Basics of C++ Programming

History of C++

C++ is an object oriented programming language. It was called “C with class”. C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both languages and create a more powerful language that could support object oriented programming features and still remain power of C. The result was C++. The major addition was class to original C language, so it was also called “C with classes”. Then in 1993, the name was changed to C++. The increment operator ++ to C suggest an enhancement in C language.
C++ can be regarded as superset of C (or C is subset of C++). Almost all C programs are also true in C++ programs.

C++ program Construction.

Before looking at how to write C++ programs consider the following simple example program.
// Sample program
// Reads values for the length and width of a rectangle
// and returns the perimeter and area of the rectangle.

```
#include <iostream.h> //for cout and cin
#include <conio.h> //for getch()
void main()
{
  int length, width;
  int perimeter, area;              // declarations
  cout <<  "Length = ";             // prompt user
  cin >> length;                    // enter length
  cout << "Width = ";               // prompt user
  cin >> width;                     // input width
  perimeter = 2*(length + width);     // compute perimeter
  area = length*width;              // compute area
  cout << endl
       << "Perimeter is " << perimeter;
  cout << endl
       << "Area is " << area
       << endl;                    // output results
  getch();
} // end of main program
```

The following points should be noted in the above program:

1. **Single line comment** (// ………….)
   Any text from the symbols // until the end of the line is ignored by the compiler. This facility allows the programmer to insert **Comments** in the program. Any program that is not very simple should also have further comments

indicating the major steps carried out and explaining any particularly complex piece of programming. This is essential if the program has to be extended or corrected at a later date. This is a kind of documentation. Also C comment type /*--------------*/ is also a valid comment type in C++.

2. The line

   #include <iostream.h>

   must start in column one. It causes the compiler to include the text of the named file (in this case iostream.h) in the program at this point. The file iostream.h is a system supplied file which has definitions in it which are required if the program is going to use stream input or output. All programs will include this file. This statement is a **preprocessor directive** -- that is it gives information to the compiler but does not cause any executable code to be produced.
3. The actual program consists of the **function** main() which commences at the line void main( ).

   All programs must have a function main( ). Note that the opening brace ({) marks the beginning of the body of the function, while the closing brace (}) indicates the end of the body of the function. The word void indicates that main() does not return a value. Running the program consists of obeying the statements in the body of the function main().
4. The body of the function main contains the actual code which is executed by the computer and is enclosed, as noted above, in braces {}.
5. Every statement which instructs the computer to do something is terminated by a semi-colon. Symbols such as main(), { } etc. are not instructions to do something and hence are not followed by a semi-colon. Preprocessor directives are instruction to the compiler itself but program statements are instruction to the computer.
6. Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as

   cout << "Length = "

   send the quoted characters to the output stream cout. The special identifier endl when sent to an output stream will cause a newline to be taken on output.
7. All variables that are used in a program must be declared and given a type. In this case all the variables are of type int, i.e. whole numbers. Thus the statement

   int length, width;

   declares to the compiler that integer variables length and width are going to be used by the program. The compiler reserves space in memory for these variables.

Variable Definition at the point of use.

In C, local variables can only be defined at the top of a function, or at the beginning of a nested block. In C++, local variables can be created at any position in the code, even between statements. Also local variables can be defined in some statements, just before their usage.

```
//to find sum of first natural number to display for loop
#include<iostream.h>
#include<conio.h>
void main()
{
      int n;
      cout<<"\nEnter Number";
      cin>>n;
      int sum=0;

      for( int i=0;i<=n;i++)
             sum+=i;
      cout<<"Sum of first n natural number:"<<sum;
      getch();
}
```

8. Values can be given to variables by the **assignment** statement, e.g. the statement
   area = length*width;
   evaluates the expression on the right-hand side of the equals sign using the current values of length and width and assigns the resulting value to the variable area.
9. Layout of the program is quite arbitrary, i.e. new lines, spaces etc. can be inserted wherever desired and will be ignored by the compiler. The prime aim of additional spaces, new lines, etc. is to make the program more readable. However superfluous spaces or new lines must not be inserted in words like main, cout, in variable names or in strings.

