immediately enclosing it or the switch statement. It is an error to use the break statement anywhere else.

The **continue** statement is placed inside a loop. Its purpose is to terminate the current iteration of the loop and jump to the following iteration. It is an error to put a continue statement outside a loop.

3.6. EXERCISES

- A) Rewrite the program given as illustrative example of for loop, where the used loop is do while.
- B) Using C++, write a program that will display number of times user has entered a wrong integer knowing that the correct one must be between 0 and 20 and the user must attempt 5 times.
- C) Write a program that will attest that an entered integer is prime or not.
- D) Write a program that will display the suite of numbers organized as followed: $x, 2x, 3x, 4x, \dots, (|x|-1)x, |x|x$.

- E) Write a program that will print factorial of an entered integer. That integer must be between 1 and 9.
- F) Write a program that will print the suite of numbers organized as followed: 2!, 3!, 4!, ..., (x-1)!, x! knowing that x must be between 1 and 9.

CHAPTER 4: POINTERS, ARRAYS AND STRUCTURES

4.1. POINTERS

Pointers are an extremely powerful programming tool. C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

Consider the following which will print the address of the variables defined:

```
#include <iostream>

using namespace std;

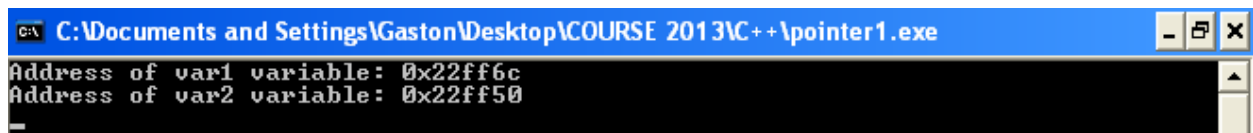
int main ()
{
    int var1;
    char var2[10];

    cout << "Address of var1 variable: ";
    cout << &var1 << endl;

    cout << "Address of var2 variable: ";
    cout << &var2 << endl;

    return 0;}
```

Output



```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\pointer1.exe
Address of var1 variable: 0x22ff6c
Address of var2 variable: 0x22ff50
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *pointer-name;
```

Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

Example for pointers application

```
#include <iostream>

using namespace std;

int main ()
{
    int  var = 20;    // actual variable declaration.
    int  *ip;         // pointer variable

    ip = &var;        // store address of var in pointer variable

    cout << "Value of var variable: ";
    cout << var << endl;

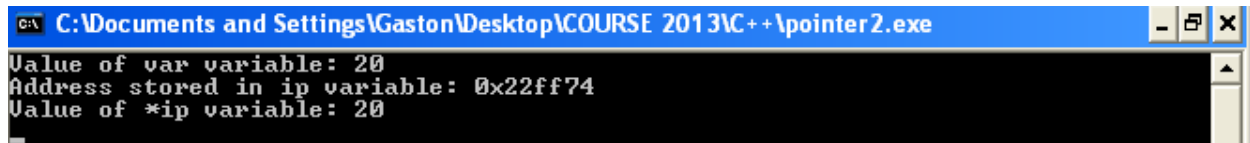
    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;
```



```
// access the value at the address available in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;

return 0;}
```

Output:



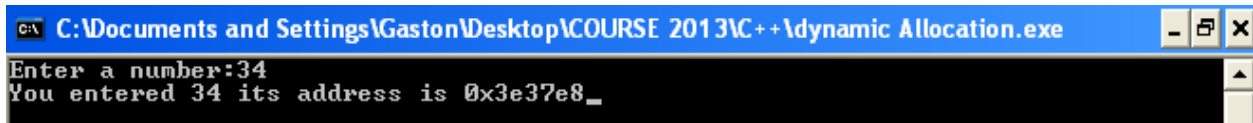
```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\pointer2.exe
Value of var variable: 20
Address stored in ip variable: 0x22ff74
Value of *ip variable: 20
```

Dynamic memory allocation

Pointers constitute the foundation of the dynamic memory allocation both in C and C++ languages. Dynamic memory allocation is a way of transforming a pointer into a space of memory that not only can hold address but also data. In C++ the functions **new()** and **delete()** are used to perform that dynamic memory allocation. After a pointer is declared, during the runtime of the program the space for data can be **allocated** or **deleted** respectively using the functions **new()** and **delete()**. The following codes show how the function **new()** is used:

```
#include<iostream>
#include<conio.h>
using namespace std;
main() {
    int *n;
    n=new(int);
    cout<<"Enter a number:";
    cin>>*n;
    cout<<"You entered "<<*n<<" its address is "<<n;
    getch();}
```

The output



4.2. NUMERIC ARRAYS AND STRINGS

It happens that programmer has to input or process or output different values of the same type. Here we can give example of the case where programmer has to record marks of a certain number of students. What programmer will do? Will he declare variables whose number is equivalent to the number of students? It may be impossible or very and very difficult! There must be a way of to keep into computer's memory the all students' marks without declaring many variables having the same type and purpose. Arrays handle that problem.

What is an array?

An array constitutes a data structure that is able to keep different values of the same data type. In case these values have **char** as type, then the array is called **string**.

Array declaration

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Structure of the array “balance”

index: 0 1 2 3 4 5 6 7 8 9

balance:

--	--	--	--	--	--	--	--	--	--

position: 1 2 3 4 5 6 7 8 9 10

Notice that the position of the last element in an array is equivalent to the size of the array while its index is equivalent to the size of the array minus one (size-1) because indices are counted starting from 0 while positions are counted starting from 1. In other words, the index of an element in an array is given by its position minus 1 (position-1).

Array manipulation

By manipulation of an array we mean input, output, initialization, calculations or any other process requiring having access to an element of the concerned array.

Particularity of strings

A string is an array of characters by character we mean alphanumeric characters which are (“a” to “z” or “A” to “Z” or “0” to “9”). The strings are more flexible in terms of manipulation than the arrays of other types because of the following reasons:

- C++ allows that a string can be initialized as a simple variable.

