



C++ for Image Processing: STL, the Preprocessor, Debugging, etc.

Lecture 3



Today

- STL (Vectors)
- Namespaces
- Preprocessing
- Debugging



The STL (Standard Template Library)

- The **STL** includes “templatised” classes and functions that implement popular algorithms and data structures
 - vectors, lists, stacks, etc.
- It also provides a number of routines to access them
- Three fundamental components: **containers**, **algorithms** and **iterators**



The three components

- **Containers:** Objects that hold other objects
 - for example, the **vector** and **list** classes
 - sequence containers: lists, vectors, queues, stacks, double-ended queues (deque)
 - associative containers: sets, maps
- **Algorithms:** they act on containers
 - they provide means to manipulate containers
 - for example, to initialise, sort, search, access, etc.
- **Iterators:** they act more or less like pointers
 - can cycle through the contents of a container like you would use a pointer to cycle through an array



Some container classes

- **bitset** - a set of bits - <bitset>
- **deque** - a double-ended queue - <deque>
- **list** - a linear list - <list>
- **queue** - a queue - <queue>
- **set** - a set of unique elements - <set>
- **stack** - a stack - <stack>
- **vector** - a dynamic array - <vector>



Storing objects in STL container classes

- The use of containers places certain requirements on the objects that are stored in them
 - failing to meet the requirements is likely to cause compiler errors with messages that are very unhelpful at determining the problem
- Requirements may vary depending on the container type and on the algorithms you apply
 - in general, object should have a default constructor, and some operations defined on them (e.g. copy, equality, inequality)
- These can be compiler-specific
 - just be aware of these if you come across problems with storing objects in STL containers

Iterators

- Iterators are classes that are defined in terms of a container
 - e.g. `vector<char>::iterator p;`
- Every container has methods that return iterators associated with the container
 - e.g. `begin()`, `end()`
- Iterators also have associated operators
 - `++`, `--`, `!=`, `==`, etc.
- Iterators can be **forward**, **bidirectional**, or provide **random access**
 - depending on the direction of move and the size of "steps" when moving along containers

C++ for Image Processing

7

Vectors

- The vector class supports an **array that can grow dynamically** as needed
 - the vector container is defined as a template class, meaning that it can be customised to hold objects of any type
- Declaring a vector is easy
 - `vector<int> intVec; vector<char> chVec(5);`
- Many member functions provide useful functionality
 - `size()`, `begin()`, `end()`, `push_back()`, `insert()`, `erase()`, etc.

C++ for Image Processing

8

Accessing a vector through an iterator

- You can access members of a vector either through **subscripting** (like arrays) or through **iterators** (like pointers for arrays)

```
vector<char> v(10);
vector<char>::iterator p;
int i;
p=v.begin(); i=0;
while (p != v.end()) {
    *p = 'a' + i;
    p++; i++;
}
```

C++ for Image Processing

9

Inserting and deleting elements in a vector

- We can put values at the end of the vector using `push_back()`
- Or we can add elements in any position using `insert()`
 - `v.insert(p, 10, 'X')`
 - this will insert 10 'X's in the vector `v` at the position where the iterator `p` is pointing
 - `v.insert(p, v2.begin(), v2.end())`
 - This will insert in `v` the entire contents of vector `v2`

C++ for Image Processing

10

Inserting and deleting elements in a vector

- We can delete values from any position of a vector using `erase()`
 - `v.erase(p, p+10)`
 - this will erase the next 10 elements of vector `v` from the current position of the iterator `p`
- Inserting and deleting elements from the within the vector can change the index of the elements in the vector
- Note that **no array-like bounds checking** is done

C++ for Image Processing

11

Namespaces

- They are a recent addition to C++
- They were introduced to **localise** the names of **identifiers** in order to **avoid conflicts**
- This is a realistic need as the C++ programming environment has seen an explosion in variable, class and function names
- Before namespaces, all names were put in a single global namespace (including the standard library)
 - conflicts were very common, esp. between third-party libraries

C++ for Image Processing

12

Namespace basics

- In essence a namespace defines a **scope**
- Anything defined within a namespace statement is within the scope of that namespace

```
namespace name {  
    //declarations  
}
```

- Inside a namespace, identifiers declared within that namespace can be referred to directly
- Outside the scope of the namespace, we need to use the **:: scope identifiers** to refer to the namespace's identifiers

Example

Efd.h

```
namespace EFDnameSpace {  
    float altitude;  
    float speed;  
  
    class flaps {  
        int degrees;  
        int status;  
    public:  
        void set_flaps (int  
            d)  
        { if (altitude < 10000)  
            degrees = d;}  
}; }
```

Efd.cpp

```
#include <iostream>  
#include "efd.h"  
using namespace std;  
  
int main() {  
    EFDnameSpace::speed = 234;  
    EFDnameSpace::altitude = 9030;  
  
    EFDnameSpace::flaps f;  
  
    f.set_flaps(15);  
}
```

Some more info

- You can split namespaces over several files or even separate them within the same file

```
namespace NS { int i;}  
// ...  
namespace NS {int j;}  
int main () {NS::i = NS::j = 10;  
    cout<< NS::i * NS::j; }
```

- A namespace **must be declared outside of all other scopes**
- A namespace **can be nested** within another namespace
 - NS1::NS2::j = 10; (NS2 is nested within NS1)

using

- If you need to make frequent references to the members of a namespace you can use **using**

- **using namespace name;**

- all identifiers of the namespace can be referred to without the scope resolution

```
using namespace EFDnameSpace;  
altitude = 8000;
```

- **using name::member**

- only a specific member of the namespace can be used without scope resolution

```
using EFDnameSpace::altitude;  
altitude = 8000;
```

