Introduction to Object Oriented Programming

Lecture 2

Function in C++

Size Of ?

```
#include <iostream.h>
void main ( )
{
cout << "char size = " << sizeof(char) << " bytes \n";
cout << "short size = " << sizeof(short) << " bytes \n";
cout << "int size = " << sizeof(int) << " bytes \n";
cout << "long size = " << sizeof(long) << " bytes \n";
cout << "bool size = " << sizeof(bool) << " bytes \n";
cout << "float size = " << sizeof(float) << " bytes \n";
cout << "double size = " << sizeof(double) << " bytes \n";
cout << "long double size = " << sizeof(long double) << " bytes \n";
cout << "char* size = " << sizeof(char*) << " bytes \n";
cout << "int* size = " << sizeof(int*) << " bytes \n";
}
```

Size Of ?

in VS C++

- char size = 1 bytes
- short size = 2 bytes
- int size = 4 bytes
- long size = 4 bytes
- bool size = 1 bytes
- float size = 4 bytes
- double size = 8 bytes
- long double size = 8 bytes
- char* size = 4 bytes
- int* size = 4 bytes

Function In C++

- A function is a named unit of a group of program statements.
- This unit can be invoked from other parts of the program.
- A programmer can solve a simple problem in C++ with a single function.
- Difficult and complex problems can be decomposed into sub-problems, each of which can be either coded directly or further decomposed.
- Decomposing difficult problems, until they are directly code-able as single C++ functions,
- These functions are combined into other functions and are ultimately used in **main()** to solve the original problem.

Writing a Function

- You have decide on what the function will *look* like:
  - Return type
  - Name
  - Types of parameters (number of parameters)
- You have to write the body (the actual code).

Syntax

- In C++, a function must be defined before it can be used any where in the program.
- The general function definition is as given below:

```
datatype function-name (parameter list)
{
        function body ;
            -
            -
}
```

Syntax

- where **datatype** specifies the type of value the function returns (e.g.: int, char, float, double, user-defined).
- If no **datatype** is mentioned, the compiler assumes it as default integer type.
- The parameter list is also known as the arguments or signature of the function, which are the variables that are sent to the function to work on.
- If the parameter list is empty, the compiler assumes that the function does not take any arguments.

Function parameters

- The parameters are *local variables* inside the body of the function.
  - When the function is called they will have the values *passed in*.
  - The function gets *a copy* of the values passed in (we will later see how to pass a *reference* to a variable).

Sample Function

Return type

Function name

parameters

```
int add2ints(int a, int b)
{
 return(a+b);
}
```

Function body

Sample Function

```
int add2nums( int firstnum, int secondnum )
 {
   int sum;
   sum = firstnum + secondnum;

   firstnum = 0;
   secondnum = 0;
   return(sum);
}
```

Testing `add2nums`

```
void main()
 {
   int y,a,b;
   cout << "Enter 2 numbers\n";
   cin >> a >> b;

   y = add2nums(a,b);

   cout << "a is " << a << endl;
   cout << "b is " << b << endl;
   cout << "y is " << y << endl;
}
```

Using functions – Math Library functions

- C++ includes a library of Math functions you can use.
- You have to know how to *call* these functions before you can use them.
- You have to know what they return.
- You don’t have to know how they work!

double sqrt(double)

- When *calling* **`sqrt`**, we have to give it a **`double`**.
- The **`sqrt`** function returns a **`double`**.
- We have to give it a **`double`**.

```
x = sqrt(y);
x = sqrt(100);
```

Table of square roots

```
int i;
for (i=1;i<10;i++)
  cout << sqrt(i) << “\n”;
```

- But I thought we had to give **`sqrt()`** a double?
- C++ does automatic *type conversion* for you.

Telling the compiler about sqrt()

- How does the compiler know about **`sqrt`** ?
- You have to tell it:

```
#include <math.h>
```

Other Math Library Functions

```
ceil    floor

cos     sin     tan

exp     log     log10   pow

fabs    fmod
```

Local variables

- Parameters and variables declared inside the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!

That’s fine! We don’t need them anymore!

Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.

Block Variables

- You can also declare variables that exist only within the *body* of a compound statement *(a block):*

```
{
int f;
        …
        …
}
```

Scope

- The *scope* of a variable is the portion of a program where the variable has meaning (where it exists).
- A global variable has global (unlimited) scope.
- A local variable’s scope is restricted to the function that declares the variable.
- A block variable’s scope is restricted to the block in which the variable is declared.

Block Scope

```
int main() {
  int y;

  {
    int a = y;
    cout << a << endl;
  }
  cout << a << endl;
}
```

Error – a doesn't exist outside the block!

Function Prototypes

- A Function prototype can be used to *tell* the compiler what a function looks like
    - So that it can be called even though the compiler has not yet seen the function definition.
- A function prototype specifies the function name, return type and parameter types.

Using a prototype

```
int counter();

int main() {
  cout << counter() << endl;
  cout << counter() << endl;
  cout << counter() << endl;
}

int counter() {
  int count = 0;
  count++;
  return(count);
}
```

Call-by-value vs. Call-by-reference

- So far we looked at functions that get a copy of what the *caller* passed in.
  - This is call-by-value, as the value is what gets passed in (the value of a variable).
- We can also define functions that are passed a *reference* to a variable.
  - This is call-by-reference, the function can change a callers variables directly.

References

- A *reference* variable is an alternative name for a variable. A *shortcut*.
- A reference variable must be initialized to *reference* another variable.
- Once the reference is initialized you can treat it just like any other variable.