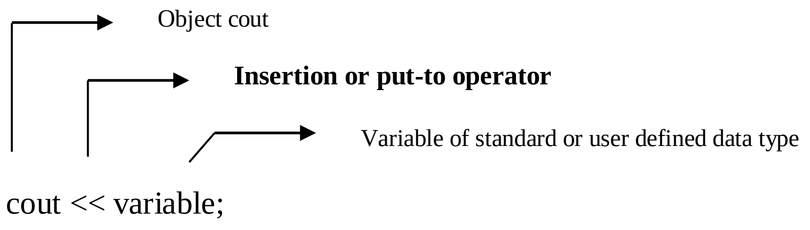
Input and Output:

C++ Supports rich set of functions for performing input and output operations. The syntax using these I/O functions is totally consistent of the device with I/O operations are performed. C++'s new features for handling I/O operations are called streams. Streams are abstractions that refer to data flow. Streams in C++ are :

- Output Stream
- Input Stream.

Output Stream:

The output stream allows us to write operations on output devices such as screen, disk etc. Output on the standard stream is performed using the cout object. C++ uses the bit-wise-left-shift operator for performing console output operation. The syntax for the standard output stream operation is as follows:
cout<<variable;
The word cout is followed by the symbol << , called the insertion or put to operator , and then with the items (variables, constants, expressions) that are to be output. Variables can e of any basic data types. The us of *cout* to perform an output operation is as shown :


Object cout
Insertion or put-to operator
Variable of standard or user defined data type
cout << variable;

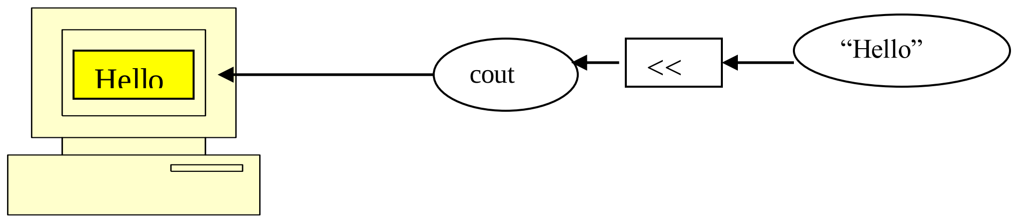


Figure: output with cout

The following are example of stream output operations:

1. cout << "hello world";
2. int age;
   cout<< age;
3. float weight;
   cout<< weight;

etc.

More than one item can be displayed using a single cout output stream object. Such out put operations in C++ are called cascaded output operations. For example

cout<<"Age is: "<<age<<"years";

This cout object will display all the items from left to right. If value of age is 30 then this stream prints

Age is : 30 years

C++ does not restricts the maximum number of items to output. The complete syntax of the standard output streams operation is as follows:

cout<<variable1<<vaariable2<<………..<<variableN;

The object cout must be associated with at least one argument. Like printf in C, A constant value can be sent as an argument to the cout object.

e.g.

cout<<'A'; //prints a constant character A

cout<<10.99; //Prints constant 10.99

cout<<" "; //prints blanks

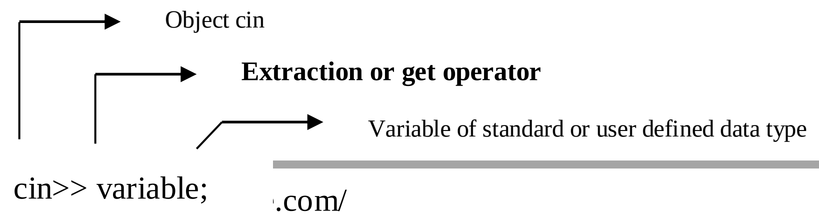
cout<<"\n", //prints new line

Input Streams:

The input stream allows us to perform read operations with input devices such as keyboard, disk etc. Input from the standard stream is performed using the cin object. C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for the standard output stream operation is as follows:

cin<<variable;