More on using

- The scope of **using** is restricted within the scope (i.e. {...}) in which it appears
- Using one namespace does not override another
 - you can have the **std** namespace and the **EFDnameSpace** in effect at the same time

The std namespace

- Standard C++ defines its entire library in the **std** namespace
 - this saves us having to qualify every name of function etc. by **std::**
 - you can still type **std::cout << "Hello World";**
- If you are using only a few names from the standard library you could specify a **using** statement for each individually
 - **using std::cout; using std::cin;**

Namespace - Example

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

namespace first
{ int x = 5;
  int y = 10;
}

namespace second
{ double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Output:
5
2.7183
10
3.1416

Namespace - Example

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

namespace first
{ int x = 5;
  int y = 10;
}

namespace second
{ double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Output:
5
10
3.1416
2.7183

Namespace - Example

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

namespace first
{ int x = 5;}

namespace second
{ double x = 3.1416;}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

Output:
5
3.1416

The Preprocessor

- The preprocessor touches your program before the compiler
- Preprocessor commands (directives) start with #
 - e.g. **#include**
- It allows you to include the contents of one file into another
 - the contents of the included file are literally pasted into the calling file

#include statements

- When including using `<...>` the compiler will search standard list of directories for these files
 - using the `-I` option of the compiler can allow you to add directories where the compiler looks for files
- When including using `"..."` the compiler looks in the current directory for the included file
 - or in another directory that you explicitly express within the `"..."` boundaries

When to #include?

- `#include` is typically used to **include header files**
- Some cases where you should `#include` the header file for a class:
 - when you create an instance of the class
 - when you call a member function on an instance of the class
 - when you access a public instance variable on an instance of the class
 - when you access a static function or member of the class
 - when you implement methods for the class
 - when the class you are declaring is a subclass of the class

Conditional compilation

- It is mainly used to avoid **multiple definitions of classes** - things can not be defined twice in C++
- If A.h includes B.h and B.h includes A.h we have a problem
- Conditional compilation can solve this

Conditional compilation

- It is good practice to “protect” your class definitions in the header files in the following way

```
#if ! defined (FILENAME_H)
#define FILENAME_H
// content of the header file goes here
#endif //ALWAYS remember this
```

Forward declarations

- Sometimes you only need to know the name of a class that you refer to in your code
 - but not any member functions or variables
- Then you can use a forward declaration
 - declare only the name of the class
- The compiler will add the name to the symbol table so that there is no error
 - but it also awaits for the class to be fully declared before operations on e.g. the class's variables and functions take place

Forward declarations: example

```
File C1.h
*****

class C2; //Forward
          declaration

class C1 {
int number;
public:
int test_func(C1 a, C2 b);
};
```

- You could achieve the same with #include <C2.h>
- Using a forward declaration **can save you time** when you compile your source code after you make changes
 - more on this later
- It is a good idea to try to use forward declarations where possible
 - mainly in the header files

Other preprocessor directives

- There are more directives you can use
 - #define, #undef
 - #error (for error message)
 - #ifdef
 - #if, #elif (else if), #else
 - etc.

The Build process

- For each source file (.cpp), the preprocessor copies in any included .h files and generates a large temporary file
- This temp file is **compiled** into an object (.o) file
- There is a final step (**linking**) that links together multiple object files and external library files into an executable
- If you change your program, you should find out which source files need recompiling -> to make this easier for large programs we use a **Makefile**

Example

main.cpp

```
#include "Point.h"
#include "Rectangle.h"
...
{ ... }
```

Point.cpp

```
#include "Point.h"
...
{ ... }
```

Rectangle.cpp

```
#include
"Rectangle.h"
...
{ ... }
```

Point.h

```
class Point
{ ... };
```

Rectangle.h

```
#include "Point.h"
class Rectangle
{ . . .
    Point bottom_left;
    Point bottom_right;
    . . .
};
```

C++ for Image Processing

31

Example: Solving redefinition

main.cpp

```
#include "Point.h"
#include "Rectangle.h"
...
{ ... }
```

Point.cpp

```
#include "Point.h"
...
{ ... }
```

Rectangle.cpp

```
#include
"Rectangle.h"
...
{ ... }
```

Point.h

```
#if !defined POINT_H
#define POINT_H
class Point
{ ... };
#endif
```

Rectangle.h

```
#include "Point.h"
class Rectangle
{ . . .
    Point bottom_left;
    Point bottom_right;
    . . .
};
```

C++ for Image Processing

32

A Makefile for example

```
# Entries to compile c++ programmes to .obj files (Compilation)
main.o: main.cpp Point.h Rectangle.h
g++ -c main.cpp
Point.o: Point.cpp Point.h
g++ -c Point.cpp
Rectangle.o: Rectangle.cpp Rectangle.h Point.h
g++ -c Rectangle.cpp

# Entries to bring the executable up to date (linking)
main: main.o Point.o Rectangle.o
g++ -o main main.o Point.o Rectangle.o

# Entries to compile and link with all warnings
g++ -o main -Wall main.cpp Point.cpp Rectangle.cpp
```

C++ for Image Processing

33

Debugging

- Your program may have:
 - Compile errors
 - Run time errors
 - Algorithmic/logical errors
 - Memory leaks
 - Numerical errors
- You can use the `gdb` debugger in the ITL
 - Material & a short tutorial to be added on the course website

C++ for Image Processing

34

What next?

- No more lectures on C++
 - week 4 lecture on Image Processing
- more labs on C++
 - **Week 4 lab:** material from today's lecture
 - **Week 5 lab:** gdb, image I/O using wxwin libraries
- After week 5
 - labs to help you with your CIP coursework

C++ for Image Processing

35