**OBJECT ORIENTED PROGRAMMING (PCC-CS503)**

Unit – 2

**Some difference between C and C++**

**Single line comments**

Comments are portions of the code ignored by the compiler which allow the user to make simple notes in the relevant areas of the source code. Comments come either in block form or as single lines.

* **Single-line comments** (informally, *C++ style*), start with // and continue until the end of the line. If the last character in a comment line is a \ the comment will continue in the next line.
* **Multi-line comments** (informally, *C style*), start with /\* and end with \*/.

**NOTE:** Since the 1999 revision, C also allows *C++ style* comments, so the informal names are largely of historical interest that serves to make a distinction of the two methods of commenting.

**Local variable declaration within function scope**

* In case of C, **the local** **variables must always be defined at the top of a block**.
* When a local variable is declared and not initialised, it will hold some garbage value (assigned by the system).
* A local variable is defined inside a **block** and is only visible from within the block.

int main()

{

int i=4; //local variable i

int j=10; //local variable j

i++;

if (j > 0)

{

printf("i is %d\n",i); // i defined in 'main' can be seen

}

if (j > 0)

{

int i=100; /\* 'i' is defined and so local to

\* this block \*/

printf("i is %d\n",i);

} /\* 'i' (value 100) dies here \*/

printf("i is %d\n",i); /\* 'i' (value 5) is now visible. \*/

}

**However, if you initialize the local variables anywhere other than the top of the block, C compiler will give an error!**

int main()

{

int i=4; //local variable i

i++;

int j=10; //local variable j can’t be initialized here

if (j > 0)

{

printf("i is %d\n",i); // i defined in 'main' can be seen

}

if (j > 0)

{

int i=100; /\* 'i' is defined and so local to

\* this block \*/

printf("i is %d\n",i);

} /\* 'i' (value 100) dies here \*/

printf("i is %d\n",i); /\* 'i' (value 5) is now visible. \*/

return 0;

}

**In case of C++, the local variables can be declared and defined anywhere in the code and not just at the start of the block.** The idea of this feature is to allow the programmer to put variable declarations near to the place you wish to use them.

int main()

{

float pi = 3.142; // Usual location for variable definitions

cout << "PI is " << pi << endl;

int Count = 1; // **C++ allows us to place a definition here but not C**.

while (Count < 10)

{

cout << Count << endl;

}

return 0;

}

**Function declaration**

Why function declaration?

Declaring a value without defining it allows you to write code that the compiler can understand without having to put all of the details. This is particularly useful if you are working with multiple source files, and you need to use a function in multiple files. You don't want to put the body of the function in multiple files, but you do need to provide a declaration for it.

Example: int add(int x, int y);

This is a function declaration; it does not provide the body of the function, but it does tell the compiler that it can use this function and expect that it will be defined somewhere.

Why function definition?

Defining something means providing all of the necessary information to create that thing in its entirety. Defining a function means providing a function body.

Example:

int func(); // function declaration

int main()

{

int x = func(); //calling function func()

}

int func()

{

return 2;

}

Since the compiler knows the return value of func, and the number of arguments it takes, it can compile the call to func even though it doesn't yet have the definition. In fact, the definition of the method func could go into another file!

NOTE: If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:

int function\_name();

Example:

//int func();

int main()

{

int x = func();

}

void func() // ERROR!- type mismatch!

{

return 2;

}

And in that case if the return type of that function is different than “int”, compiler would show an error.

**In OOP, since there is bottom-up approach, so declaration is actually not a necessity.**

**Function overloading**

Function overloading is a feature of a programming language that allows one to have many functions with same name but with different signatures.

// Different signatures of the same function- add()

Int add (int a, int b);

Void add (int a, int b);

Int add (int a, int b, int c);

Int add (int a, double b);

This technique is used to enhance the readability of the program because we don’t need to use different names for the same action again and again.

There are two ways to overload a function, i.e. −

* Having different number of arguments
* Having different argument types

Function overloading is normally done when we have to perform one single operation with different number or types of arguments.

**C does not support function overloading but C++ does include this feature.**

Example:

#include <iostream.h>

int add(int x, int y) {  
  return x + y;  
}

int add(int x, int y, int z) {  
  return x + y +z;  
}

double add(double x, double y) {  
  return x + y;  
}  
  
int main() {  
  int myNum1 = add(8,5,9);  
  double myNum2 = add(4.3,6.26);  
  cout << "Int: " << myNum1 << "\n";  
  cout << "Double: " << myNum2;  
  return 0;  
}

**Prog: Overload the function “volume” for a cube, a cylinder and a cone.**

**Stronger type checking**

* ***Type checking*** checks and enforces the rules of the type system to prevent type errors from happening.

int a = ghana; // compile time err!

