Object Oriented Methodology

Week – *2*, Lecture – *2*

Introduction to C++

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Let us go back to C … for a while !!

```c
#include<stdio.h>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        printf("i + j = %d\n", add(i, j));
        printf("i + j + k = %d\n", add_to_result(k));
        printf("i * j = %ld\n", multiply(i, j));
        printf("i * j * k = %ld\n", multiply_with_result(k));
}
```

Let us go back to C … for a while !!

```c
#include<stdio.h>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        printf("i + j = %d\n", add(i, j));
        printf("i + j + k = %d\n", add_to_result(k));
        printf("i * j = %ld\n", multiply(i, j));
        printf("i * j * k = %ld\n", multiply_with_result(k));
}
```

What can you infer from this C code?

Let us go back to C … for a while !!

```c
#include<stdio.h>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        printf("i + j = %d\n", add(i, j));
        printf("i + j + k = %d\n", add_to_result(k));
        printf("i * j = %ld\n", multiply(i, j));
        printf("i * j * k = %ld\n", multiply_with_result(k));
}
```

There are two methods related to addition:

1. `add()`
2. `add_to_result()`

These may be declared in the header file `add.h`

Let us go back to C … for a while !!

```c
#include<stdio.h>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        printf("i + j = %d\n", add(i, j));
        printf("i + j + k = %d\n", add_to_result(k));
        printf("i * j = %ld\n", multiply(i, j));
        printf("i * j * k = %ld\n", multiply_with_result(k));
}
```

Similarly, there are two methods related to addition:

1. `multiply()`
2. `multiply_with_result()`

These may be declared in the header file , `multiply.h`

Let us go back to C … for a while !!

```c
#include<stdio.h>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        printf("i + j = %d\n", add(i, j));
        printf("i + j + k = %d\n", add_to_result(k));
        printf("i * j = %ld\n", multiply(i, j));
        printf("i * j * k = %ld\n", multiply_with_result(k));
}
```

All the above observations are correct, so this looks good to execute …

Let us go back to C … for a while !!

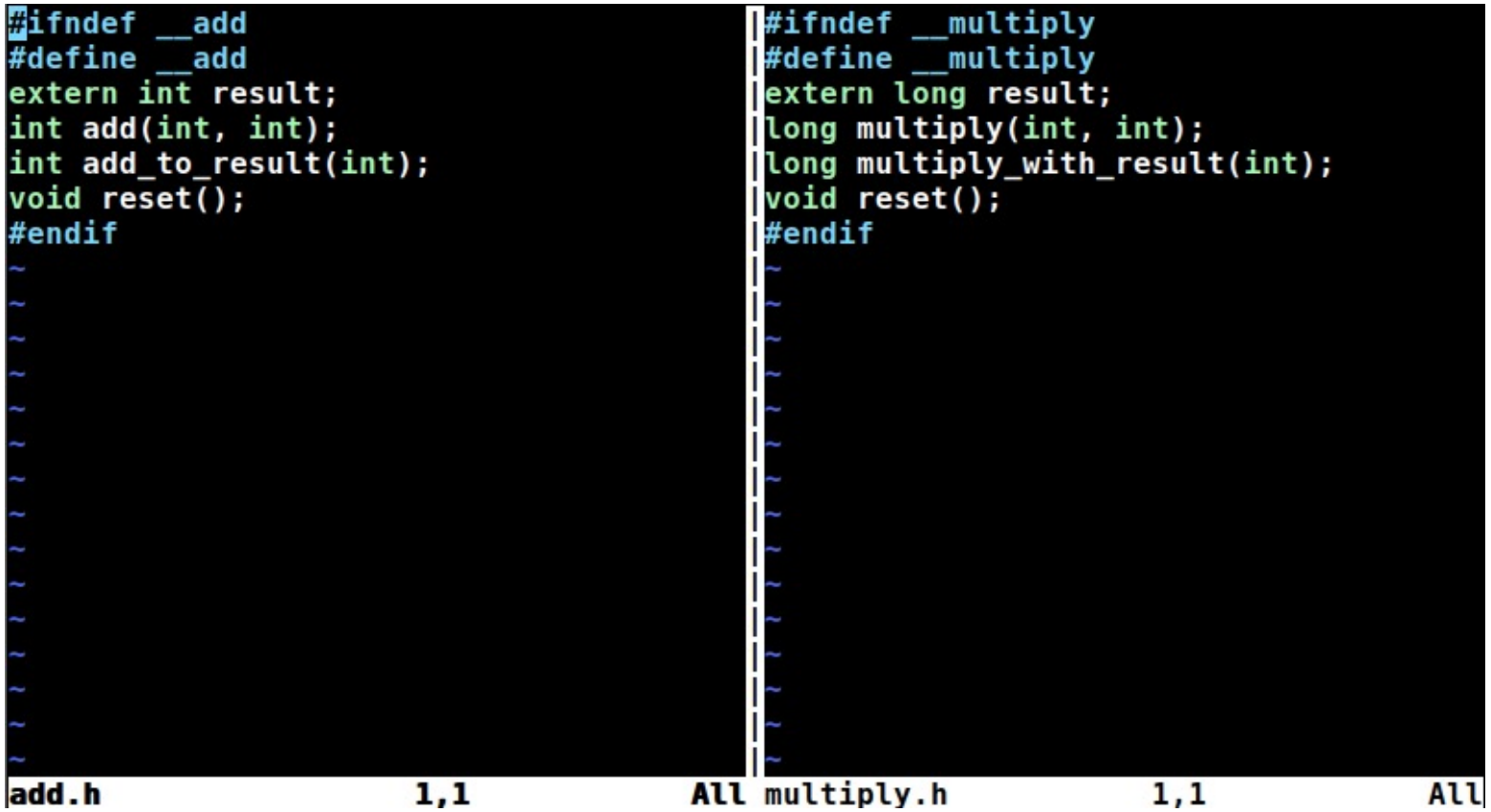
```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C$ make
gcc -o arithmetic-example main.c add.c multiply.c
In file included from main.c:4:0:
multiply.h:3:13: error: conflicting types for ‘result’
 extern long result;
             ^~~~~~
In file included from main.c:3:0:
add.h:3:12: note: previous declaration of ‘result’ was here
 extern int result;
            ^~~~~~
makefile:2: recipe for target 'arithmetic-example' failed
make: *** [arithmetic-example] Error 1
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C$
```

