C++ and Object Oriented Programming

Unit – 1 (Topic – 2)

FUNCTIONS IN C++

main() in c++

-> Generally main() of c and c++ both are same.

-> We know that by default any function returns an int value.

-> In c language compiler return a warning if we create a main() which does not return any value.

-> Therefore in c language we write void main(), So we did not get any warning or error message.

-> Same way in c++ if we create a main() and does not return any value it returns an error.

-> So, to remove this error generally we write int main() instead of void main().

-> Even if we write void main() then also it is valid.

-> We can use any one format from the following

```
int main();
int main(int argc, char * argv[]);
void main();
void main(int argc, char * argv[]);
```

Function Prototype

-> Function prototype means declaration of the function with its types of argument and return type.

-> In C++ if we call any function without declaration or definition then it return an error " Function Should Have A Prototype".

For Example:

```
int add(int, int);
int sub(int, int , int);
float mul(float, float);
```

Note: In C++ we must have to include a header file from which we want to use functions in our program, otherwise it will return error related to function prototype.

Call By value & Call By reference

Call By Value:

-> In call by value calling function pass copy of the argument to the called function.

-> Therefore if we change value of argument inside the called function it does not reflect to the calling function.

For Example:

```
void swap(int, int);
void main()
{
        int a=1,b=2;
        swap(a,b);
        cout<<“\n After swap a=“<<a<<“\n b=“<<b;
}
void swap(int  x, int y)
{
        int z=x;
        x=y;
        y=z;
}
```

Call By value & Call By reference

Call By Reference:

-> In call by reference calling function pass reference (address) of the argument to the called function.

-> Therefore if we change value of argument inside the called function it reflects to the calling function.

For Example:

```
void swap(int *,  int *);
void main()
{
        int a=1,b=2;
        swap(&a,&b);
        cout<<“\n After swap a=“<<a<<“\n b=“<<b;
}
void swap(int  *x, int *y)
{
        int z=*x;
        *x=*y;
        *y=z;
}
```

Return By reference

Return By Reference:

-> We know that reference variable creates an alias of the variable.

-> As we pass any type of argument to the called function same way we can return any type of value.

For Example:

```
int & max(int &,  int &);
void main()
{
        int a=1,b=2,c,x=10,y=20,z=30;
        c = max(a, b);
        cout<<“\n max value =“<<c;
        max(x,y)=z;
        cout<<“\n x=“<<x<<“\ny=“<<y<<“\nz=“<<z;
}
int & max(int  x, int y)
{
        int z=x;
        x=y;
        y=z;
}
```

Inline function

inline function:

-> C++ **inline** function is powerful concept that is commonly used with classes.

-> If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

-> Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

-> To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function.

-> The compiler can ignore the inline qualifier in case defined function is more than a line.

-> A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Syntax:

```
inline functionhader
{
        function body;
}
```

Inline function

inline function:

-> In above syntax keyword **inline** indicate that the function is inline.

-> functionheader contains return type of the function, function name and types of argument.

-> function body contains the code of the function.

-> Note that the inline keyword only sends a request, not a command, to the compiler.

-> The compiler may ignore this request if the functions definition is too long or too complicated and compile the function as a normal function.

-> We must have to define inline function before the main().

-> Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch or a goto exist.
2. For functions not returning values, if a return statement exist.
3. If functions contain static variables.
4. If inline functions are recursive.

Inline function

inline function:

-> For Example:

```
#include<iostream.h>
#include<conio.h>
inline int mul(int x, int y)
{
        return (x*y);
}
int main()
{
        int x, y, z;
        clrscr();
        cout<<“Enter x and y:”;
        cin>>x>>y;
        z = mul(x,y);
        cout<<“\n z =“<<z;
        getch();
}
```

Function with default argument

Default Arguments:

-> C++ allows us to call a function 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.

-> Default values are specified when the function is declared.

-> The compiler looks at the prototype to see how many arguments a functions uses and alerts the program for possible default values.

-> the default value is specified similar to variable initialization.

