COMP 322: Introduction to C++

Chad Zammar, PhD
Jan 22, 2021

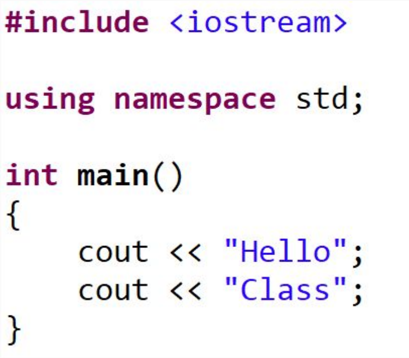
Lecture 3
(C++ basics)

- Quick recap
- Standard input & output
- Namespaces
- Functions
- Scope and lifetime of a variable

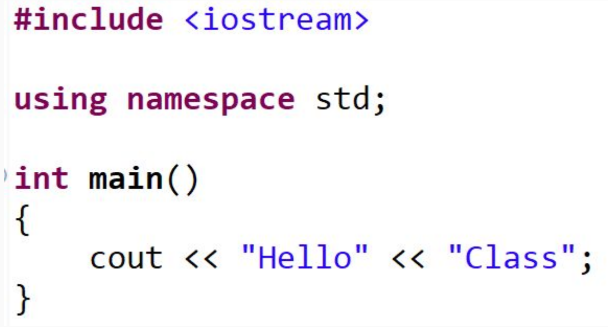
Standard input/output

- C++ uses "streams" for reading from (input) and writing to (output) a media
    - Media can be a keyboard, screen, file, printer, etc.
- Input and output streams are provided by the iostream header file
    - #include <iostream>
- cout stream object is used to print on screen
    - cout << "some message";
    - <<: insertion operator
- Default standard output is the screen
- Similar to printf() in c, system.out.println() in java

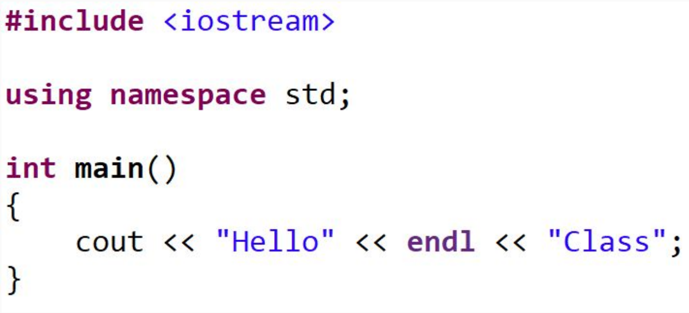
Standard input/output



```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello";
    cout << "Class";
}
```



```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello" << "Class";
}
```



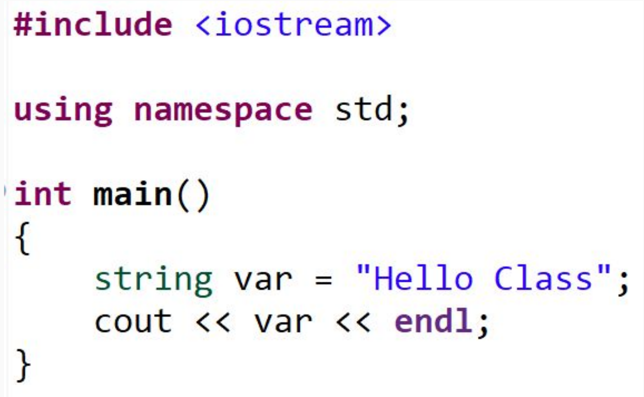
```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello" << endl << "Class";
}
```