E.g.: `char name[] = “BAZIRUWIHA”;`

name

B	A	Z	I	R	U	W	I	H	A	\0
---	---	---	---	---	---	---	---	---	---	----

NB: The value noted like ‘\0’ is a character that indicates the end of the string. Once the size of the sting is greater than the number of characters, the remaining part of the string becomes useless.

E.g.: `char name[15]="NIYIKIZA";`

N	I	Y	I	K	I	Z	A	\0						
---	---	---	---	---	---	---	---	----	--	--	--	--	--	--

- You don't need a loop to display the content of a string. Just write like : `cout<<name;`

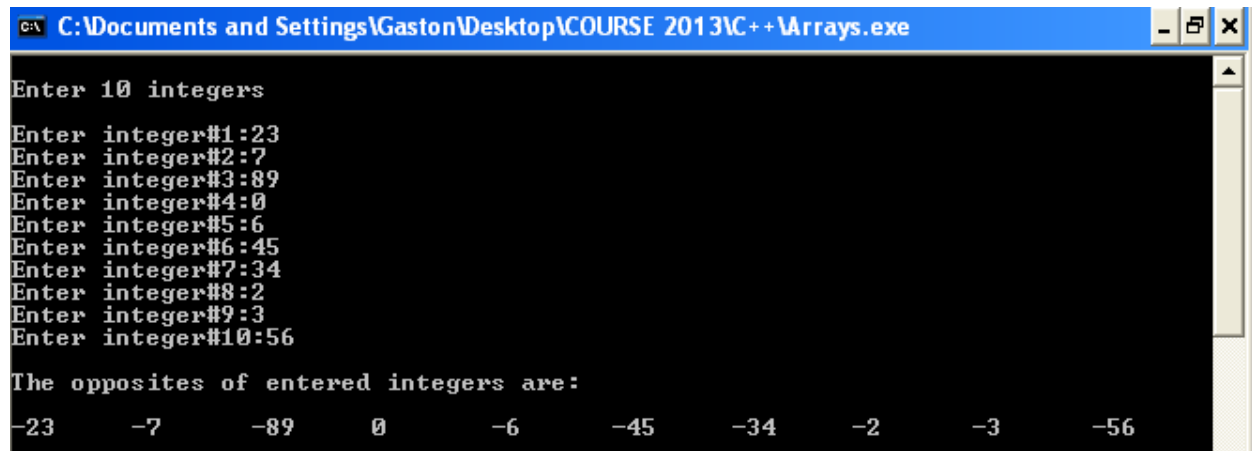
Example for application

Using C++, the following program will allow user to record 10 integers in an array and to display the opposite of each entered integer.

```
#include<iostream>
#include<conio.h>
using namespace std;
main(){
    int t[10]; //declaration of array
    int i; //variable to be used for index access
    cout<<endl<<"Enter 10 integers"<<endl<<endl;
    for(i=0; i<10; i++){
        cout<<"Enter integer#"<<i+1<<":";
        cin>>t[i];
    } //end of loop for recording
    cout<<endl<<"The      opposites      of      entered      integers
are:"<<endl<<endl;
    for(i=0; i<10; i++)
        cout<<t[i]*-1<<"\t";
```

```
getch();}
```

Example of output



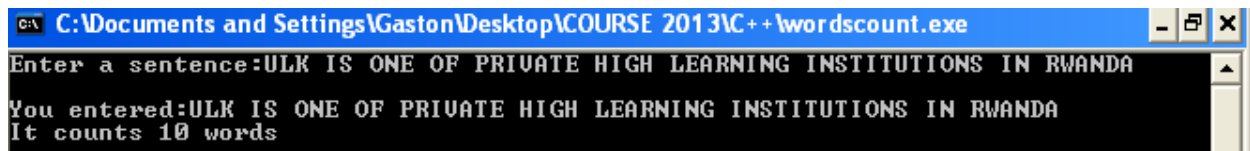
```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\Arrays.exe
Enter 10 integers
Enter integer#1:23
Enter integer#2:7
Enter integer#3:89
Enter integer#4:0
Enter integer#5:6
Enter integer#6:45
Enter integer#7:34
Enter integer#8:2
Enter integer#9:3
Enter integer#10:56
The opposites of entered integers are:
-23    -7    -89    0    -6    -45    -34    -2    -3    -56
```

The following program will display number of words composing a sentence entered by user.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    char str[100];
    int words=0;
    int i;
    cout<<"Enter a sentence:";
    gets(str);/*gets is an input function that allows blank
               spaces*/
    cout<<endl<<"You entered:"<<str;
    for(i=0;str[i]!='\0';i++){
        if(str[i]==' ')
            words++;
    }
}
```

```
cout<<endl<<"It counts "<<words+1<<" words";  
getch();  
return 0; }
```

Example of output



```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\wordscout.exe  
Enter a sentence:ULK IS ONE OF PRIVATE HIGH LEARNING INSTITUTIONS IN RWANDA  
You entered:ULK IS ONE OF PRIVATE HIGH LEARNING INSTITUTIONS IN RWANDA  
It counts 10 words
```

4.3. STRUCTURES

C/C++ arrays allow you to define variables that combine several data items of the same type but **structure** is user defined data type which allows you to combine data items of different kinds. They constitute **personalized** or **customized** types that programmer is expecting to use in the rest of his program. They must be placed at the beginning of the program just after the declaration of header files.

Structures are used to represent a record. Suppose you want to keep track of your students' profiles. You might want to track the following attributes about each student:

- Roll number
- First name
- Second name
- Sex
- Department code

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this:

```

struct structure_name
{
    type member1;
    type member2;
    ...
    type memberN;
} [one or more structure variables];

```

The **structure name** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Student structure:

