# 122COM: Intermediate C++

David Croft

Coventry University

david.croft@coventry.ac.uk

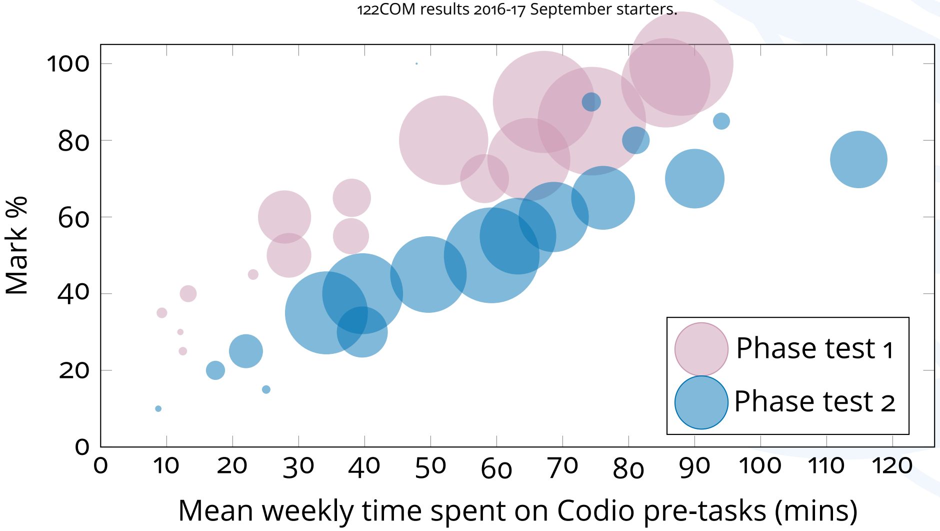
2018

## Overview

## Expectations

C

You have all attempted the green Codio exercises for this week.



## Introduction

## Expectations

C

Have been trying to make C++ as similar to Python as possible.

- Minimize differences, make it easier to adapt.

C++ is however a very different language.

- Will now start to move away from Pythonesque code.
- Will make it harder to translate from one language to another.
- Will make it easier to write C++ programs.

Am not expecting proficiency with all these features in this session.

- Making you aware so you can do further investigations.

## Standard Template Library

I

What is the Standard Template Library (STL)?

- Modern C++ is really a language of two parts.
  - The core language, `if`, `for`, classes, C-style arrays, raw pointers etc.
  - The STL, more advanced containers and algorithms, `vector<>`, `array<>`, smartpoints, strings etc.
  - ● More complicated in reality, but we don't care right now.
- It is possible to write C++ code without using any of the STL.
  - Not recommended.
  - The STL can be seen as collection of additional functionality added onto core C++.

# Iterators

## Standards

One of the key features of the STL is that everything is standarised.

- Standards have been agreed.
- If everyone follows them then everyones code will work together.

- One of the most significant standards is that all container objects should have an iterator interface.

## iterators

I

Similar to pointers, special kind of variable.

- Act as a combined value and position of an element in a sequence.
- An iterator has a position in a sequence.
  - Can move an iterator forwards and/or backwards through a sequence.
  - Best way `next(it)`, `prev(it,2)`.
  - Also works (sometimes) `it++`, `it-2`.
- An iterator refers to a value in a sequence.
  - Can read and write to elements in a sequence.
  - `cout << *it;`      `*it == 42;`.
- `it` is the position, `*it` is the value.
  - Like pointers.

## Iterator locations

I

Notice that `end()` is 1 **PAST** the last element of the container.
DeclareAcronyn

```
vector<int> seq { 11, 22, 33, 44, 55 };

// prints 11
cout << *seq.begin() << endl;

// both print 55
cout << *prev( seq.end() ) << endl;
cout << *(seq.end()-1) << endl;

// prints 5 because seq contains 5 elements
cout << seq.end() - seq.begin() << endl;

/* prints random value, value at
   seq.end() is NOT part of the vector */
cout << *seq.end() << endl;
```
lec_iter_visualisation.cpp

| Element | Value | |
|---|---|---|
| 0 | 11 | ← `begin()` |
| 1 | 22 | |
| 2 | 33 | |
| 3 | 44 | |
| 4 | 55 | |
| | | ← `end()` |

## iterators II

How would we actually use iterators?

```
vector<int> seq { 2, 4, 6, 8, 10 };

// traditional index based iteration of a sequence;
for( int i=0; i<seq.size(); ++i )
    cout << seq[i] << endl;

// iteration of a sequence using iterators
for( vector<int>::iterator it=seq.begin(); it!=seq.end(); it=next(it) )
    cout << *it << endl;

/* iteration using iterators, note that the "it" variable is still of type
    vector<int>::iterator, using auto simply means that we don't have to
    type all that and the compiler figures it out for us */
for( auto it=seq.begin(); it!=seq.end(); it=next(it) )
    cout << *it << endl;
```

lec_iterators.cpp

## iterators III

I

What's the point? What was wrong with index based iteration?