Call-by-value

```
#include <iostream.h>
int add2nums( int firstnum, int secondnum );//prototype
int main()
 {
  int y,a,b;
  cout << "Enter 2 numbers\n";
  cin >> a >> b;
  y = add2nums(a,b);
  cout << "a is " << a << endl;
  cout << "b is " << b << endl;
  cout << "y is " << y << endl;
  return 0;
}

int add2nums( int firstnum, int secondnum )
 {
  int a = firstnum + secondnum;
  return a;
 };
```

Call-by-reference

```
#include <iostream.h>
int add2nums( int &firstnum, int &secondnum );//prototype
int main()
 {
  int y,a,b;
  cout << "Enter 2 numbers\n";
  cin >> a >> b;
  y = add2nums(a,b);
  cout << "a is " << a << endl;
  cout << "b is " << b << endl;
  cout << "y is " << y << endl;
  return 0;
}
int add2nums( int &firstnum, int &secondnum )
 {
  int a = firstnum + secondnum;
  firstnum=0 ; secondnum=0;
  return a;
 };
```

Inline Function

- One of the main objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for task such as jumping to the function, saving register, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

Inline Function

- C++ has a special solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function codes. The inline function is defined as follows:

```
inline function-header
{
   function-body;
}
```

Inline Function

Example:

```
inline int cube( int a )
{
   return (a*a*a);
}
```

- It is easy to make a function inline. All we need to do is prefix the keyword inline to the function definition. All inline functions must be defined before they are used.

Inline Function

- There are few situations where an inline function may not work:

- For a function returning values; if a return statement exists.
- For a function not returning any values; if a loop, switch or goto statement exists.
- If a function is recursive.

Function Templates

- we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is as follows:

```
template <class T>
return-type function-name( argument of type T )
{
    // body of function with Type T
}
```

Function Templates

- The following example declares a swap() function template that will swap two values of a given type of data.

```
template <class T>
void swap(T &x, T &y )
{
   T temp = x ;
   x = y ;
   y = temp ;
}
```

Function Templates

```
#include<iostream.h>
template<class T>
void swap( T &x, T &y )
{
    T temp = x;
    x = y;
    y = temp;
}
void fun(int m, int n, float a, float b)
{
    cout<<"\n m and n before swap : "<<m<<" " <<n;
    swap(m, n);
    cout<<"\n m and n after swap : "<<m<<" " <<n;
    cout<<"\n a and b before swap : "<<a<<" " <<b;
    swap(a, b);
    cout<<"\n a and b after swap : "<<a<<" " <<b;
}
void main()
{
    fun(100, 200, 11.22, 33.44);
}
```

Function Templates with Multiple Parameters

- We can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```
 template < class T1, class T2 >
return-type function-name(argument of types T1, T2,…)
{
   // body of the function
}
```

Function Templates with Multiple Parameters

```
#include<iostream.h>
template < class T1, class T2 >
void display( T1 x, T2 y )
{
     cout<<“x=“<<x<<endl;
    cout<<“y=“<<y<<endl;
}
void main()
{
      display(100, “ABC”);
     display(11.22,’c’);
}
```

Function Overloading

- Function overloading is a concept where several function declarations are specified with a single and a same function name within the same scope. Such functions are said to be overloaded. C++ allows functions to have the same name. Such functions can only be distinguished by their number and type of arguments.

Function Overloading

- Example

```
float divide(int a, int b) ;
float divide(float a, float b) ;
```

- The function **divide(),** which takes two integer inputs, is different from the function **divide()** which takes two float inputs.

What is the need for function overloading?

- Every object has characteristics and associated behavior. An object may behave differently with change in its characteristics. Therefore, in order to simulate real world objects in programming environment, it is necessary to have function overloading.

What is the need for function overloading?

For Example:

```
float AddNumber(float a, float b)
{
        return a + b ;
}
int AddNumber(int a, int b)
{
        return a + b ;
}
void main()
{
        cout<<AddNumber( 21, 36 ) ;
        cout<<AddNumber( 6.72, 2.22 ) ;
}
```

How to implement function overloading?

- The key to function overloading is the function's argument list which is also known as function signature. It is the signature and not the function type that enables function overloading.
- If two functions have the same number and type of arguments in the same order, they are said to have the same signature.

How to implement function overloading?

```
void abc(int a, float b)
void abc(int x, float y)
```

- Both these functions have the same signature.

Sample for Function Overloading

- C++ allows you to overload a function provided the function has the same name but different signatures. The signature can differ in the number of arguments or in the type of arguments, or both. To overload a function, all you need to do is, declare and define all the functions with the same name but different signatures.

Sample for Function Overloading

```
void prnsqr(int i);
void prnsqr(char c);
void prnsqr(float f);
void prnsqr(double d);
void prnsqr(int i)
{
        cout<<"\n Integer "<< I <<"'s square is "<<I * i<<"\n";
}
void prnsqr(char c)
{
        cout<<"\n Character "<< c <<" thus no square "<<"\n";
}
void prnsqr(float f)
{
        cout<<"\n Float "<< f <<"'s square is "<<f*f<<"\n";
}
void prnsqr(double d)
{
        cout<<"\n Double "<<d<<"'s square is "<<d*d<<"\n";
}
```

Sample for Function Overloading

- When a function, with same name, is declared more than once in the program, the compiler will interpret the second declaration as follows:
- If the signature of subsequent function matches the previous function, then the second is treated as the re-declaration of the first.
- If the signature of both the functions match exactly, but the return type differs, then the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error.

For example,

```
float square(float f);
double square(float x); //error
```

The End