```

struct Students
{
    int Roll_number;
    char First_name[20];
    char Second_name[16];
    char Sex;
    char Department_code[5];
}Student_data;

```

Manipulation of a structure

After a structure is declared, it can be manipulated through a variable having the name of the structure as its data type. As an element of an array refers to its index in order to be accessed for any manipulation, the member of a structure is also accessed and manipulated referring to that variable and the symbol of period (.) is used to separate the main variable name and structure member name.

The following codes of C++ explain more about structure manipulation:

```

#include<iostream>
#include<conio.h>
using namespace std;

struct Students
{
    int Roll_number;
    char First_name[20];
    char Second_name[16];
    char Sex;
    char Department_code[5];
};

main(){
    Students Student_data;
    cout<<"Enter the roll number:";
    cin>>Student_data.Roll_number;
    cout<<"First name:";
    cin>>Student_data.First_name;
    cout<<"Second name:";
    cin>>Student_data.Second_name;
    cout<<"Sex:";
    cin>>Student_data.Sex;
    cout<<"The department:";
    cin>>Student_data.Department_code;
    cout<<endl<<"You entered:"<<endl<<"-----"<<endl;
    cout<<Student_data.Roll_number<<endl;
    cout<<Student_data.First_name<<endl;
}

```



```

cout<<Student_data.Second_name<<endl;

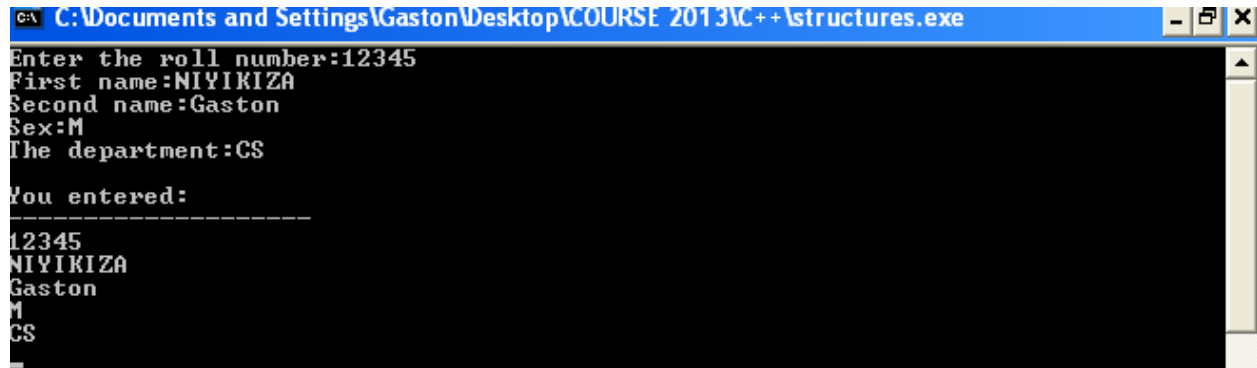
cout<<Student_data.Sex<<endl;

cout<<Student_data.Department_code<<endl;

getch(); }

```

Illustrative output:



```

C:\Documents and Settings\Gaston\Desktop\COURSE 2013\VC++\structures.exe
Enter the roll number:12345
First name:NIYIKIZA
Second name:Gaston
Sex:M
The department:CS
You entered:
-----
12345
NIYIKIZA
Gaston
M
CS

```

4.4. ARRAYS AND POINTERS

There is a close relationship between pointers and arrays. Two aspects expressing that relationship are:

- The name of an array constitutes a pointer that keeps the referential address of the first element of that array. That address is qualified referential because the address of each remaining element of the array is given by the address of the first element plus its index. Let us using a pointer, navigate an array of 10 integers for recording and displaying the opposite of each integer.

```

#include<iostream>

#include<conio.h>

using namespace std;

main() {

    int t[10]; //declaration of array

    int *i; //pointer to replace index

```

```

cout<<endl<<"Enter 10 integers"<<endl<<endl;
for(i=t;i<(t+10);i++){
    cout<<"Enter integer for adress "<<i<<":";
    cin>>*i;
} //end of loop for recording

cout<<endl<<"The      opposites      of      entered      integers
are:"<<endl<<endl;

for(i=t;i<(t+10);i++)
    cout<<*i*-1<<"\t";

getch();
}

```

Illustrative output

```

Enter 10 integers
Enter integer for adress 0x22ff40:10
Enter integer for adress 0x22ff44:23
Enter integer for adress 0x22ff48:-4
Enter integer for adress 0x22ff4c:56
Enter integer for adress 0x22ff50:-70
Enter integer for adress 0x22ff54:-19
Enter integer for adress 0x22ff58:50
Enter integer for adress 0x22ff5c:90
Enter integer for adress 0x22ff60:-95
Enter integer for adress 0x22ff64:100

The opposites of entered integers are:
-10    -23    -4    -56    -70    19    -50    -90    95    -100

```

NB: The values following the message “Enter integer for address” are the addresses of elements of the arrays. **E.g.:** 0x22ff40

- Another aspect expressing the relationship between array and pointer is that basing on a pointer an array of SIZE elements can be created. This way of creating an array basing on a pointer is called “**Dynamic memory allocation**” in other words that array is dynamic. The function **new()** is used to transform a pointer into an array. Let us modify the

program used to illustrate the first aspect where instead of using 10 as size of the array, user himself will determine the size of the array.