Output:
HelloClass

Output:
HelloClass

Output:
Hello
Class

Standard input/output



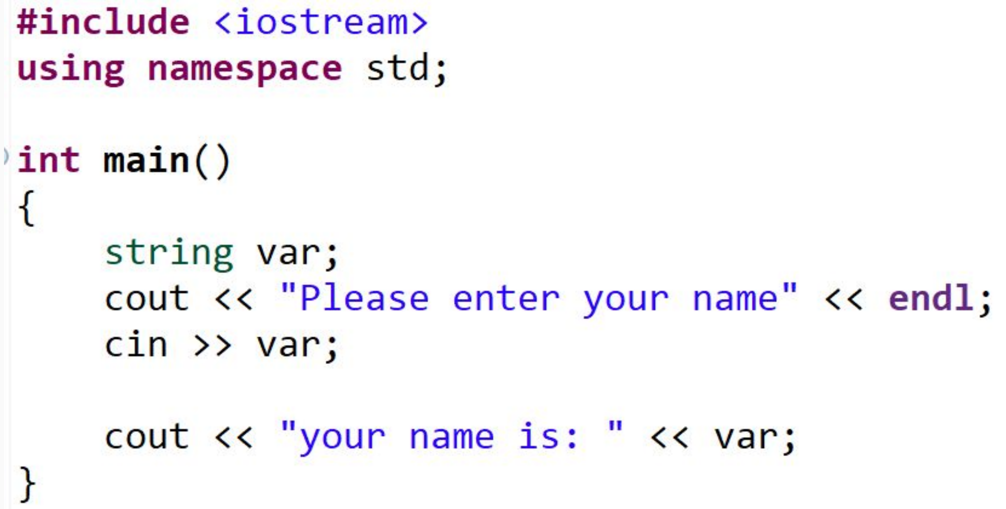
```cpp
#include <iostream>

using namespace std;

int main()
{
    string var = "Hello Class";
    cout << var << endl;
}
```

Output:
Hello Class

Standard input/output

- cin stream object is used to read from the keyboard
    - cin >> x;
    - >>: extraction operator
- Cin can read strings but limited to one word
    - cin >> stringVariable;
- Use getline function to read a full sentence
    - getline(cin, stringVariable);
- Similar to scanf() in c, scanner class in java

Standard input/output



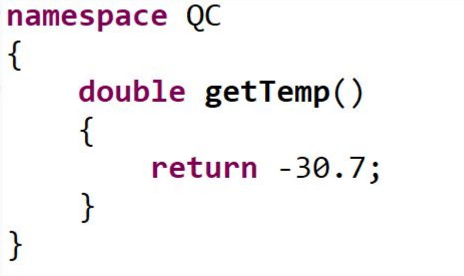
```
#include <iostream>
using namespace std;

int main()
{
    string var;
    cout << "Please enter your name" << endl;
    cin >> var;

    cout << "your name is: " << var;
}
```

Namespaces

- A name can represent only one variable within the same scope
- Large projects consists of multiple modules of code provided by different programmers
    - What happens if one module has a variable name that is the same as another variable in different module? Name conflict (also called name collision)
- Namespaces solve the name conflict problem