Let us go back to C … for a while !!

```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C$ make
gcc -o arithmetic-example main.c add.c multiply.c
In file included from main.c:4:0:
multiply.h:3:13: error: conflicting types for ‘result’
 extern long result;
             ^~~~~~
In file included from main.c:3:0:
add.h:3:12: note: previous declaration of ‘result’ was here
 extern int result;
            ^~~~~~
makefile:2: recipe for target 'arithmetic-example' failed
make: *** [arithmetic-example] Error 1
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C$
```

Oops… there is an error regarding multiple declarations for the identifier `result`

Let us go back to C … for a while !!



```
#ifndef __add
#define __add
extern int result;
int add(int, int);
int add_to_result(int);
void reset();
#endif
```

add.h 1,1 All

```
#ifndef __multiply
#define __multiply
extern long result;
long multiply(int, int);
long multiply_with_result(int);
void reset();
#endif
```

multiply.h 1,1 All

That does seem to be the case !!

Let us go back to C … for a while !!

```
#ifndef __add
#define __add
extern int result;
int add(int, int);
int add_to_result(int);
void reset();
#endif
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
add.h                                   1,1           All
```

```
#ifndef __multiply
#define __multiply
extern long result;
long multiply(int, int);
long multiply_with_result(int);
void reset();
#endif
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
multiply.h                              1,1           All
```

But if we assume that the two files came from different sources, it is certainly plausible to see such coincidences ☹

C++ to the rescue …

The problem with the C code we saw is that an extern variable is accessible "everywhere"

- In essence, the *space* in which it lives is the *global* space

C++ introduced a mechanism to restrict even global variables to a "smaller space"

This mechanism is known as a *namespace*

A namespace is a collection of C++ statements – declarations as well as definitions

All identifiers within a namespace must be unique, but there can be duplicates outside

Namespaces can be hierarchical – you can have one namespace as a child of another

- It might be confusing though, specially when the code of a namespace is *split across multiple locations* …
- … it is possible, by the way, to do so …

To access any variable or function within a namespace, prefix the namespace name followed by `::`