```
#include<iostream>
#include<conio.h>
using namespace std;
main(){
    int *t;//declaration of a pointer
    int i,N;//i for index and N for size
    do{
        cout<<"How many?:";
        cin>>N;
    }while(N<=0); //constrainting user to record a size >=1
    cout<<endl<<"Enter "<<N<<" integers"<<endl<<endl;
    t=new int[N]; //Allocate memory for N integers
    for(i=0;i<N;i++){
        cout<<"Enter integer#"<<i+1<<" ":";
        cin>>t[i];
    } //end of loop for recording
    cout<<endl<<"The opposites of entered integers
are:"<<endl<<endl;
    for(i=0;i<N;i++)
        cout<<t[i]*-1<<"\t";
    getch();
}
```

Illustrative output

```
How many?:5
Enter 5 integers
Enter integer#1:12
Enter integer#2:34
Enter integer#3:-90
Enter integer#4:25
Enter integer#5:34

The opposites of entered integers are:
-12    -34    90    -25    -34
```

4.5. POINTERS AND STRUCTURES

As in previous section we discussed about the usage of pointers in arrays creation and manipulation, this section does the same. It is important to notice that once the variable to be used for structure manipulation is a pointer, the symbol (->) is placed between the structure variable name and its structure member.

The following codes explain more about the manipulation of members of a structure basing using a pointer:

```
#include<iostream>
#include<conio.h>
using namespace std;

struct Students
{
    int Roll_number;
    char First_name[20];
    char Second_name[16];
    char Sex;
    char Department_code[5];
};

main(){
    Students *Student_data;//declaration of poiter
    Student_data=new(Students);//Allocation of memory
    cout<<"Enter the roll number:";
    cin>>Student_data->Roll_number;
    cout<<"First name:";
    cin>>Student_data->First_name;
```

```

    cout<<"Second name:";
    cin>>Student_data->Second_name;
    cout<<"Sex:";
    cin>>Student_data->Sex;
    cout<<"The department:";
    cin>>Student_data->Department_code;
    cout<<endl<<"You      entered:"<<endl<<"-----"
"<<endl;

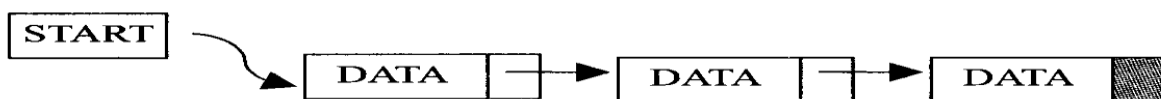
    cout<<Student_data->Roll_number<<endl;
    cout<<Student_data->First_name<<endl;
    cout<<Student_data->Second_name<<endl;
    cout<<Student_data->Sex<<endl;
    cout<<Student_data->Department_code<<endl;
    getch();
}

```

The usage of pointers and structures becomes compulsory when programmer has to use the linked lists in his program.

What is a linked list?

A linked list is a dynamic data structure that is defined by programmer in such way there are into computer's memory different blocs of data that are interlinked. It may look like following figure:



Each bloc has the part keeping data and another part keeping the address or reference of the following block of the list.

Why the use of linked list?

Consider a program that has to manage a text file which is subject to be opened for appending (adding) and continual process of its data. The linked list will help to allocate memory that the file needs exactly because each record in a file may be loaded into one bloc of the linked list the more the file is increasing its records more its loading will need many bloc of a linked list. The illustrative examples about linked lists will be seen in the module entitled as “**Data structures and Algorithms**”. Then a rendezvous is given to you in second year computer science department.

4.6. EXERCISES

- A) Using C++, write a program able to display the prime numbers entered in an array of N integers. Once there is no prime number the program must display the message “NOTHING TO BE DISPLAYED”.
- B) Use C++ to write a program that will allow user to record N integers in an array and get at the output the recorded integers associated with their number of occurrences. E.g.: assume that N=5 and the array contains the integers 10, 14,12,10,14. At the output there must be a display looking like followed:
10:2 occurrences
14: 2 occurrences
12: 1 occurrence
- C) Write a program in C++ that will allow user to record names and the brut salary of N employees. The program must print the list of employee organized by ascending order of their brut salary.

D) Create in C++ a program that will allow user to record integers in an array of N elements and get the output described as followed:

N=6, content of array is 9,-6, 5,-3, 2, 7 then the output must look like

9	-6	5	etc.
---	----	---	------

1	-1-1-1-1-1-1	1
---	--------------	---

22	-2-2-2-2-2	22
----	------------	----

33	-3-3-3-3	333
----	----------	-----

4444	-4-4-4	4444
------	--------	------

55555	-5-5	55555
-------	------	-------

666666	-6
--------	----

7777777

88888888

999999999

CHAPTER 5: FUNCTIONS

5.1. INTRODUCTION

The function usage in programming has evolved with the **modular programming** concept. Modular Programming is the act of designing and writing programs using functions, that each one performs a single well-defined task. A function is a set of codes that are independently executed under the name of the function. Every C++ program has at least one function which is **main()**, and all the most trivial programs can define additional functions.

The main advantages of functions usage in programming are **task sharing** and **debug facilitation** in program creation process.

Depending on the nature of the task it has to perform, a function can **return** a value or not. In case a function doesn't have to return a value, it is called **procedure** and is declared with the type **void** while a function having to return a value is declared with other types like int, float, char, double, etc. the type of a function specifies the type of the value it has to return. Briefly the meaning of void is "nothing" which means that the function will not return any value.

What is the meaning of returning a value?

To return a value is an action performed by a statement having as syntax `return value or expression.`

Example: `return 0;` or `return n*2;`

Once this statement is executed, the execution of the concerned function is ended and a space of memory containing the returned value is temporally created, and that value is accessible and usable by the function that has triggered (calling function) that execution.

Before determining functions and procedures of a program, the general problem concerned by that program has to be divided into sub problems. The solution of each sub problem will constitute a function.

5.2. FUNCTION DECLARATION, DEFINITION AND CALL

The usage of functions in programming implicates three actions which are **declaration**, **definition** and **call**.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function or in other words the codes to realize the task of the function while the **call** constitutes a statement that triggers the execution of a defined function after all the parameters are initialized to the values of **arguments**.

Declaration of a function

The same as variables are declared before they are manipulated, functions are also declared before they are defined and called. However the same as a variable can be declared and initialized with the same statement (e.g.: `int b=10;` in stead of `int b;` first then `b=10` after), it is possible to combine declaration and definition of a function into one stage of the program.

Syntax of function declaration:

```
Return_type function_Name(parameters list);
```

E.g.:

```
a) int max(int num1, int num2);
```

```
b) void swap(int *x, int &y);
```

Definition of a function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
    body of the function
}
```

```
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type:** A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters. They constitute the basic inputs for the function and get values or initialized by argument when the function is called.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Illustrative example:

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
```

```

int max(int num1, int num2)
{
    // local variable declaration
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}

```

Call of a function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. For example:

```

#include <iostream>
using namespace std;

// function declaration
int max(int num1, int num2);

```

```

int main ()
{
    // local variable declaration:
    int a = 100;
    int b = 200;
    int ret;

    // calling a function to get max value.
    ret = max(a, b);

    cout << "Max value is : " << ret << endl;

    return 0;
}

// function returning the max between two numbers
int max(int num1, int num2)
{
    // local variable declaration
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}

```

5.3. PARAMETERS, ARGUMENTS AND SCOPE OF VARIABLES

A function as a set of codes supposed to be executed as an independent entity of a program. It may perform a routine process that can be needed at different level of a program and requiring basic inputs for execution.

Parameters

Parameters as we said above, parameters constitute variables keeping values that are used as basic inputs for the function. Parameters are associated with function declaration and definition. They are one of the ways of coupling the calling with the called function, where the calling function provides the required input to the called function.

A parameter can be a classic variable, a pointer, an address of a variable, an array or a structure.

Arguments

An **argument** is the value that is passed to the function in place of a parameter. When a function is called, all of the parameters of the function are created as variables, and the value of the arguments are copied into the parameters.

There are 3 primary methods of passing arguments to functions: pass by value, pass by reference, and pass by address.

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument is passed to the function.

```
using namespace std;
void foo(int y)
{
    cout << "y = " << y << endl;
}

int main()
{
    foo(5); // first call
```

```

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call

    return 0;
}

```

One way to allow functions to modify the value of argument is by using pass by reference. In **pass by reference**, we declare the function parameters as references rather than normal variables:

```

void AddOne(int &y) // y is a reference variable
{
    y = y + 1;
}

```

When the function is called, y will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

There is one more way to pass variables to functions, and that is by address. **Passing an argument by address** involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

Here is an example of a function that takes a parameter passed by address:

```

void foo(int *pValue)
{
    *pValue = 6;
}

int main()
{
    int nValue = 5;

    cout << "nValue = " << nValue << endl;
    foo(&nValue);
    cout << "nValue = " << nValue << endl;
    return 0; }

```

Scope of variables

A scope is a region of the program and broadly speaking there are three places where variables can be declared:

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

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

int main ()
{
    // Local variable declaration:
    int a, b;
    int c;

    // actual initialization
    a = 10;
    b = 20;
    c = a + b;

    cout << c;

    return 0;
}
```

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the lifetime of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

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

// Global variable declaration:
int g;

int main ()
{
    // Local variable declaration:
    int a, b;

    // actual initialization
    a = 10;
    b = 20;
    g = a + b;

    cout << g;

    return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example:

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

// Global variable declaration:
int g = 20;

int main ()
{
    // Local variable declaration:
    int g = 10;

    cout << g;

    return 0;
}
```


When the above code is compiled and executed, it produces following result:

10

NB: When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initialser
Int	0
Char	'\0'
Float	0
Double	0
Pointer	NULL

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result.

Example of application of functions

Consider illustrative example of using switch case usage that we saw in chapter 3. The problem was described as “*The following program will allow user to record a number and choose one of the following options concerning the output: absolute value, opposite number, double of the number and the square of the number*”. The program is going to be modified in such way it contains functions each one performing a task as described in the following table:

Name of function	Short description of the task
Menu	For displaying the menu for user
AbV	For displaying the absolute value

OppV	For displaying opposite value
DoubleV	For calculating and displaying the double
SquareV	For calculating and displaying the square
Record	For recording the number for which the chosen output will be printed
Main	Compulsory function to make the program run

The codes will look like followed:

```
#include<iostream>

#include<conio.h>

using namespace std;

int Menu();/* declaration of the function Menu*/
int Record();/* declaration of the fuction Record*/
void AbV(int n); /* declaration of the function AbV*/
void OppV(int *n);
void DoubleV(int &n); /* declaration of the function DoubleV*/
void SquareV(int n);/* declaration of the function SquareV*/
main(){

    int n;

    int choice;

    do{

        choice=Menu();

        n=Record();

        switch(choice){

            case 1:

                AbV(n);
```

```

        break;

    case 2:
        OppV(&n);
        break;

    case 3:
        DoubleV(n);
        break;

    case 4:
        SquareV(n);
        break;

    case 5:
        exit(-1);

    default:
        cout<<endl<<"Wrong choice";
        break;

}

}while(choice!=5);

getch();
}

//Definition of functions
int Menu(){
    int choice;

    cout<<endl<<"1 to display absolute value";

```

```

    cout<<endl<<"2 to display opposite number";
    cout<<endl<<"3 to display the double of the number";
    cout<<endl<<"4 to display the square of the number";
    cout<<endl<<"5 to quit program";
    cout<<endl<<endl<<"Your choice please:";
    cin>>choice;
    return choice;
} //end of function Menu

int Record() {
    int n;
    cout<<endl<<"Enter a number:";
    cin>>n;
    return n;
} //end of Function Record

void AbV(int n) {
    if(n>=0)
        cout<<endl<<"The absolute value of "<<n<<" is "<<n;
    else
        cout<<endl<<"The absolute value of "<<n<<" is "<<n*-1;
} //end of Function AbV

void OppV(int *n) {
    cout<<endl<<"The opposite value of "<<*n<<" is "<<*n*-1;
}

void DoubleV(int& n) {

```

```

        cout<<endl<<"The double of "<<n<<" is "<<n*2;
    }

void SquareV(int n){
    cout<<endl<<"The square of "<<n<<" is "<<n*n;}

```

5.4. FUNCTION OVERLOADING

Function overloading is the general concept of c++. **A function can be declared more than once with different operations.** This is called function overloading. In “C” language, the same function name is illegal to declare more than once. But c++ is benefited with this feature. It is the compiler job which one is the right to choose. If it makes sense to you then I should say that one function name for different operations have the advantage of good readability of a program.

