A brief introduction to C++

Rupert Nash
r.nash@epcc.ed.ac.uk

13 June 2018

References

- Bjarne Stroustrup, "Programming: Principles and Practice Using C++" (2nd Ed.). Assumes very little but it's long
- Bjarne Stroustrup, "A Tour of C++". Assumes you're an experience programmer and is quite brief.
- Best online reference I've found is `http://en.cppreference.com/` (Comes other human languages too!)
- Scott Meyers, "Effective Modern C++", 2014. This is the book to get once you know your way around C++, but you want to improve. Teaches lots of techniques and rules of thumb for writing correct, idiomatic, maintainable code.
- `stackoverflow.com` has a lot of good questions about C++ (look for ones with at least 100 up-votes).

Misunderstood monsters: Frankenstein's vs. C++

 [1]

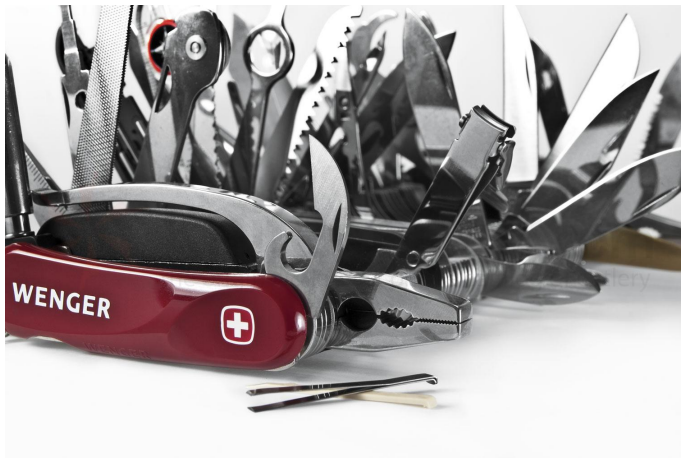
[1] By Universal Studios - Dr. Macro, Public Domain, https://commons.wikimedia.org/w/index.php?curid=3558176

Misunderstood monsters: Frankenstein's vs. C++

- Large: the C++11 standard is about 1300 pages
- Composed of many parts: C, classes, generics, functional programming, exceptions, the vast library, ...
- Inspires dread in those do not understand it
- Dangerous: "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." – Bjarne Stroustrup
- "Expert friendly"

Octopus vs Swiss Army knife

"C++ is an octopus made by nailing extra legs onto a dog" – Steve Taylor



But you can cut off some extra legs to get the right dog for your program!

The philosophy of C++

- General purpose
- Flexible by allowing developers to build abstractions (and provides a large number through the library)
- Performance and efficiency always targeted "You only pay for what you use"
- Use the powerful type system to express intent.

These few lectures

We could spend all semester going in depth on C++, but we've got three lectures plus a "drop in".
So I've picked a handful of features to cover today that you really need to write C++ for HPC:

- References and memory
- Overloading
- Classes
- Templates

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

```
$ g++ --std=c++11 hello.cpp -o hello
$ ./hello
Hello, world!
```

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

Include the `iostream` standard library header which gives us a way to communicate with the file system

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

`std` is the standard library namespace. A namespace allows scoping of names (much like a filesystem has directories).

The scope resolution operator `::` lets us access a declaration from inside a namespace.

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

cout represents console output (i.e. stdout)

Hello!

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello, world!" << std::endl;
}
```

The standard library uses the bitwise left shift operator (<<) to mean stream insertion - i.e. output the right hand side to the left.

(Similarly, the right shift (>>) operator is used for extraction, i.e. input.)

Don't use malloc again

While you can use `malloc` and `free` in C++, you should not. Instead, if you need to directly allocate memory use `new` and `delete`.