The word cin is followed by the symbol >> and then with variable, into which input data is to be stored. The use of *cin* to perform an input operation is as shown :


Object cin
Extraction or get operator
Variable of standard or user defined data type
cin>> variable;

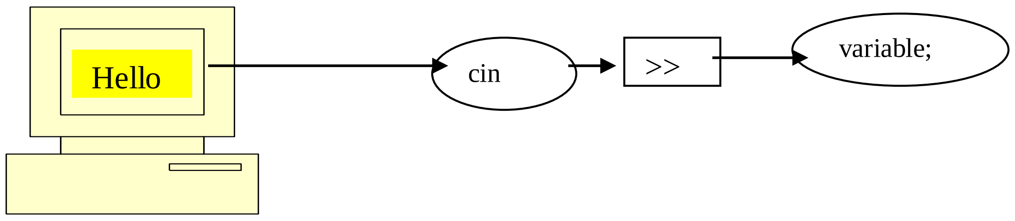


Figure: Input with cin

Following Examples show the stream input operations;

1. int amount;
   cin>>amount;
2. float weight;
   cin>>weight;
3. char name[20];
   cin>>name;   etc

Input of more than one item can also be performed using the cin input stream object. Such input operation in C++ are called cascaded input operations. For example, reading the name of a person his address, age can be performed by the cin as:

cin>> name>>address>>age;

The cin object reads the items from left to right. The complete syntax of the standard input streams operations is as follows:

cin>>var1>>var2>>var3>>……………….>>varN;

e.g. cin>>i>>j>>k>>l;

The following are two important points to be noted about the stream operations.

- Streams do not require explicit data type specification in I/O statement.
- Streams do not require explicit address operator prior to the variable in the input statement.

In C printf and scanf functions, format strings (%d,%s,%c etc) and address operator (&) are necessary but in cin stream format specification is not necessary and in the cout stream format specification is optional. **Format-free I/O is special features of C++ which make I/O operation comfortable.**

Note: The operator << and >> are the bit-wise left-shift and right-shift operators that are used in C and C++. In C++ the operator can be overloaded i.e. same operator can perform different activities depending on the context.

Manipulators:

Manipulators are instructions to the output stream that modify the output in various ways. For example endl , setw etc.

The endl Manipulator:

The endl manipulator causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line. It has same effect as sending the '\n' character but somewhat different. It is a manipulator that sends a newline to the stream

and *flushes* the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output). Unlike '\n' it also causes the output buffer to be flushed but this happens invisibly.
e.g.

```
cout << endl<< "Perimeter is " << perimeter;
  cout << endl << "Area is " << area << endl;
```

The setw Manupulator:

To use setw manipulator the "iomanip.h" header file must be included. The setw manipulator causes the number or string that follows it in the stream to be printed within a field n characters wide , where n is the argument used with setw as stew(n). The value is right justified within the field.
e.g.

```
//demonstrates setw manipulator
#include<iostream.h>
#include<iomanip.h>
void      main()
{
           long pop1=5425678,pop2=47000;pop3=76890;
            cout<<setw(8)<<"LOCATION"<<setw(12)<< "POPULATION"<<endl
          <<setw(8)<< "Patan"<<setw(12)<<pop1<<endl
          <<setw(8)<< "Khotang" <<setw(12)<<pop2<<endl
          <<setw(8)<< "Butwal" <<setw(12) <<pop3<<endl;
}
```

The output of this program is:

```
LOCATION   POPULATION
   Patan          5425678
 Khotang           47000
  Butwal          76890
```

Manipulators come in two flavors: those that take and arguments and those that don't take arguments. Followings are the some important non-argument manipulators .