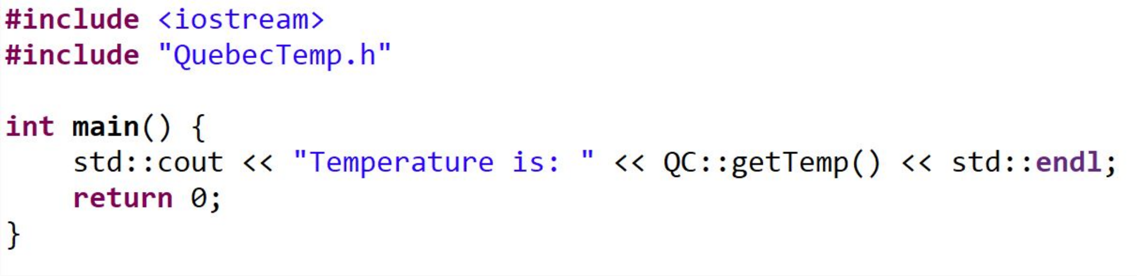
Namespaces

QuebecTemp.h



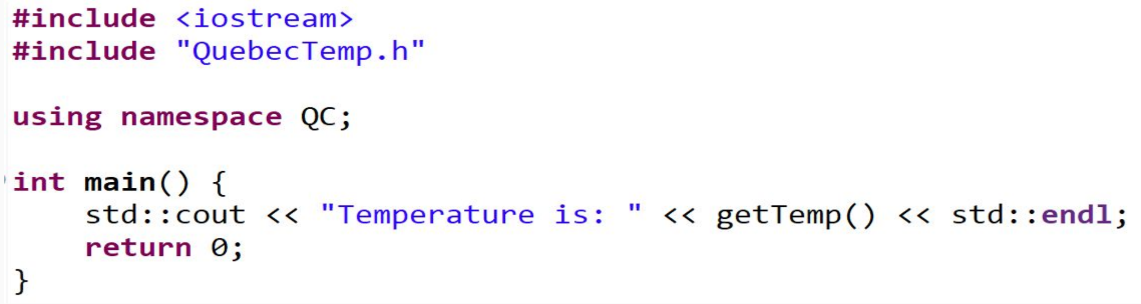
```
namespace QC
{
    double getTemp()
    {
        return -30.7;
    }
}
```

main.cpp



```
#include <iostream>
#include "QuebecTemp.h"

int main() {
    std::cout << "Temperature is: " << QC::getTemp() << std::endl;
    return 0;
}
```

Or also: main.cpp

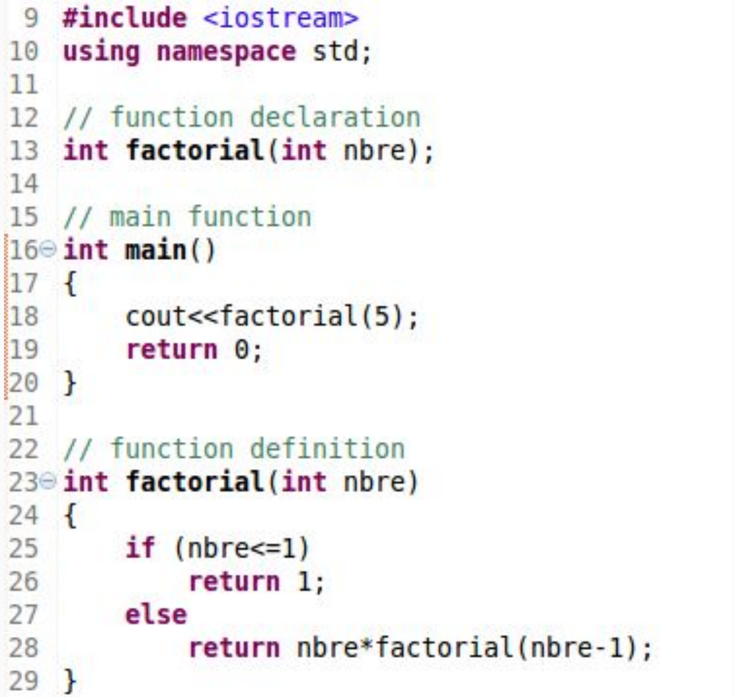


```
#include <iostream>
#include "QuebecTemp.h"

using namespace QC;

int main() {
    std::cout << "Temperature is: " << getTemp() << std::endl;
    return 0;
}
```

Functions

- Same as in C and java
- Should be declared before being used
- Declaration should include the name, return type and arguments type
  - Also called prototype or signature of a function
- If the function doesn't return a value, its return type should be declared ***void***
- Functions can be recursive

Recursive Function: example



```cpp
#include <iostream>
using namespace std;

// function declaration
int factorial(int nbre);

// main function
int main()
{
    cout<<factorial(5);
    return 0;
}

// function definition
int factorial(int nbre)
{
    if (nbre<=1)
        return 1;
    else
        return nbre*factorial(nbre-1);
}
```

The factorial function in this example is not optimal because it is not “tail-recursive”. Can you rewrite it in a more optimal way?