Example

```

#include <iostream>
using namespace std;

void print(int i) {
    cout << " Here is int " << i << endl;
}

void print(double f) {
    cout << " Here is float " << f << endl;
}

void print(char* c) {
    cout << " Here is char* " << c << endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
}

```

The following is the output of the above example:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

5.5. INLINE FUNCTIONS

Although you've already learned about basic functions in c++, there is more: the inline function. Inline functions are not always important, but it is good to understand them. The basic idea is to save time at a cost in space. Inline functions are a lot like a placeholder. Once you define an inline function, using the 'inline' keyword, whenever you call that function the compiler will replace the function call with the actual code from the function.

How does this make the program go faster? Simple, function calls are simply more time consuming than writing all of the code without functions. To go through your program and replace a function you have used 100 times with the code from the function would be time consuming not too bright. Of course, by using the inline function to replace the function calls with code you will also greatly increase the size of your program.

Example Inline Function

```
#include <iostream>

using namespace std;

inline void hello()
{
    cout<<"hello";
}

int main()
{
    hello(); //Call it like a normal function...
    cin.get();
}
```

However, once the program is compiled, the call to hello(); will be replaced by the code making up the function.

A WORD OF WARNING: Inline functions are very good for saving time, but if you use them too often or with large functions you will have a tremendously large program. Sometimes large programs are actually less efficient, and therefore they will run more slowly than before. Inline functions are best for small functions that are called often.

5.6. EXERCISES

Using functions, write programs in C++ that will perform the following tasks:

- A) To record in N integers in an array, to display unsorted and sorted content of the array and to search a number in the array where the position of a found number is indicated.
- B) Using C++ language, write a program that will perform the following tasks:
 - 1 To print factorial of an entered number;
 - 2 To attest if an entered number is prime or not;
 - 3 To print the suit of $1!, 2!, \dots, (n-1)!, n!$;
 - 4 To print the suit of $|n|n, (|n|-1)n, \dots, n$; and
 - 5 To quit the program.

Make sure that the concerned number is received into parameter.

- C) A program that displays number of prime, number of positive, number of odd and number of multiple of N integers recorded in a dynamic array.

CHAPTER 6: INTRODUCTION TO OOP

In structured programming, the focus is mainly put on designing functions that manipulate data. However, data and those functions are completely decoupled. What we have seen of C++ so far

belongs to this paradigm. C++ however supports many programming paradigms. Apart from the structured programming, it supports Object - Oriented programming and Generic programming. One of major concepts of Object -Oriented programming is *Encapsulation*: It is the mechanism that binds together the code and the data it manipulates, and keeps both safe from outside interference and misuse. In C++ and a number of other Object -Oriented programming, that mechanism is implemented via the concept of a class.

6.1. A CLASS

A class is an entity that describes data and the functions used to manipulate those data. As the name indicates, the class is a description of the common features of a category of objects. One particular instance of that category is called an object of that class. In other words, an object is an instance of a class. A class definition must be followed either by a semicolon or a list of declarations. For example we defined the Box data type using the keyword **class** as follows:

C++ Class Definitions:

```
class Box
{
    public:
        double length;    // Length of a box
        double breadth;   // Breadth of a box
        double height;    // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;           // Declare Box1 of type Box
Box Box2;           // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try following example to make the things clear:

```
#include <iostream>

using namespace std;

class Box
{
    public:
        double length;    // Length of a box
        double breadth;    // Breadth of a box
        double height;    // Height of a box
};

int main( )
{
    Box Box1;           // Declare Box1 of type Box
    Box Box2;           // Declare Box2 of type Box
    double volume = 0.0;    // Store the volume of a box here

    // box 1 specification
    Box1.height = 5.0;
    Box1.length = 6.0;
    Box1.breadth = 7.0;

    // box 2 specification
    Box2.height = 10.0;
    Box2.length = 12.0;
    Box2.breadth = 13.0;
    // volume of box 1
```

```

    volume = Box1.height * Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.height * Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " << volume <<endl;
    return 0;
}

```

When the above code is compiled and executed, it produces following result:

```

Volume of Box1 : 210
Volume of Box2 : 1560

```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

6.2. CLASS CONSTRUCTOR AND DESTRUCTOR FUNCTIONS

The Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explain the concept of constructor:

```

#include <iostream>

```

```

using namespace std;

class Line
{
    public:
        void setLength( double len );
        double getLength( void );
        Line(); // This is the constructor

    private:
        double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

// Main function for the program
int main( )
{
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;

    return 0;
}

```

When the above code is compiled and executed, it produces following result:

```

Object is being created
Length of line : 6

```

Parameterized Constructor:

A default constructor does not have any parameter but if you need a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
    public:
        void setLength( double len );
        double getLength( void );
        Line(double len); // This is the constructor

    private:
        double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

// Main function for the program
int main( )
{
    Line line(10.0);

    // get initially set length.
```

```

    cout << "Length of line : " << line.getLength() <<endl;
    // set line length again
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;

    return 0;
}

```

When the above code is compiled and executed, it produces following result:

```

Object is being created, length = 10
Length of line : 10
Length of line : 6

```

The Class Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explain the concept of destructor:

```

#include <iostream>
using namespace std;
class Line
{
    public:
        void setLength( double len );
        double getLength( void );
        Line();    // This is the constructor declaration
        ~Line();   // This is the destructor: declaration
}

```

```

    private:
        double length;
};
// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}
double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
    Line line;
    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;
    return 0;
}

```

When the above code is compiled and executed, it produces following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

6.3. CLASSES INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Base & Derived Classes

A class can be derived from more than one class, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public**, **protected**, or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>
```

```

using namespace std;