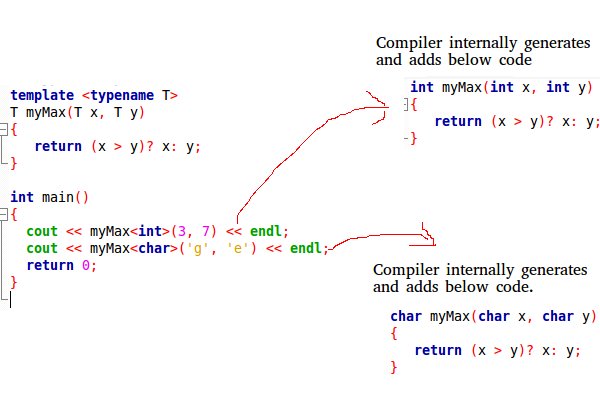
int a = “123”; // compile time err!

string x=123; // compile time err!

* A ***type error*** happens when an expression produces a value outside the set of values it is supposed to have.
* ***Strong type checking*** prevents all type errors from happening, either at compile time or at run time or partly at compile time and partly at run time.
* A ***strongly-typed language*** is one that uses strong type checking.
* ***Weak type checking*** does not prevent type errors from happening.
* A ***weakly-typed language*** is one that uses weak type checking.

**C++ which is otherwise a strongly typed language can allow flexibility in type-checking using the concept of Templates. C however has no concept of Templates.**

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.



**Reference variable**

* C++ introduces a new kind of variable known as ***Reference Variable.*** It provides an alias (alternative name) for a previously defined variable.
* However, there is no concept of reference variables in C.
* A reference variable must be initialized at the time of declaration. This establishes the correspondences between the reference and the data object which it references to.
* A reference variable once initialized cannot be reinitialized.

**Define:**

[data\_type] &[reference\_variable]=[regular\_variable];

#include <iostream.h>

#include <conio.h>

int main()

{

clrscr();

int a=10; //existing variable a,c

int c=20;

int &b=a; // reference variable b

cout<< " value of a :"<< a << endl; //10

cout<< " value of b :"<< b << endl; //10

b=c; //ERROR: We are just reassigning the value but not rebinding the reference.

cout<<"value of variables after reassignment:"<<endl;

cout<< " value of a :"<< a << endl; //20

cout<< " value of b :"<< b << endl; //20

cout<< " value of c :"<< c << endl; //20

cout<<" address of variables after reassignment:"<<endl;

cout<< " address of a :"<< &a << endl; //2004

cout<< " address of b :"<< &b << endl; //2004

cout<< " address of c : "<<&c << endl; //8002

//b=b+10; //increment ref variable

a=a+10; //increment ref variable

cout<<"\nAFTER ADDING 10 INTO REFERENCE VARIABLE \n";

cout<< " value of a :"<< a << endl; //30

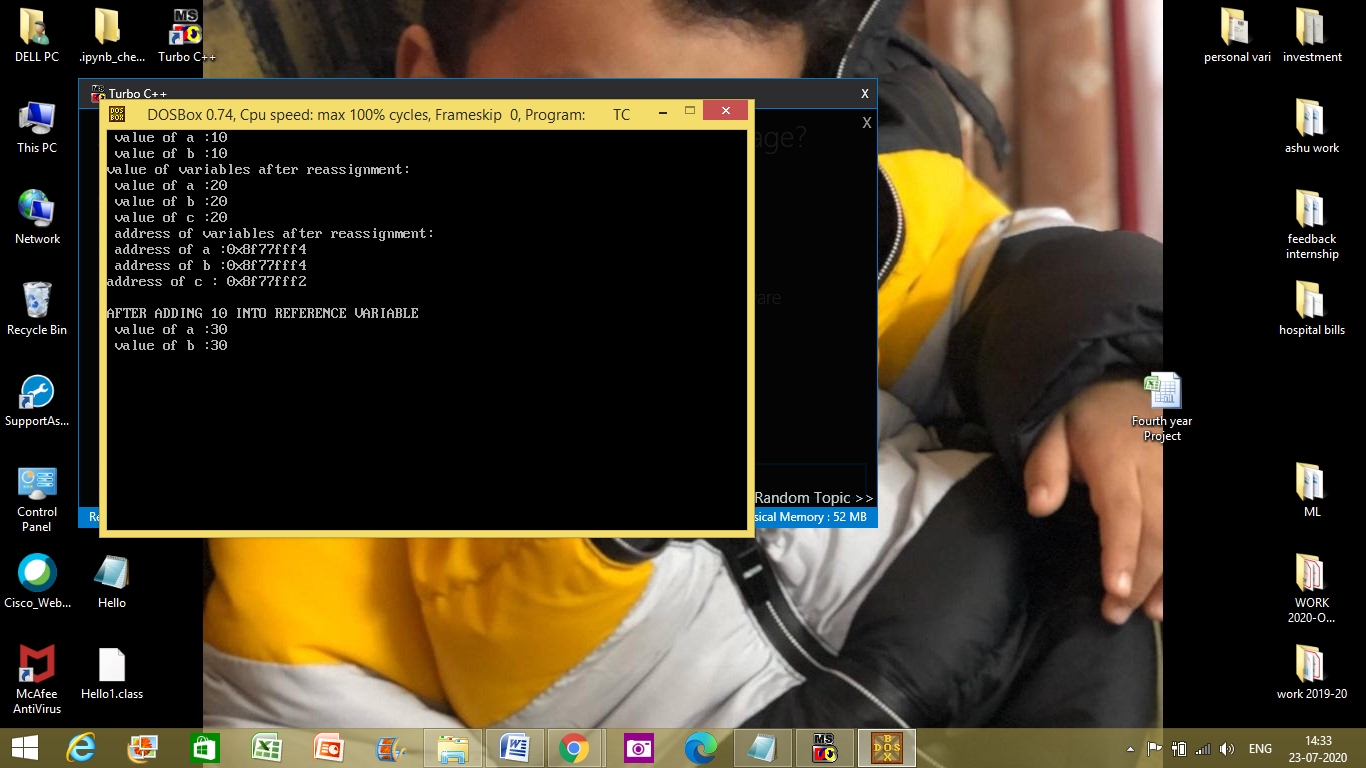
cout<< " value of b :"<< b << endl; //30