Table: No-argument Manipulators

Manipulators	Purpose
Ws	Turn on whitespace skipping on input
Dec	convert to decimal
Oct	Convert to octal
Hex	convert to Hexadecimal
Endl	insert newling and flush the output stream
Ends	Insert null character to terminate an output string
Flush	flush the output stream
Lock	lock the file handle
Unlock	Unlock the file handle

```
//demonstrates  manipulators
#include<iostream.h>
#include<iomanip.h>
void main()
{
  long pop1=2425678;
  cout<<setw(8)<<"LOCATION"<<setw(12)<< "POPULATION"<<endl
          <<setw(8)<< "Patan"<<setw(12)<<hex<<pop1<<endl
          <<setw(8)<< "Khotang" <<setw(12)<<oct<<pop1<<endl
          <<setw(8)<< "Butwal" <<setw(12) <<dec<<pop1<<endl;
}
```

Output:

```
LOCATION  POPULATION
   Patan      25034e
 Khotang    11201516
  Butwal     2425678
```

There are manipulators that take arguments for example setw(), setfill() setprecision() etc e.g.

cout<<setw(12)<<setfill(64)<<"string";

In this statement setw(12) describes the field width 12 and setfill(64) fills the blanks with character specified with integer arguments. The output of this statement is :

@@@@@@string where @ is the character corresponding to int 64

cout<<setprecision(5)<<12.456789;

The output of this statement is : 12.456 i.e. setprecision() describes the no of digits to be displayed.

Data Types:

Variables

A variable is the name used for the quantities which are manipulated by a computer program. i.e. it is a named storage location in memory. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

In order to distinguish between different variables, they must be given **identifiers**, names which distinguish them from all other variables. The rules of C++ for valid identifiers state that:

An identifier must:

- start with a letter
- consist only of letters, the digits 0-9, or the underscore symbol _
- not be a **reserved word**

For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers.

The following are valid identifiers

length days_in_year DataSet1 Profit95
Int _Pressure first_one first_1
although using _Pressure is not recommended.
The following are invalid:
days-in-year 1 data int first .val throw
Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word main in a program is quite different from the word Main or the word MAIN.

Reserved words

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

C++ Reserved Words

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	const	const_cast	continue
default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

Some of these reserved words may not be treated as reserved by older compilers. However it is better to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the PC, which add near, far, huge, cdecl, and pascal.

Declaration of variables

In C++ (as in many other programming languages) all the variables that a program is going to use must be **declared** prior to use. Declaration of a variable serves two purposes:

- It associates a **type** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.

- It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

Scope Resolution Operator(::)

C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the scope resolution operator.

:: GloabalVariableName

The global variable to be accessed must be preceded by the scope resolution operator. It directs the compiler to access a global variable, instead of one defined as a local variable. The scope resolution operator permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

```
//An example of use of scoperesolution operator ::
#include<iostream.h>
#include<conio.h>
int x=5;
void main()
{
     int x=15;
     cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
     {
          int x=25;
          cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
     }
     cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
     cout<<”Global +Local=”<<::x +x;

   getch();
}
```

Reference Variables:
C++ introduces a new kind of variable known as reference variable. A reference variable provides an alias (Alternative name) of the variable that is previously defined. For example, if we make the variable **sum** a reference to the variable **total**, the **sum** and **total** can be used interchangeably to represent that variable.

Syntax for defining reference variable
Data_type & reference_name = variable_nane

Example:

```
   int total=100 ;
   int &sum=total;
```

Here total is int variable already declared. Sum is the alias for variable total. Both the variable refer to the same data 100 in the memory.

```
cout<<total;  and cout<<sum; gives the same output 100.
 And total= total+100;
Cout<<sum;    //gives output 200
```

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it means. The initialization of reference variable is completely different from assignment to it.
A major application of the reference variables is in passing arguments to function.

```
//An example of reference
#include<iostream.h>
#include<conio.h>

void main()
{
      int x=5;
      int &y=x;
      //y is alias of x
      cout<<"x="<<x<<"and y="<<y<<endl;
      y++;
      //y is reference of x;
      cout<<"x="<<x<<"and y="<<y<<endl;
      getch();
}
```

