CSE202: OBJECT ORIENTED PROGRAMMING

Lecture 6-7

What is function????

- Function is a self contained block of statements that perform a coherent task of some kind.
- Every C++ program can be a thought of the collection of functions.
- main( ) is also a function.

Types of Functions.

- Library functions
  - These are the in- -built functions of ‘C++ ’library.
  - These are already defined in header files.
  - e.g. Cout<<; is a function which is used to print at output. It is defined in ‘iostream.h ’ file .
- User defined functions.
  - Programmer can create their own function in C++ to perform specific task

Why use function?

- Writing functions avoids rewriting of the same code again and again in the program.
- Using function large programs can be reduced to smaller ones. It is easy to debug and find out the errors in it.
- Using a function it becomes easier to write program to keep track of what they are doing.

```
//Function Declaration
  retn_type  func_name(data_type 1,data_type par2);

//Function Defination
rent_type  func_name(data_type
  par1,data_type par2)
{
// body of the function
}

//Function Call
func_name(par1,par2);
```

Function prototype

- A prototype statement helps the compiler to check the return type and arguments type of the function.
- A prototype function consist of the functions return type, name and argument list.
- Example
  - int sum(int x, int y);

Example

```
#include<iostream.h>
#include<conio.h>
void sum(int, int);               // function declaration
main()
{
int a, b;
   cout<<“enter the two no”;
   cin>>a>>b;
   sum(a,b);                    // function calling

}
 void sum( int x, int y)                // function definition
{
   int c=x+y;
  cout<< “ sum is”<<c;
}
```

```
#include<conio.h>
int sum(int, int);
main()
{
int a=10,b=20;
int c=sum(a,b);                                    /*actual
  arguments
Cout<<“sum is” << c;
getch();
}
int sum(int x, int y) /*formal arguments
{
int s;
s=x+y;
  return(s);                     /*return value
}
```

Where does the execution of the program starts?

a) user-defined function
b) main function
c) void function
d) else function

Where does the execution of the program starts?

a) user-defined function

b) main function

c) void function

d) else function

What are mandatory parts in the function declaration?

a) return type, function name

b) return type, function name, parameters

c) parameters, function name

d) parameters, variables

What are mandatory parts in the function declaration?

a) return type, function name

b) return type, function name, parameters

c) parameters, function name

d) parameters, variables

What is the scope of the variable declared in the user defined function?

a) whole program

b) only inside the {} block

c) the main function

d) header section

What is the scope of the variable declared in the user defined function?

a) whole program

b) only inside the {} block

c) the main function

d) header section

Categories of functions

1. A function with no parameter and no return value
2. A function with parameter and no return value
3. A function with parameter and return value
4. A function without parameter and return value

A function with no parameter and no return value

```
#include<iostream>
using namespace std;
int main()
{
void print();                         /*func declaration
cout<<“no parameter and no return value”;
print();                              /*func calling
}
void print()              /*func definition
{
for(int i=1;i<=30;i++)
{
cout<<“*”;
}
Cout<<“\n”;
}
```

A function with no parameter and no return value

- There is no data transfer between calling and called function
- The function is only executed and nothing is obtained
- Such functions may be used to print some messages, draw stars etc

A function with parameter and no return value

```
#include<iostream>
using namespace std;
 int main()
{
int a=10,b=20;
void mul(int,int);
mul(a,b);                    /*actual arguments
getch();
}
void mul(int x, int y) /*formal arguments
{
int s;
s=x*y;
cout<<“mul is” << s;
}
```

A function with parameter and return value

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

{


int a=10,b=20,c;

int max(int,int);

c=max(a,b);

cout<<“greatest no is” <<c;

}
```

```
int max(int x, int y)
{
if(x>y)
return(x);
else
{
return(y);
}
}
```

A function without parameter and return value

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

int main()
{
int a=10,b=20;
int sum();
int c=sum();    /*actual arguments
Cout<<“sum is”<< c;
}
int sum()                          /*formal arguments
{
int x=10,y=20;
return(x+y);        /*return value
}
```

What is the default return type of a function ?

A. int

B. void

C. float

D. char

What is the default return type of a function ?