- `::` is known as the scope resolution operator

C++ Namespaces

```
#ifndef __add
#define __add
namespace add_ns
{
        extern int result;
        int add(int, int);
        int add_to_result(int);
        void reset();
}
#endif
~
~
~
~
~
~
~
~
~
~
~
~
~
~
add.h                   1,1           All
```

```
#ifndef __multiply
#define __multiply
namespace multiply_ns
{
        extern long result;
        long multiply(int, int);
        long multiply_with_result(int);
        void reset();
}
#endif
~
~
~
~
~
~
~
~
~
~
~
~
~
~
multiply.h              1,1           All
```

C++ Namespaces

add.h
```
#ifndef __add
#define __add
namespace add_ns
{
        extern int result;
        int add(int, int);
        int add_to_result(int);
        void reset();
}
#endif
```
1,1 All

multiply.h
```
#ifndef __multiply
#define __multiply
namespace multiply_ns
{
        extern long result;
        long multiply(int, int);
        long multiply_with_result(int);
        void reset();
}
#endif
```
1,1 All

Adding code to a namespace is no different than creating a code block, and adding the code inside it

C++ Namespaces

```
#include<iostream>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        std::cout<<"i + j = "<<add_ns::add(i, j)<<"\n";
        std::cout<<"i + j + k = "<<add_ns::add_to_result(k)<<"\n";
        std::cout<<"i * j = "<<multiply_ns::multiply(i, j)<<"\n";
        std::cout<<"i * j * k = "<<multiply_ns::multiply_with_result(k)<<"\n";
}
```

C++ Namespaces

```
#include<iostream>

#include "add.h"
#include "multiply.h"

int main()
{
        int i = 5, j = 10, k = 20;
        std::cout<<"i + j = "<<add_ns::add(i, j)<<"\n";
        std::cout<<"i + j + k = "<<add_ns::add_to_result(k)<<"\n";
        std::cout<<"i * j = "<<multiply_ns::multiply(i, j)<<"\n";
        std::cout<<"i * j * k = "<<multiply_ns::multiply_with_result(k)<<"\n";
}
```

The way to access the functions become a little tedious though, since you need to provide their fully qualified names

C++ Namespaces

```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$ make
g++ -o arithmetic-example main.cpp add.cpp multiply.cpp
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$ ./arithmetic-example
i + j = 15
i + j + k = 35
i * j = 50
i * j * k = 1000
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$
```

C++ Namespaces

```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$ make
g++ -o arithmetic-example main.cpp add.cpp multiply.cpp
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$ ./arithmetic-example
i + j = 15
i + j + k = 35
i * j = 50
i * j * k = 1000
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2/NamespaceExample/C++$
```

Now ... *All Izz Well* … (you may sing the song too, I can wait :-D)

Some more C++ basics

Most header files in C++ *do not* have the extension ".h"

- There is nothing sacred about it though… on some implementations, you may find the ".h" extension too

The `stdio.h` equivalent in C++ is `iostream`

The `printf()` equivalent in C++ is `cout`

- Unlike `printf()`, `cout` is not a function, but a *stream* (check your Homework)

The `scanf()` equivalent in C++ is `cin`

- Similar to `cout`, `cin` too is a stream and not a function

There are two major differences though between these two pairs

- You don't need to provide format specifiers to `cin` or `cout`, similar to `scanf()` and `printf()`
- To take multiple inputs or print multiple pieces of output, you have to use the << and >> operators
- The insertion operator (<<) is used with `cout` and the extraction operator (>>) is used with `cin`

Hello World – better late than never !!

```
#include<iostream>

using namespace std;

int main()
{
        cout<<"Hello World !!"<<endl;
        return 0;
}
```

Finally, the customary *Hello World* program is here !!

Hello World – better late than never !!

```cpp
#include<iostream>

using namespace std;

int main()
{
        cout<<"Hello World !!"<<endl;
        return 0;
}
```

Here, `std` is a special namespace, which contains all the declarations of the C++ standard library

Hello World – better late than never !!

```
#include<iostream>

using namespace std;

int main()
{
        cout<<"Hello World !!"<<endl;
        return 0;
}
```

The `using` keyword pulls up all the declarations from that namespace, so you are not required to write the fully qualified names of the functions or variables from that namespace …

Hello World – better late than never !!

```
#include<iostream>

using namespace std;

int main()
{
        cout<<"Hello World !!"<<endl;
        return 0;
}
```

This is why we wrote `cout` and not `std::cout`

Hello World – better late than never !!

```
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2$ g++ HelloWorld.cpp
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2$ ./a.out
Hello World !!
saurabh@saurabh-VirtualBox:~/C++/examples/Week 2$
```

Executing C++ programs is very similar to executing C programs

We just use `g++` instead of `gcc`, with the default executable name still being `a.out` !!