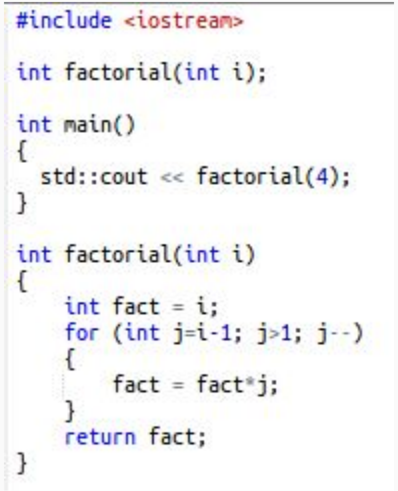
Factorial is the number of permutations for a set of objects.

Quiz

- Rewrite the factorial function but in an iterative (non-recursive) fashion.

Quiz

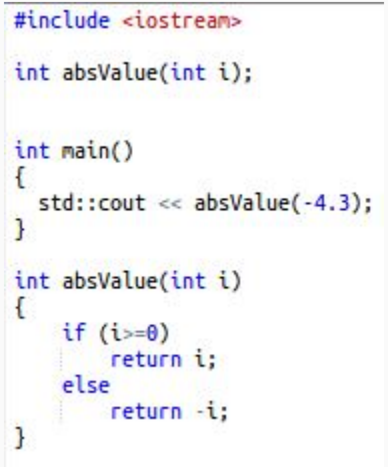
- **Rewrite the factorial function but in an iterative (non-recursive) fashion.**



```
#include <iostream>

int factorial(int i);

int main()
{
  std::cout << factorial(4);
}

int factorial(int i)
{
    int fact = i;
    for (int j=i-1; j>1; j--)
    {
        fact = fact*j;
    }
    return fact;
}
```

Function overloading

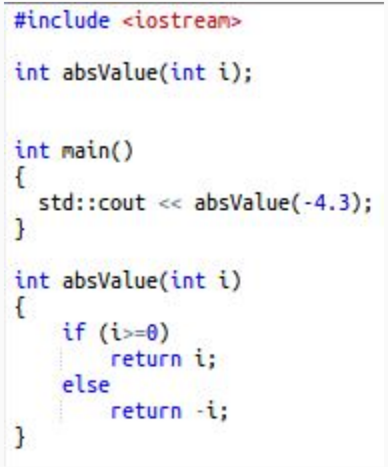
- **What's the output of the following code?**



```cpp
#include <iostream>

int absValue(int i);


int main()
{
  std::cout << absValue(-4.3);
}

int absValue(int i)
{
    if (i>=0)
        return i;
    else
        return -i;
}
```

Function overloading

- What's the output of the following code? (answer is 4 because of implicit conversion from double to int)

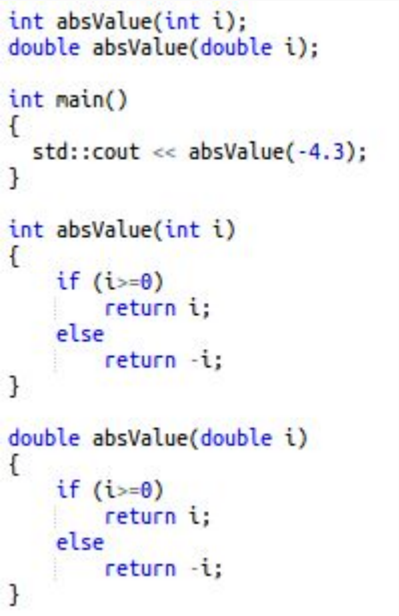


```
#include <iostream>

int absValue(int i);


int main()
{
  std::cout << absValue(-4.3);
}

int absValue(int i)
{
    if (i>=0)
        return i;
    else
        return -i;
}
```

Function overloading

- Multiple functions may have the same name but different number of arguments
  - Int max(int i, int j);
  - Int max(int i, int j, int k);
- Multiple functions may have the same name and same number of arguments but different types
  - Int max(int i, int j);
  - float max(float i, float j);
- Changing only the return type is not enough