A. int

B. void

C. float

D. char

What is the output of this program?

```
#include < iostream >
using namespace std;
void fun(int x, int y)
{
x = 20;
y = 10;
}
int main()
{
int x = 10;
fun(x, x);
cout << x;
return 0;
}
```

- **A.** 10
- **B.** 20
- **C.** compile time error
- **D.** none of the mentioned

What is the output of this program?

```
#include < iostream >
using namespace std;
void fun(int x, int y)
{
x = 20;
y = 10;
}
int main()
{
int x = 10;
fun(x, x);
cout << x;
return 0;
}
```

- **A.** 10
- **B.** 20
- **C.** compile time error
- **D.** none of the mentioned

```
#include < iostream >
using namespace std;
int fun(int x, int y)
{
x = 20;
y = 10;
return x;
}
int main()
{
int x = 10;
x=fun(x, x);
cout << x;
return 0;
}
```

A. A function with no parameter and no return value
B. A function with parameter and no return value
C. A function with parameter and return value
D. A function without parameter and return value

Default arguments

- In the function prototype declaration , the default values are given. Whenever a call is made to function without specifying an argument , the program will automatically assign values to the parameters from the default function prototype.
- Default arguments facilitate easy development and maintenance of programs.

Example

- **void function(int x, int y, int z = 0)** Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.
- **Void function(int x, int z = 0, int y)** Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

code

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
{ // Both z and w are initialised to zero
    return (x + y + z + w); // return sum of all parameter values
}
int main()
{
    cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
    cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
    cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
    return 0;
}
```

Characteristics for defining the default arguments

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Default Arguments

```
#include <iostream>

using namespace std;

void display(char = '*', int =
1);

int main()

{

  cout<<"No argument
passed:\n";

 display();

  cout<<"\n\nFirst argument
passed:\n";

 display('#');

 cout<<"\n\nBoth argument
display('$', 5);

 return 0;

}

void display(char c, int n){

 for(int i = 1; i <=n; ++i) {

    cout<<c;

 }

  cout<<endl;

}
```

If the user didn't supply the user value means, then what value will it take?

- **A.** default value
- **B.** rise an error
- **C.** both a & b
- **D.** none of the mentioned

If the user didn't supply the user value means, then what value will it take?

- **A.** default value
- **B.** rise an error
- **C.** both a & b
- **D.** none of the mentioned

Where does the default parameter can be placed by the user?

A. leftmost

B. rightmost

C. both a & b

D. none of the mentioned

Where does the default parameter can be placed by the user?

A. leftmost

B. rightmost

C. both a & b

D. none of the mentioned

Which value will it take when both user and default values are given?

A. user value

B. default value

C. custom value

D. none of the mentioned

Which value will it take when both user and default values are given?

A. user value

B. default value

C. custom value

D. none of the mentioned

Manipulators in C++

- The unformatted I/O statements that it is impossible to display output in a required user format or input the values in the desired form. In certain situations, we may need to format the I/O as per user requirements. For example :
- 1) The square root of a number should be displayed upto 2 decimal places.
  2) The number to be inputted must be in a hexadecimal form.
- To overcome the problems of unformatted I/O operations in C++, the concept of manipulators was introduced.
- The manipulators in C++ are special functions that can be used with insertion (<<) and extraction (>>) operators to manipulate or format the data in the desired way. Certain manipulators are used with << operator to display the output in a particular format, whereas certain manipulators are used with >> operator to input the data in the desired form. The manipulators are used to set field widths, set precision, inserting a new line, skipping white space etc. In a single I/O statement, we can have more than one manipulator, which can be chained as shown
- cout<<manipl<<var1<<manip2<<var2;
- cout<<manipl<<manip2<<var1;

Manipulator	Purpose	Header File
endl	causes line feed to be inserted i.e. '\n'	iostream.h
dec,oct,hex	set the desired number system	iostream.h
setbase (b)	output integers in base b	iomanip.h
setw(w)	read or write values to w characters	iomanip.h
setfill (c)	fills the whitespace with character c	iomanip.h
setprecision(n)	set floating point precision to n	iomanip.h

endl Manipulator

- The endl manipulator stands for endline and is used with an insertion operator (<<) that moves the cursor to the next line. If we do not use endl, the next output will be displayed in the same line. The endl has the same function as that of ‘\n.’

code

```
#include<iostraam.h>
#include<conio.h>
int main() {
 cout<<"Entar name"<<endl;
 cout<<"My name is Ram";
 return 0;
}
```

Output

```
Enter name
My name is Ram
```

Dec, Oct , Hex Manipulator

All the numbers are displayed and read in decimal notation by default. However, you may change the base of an integer value to octal or hexadecimal or back to a decimal using the manipulator’s oct, hex or dec, respectively. These manipulators are preceded by the appropriate variables to be used with.