Passing by reference

We can pass parameters in function in C++ by reference .When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function i.e. when function is working with its own arguments , it is actually working on the original data.
Example

```
void fun(int &a)
//a is reference variable
{
     a=a+10;
}
```

```
 int main()
 {
     int x=100;
     fun(x);  //CALL
     cout<<x;  //prints 110
}
```

when function call fun(x) is executed, the following initialization occurs:
`int &a=x;` i.e. a is an alias for x and represent the same data in memory. So updating a in function causes the update the data represented by x. this type of function call is known as *Call by rererence*.

```
//pass by reference
#include<iostream.h>
#include<conio.h>
void swap(int &, int &);
void main()
{
      int a=5,b=9;
      cout<<"Before Swapping: a="<<a<<" and b="<<b<<endl;
      swap(a,b);//call by reference
      cout<<"After Swapping: a="<<a<<" and b="<<b<<endl;
      getch();
}

void swap(int &x, int &y)
{
      int temp;
      temp=x;
      x=y;
      y=temp;
}
```

Return by reference

A function can return a value by reference. This is a unique feature of C++. Normally function is invoked only on the right hand side of the equal sign. But we can use it on the left side of equal sign and the value returned is put on the right side.

```
//returning by reference from a function as a parameter
#include<iostream.h>
#include<conio.h>
int x=5,y=15;//globel variable
int &setx();
void main()
{
      setx()=y;
```

```
   //assign value of y to the variable
   //returned by the function
   cout<<"x="<<x<<endl;
   getch();
}
int &setx()
{
   //display global value of x
      cout<<"x="<<x<<endl;
      return x;
}
```

Inline Function:

A inline function is a short-code function written and placed before main function and compiled as inline code. The prototyping is not required for inline function. It starts with keyword **inline** . In ordinary functions, when function is invoked the control is passed to the calling function and after executing the function the control is passed back to the calling program.

But , when inline function is called, the inline code of the function is inserted at the place of call and compiled with the other source code together. That is the main feature of inline function and different from the ordinary function. So using inline function executing time is reduced because there is no transfer and return back to control. But if function has long code inline function is not suitable because it increases the length of source code due to inline compilation.

```
// Inline Function
//saves memory, the call to function cause the same code to be
//executed;the function need not be duplicated in memory
#include<iostream.h>
#include<conio.h>
inline float lbstokg(float pound)
{
      return (0.453592*pound);
}

void main()
{
      float lbs1=50,lbs2=100;
      cout<<"Weinght in Kg:"<<lbstokg(lbs1)<<endl;
      cout<<"Weinght in Kg:"<<lbstokg(lbs2)<<endl;
      getch();
}
```

Default Arguments

In C++ a function can be called without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. The default value are specified when function is declared.

The default value is specified similar to the variable initialization . The prototype for the declaration of default value of an argument looks like

```
float  amount(float p, int  time, float rate=0.10);
```

// declares a default value of 0.10 to the argument rate.

The call of this function as

```
value = amount(4000,5);// one argument missing for rate
```

passes the value 4000 to p , 5 to time and the function looks the prototype for missing argument that is declared as default value 0.10 the the function uses the default value 0.10 for the third argument. But the call

```
 value = amount(4000,5,0.15);
```

no argument is missing , in this case function uses this value 0.15 for rate.

Note : only the trailing arguments can have default value. We must add default from right to left. E.g.

```
int add( int  a,  int b =9, int  c= 10 );  // legal
int add(int a=8, int b, int c); // illegal
int add(int a, int b = 9, int c);  //illegal
int add( int a=8, int b=9,int c=10) // legal
```

```
//default arguments in function
//define default values for arguments that are not passed when
//a function call is made
#include<iostream.h>
#include<conio.h>
void marks_tot(int m1=40,int m2=40, int m3=40 );
void main()
{
   //imagine 40 is given if absent in exam.
      marks_tot();
   marks_tot(55);
   marks_tot(66,77);
   marks_tot(75,85,92);
   getch();
}

void marks_tot(int m1, int m2, int m3)
{
      cout<<"Total marks"<<(m1+m2+m3)<<endl;
}
```

Const Arguments

When arguments are passed by reference to the function, the function can modify the variables in the calling program. Using the constant arguments in the function , the variables in the calling program can not be modified. const qualifier is used for it.

e.g.

```
void func(int&,const int&);
 void  main()
{
     int a=10, b=20;
     func(a,b);
}
void func(int& x, int &y)
{
     x=100;
     y=200; // error  since y is constant argument
}
```

Function overloading

Overloading refers to the use of same thing for different purpose. When same function name is used for different tasks this is known as function overloading. Function overloading is one of the important feature of C++ and any other OO languages.