- Index based iteration doesn't work with structures like linked lists.
  - You can't just get the $n^{th}$ element, you have to step through.
- Can modify the sequence as you iterate (depends on other factors).
- Lets you write generic functions.
  - Write one function and it works for multiple containers, arrays, vectors, linked lists, trees.
  - Write one function and it works backwards and forwards (iterators, reverse iterators).
  - Write one function and it works on whole ranges and subsections.
  - ● Not always that simple in reality (e.g. `ForwardIterator` vs `RandomAccessIterator`).

## iterators IV

I

```
template<typename ITER>
void print_with_hyphens( ITER begin, ITER end )
{     for( ITER it=begin; it!=end; it=next(it) )
          cout << *it << "-";
     cout << endl;
}

int main()
{     vector<int> v { 2, 4, 6, 8, 10, 12 };
     list<int> l { 2, 4, 6, 8, 10, 12 };

     // print the entire sequence regardless of container type
     print_with_hyphens( v.begin(), v.end() );
     print_with_hyphens( l.begin(), l.end() );

     // print the first half of the sequence
     print_with_hyphens( v.begin(), next( v.begin(), v.size()/2 ) );

     // print the sequence in reverse order. rbegin returns a reverse iterator
     print_with_hyphens( v.rbegin(), v.rend() );
```

lec_iter_ranges.cpp

## Break

## iterators IV

I

```
int main()
{
    vector<int> v { 2, 4, 6, 8, 10, 12 };

    // index iteration approach
    int total = 0;
    for( int i=0; i<v.size(); ++i )
        total += v[i];
    cout << total << endl;

    // functional programming and iterator approach
    cout << accumulate( v.begin(), v.end(), 0 ) << endl;

    /* parallel implementations coming soon? C++17 maybe?
    cout << reduce( parallel::par, v.begin(), v.end(), 0 ) << endl;
    */

    return 0;
}
```

lec_functional.cpp

## iterators V

I

With iterators a single templated function can work on...

- Multiple container types.
  - `vector`, `list`, `set`, `array`, `queue`, `stack`.
- Process the container forwards or backwards.
  - `vector.begin()` gives an `iterator`.
  - `vector.rbegin()` gives a `reverse_iterator`.
- Process subsections of a container.
  - Basically Python slicing.
- Lots of really useful pre-existing functions all taking iterators.
  - Check out the `algorithm` library.
  - `find()`, `copy()`, `replace_if()`, `reverse()`, `next_permutation()` etc.

# Overloading

## Static typing problems

C

```
def add( a, b ):
  return a + b

add( 123, 456 )           # works
add( "Monty", "Python" ) # works
```

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

int main()
{
  add( 123, 456 );         // works
  add( "Monty", "C++" ); // error
}
```

In Python if you have a function the parameters can take any variable type.

C++ is statically typed so have to explicitly state parameter types.

- Can't reuse functions in same way.

## Overloading

Overloading allows different version of function to share single name.

```
// version 1
int add( int a, int b ) { return a + b; }

// version 2
string add( string a, string b ) { return a + b; }

// version 3
int add( int a, int b, int c ) { return a + b + c; }

int main()
{
  int i = add( 1, 2 );                  // calls version 1
  string f = add( "Monty", "Python" ); // calls v2
  int j = add( 1, 2, 3 );               // calls version 3
}
```
lec_overloading.cpp

- Compiler deduces which version to call based on parameter and return types.
- But what about `add(1, "hi");`?

## Overloading

Overloading allows different version of function to share single name.

```cpp
// version 1
int add( int a, int b ) { return a + b; }

// version 2
string add( string a, string b ) { return a + b; }

// version 3
int add( int a, int b, int c ) { return a + b + c; }

int main()
{
  int i = add( 1, 2 );           // calls version 1
  string f = add( "Monty", "Python" ); // calls v2
  int j = add( 1, 2, 3 );        // calls version 3
}
```
lec_overloading.cpp

- Compiler deduces which version to call based on parameter and return types.
- But what about `add(1, "hi");`?

  No version that takes that combination so compiler error.

# Templates

## What's the problem?

Overloaded functions allow us to handle different variable types.
Problem.

- There are a lot of types.
  - Just the C++ primates have 13 variants.
- Need a function for each one.
  - At least 13 versions of the function.