// Base class
class Shape
{
    public:
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
};

// Derived class
class Rectangle: public Shape
{
    public:
        int getArea()
        {
            return (width * height);
        }
};

int main(void)
{
    Rectangle Rect;

    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;

    return 0;
}

```

When the above code is compiled and executed, it produces following result:

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public**, **protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Multiple Inheritances:

A C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public**, **protected**, or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape
{
    public:
        void setWidth(int w)
        {
            width = w;
        }
}
```

```

        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
};

// Base class PaintCost
class PaintCost
{
    public:
        int getCost(int area)
        {
            return area * 70;
        }
};

// Derived class
class Rectangle: public Shape, public PaintCost
{
    public:
        int getArea()
        {
            return (width * height);
        }
};

int main(void)
{
    Rectangle Rect;
    int area;

    Rect.setWidth(5);
    Rect.setHeight(7);

    area = Rect.getArea();

    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;

    // Print the total cost of painting
    cout << "Total paint cost: $" << Rect.getCost(area) << endl;
}

```

```
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Total area: 35
Total paint cost: $2450
```

6.4. POLYMORPHISM IN C++

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};

class Rectangle: public Shape{
public:
    Rectangle( int a=0, int b=0)
```

```

        {
            Shape(a, b);
        }
        int area ()
        {
            cout << "Rectangle class area :" <<endl;
            return (width * height);
        }
};
class Triangle: public Shape{
public:
    Triangle( int a=0, int b=0)
    {
        Shape(a, b);
    }
    int area ()
    {
        cout << "Rectangle class area :" <<endl;
        return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);

    // store the address of Rectangle
    shape = &rec;
    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;
    // call triangle area.
    shape->area();

    return 0;
}

```

When the above code is compiled and executed, it produces following result:

```
Parent class area
```

Parent class area

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    virtual int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces following result:

Rectangle class area
Triangle class area

This time the compiler looks at the contents of the pointer instead of its type. Hence since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

6.5. DATA ABSTRACTION

Data abstraction refers to, providing only essential information to the outside world and hiding their background details ie. to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players BUT you do not know it's internal detail that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data ie. state without actually knowing how class has been implemented internally.

In C++ we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( )
```

```
{  
    cout << "Hello C++" <<endl;  
    return 0;  
}
```

Here you don't need to understand how **cout** displays the text on the user's screen. You need only know the public interface and the underlying implementation of `cout` is free to change.

Access Labels Enforce Abstraction:

In C++ we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to

see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

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

class Adder{
public:
    // constructor
    Adder(int i = 0)
    {
        total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
        total += number;
    }
    // interface to outside world
    int getTotal()
    {
        return total;
    };
private:
    // hidden data from outside world
    int total;
};

int main( )
{
    Adder a;

    a.addNum(10);
    a.addNum(20);
    a.addNum(30);

    cout << "Total " << a.getTotal() << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

6.6. DATA ENCAPSULATION

All C++ programs are composed of following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private**, **protected** and **public** members. By default, all items defined in a class are private. For example:

```
class Box
{
    public:
        double getVolume(void)
```

```
    {  
        return length * breadth * height;  
    }  
private:  
    double length;        // Length of a box  
    double breadth;       // Breadth of a box  
    double height;        // Height of a box  
};
```

The variables `length`, `breadth`, and `height` are **private**. This means that they can be accessed only by other members of the `Box` class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction.

CHAPTER 7: FILE PROCESSING

7.1. NOTIONS OF FILES IN PROGRAMMING

Up to now, the program we were creating was supposed to keep data only into volatile memory. This means that once the program is closed the recorded data are lost. Files constitute objects that are used to keep data even when the program is no longer running. In other words, file processing will allow programmer to use both volatile and storage memories. In file processing, a file can be created, opened, loaded, updated and closed.

7.2. FILE MANIPULATION STREAMS

The streams of C++ we were familiar with were `cin>>` and `cout<<` with inclusion of the header file `<iostream>`. However these streams were used for reading data from an input device like keyboard and writing information on the output device like monitor. To manipulate files there is need of including the header file `<fstream>` in your program. This header file provides to the program in which it is included, the file streams that are used to manipulate files. File manipulation implicates opening, loading, writing and closing it.

Opening a file

Generally, before any other operation is performed on a file, it is opened. C++ supports that programmer specifies the mode in which the file will be opened. These mode are presented in the following table:

Mode	Description
<code>ios::in</code>	Open for input operations.
<code>ios::out</code>	Open for output operations.
<code>ios::binary</code>	Open in binary mode.
<code>ios::ate</code>	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
<code>ios::app</code>	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for

	output-only operations.
<code>ios::trunk</code>	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters). Since the first task that is performed on a file stream object is generally to open a file, these three

classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a Boolean value of `true` in the case that indeed the stream object is associated with an open file, or `false` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

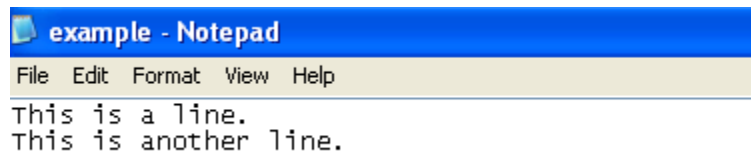
Data output operations on text files are performed in the same way we operated with `cout`:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n";
        myfile << "This is another line.\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

Output

The file called `example.txt` is created and its location will be the same one as the source code file of this program.