```
#include<iostream>
int main() {
 int i;
 cout<<"Enter hexadecimal number =";
 cin>>hex>>i;
 cout:<<"Hexadecimal value = "<<hex<<i<<endl;
 cout<<"Octal Value = "<<oct<<i<<endl;
 cout<<"Dcimal Value = "<<dec<<i<<endl;
  return 0;
}
```

Output :

```
Enter hexadecimal = f
Hexadecimal = f
Octal value = 17
Decimal value = 15
```

setbase(b) Manipulator

The setbase () manipulator is used to change the base of a numeric value during inputting and outputting. It is an alternative to Dec, Oct and hex manipulators. It is a function that takes a single integer argument(b) having values 8, 10 or 16 to set the base of the numeric value to octal, decimal and hexadecimal, respectively. The default base is 10.

```
#include<iostream.h>
#include<iomanip.h>
int main() {
 int num;
 cout<<"Enter number in Octal form = ";
 cin>>setbase(8)>>num;
 cout<<"Value of number in decimal form =
   "<<setbase(10)<<num<<endl;
 cout<<"Value of number in octal form = "<<setbase(8)<<num<<endl;
 cout<<"Value of number in hexadecimal form = "<<setbase(l6)<<num;
 return 0;
}
```

Output

```
Enter numberin Octal form = 21
Value of number in decimal form = 17
Value of number in octal form = 21
Value of number in hexadecimal form = 11
```

setw(w) Manipulator

- The setw() stands for set width. It is a function that takes a single integer argument which specifies the amount of space used to display the required value. We typically use the setw() manipulator for displaying output so that it becomes more understandable. It can be used to format only one value at a time.

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
int main() {
 int age = 22,rollno = 910l;
 cout<<setw(l2)<<"My Rollno
   is"<<setw(8)<<rollno<<endl;
 cout<<setw(l2)<<"My Age is"<<setw(8)<<age;
 getch();
 return 0;
}
```

setfill(c) Manipulator

- The setfill() manipulator is used in conjunction with the setw() manipulator. The compiler leaves the empty spaces on the left side of the required value if the set width is greater than the needed space. If you wish to fill the blank space with an alternative character instead of a blank space, you can use the setfill () manipulator.
- Generally, you will not want to change the fill character. However, one common example of when you may want to is creating a program that prints cheques. To prevent the cheque amount from being altered by the user, computer-generated cheque amounts are usually printed with leading asterisks(*). This is done in C++ with a setfill() manipulator

```
#inclucle<iostream.h>
#include<iomanip.h>
#include<conio.h>
int main() {
 int age = 22,rollno = 910l; cout<<setfill('#');
 cout<<setw(4)<<age<<setw(6)<<rollno<<endl;
 cout<<setw(6)<<age<<setw(8)<<rollno;
 getch();
 return 0;
}
```

Output :

```
##22##9101
####22####9101
```

setprecision(n) Manipulator

The setprecision() manipulator is used to control the precision of floating-point numbers, i.e. the number of digits to the right of the decimal point. By default, the precision of the floating-point number displayed is 6. This precision can be modified by using a setprecision () manipulator. This function takes an integer argument n that specifies the number of digits displayed after the decimal point. The floating-point number will be rounded to the specified precision.

```
#include<iostream.h>
#includi<iomanip.h>
#inelude<eonio.h>
int main() {
 float a = 129.455396;
 cout<<setprecision(2)<<a<<endl;
 cout<<setprecision(3)<<a;
 getch();
 return 0;
}
```