getch();

return 0;

}

The output:



#include <iostream.h>

**int** main()

{

**int** a=10;

**int** &b=a; // reference variable

**int** \*p=&a; // pointer variable

cout<< " value of a :"<< a << endl; //10

cout<< " value of b :"<< b << endl; //10

cout<< " value of p :"<< p << endl; //2004

cout<< " value at \*p :"<< \*p << endl; //10

b=b+10;

cout<<"\nAFTER ADDING 10 INTO REFERENCE VARIABLE \n";

cout<< " value of a :"<< a << endl;

cout<< " value of b :"<< b << endl;

cout<< " address of a :"<< &a << endl; //2004

cout<< " address of b :"<< &b << endl; //2004

cout<< " address of p :"<< &p << endl; //8006

**return** 0;

}

**Pointer vs Reference variable:**

1. A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
2. Pointer can be assigned NULL directly, whereas reference cannot.
3. Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
4. A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
5. A pointer to a class/struct uses ‘🡪’ (arrow operator) to access it’s members whereas a reference uses a ‘.’ (dot operator)
6. A pointer needs to be de-referenced with \* to access the memory location it points to, whereas a reference can be used directly.

**Parameter passing – value vs. reference**

Parameters are the data values that are passed from the calling function to a called function.

In C/C++, there are two types of parameters:

* **Actual Parameters:** parameters that are specified in calling function e.g. swap(10, 20); // here 10 & 20 are actual parameters
* **Formal Parameters:** parameters that are declared at called function e.g. swap(int x, int y) // x & y are formal arguments

When a function gets executed, the copies of actual parameter values are copied into formal parameters.

In C we can pass parameters in two different ways:

* Call by value, and
* Call by address

**Call by value:** In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. **The changes made on the formal parameters does not effect the values of actual parameters**. That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example:

**Swapping numbers using Function Call by Value**

#include<conio.h>

#include<iostream.h>

void swap(int x, int y) **// called function, x & y are formal arguments**

{

int temp;

temp = x;

x = y;

y = temp;

//cout<<x<<y; //40 10

}

void main() **//calling function**

{

int a, b; **// a & b are actual arguments**

clrscr();

a = 10;

b = 40;

cout << "(a,b) = (" << a << ", " << b << ")\n";

**swap(a, b); //function call**

cout << "(a,b) = (" << a << ", " << b << ")\n";

getch();

}

O/P:

(a,b) = (10, 40)

(a,b) = (10, 40)

**Call by Address:** In **Call by Address** parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be **pointer** variables.

That means in call by address parameter passing method, the address of the actual parameters is passed to the called function and is received by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So **the changes made on the formal parameters effects the values of actual parameters**. For example:

**Swapping numbers using Function Call by Address**

#include<conio.h>

#include<iostream.h>

void swap(int \*x, int \*y) { //called func

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

void main() { // calling func

int a, b;

clrscr();

a = 10;

b = 40;

cout << "(a,b) = (" << a << ", " << b << ")\n";

swap(&a, &b);

cout << "(a,b) = (" << a << ", " << b << ")\n";

getch();

}

O/P:

(a,b) = (10, 40)

(a,b) = (40, 10)

Point of difference: In C we can pass parameters in two different ways:

* Call by value, or
* call by address,

In C++, apart from the above two ways, we can get another technique, called “Call by reference”. Let us see the effect of these, and how they work.

**Call by Reference:** This technique only works in C++ and not in C. In this case, we have to put & before variable name at the function definition. The technique or working however remains same as that in “call by address” in case of C.

**Swapping numbers using Function Call by Reference**

#include<conio.h>

#include<iostream.h>

void swap(int &x, int &y) {

int temp;

temp = x;

x = y;

y = temp;

}

void main() {

int a, b;

clrscr();

a = 10;

b = 40;

cout << "(a,b) = (" << a << ", " << b << ")\n";

swap(a, b);

cout << "(a,b) = (" << a << ", " << b << ")\n";

getch();

}

O/P:

(a,b) = (10, 40)

(a,b) = (40, 10)

**Passing pointer by value or reference**