Data input from a file can also be performed in the same way that we did with `cin`:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( myfile.good() )
        {
            getline (myfile,line);
            cout << line << endl;
        }
        myfile.close();
    }
}
```

```
else cout << "Unable to open file";  
  
return 0;  
}
```

The output



Comments

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `good()` that returns true in the case that the stream is ready for input/output operations. We have created a while loop that finishes when indeed `myfile.good()` is no longer true, which will happen either if the end of the file has been reached or if some other error occurred.

Checking state flags

In addition to `good()`, which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a Boolean value):

`bad()`

Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

`fail()`

Returns true in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

`eof()`

Returns true if a file open for reading has reached the end.

`good()`

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

Get and put stream pointers

All input/output streams objects have, at least, one internal stream pointer: `ifstream`, like `istream`, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.

`ofstream`, like `ostream`, has a pointer known as the *put pointer* that points to the location where the next element has to be written.

Finally, `fstream`, inherits both, the get and the put pointers, from `iostream` (which is itself derived from both `istream` and `ostream`).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type `pos_type`, which is an integer data type representing the current position of the get stream pointer (in the case of `tellg`) or the put stream pointer (in the case of `tellp`).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );  
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions `tellg` and `tellp`: the member type `pos_type`, which is an integer value.

The other prototype for these functions is:

```
seekg ( offset, direction );  
seekp ( offset, direction );
```

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter `direction`. The `offset` is of the member type `off_type`, which is also an integer type. And `direction` is of type `seekdir`, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

<code>ios::beg</code>	offset counted from the beginning of the stream
<code>ios::cur</code>	offset counted from the current position of the stream pointer
<code>ios::end</code>	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```
// obtaining file size  
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main () {  
    long begin,end;  
    ifstream myfile ("example.txt");  
    begin = myfile.tellg();  
    myfile.seekg (0, ios::end);  
    end = myfile.tellg();  
    myfile.close();  
    cout << "size is: " << (end-begin) << " bytes.\n";  
    return 0;  
}
```

Output

A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\Files3.exe". The command prompt itself is black with white text. The first line shows the command "size is: 40 bytes." followed by a newline character.

```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\Files3.exe
size is: 40 bytes.
```

Binary files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like `getline` is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: `write` and `read`. The first one (`write`) is a member function of `ostream` inherited by `ofstream`. And `read` is a member function of `istream` that is inherited by `ifstream`. Objects of class `fstream` have both members. Their prototypes are:

```
write ( memory_block, size );
read  ( memory_block, size );
```

Where `memory_block` is of type "pointer to char" (`char*`), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The `size` parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
    ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
    if (file.is_open())
```

```

{
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "The complete file content is in memory"<<endl;
    cout<<memblock;

    delete[] memblock;
}
else cout << "Unable to open file";
cin.get();
return 0;
}

```

Clarifications

In this example the entire file is read and stored in a memory block. Let's examine how this is done:

First, the file is open with the `ios::ate` flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member `tellg()`, we will directly obtain the size of the file. Notice the type we have used to declare variable `size`:

```
ifstream::pos_type size;
```

`ifstream::pos_type` is a specific type used for buffer and file positioning and is the type returned by `file.tellg()`. This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use `int`:

```
int size;
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);  
file.read (memblock, size);  
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

7.3. THE USE OF STRUCTURES AND ARRAYS IN FILES PROCESSING

The following program will allow user to record names, sex and age of clients in a text file.

```
#include<iostream>  
#include<fstream>  
using namespace std;  
struct person{  
    char names[20];  
    char sex;  
    int age;  
};  
main(){  
    person data[5];
```

```

int i;

ofstream file;

cout<<"Enter data for 5 clients"<<endl;

file.open("client.txt");

//writing on file the content of array of structure
for(i=0;i<5;i++){

    cout<<"Names#"<<i+1<<":";

    cin>>data[i].names;

    cout<<"Sex#"<<i+1<<":";

    cin>>data[i].sex;

    cout<<"Age#"<<i+1<<":";

    cin>>data[i].age;

    //writing on file client.txt
    file<<data[i].names<<"\t"<<data[i].sex<<"\t"<<data[i].age<<"\n";

}

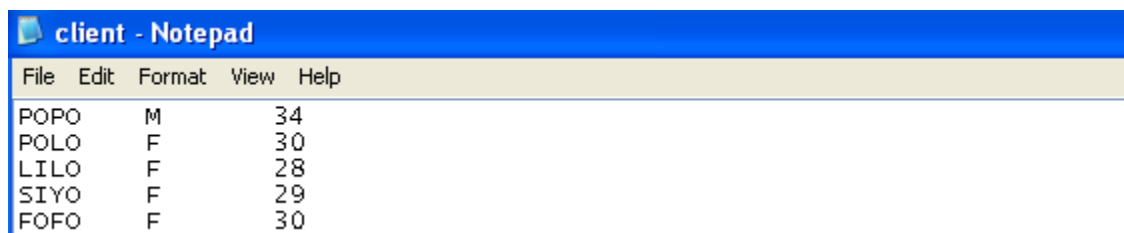
    file.close();

}

```

The output

The program presented here above will create a file called “client.txt” as it is shown here bellow. By default the location of the created file is the same as the one of the source codes file of the program.



client - Notepad		
File	Edit	Format View Help
POPO	M	34
POLO	F	30
LILO	F	28
SIYO	F	29
FOFO	F	30

The following codes will read the content of the file client.txt.

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
main(){
    string record;
    ifstream file;
    file.open("client.txt");
    if(file.is_open()){
        cout<<"The content of the file"<<endl;
        cout<<"-----"<<endl;
        while(file.good()){
            getline (file,record);
            cout<<record<<endl;
        }
    }
    else
        cout<<"Error in file opening";
    cin.get();
}
```

The output

```
C:\Documents and Settings\Gaston\Desktop\COURSE 2013\C++\ReadingFile.exe
The content of the file
-----
POPO M      34
POLO F      30
LILO F      28
SIYO F      29
FOFO F      30
```

7.4. EXERCISES

- A) What is a file? What is its importance in programming?
- B) What is the difference between a text and binary files?
- C) Write a program that will write on a file the content of an array of N integers. Use C++ of course.
- D) Using C++, write a program that will allow user to manage the contacts of his customers knowing that the contact of each customer is described by name, type (“company” or “individual”), telephone number. The management consists of adding contact, searching customer and displaying the content of the file.