- Output :
- 129.46
- 129.455

MCQ

- **1. _______________are used to format the data display in CPP?**

  a. Iterators
  b. Punctuators
  c. Manipulators
  d. Allocators

MCQ

- **1. ______________are used to format the data display in CPP?**

  a. Iterators
  b. Punctuators
  c. Manipulators
  d. Allocators

2. Which of the following manipulator is used for the representing octal equivalent of a given decimal number ?

a. oct
b. setbase(8)
c. tobase(8)
d. both a and b

2. Which of the following manipulator is used for the representing octal equivalent of a given decimal number ?

a. oct
b. setbase(8)
c. tobase(8)
d. both a and b

3.Predict the output:

```
float x= 3.1496;
cout<<setpricision(2)<<x;
```

a. 3.14
b. 3.15
c. 3.00
d. None of these

3.Predict the output:

```
float x= 3.1496;
cout<<setpricision(2)<<x;
```

a. 3.14
b. 3.15
c. 3.00
d. None of these

ANSWER: d. None of these

Explanation: setprecision(int) is predefined manipulator. Due to spelling mistake here, compiler will generate error.

FUNCTION OVERLOADING

Overloading in C++

- ❑What is overloading
  - – Overloading means assigning multiple meanings to a function name or operator symbol
  - – It allows multiple definitions of a function with the same name, but different signatures.
- ❑C++ supports
  - – Function overloading
  - – Operator overloading

Why is Overloading Useful?

- Function overloading allows functions that conceptually perform the same task on objects of different types to be given the same name.

- Operator overloading provides a convenient notation for manipulating user-defined objects with conventional operators.

Function Overloading

- Is the process of using the same name for two or more functions
- Requires each redefinition of a function to use a different function signature that is:
  - different types of parameters,
  - or sequence of parameters,
  - or number of parameters
- Is used so that a programmer does not have to remember multiple function names

```
Void sum(int,int);
Void sum(double,double);
Void sum(char,char);
main()
{
int a=10,b=20 ;
double c=7.52,d=8.14;
char e=‘a’ , f=‘b’ ;
sum(a,b);
sum(c,d);
sum(e,f);
}
```

```
void sum(int x, int y)
{
cout<<“\n sum of integers
   are”<<x+y;
}
void sum(double x, double y)
{
cout<<“\n sum of two floating
   no are”<<x+y;
}
void sum(char x, char y)
{
cout<<“\n sum of characters
   are”<<x+y;
}
```

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

Type Conversion:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
    std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Which of the following permits function overloading on c++?

a) type

b) number of arguments

c) type & number of arguments

d) number of objects

Which of the following permits function overloading on c++?

a) type

b) number of arguments

c) type & number of arguments

d) number of objects

In which of the following we cannot overload the function?

a) return function

b) caller

c) called function

d) main function

In which of the following we cannot overload the function?