Function overloading



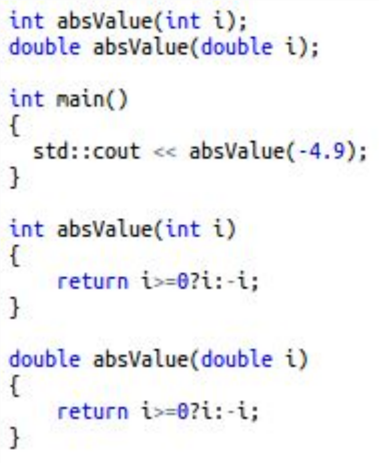
```
int absValue(int i);
double absValue(double i);

int main()
{
  std::cout << absValue(-4.3);
}

int absValue(int i)
{
    if (i>=0)
        return i;
    else
        return -i;
}

double absValue(double i)
{
    if (i>=0)
        return i;
    else
        return -i;
}
```

Quiz

- Rewrite the absolute value function from previous example using the ternary operator ?:

Quiz

- **Rewrite the absolute value function from previous example using the ternary operator ?:**



```
int absValue(int i);
double absValue(double i);

int main()
{
  std::cout << absValue(-4.9);
}

int absValue(int i)
{
    return i>=0?i:-i;
}

double absValue(double i)
{
    return i>=0?i:-i;
}
```

More about variables ...

- Variables have:
  - Name
  - Type
  - Address
  - Scope
  - Life span

Scope and lifetime of a variable

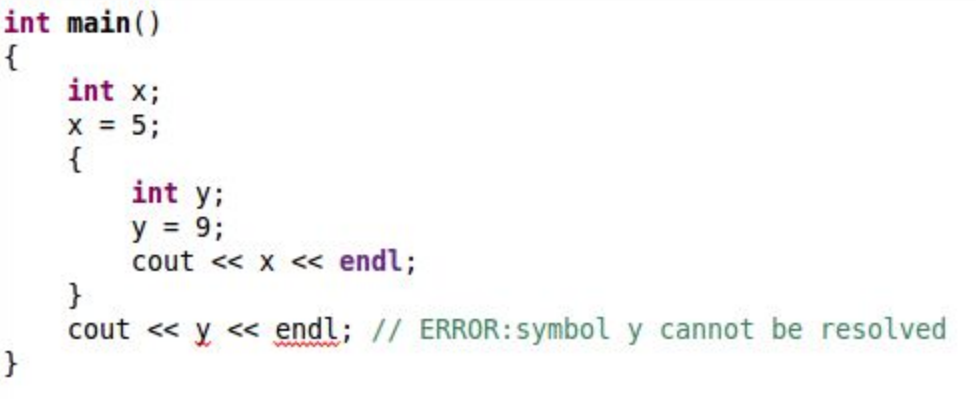
- When declaring variables we specify the name and type, but we should also keep in mind their scope and lifetime
- Scope of a variable
  - A section of the program where the variable is visible (accessible)
- Lifetime of a variable
  - The time span where the state of a variable is valid (meaning that the variable has a valid memory)

Scope and lifetime of a variable

- Local variables (that are non-static) have their lifetime ends at the same time when their scope ends
  - Local variables may also be called automatic variables because they are automatically destroyed at the end of their scope
  - Scope of local variables is comprised from the moment they are declared until the end of the block or function where they reside (in other terms, until the execution hits a closing bracket })

Scope and lifetime of a variable

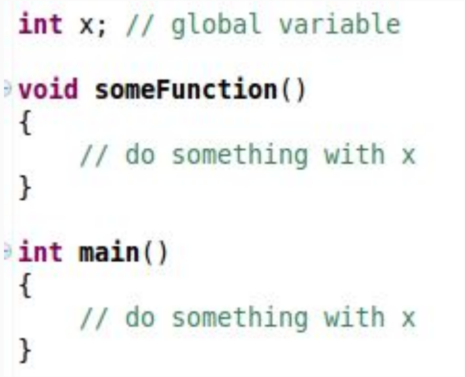
- **Local variables (that are non-static) have their lifetime ends at the same time when their scope ends**



```
int main()
{
    int x;
    x = 5;
    {
        int y;
        y = 9;
        cout << x << endl;
    }
    cout << y << endl; // ERROR:symbol y cannot be resolved
}
```