```python
def add( a, b ):
  return a + b;

def main():
  i = add( -1, 2 )
  f = add( 1.1, 2.2 )
  s = add( "Monty", "Python" )
  l = add( [1,2,3], [4,5,6] )
```

lec_without_templates.py

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

unsigned int add( unsigned int a, unsigned int b )
{
    return a + b;
}

float add( float a, float b ) {    return a + b; }

char add( char a, char b ) { return a + b; }

string add( string a, string b ) { return a + b; }

vector<int> add( vector<int> a, vector<int> b )
{
    a.insert( a.begin(), b.begin(), b.end() );
    return a;
}

int main()
{
    int i = add( -1, 2 );
    unsigned int ui = add( 1, 2 );
    float f = add( 1.1f, 2.2f );
    char c = add( 'A', '3' );
    string s = add( "Monty", "C++" );
    vector<int> v = add( vector<int>{1,2,3},
    ↪   vector<int>{4,5,6} );

    cout << i << " " << ui << " " << f << " " << c << " "
    ↪   << s << " ";
    for( int i : v ) cout << i << ", "; cout << endl;
}
```

lec_without_templates.cpp

## What's the solution?

So if we want an `add()` function for ints, strings, floats, doubles, shorts, longs, unsigned ints....

- Have to write a copy of the function for each type.
  - Programming advice - anytime you are writing same code over and over with little tweaks there is probably a better way.
- There is a cheat.
  - Still have to have a separate function for each variable type.
  - But can get the compiler to write them for us.
  - Templates!

## How?

C

The compiler will generate the necessary versions of the templated functions/classes/etc.

- Works out what type the template identifier needs to be.
- Effectively does a find-replace in the code.
  - More complicated in reality, don't care.

```
template<typename TYPE>
TYPE add( TYPE a, TYPE b )
{
    return a + b;
}

int main()
{
    int i = add( 1, 2 );
    float f = add( 1.1f, 2.2f );

    cout << i << " " << f << endl;
}
```

lec_with_templates.cpp

## How?

C

The compiler will generate the necessary versions of the templated functions/classes/etc.

- Works out what type the template identifier needs to be.
- Effectively does a find-replace in the code.
  - More complicated in reality, don't care.

```
template<typename TYPE>
TYPE add( TYPE a, TYPE b )
{
    return a + b;
}

int main()
{
    int i = add( 1, 2 );
    float f = add( 1.1f, 2.2f );

    cout << i << " " << f << endl;
}
```

lec_with_templates.cpp

Compiler generated code.

*Doesn't actually look like this at all.*

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

float add( float a, float b )
{
    return a + b;
}
```

## Advantages/Disadvantages

I

Can be used for functions and classes.

Advantages

- Write less code.
- Reuseable code.

Disadvantages

- Takes longer to compile.
- All the additional code generated can cause code bloat.
- Hideous compiler errors.
  - Code is generated by compiler so doesn’t look like normal code.
  - If it goes wrong the errors messages can be very confusing.
  - Harder to debug since you never see the actual code producing the errors.

## STL examples

I

The Standard Template Library (STL) makes heavy use of templates.

- Containers for any type.
- Functions for any type.

```
vector<int> v { 2, 4, 6, 8, 10, 12 };       // storing ints
vector<float> f { 2.2, 4.4, 6.6 };          // storing floats

list<int> l { 2, 4, 6, 8, 10, 12 };         // storing ints differently
list<string> s { "apple", "ball", "cat" };

pair<int,float> p { 123, 45.6f };           // double template

vector< vector<int> > vv { { 1, 2, 3}, { 4, 5, 6 } }; // template in a template

cout << max( 1, 99 ) << endl;              // same function, different input types
cout << max( 'A', 'Z' ) << endl;
```

lec_stl_template_examples.cpp

## Typedefs

A

One issue of the STL is that our variable names can start to get out of hand. Imagine a variable representing a line.

- A line is made up of two points.
- Points are made up of two co-ordinates, x & y.
  - So can represent it as a pair of pairs.
- Could define a custom class.
  - Quite a lot of hassle.
- Could make it out of existing STL containers.
  - But horrific to type.

```
#include <utility>
pair< pair<int,int>, pair<int,int> > line1;
pair< pair<int,int>, pair<int,int> > line2;
```

## Typedefs

A

Or use `typedef` declaration to give an alias to a complex type name.

- Can write the alias in code instead of unpleasant templated type.