a) return function

b) caller

c) called function

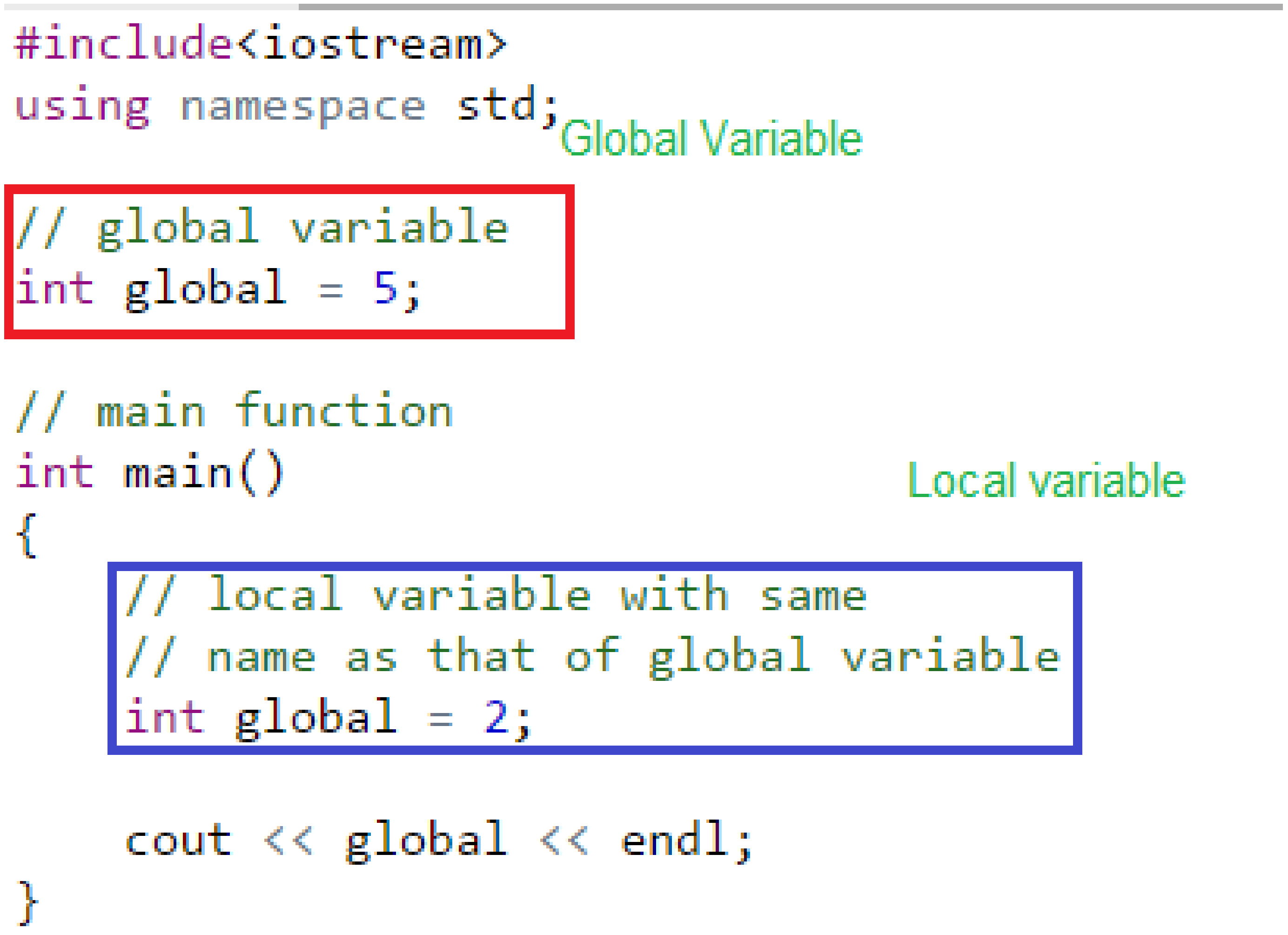
d) main function

Scope of Variables in C++

The scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables

Example

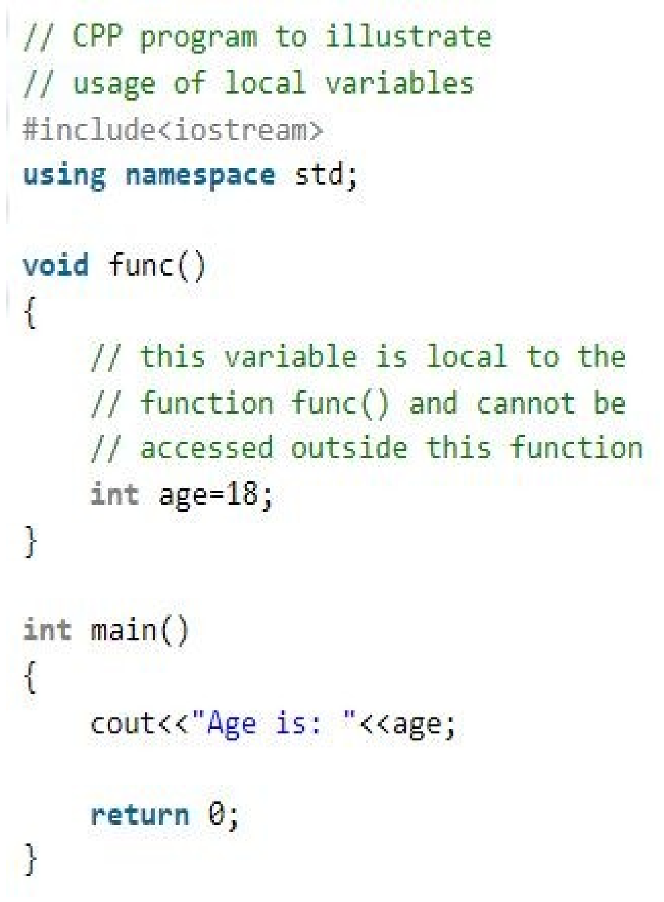


```
#include<iostream>
using namespace std;

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Local Variables

Variables defined within a function or block are said to be local to those functions.

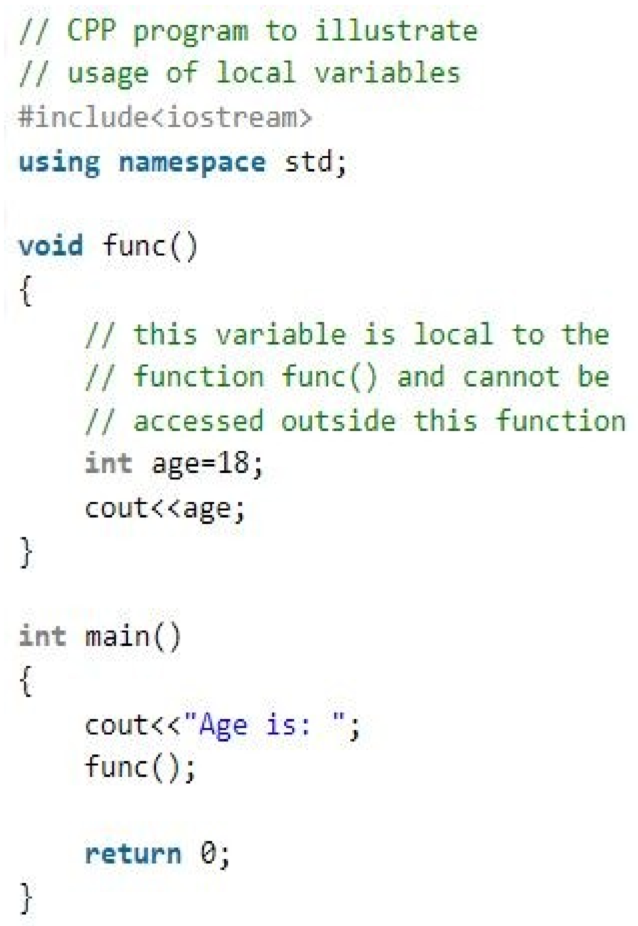
- Anything between ‘{‘ and ‘}’ is said to inside a block.
- Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.
- **Declaring local variables**: Local variables are declared inside a block.



```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
}

int main()
{
    cout<<"Age is: "<<age;

    return 0;
}
```

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.

Rectified Program : To correct the above error we have to display the value of variable age from the function func() only. This is shown in the below program:



```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}

int main()
{
    cout<<"Age is: ";
    func();

    return 0;
}
```

- Output:
- Age is: 18

Global Variables

As the name suggests, Global Variables can be accessed from any part of the program.

- They are available through out the life time of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- **Declaring global variables**: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
// CPP program to illustrate
// usage of global variables
#include<iostream>
using namespace std;

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
}
```

```
int main()
{
    display();

    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

Output:

```
5
 10
```

What if there exists a local variable with the same name as that of global variable inside a function?

If there is a variable inside a function with the same name as that of a global variable and if the function tries to access the variable with that name, then which variable will be given precedence? Local variable or Global variable?

- Usually when two variable with same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.
- Whenever there is a local variable defined with same name as that of a global variable then the **compiler will give precedence to the local variable**

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

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable

    int global = 2;
    cout << global << endl;
}
```

How to access a global variable when there is a local variable with same name?

if we want to access global variable when there is a local variable with same name?
To solve this problem we will need to use the **<u>scope resolution operator.</u>**

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

// Global x
int x = 0;

int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is "
     << ::x;
  cout<< "\nValue of local x is "
     << x;
  return 0;
}
```

Q1. To reveal the hidden scope of the variable which operator is used?

1. Scope Resolution operator
2. Address operator
3. Assignment operator
4. Pointer operator

Q1. To reveal the hidden scope of the variable which operator is used?

1. **Scope Resolution operator**
2. Address operator
3. Assignment operator
4. Pointer operator

Q2. What is the output of the following code snippet?