Scope and lifetime of a variable

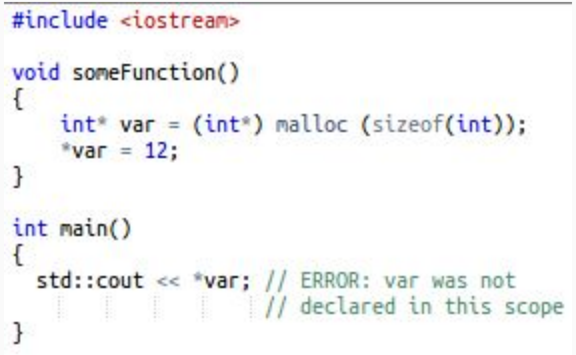
- Global variables have their lifetime ends when the execution of the program ends
  - Usually declared at the top of the file outside of any function or block
  - They have global scope



```
int x; // global variable

void someFunction()
{
    // do something with x
}

int main()
{
    // do something with x
}
```

Scope and lifetime of a variable

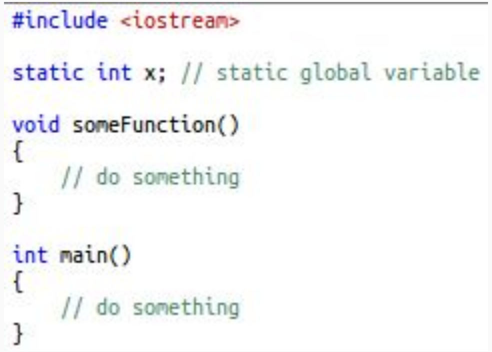
- Dynamically allocated variables have their lifetime starts when we explicitly allocate them (operator new, or malloc) and ends when we explicitly deallocate them (operator delete, or free)
  - Their lifetime is not decided by their scope (they may live even when they are out of scope)
  - We will get back to this in later chapters
  - The sample code provided has a memory leak
  - and assuming that someFunction() was being called before the cout statement.



```
#include <iostream>

void someFunction()
{
    int* var = (int*) malloc (sizeof(int));
    *var = 12;
}

int main()
{
  std::cout << *var; // ERROR: var was not
                     // declared in this scope
}
```

Scope and lifetime of a variable (static)

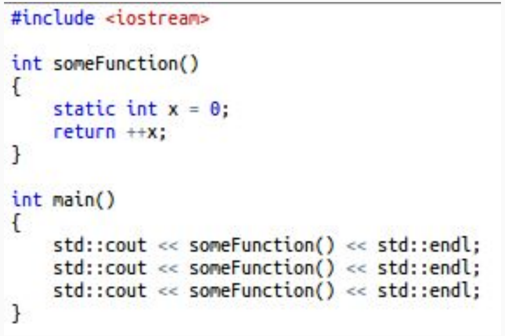
- **Global static variables have their lifetime ends when the execution of the program ends but their scope is limited to the file in which they are declared (file scope)**
    - **Scope is affected (reduced) but not the lifetime**



```
#include <iostream>

static int x; // static global variable

void someFunction()
{
    // do something
}

int main()
{
    // do something
}
```

Scope and lifetime of a variable (static)

- **Local static variables have their lifetime ends when the execution of the program ends but their scope is limited to the function in which they are declared (function scope)**
  - **Lifetime is affected (extended) but not the scope**



```cpp
#include <iostream>

int someFunction()
{
    static int x = 0;
    return ++x;
}

int main()
{
    std::cout << someFunction() << std::endl;
    std::cout << someFunction() << std::endl;
    std::cout << someFunction() << std::endl;
}
```

Reading assignment for next week

- Variable scope
- Namespaces
- What does "static" mean?
- Pointers
- Passing arguments by value VS passing arguments by reference