When an overloaded function is called the function with matching arguments and return type is invoked.

e.g.

void border(); //function with no arguments
void border(int); // function with one int argument
void border(float); //function with one float argument
void border(int, float);// function with one int and one float arguments

For overloading a function prototype for each and the definition for each function that share same name is compulsory.

```
//function overloading
//multiple function with same name
#include<iostream.h>
#include<conio.h>
int max(int ,int);
long max(long, long);
float max(float,float);
char max(char,char);
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}

int max(int i1, int i2)
{
     return(i1>i2?i1:i2);
}

long max(long l1, long l2)
{
     return(l1>l2?l1:l2);
}

float max(float f1, float f2)
{
     return(f1>f2?f1:f2);
}

char max(char c1, char c2)
{
     return(c1>c2?c1:c2);
}
```

Structure review

An structure is an user defined data type which contains the collection of different types of data under the same name. A structure is compared with records in other languages.

Syntax:

```
        struct  tag-name
        {
                data-type var-name;
                data-type var-name;
                ……………………..
        } ; //end of structure
 e.g. struct currency
        {
                int rs;
                float ps;
        };
```

declaring variable of structure
 tag-name st-var-name;   e.g. currency c1,c2;

Initialization of structure variable: structure variable can be initialized at the time of declaraction as
currency c1 ={ 144,56.7};
 However each member data of structure variable can be initialized separately after declaration as
currency c1;
c1.rs=144;
c1.ps=56.7;
But initialization all member data at once listing is illegal after declaration as
currency c1;
c1={144,56.7}   // error:

Computation using structure

A simple Example.

```
#include<iostream.h>
 struct currency
{
      int rupees;
      float paise;
};          // currency is name for struct currency

void main()
{
      currency c1,c3;
      currency c2 ={123,56.4};
      cout<<"Enter Rupees:"; cin>> c1.rupees;
      cout<<"Enter paises"; cin>> c1. paise;
      c3.paise = c1.paise+ c2.paise;
      if(c3.paise>=100.0)
      {
       c3.paise-=100.0 ;
       c3.rupees++;
      }
      c3.rupees+=c2.rupees+c1.rupees;
      cout<<"Rs." <<c1.rupees<<" Ps. " <<c1.paise<<" + ";
      cout<<"Rs." <<c2.rupees<<" Ps. "<<c2.paise<<" = ";
      cout<< "Rs."<<c3.rupees<<" Ps."<<c3.paise<<endl;
}
```

Passing structure as function arguments

A function can receive the structure as parameter and is able to access and operate on the individual elements of the structure. When passing the structure variable as the function argument the whole variable is not passed but only the reference (address) of the structure variable is passed. The following example shows the passing structure as function argument.

```
//passing structure as function argument
#include<iostream.h>
#include<conio.h>
struct currency
{
      int rs;
      float ps;
};

currency addcurr( currency,currency );
void main()
{
      currency c3;
      currency c1={100,58.5};
      currency c2={200,62.8};
      c3 = addcurr(c1,c2);
      cout<<"Sum is Rs."<<c3.rs<<"Ps."<<c3.ps<<endl;
   getch();
}
currency addcurr(currency cc1, currency cc2)
{
      currency cc3={0,0.0};
      cc3.ps=cc2.ps+cc1.ps;
      if(cc3.ps>=100.0)
      {
            cc3.ps-=100.0;
            cc3.rs++;
      }
      cc3.rs+=cc1.rs+cc2.rs;
 return cc3;
 }
```