```
#include <iostream>
using namespace std;
int x = 1, y = 2, z = 3;
int main()
{
cout<<” x =”<<x<<” y = ” <<y<<” z
      = “<< z<<endl;
{
    int x = 10;
    float y = 20;
    cout<<” x =”<<::x<<” y = ”
      <<::y<<” z = “<< z<<endl;
    {
            int z = 100;
        cout<<” x =”<<x<<” y = ”
      <<y<<” z = “<< z<<endl;
    }
}
return 0;
}
```

A). x =1 y = 2 z = 3 x =1 y = 2 z = 3 x =1 y = 2 z = 100

B). x =1 y = 2 z = 3 x =10 y = 20 z = 3 x =1 y = 2 z = 100

C). x =1 y = 2 z = 3 x =1 y = 2 z = 3 x =10 y = 20 z = 100

D). x =1 y = 2 z = 3 x =10 y = 20 z = 3 x =10 y = 20 z = 100

Q2. What is the output of the following code snippet?

```
#include <iostream>
using namespace std;
int x = 1, y = 2, z = 3;
int main()
{
cout<<" x ="<<x<<" y = " <<y<<" z
      = "<< z<<endl;
{
    int x = 10;
    float y = 20;
    cout<<" x ="<<::x<<" y = "
      <<::y<<" z = "<< z<<endl;
    {
            int z = 100;
        cout<<" x ="<<x<<" y = "
      <<y<<" z = "<< z<<endl;
    }
}
return 0;
}
```

A). x =1 y = 2 z = 3 x =1 y = 2 z = 3 x =1 y = 2 z = 100

B). x =1 y = 2 z = 3 x =10 y = 20 z = 3 x =1 y = 2 z = 100

C). x =1 y = 2 z = 3 x =1 y = 2 z = 3 x =10 y = 20 z = 100

D). x =1 y = 2 z = 3 x =10 y = 20 z = 3 x =10 y = 20 z = 100

C++ Friend function

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
    friend data_type function_name(argum
   ent/
   s);              // syntax of friend function.
};
```

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

friend function Example

```
#include <iostream>
using namespace std;
class Box
{
    private:
        int length;
    public:
        Box(): length(0) { }
        friend int printLength(Box);
      //friend  function
};
int printLength(Box b)
{
   b.length += 10;
    return b.length;
}
```

```
int main()
{
    Box b;
 cout<<"Length of box: "<< print
     Length(b)<<endl;
    return 0;
}
```

- **Output:**
- Length of box: 10

Let's see a simple example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
    int x;
    public:
    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);         // friend function.
};
class B
{
    int y;
    public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);                     // friend funct
        ion
};
```