```
pair< pair<int,int>, pair<int,int> > line1;
pair< pair<int,int>, pair<int,int> > line2;
```

Before.

```
typedef pair< pair<int,int>, pair<int,int> > Line;

Line line1;
Line line2;
```

After.

Break

# Libraries

## Libraries

As you start writing more advanced programs you will need to...

- Use 3$^{rd}$ party libraries.
- Organise your code into separate files.
- Same concepts as Python's `import`.

## Not `import`

I

As with Python the statements go at the top of the source file.
In C++ `#include` statements are used to import additional files.

- Two variations available.
  - `#include "filename"`
  - `#include <filename>`
- `#include "filename"` will search the local directory.
  - Normally used for user written files.
- `#include <filename>` will search the include directories.
  - Will usually be several pre-set, more can be listed at compile time.
  - Normally used for installed libraries.

I

Best practise if to spread over multiple files:

- Header (.h) and source (.cpp) files.
- Compile each source file separately.

Header says what functions/classes are available.

lec_print_two.h

```
#ifndef LEC_PRINT_TWO_H
#define LEC_PRINT_TWO_H

#include <iostream>
using namespace std;

void print_two( int a, int b );

#endif
```

Source provides the actual implementation.

lec_print_two.cpp

```
#include "lec_print_two.h"

void print_two( int a, int b )
{
  cout << a << " " << b << endl;
}
```

I

If main program wants to use `print_two()`.

- Include header for that function (`lec_print_two.h`).

lec_include.cpp

```
#include "lec_print_two.h"

int main()
{
   print_two( 42, 69 );

   return 0;
}
```

## Multi file compilation

A

1. Compile just `lec_print_two.cpp` into an object file `lec_print_two.o`

   ```
   g++ --std=c++14 -c lec_print_two.cpp
   ```
2. Compile just `lec_include.cpp` into an object file `lec_include.o`

   ```
   g++ --std=c++14 -c lec_include.cpp
   ```
3. Link the object files together and create the final executable.

   ```
   g++ --std=c++14 lec_include.o lec_print_two.o -o lec_include
   ```

Also possible to compile with

```
g++ --std=c++14 lec_include.cpp lec_print_two.cpp -o lec_include
```

- But this means re-compiling the `lec_print_two.cpp` code every time even if it hasn't changed.

## Why?

- Faster compilation.
  - If code changes only re-compile the parts that have changed.
- Especially important with large 3rd party libraries e.g. OpenCV, Boost etc.
  - Can take hours to compile from scratch.
  - Many libraries are available pre-compiled, i.e. available as .o files.
- Well worth your time to investigate Makefiles and CMake.
  - Automates the compilation steps.

## Sqlite3 example

A

In the case of the sqlite3 library.
Often you will be able to use the library like so:

```
g++ --std=c++14 myprog.cpp -lsqlite3
```

- `-lsqlite3` - There's a pre-compiled file called `libsqlite3`, go link to that.

Worst case scenario:

```
g++ --std=c++14 -I/usr/include/sqlite3 myprog.cpp
 -L/usr/lib/sqlite3 -lsqlite3
```

- `-I/usr/include/sqlite3` - Here's where the .h files are.
- `-L/usr/lib/sqlite3` - Here's where the .o (and .so) files are.
- `-lsqlite3` - There's a pre-compiled file called `libsqlite3`, go link to that.

# Recap

## Investigate further

A

C++ is a large language. Many additional features not covered but important, especially for optimisation.

- `const` variables.
  - Once the value has been set, can't be changed.
- `const` member functions.
  - Do not modify the attributes of a class object.
- C++ lambda functions.
  - Same benefits as Python.
- File input/output.
  - Don't forget binary files.
- The preprocessor.

## Recap

- 🟢 C++ is a high level language.
- 🟢 Compiled.
- 🟢 Statically typed.
- 🟢 Arrays cannot be resized.
  - 🟡 Use new STL arrays.
- 🟢 Vectors can be resized.
- 🔴 Investigate C++ classes.
- 🔴 Investigate STL Algorithm Library.

## Why do I care?

Nothing making you use these features.

- Everyone.
    - Modern C++ has significant differences to legacy C++.
    - Best practises.
    - Employability.
    - Safer code.
    - Less code.
    - Less buggy code.
    - Faster written/running code.

## Expectations

C

- Complete the yellow Codio exercises for this week.
- Attempt the green Codio exercises for next week.

- If you have spare time attempt the red Codio exercises.

- If you are having issues come to the PSC.

  `https://gitlab.com/coventry-university/programming-support-lab/wikis/home`

# The End