More data types – `bool` and `string`

C++ does have two more data types of interest – `bool` and `string`

The `bool` data type is a primitive data type

- It is boolean in nature, i.e. variables of bool type can take only two possible values – `true` and `false`

Another data type, which not a primitive type, but often used like one is `string`

- Strings are actually objects, however, they do support ”some special syntax”
- For instance, unlike objects in general, you can “add” two strings using the “+” operator
- It is possible due to a C++ feature, called *Operator Overloading* (we will see it in later weeks)
- Similar to C, you can also treat a `string` object like a `char` array …
- … i.e., access or change a particular character in the string using a subscript notation
- You will have to include the `<string>` header for using the strings
- The namespace in which string is defined is `std`

Strings and Booleans…

```cpp
#include<iostream>
#include<string>

using namespace std;

int main()
{
        string s1 = "This is a string";
        string s2 = "... and this is another string";
        string s3 = s1+s2; // '+' here means concatenation
        cout<<"s1 = "<<s1<<endl;
        cout<<"s2 = "<<s2<<endl;
        cout<<"s3 = s1 + s2 = "<<s3<<endl;
        bool b1 = s1 == s2;
        cout<<"s1 == s2 ? "<<b1<<endl;
        bool b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
        // Let us change s3 slightly
        s3[0] = 't';
        cout<<"New s3 = "<<s3<<endl;
        b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
}
```

Strings and Booleans…

```
#include<iostream>
#include<string>

using namespace std;

int main()
{
        string s1 = "This is a string";
        string s2 = "... and this is another string";
        string s3 = s1+s2; // '+' here means concatenation
        cout<<"s1 = "<<s1<<endl;
        cout<<"s2 = "<<s2<<endl;
        cout<<"s3 = s1 + s2 = "<<s3<<endl;
        bool b1 = s1 == s2;
        cout<<"s1 == s2 ? "<<b1<<endl;
        bool b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
        // Let us change s3 slightly
        s3[0] = 't';
        cout<<"New s3 = "<<s3<<endl;
        b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
}
```

An instance of `string` can be created simply by assigning it the required string in double quotes

You can also access particular characters in the string using the subscript notation, as if it is a `char` array (similar to the way you did in C)

You can also use the `+` operator on `string` objects to concatenate them

Strings and Booleans…

```cpp
#include<iostream>
#include<string>

using namespace std;

int main()
{
        string s1 = "This is a string";
        string s2 = "... and this is another string";
        string s3 = s1+s2; // '+' here means concatenation
        cout<<"s1 = "<<s1<<endl;
        cout<<"s2 = "<<s2<<endl;
        cout<<"s3 = s1 + s2 = "<<s3<<endl;
        bool b1 = s1 == s2;
        cout<<"s1 == s2 ? "<<b1<<endl;
        bool b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
        // Let us change s3 slightly
        s3[0] = 't';
        cout<<"New s3 = "<<s3<<endl;
        b2 = s3 == (s1 + s2);
        cout<<"s3 == (s1 + s2) ? "<<b2<<endl;
}
```

bool variables store one of the two values – `true` or `false`

`false` is represented by `0`, and `true` is represented by `1`

`bool` types can also be used to store result of conditions

Strings and Booleans…

```
s1 = This is a string
s2 = ... and this is another string
s3 = s1 + s2 = This is a string... and this is another string
s1 == s2 ? 0
s3 == (s1 + s2) ? 1
New s3 = this is a string... and this is another string
s3 == (s1 + s2) ? 0
```

The output produced by the shown program …

Here, `1` is representing `true`, and `0` means `false` (check out the homework)

Homework !!

Explore the `string` class at depth (it is going to be very handy for your programming tasks)

- These are some places you can have a look:
  https://www.geeksforgeeks.org/stdstring-class-in-c/
  https://www.w3schools.com/cpp/cpp_strings.asp
  http://www.cplusplus.com/reference/string/string/

The `bool` variables, when printed, will probably show `1` or `0`, instead of `true` or `false`

- See if you can print `true` or `false` instead of `1` and `0`. The following link maybe helpful:
  http://www.cplusplus.com/reference/ios/boolalpha/