```
void min(A a,B b)
{
    if(a.x<=b.y)
    std::cout << a.x << std::endl;
    else
    std::cout << b.y << std::endl;
}
   int main()
{
   A a;
   B b;
   a.setdata(10);
   b.setdata(20);
   min(a,b);
    return 0;
 }
```

- **Output:**
- 10

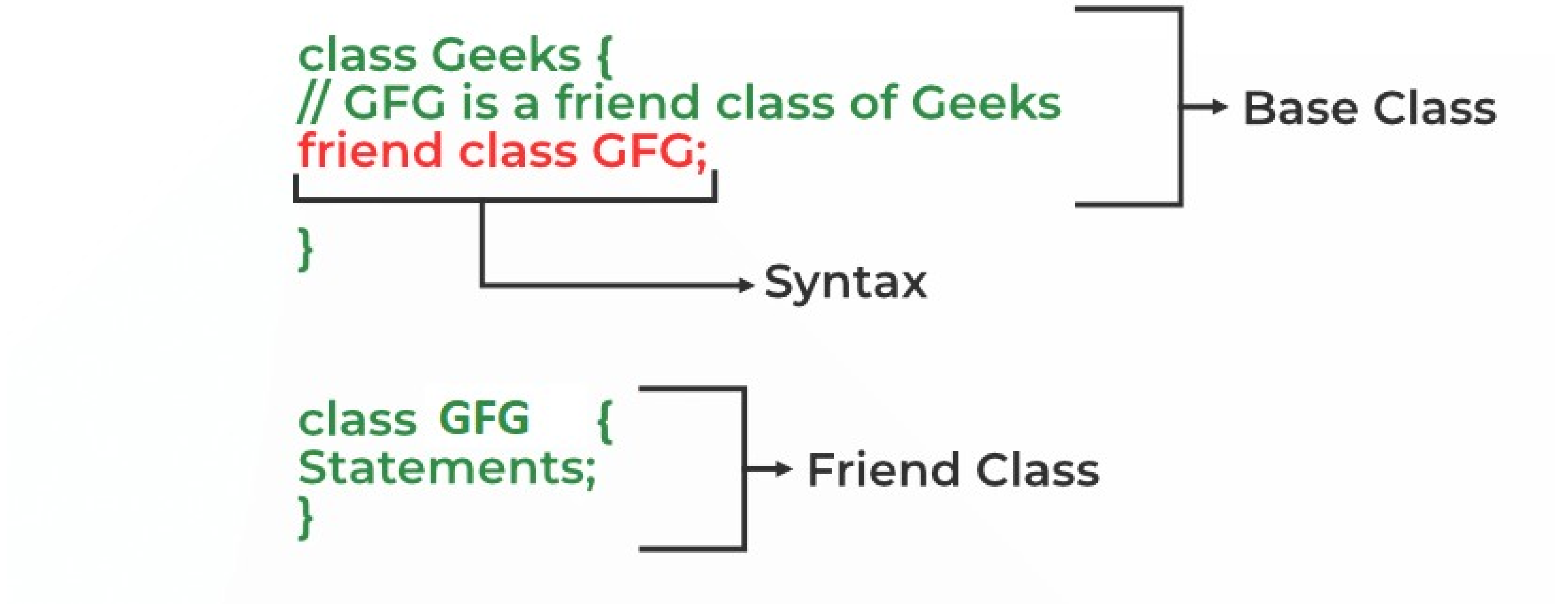
In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

- A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.
- We can declare a friend class in C++ by using the **friend** keyword.

Syntax:

friend class class_name; // declared in the base class



```
class Geeks {
// GFG is a friend class of Geeks
friend class GFG;

}

class  GFG    {
Statements;
}
```

Syntax:

friend class class_name; // declared in the base class

```
class A{

.....

friend class B;
};

class B{

.....
};
```

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

class Test {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    Test()
    {
        private_variable = 10;
        protected_variable = 99;
    }

    // friend class declaration
    friend class F;
};
```

```
class F {
public:
    void display(Test& t)
    {
        cout << "The value of Private Variable = "
             << t.private_variable << endl;
        cout << "The value of Protected Variable = "
             << t.protected_variable;
    }
};


int main()
{
    Test g;
    F fri;
    fri.display(g);
    return 0;
}
```

output

```
The value of Private Variable = 10 The value of Protected Variable = 99
```

Q1. A friend class can access ____________________ members of other class in which it is declared as friend.

A. private
B. protected
C. public
D. Both A and B

Q1. A friend class can access ______________________ members of other class in which it is declared as friend.

A. private
B. protected
C. public
D. Both A and B

Q2. A friend function can be

A. A method of another class
B. A global function
C. Both A and B
D. None of the above

Q2. A friend function can be

A. A method of another class
B. A global function
C. Both A and B
D. None of the above

Q3. If class A is a friend of B, then B doesn’t become a friend of A automatically

A. TRUE
B. FALSE
C. Can be true and false
D. Can not say

Q3. If class A is a friend of B, then B doesn’t become a friend of A automatically

A. TRUE

B. FALSE

C. Can be true and false

D. Can not say

Q4.Which of the following is false?

A. Friendship is not inherited
B. The concept of friends is there in Java.
C. Both A and B
D. None of the above

Q4.Which of the following is false?

A. Friendship is not inherited
B. The concept of friends is there in Java.
C. Both A and B
D. None of the above

Q5. Which of the following is correct about friend functions?

A. Friend functions use the dot operator to access members of a class using class objects
B. Friend functions can be private or public
C. Friend cannot access the members of the class directly
D. All of the above

Q5. Which of the following is correct about friend functions?

A. Friend functions use the dot operator to access members of a class using class objects
B. Friend functions can be private or public
C. Friend cannot access the members of the class directly
D. All of the above