OOP345 – Inline Functions & Default Arguments

Sukhbir Tatla

sukhbir.tatla@senecacollege.ca

INLINE FUNCTION

- DEFINITION:

In C++,the functions that are not actually called, rather their code is expanded in line at the point of each invocation such functions are called as inline functions.

SYNTAX

```
return_type fun_name( );    //function declaration
int main( )
{
   ------------------------
   fun_name( );             //function call
   return 0;
}
inline return_type fun_name( );     //function definition
{
   -------
}
```

WHY THERE IS A NEED OF INLINE FUNCTIONS?

- A significant amount of overhead is generated by calling and return mechanism of function.
- While calling the function arguments are pushed onto the stack and saved on various registers and restore when function returns , this will take more time to run.
- If we expand a function code inline then function call produce faster run times.

SOME IMPORTANT POINTS

- Inline function process is similar to using a macro.
- Inline is actually just a request, not a command.
- By marking it as inline, you can put a function definition in a header file.

When to use inline?

- Function can be made as inline as per programmer need. Some useful recommendation are mentioned below:

  - Use inline function when performance is needed.
  - Use inline function over macros.
  - Prefer to use inline keyword outside the class with the function definition to hide implementation details.

Consider the following example

```
int sum(int a, int b)
{
        return a + b;
}
void print_sum()
{
        int r = sum(5,6);
        printf("%d\n", r);
}
```

If you declare your function as inline:

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

- The compiler will replace the actual function call to the actual body of your function, thus, the resulting binary will have something like:

```
void print sum()
{
        int r = 5 + 6;
        printf("%d\n", r);
}
```

Program

```
#include <iostream>
 using namespace std;
 inline int sqr (int x)            //defined function as inlined
 {
        int y;
        y = x * x;
        return y;
 }
 int main()
 {
            int a =3, b;                  // declaration of variables
            b = sqr(a);                    //function call
            cout <<b;
            return 0;
 }
```

Program

```
#include <iostream.h>
 using namespace std;
inline int Max(int x, int y)      //defined function as inlined
{
        return (x > y)? x : y;
}
int main( )                  // Main function for the program
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
  return 0;
}
```

- When the above code is compiled and executed, it produces the following result :

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

Function Vs Inline Function

- In many places we create the functions for small work/functionality which contain simple and less number of executable instruction.

- Imagine their calling overhead each time they are being called by callers.

How Function Calls Work?

- A function call uses the program call "**stack**"
  - Stack frame is "pushed" when the call is made
  - Execution jumps to the function's code block
  - Function's code block is executed
  - Execution returns to just after where call was made
  - Stack frame is "popped" (variables in it destroyed)
- This incurs a (small) performance cost
  - Copying arguments, other info into the stack frame
  - Stack frame management
  - Copying function result back out of the stack frame
- Too much run time overhead

Function Vs Inline Function

- The C++ inline function provides an alternative. With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code.
- Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.

SOME OF THE SITUATION WHERE INLINE EXPANSION MAY NOT WORK

- If the function code is large.
- If the function is recursive function .
- If the function contains static variables.
- For function returning values, if a loop or a switch case exists.

Function Default Arguments

Default Arguments

- Some functions can take several arguments
  - Can increase function flexibility
  - Can reduce proliferation of near-identical functions

- But, callers must supply all of these arguments
  - Even for ones that aren't "important"

- We can provide defaults for some arguments
  - Caller doesn't have to fill these in

Required vs. Default Arguments

- Function with required argument

```
// call as foo(2); (prints 2)
void foo(int a);
void foo(int a) {cout << a << endl;}
```

- Function with default argument
  - Notice only the *declaration* gives the default value

```
// can call as foo(2); (prints 2)
// or can call as foo();  (prints 3)
void foo(int a = 3);
void foo(int a) {cout << a << endl;}
```

Defaults with Multiple Arguments

- Function with one of two arguments defaulted

```
// can call as foo(2); (prints 2 3)
// or can call as foo(2, 4); (prints 2 4)
void foo(int a, int b = 3);
void foo(int a, int b)
  {cout << a << " " << b << endl;}
```

- Same function, with both arguments defaulted

```
// can call as foo(); (prints 1 3)
// or can call as foo(2); (prints 2 3)
// or can call as foo(2, 4); (prints 2 4)
void foo(int a = 1, int b = 3);
void foo(int a, int b)
  {cout << a << " " << b << endl;}
```

Default Argument Limitations

- Watch out for ambiguous signatures
  - `foo();` and `foo(int a = 2);` for example

    //Function Overloading (Will Learn Later).
- Can only default the rightmost arguments
  - Can't declare:

    ```
    void foo(int a = 1, int b);
    ```

    // Generates Compilation Error.
- Caller *must* supply leftmost arguments
  - Even if they're the same as the defaults

THANK YOU!

Any Questions Please?