For Example:

```
int add(int a, int b, int c=20);
```

-> The above prototype declares a default value of 20 to the argument c.

-> A function call

```
sum = add(10,20);
```

-> Passes the value of 10 to a and 20 to b and then lets the function use default value of 20 for c.

-> The call

```
sum = add(10,20,30);
```

-> Passes an explicit value of 30 to c.

Function with default argument

Default Arguments:

Rules Of Default arguments:

-> Only trailing arguments can have default values.

-> We must add defaults from right to left.

-> We can not provide a default value to a particular argument in the middle of an argument list.

-> Some examples of function declaration with default values.

```
int add(int a, int b=20);                  // legal
int add(int a, int b=20, int c=30);        //legal
int add(int a=10,int b=20, int c=3);       // legal
int add(int a=10, int b, int c=30);        // illegal
int add(int a=10, int b, int c);           // illegal
int add(int a, int b=20, int c);           // illegal
```

-> Default arguments are useful in situation where some arguments always have the same value.

-> For example bank interest.

Function with default argument

Default Arguments:

```
#include<iostream.h>
#include<conio.h>
int add(int a, int b=100);
int main()
{
        int a, b, c, d;
        clrscr();
        cout<<“Enter 2 Nos.”;
        cin>>a>>b;
        c = add(a, b);
        d = add(a);
        cout<<“\n Answer without default:”<<c;
        cout<<“Answer with default=“<<d;
        getch();
        return 0;
}
int add(int a, int b)
{
        return(a+b);                    }
```

Constant arguments

Constant Argument:

-> In C++, an argument to a function can be declared as const as show below.

```
int strlen(const char *p);
```

-> The qualifier const tells the compiler that the function should not modify the argument.

-> The compiler will generate an error when this condition is violated.

-> This type of declaration is significant only when we pass arguments by reference or pointers.

Constant arguments

Constant Argument:

```
#include<iostream.h>
#include<conio.h>

int strlen1(const char *s);

void main()
{
        char *s;
        int l;
        clrscr();
        cout<<"Enter a string:";
        cin>>s;
        l = strlen1(s);
        cout<<"\n Length="<<l;
        getch();
}

int strlen1(const char *s)
{
        int i;
        for(i=0;s[i]!='\0';i++);
        return i;
}
```

Function Overloading

Function Overloading:

-> Overloading refers to the use of the same thing for different purposes.

-> C++ also permits overloading of functions.

-> This means that we can use the same function name to create functions that perform a variety of different tasks.

-> This is known as function overloading.

-> Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists.

-> The function would perform different operations depending upon the argument list in the function call.

-> **The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.**

For Example:

```
        int add(int a, int b);                  // prototype1
        int add(int a, int b, int c);           // prototype2
        float add(float a, float b);            // prototype3
//Function Calls
        cout<<add(5,10);                        //uses prototype1
        cout<<add(2,3,4);                       //uses prototype2
        cout<<add(12.5,34.6);                   //uses prototype3
```

Function Overloading

Function Overloading:

-> A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution..

-> A best match must be unique.

-> The function selection involves the following steps.

1. The compiler first tries to find an exact match.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments such as,

    char to int

    float to double

3. When either of them fails, the compiler tries to use the built-in conversions (implicit conversions).

4. If all of the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built-in conversions to find a unique match.

Note : If in any case compiler find more than one match then it will return **"ambiguous Functions"** error.

Function Overloading

Function Overloading:

```
#include<iostream.h>
#include<conio.h>
int add(int a, int b);
int add(int a, int b, int c);
void main()
{
        int a,b,c,d,e;
        clrscr();
        cout<<"Enter 3 numbers:";
        cin>>a>>b>>c;
        d = add(a,b);
        e = add(a,b,c);
        cout<<"\n Addition of Two Numbers="<<d;
        cout<<"\n Addition of Three Numbers="<<e;
        getch();
}
int add(int a,int b)
{
        return(a+b);
}
int add(int a, int b, int c)
{
        return(a+b+c);
}
```