```
int* x = nullptr;
x = new int;
*x = 42;
std::cout << "The␣answer␣is␣"<< *x <<std::endl;
// The answer is 42
delete x;
```

Slightly different for arrays

```
int* squares = nullptr;
squares = new int[5];
for (auto i=0; i<5; ++i)
  squares[i] = i*i;

// Do something
delete[] squares;
```

Note the square brackets!

References

As well as pointers and values, C++ has the concept of a reference. They are like pointers in that they don't copy the thing-that-is-referred-to, but syntactically they are just like a value.

```
double pi = 3.14;
double& pr = pi;
std::cout << pr << std::endl;

void iLoveIntegers(double& x) {
  x = 3;
}
iLoveIntegers(pi);
\\ prints 3
std::cout << pi << std::endl;
```

References: whys and whens

References have simpler syntax.
References are safer than pointers: a reference cannot be null and cannot be reseated (must be bound when it's defined)

```
double twopi = 6.24;
pr = twopi; \\ Error will not compile
```

And you can rely on it being valid, unless there is an evil coder around:

```
char& get() {
  char x = '!';
  return x;
  // Many compilers will warn about this
}
```

Use a reference by default, until you need a pointer. (E.g. need to reseat or interface with C.)

Function overloading

You can have multiple functions with the same name but different arguments.

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

When you call `sum`, the compiler knows the types of the arguments and will try to find the best match from all the candidates with the name.

The compiler will also try to use any built-in or user-defined conversion rules.

What happens here?

```
int i1 = 1;
int i2 = 2;
double d1 = 1.0;
double d2 = 2.0;
unsigned u42 = 42;

std::cout << sum(i1, i2) << std::endl;
std::cout << sum(3, 72) << std::endl;
std::cout << sum(i1, u42) << std::endl;
std::cout << sum(d2, d1) << std::endl;
std::cout << sum(d2, i1) << std::endl;
std::cout << sum(d2, 1.0f) << std::endl;
```

Operators are functions

C++ operators, for the non-built in types, are just functions with odd names, e.g.:

```
Vector operator+(const Vector& a,
                 const Vector& b);
```

You can then use the natural syntax when manipulating these in other code:

```
Vector c = a + b;
```

We'll come back to this.

Objects in C++

You define a class just like a `struct`, but using the keyword `class` instead.[1]
As well as data members, instantiations of classes (i.e. objects) can have member functions.

```
class Complex {
public:
  Complex();
  Complex(float re, float im);
  Complex(const Complex&) = default;
  float magnitude() const;
private:
  float real;
  float imag;
};
```

[1] Technically, the only difference is the default visibility of declarations.

Creating complexes

When you create an instance of your class, you usually need to provide a constructor. Note we provide two overloads - a default one needing no arguments and one that initialises with a value. We also tell the compiler to create a default copy constructor for us.

```
Complex::Complex() :
  real(0), imag(0) {
}
Complex::Complex(float re, float im) :
  real(re), imag(im) {
}
// This is roughly what the compiler will create
// Complex::Complex(const Complex& c) :
//   real(c.real), imag(c.imag) {
// }
```

Creating complexes

We can now create these numbers

```
Complex zero;
Complex imaginary_unit(0, 1);
Complex copy_of_i(imaginary_unit);
```

What about actually doing something?

```
float Complex::magnitude() const {
  return sqrt(real*real + imag*imag);
}
```

- The `const` at the end of the declaration says that this function will not alter the instance it is working on. Add this whenever possible!
- We can access the members by just giving their names.
- The instance that the function is working on is also available as a pointer, called `this`, so could rewrite as:

```
float Complex::magnitude() const {
  return this->real*this->real +
           this->imag*this->imag;
}
```

Complex numbers have the usual arithmetic operations: $(+ - \times \div)$
We have to provide operator overloads, like

```
Complex operator+(const Complex& a,
                  const Complex& b) {
  return Complex{a.real+b.real, a.imag+b.imag};
}
```

This is just a function (with an unusual name) that takes two complex numbers and returns one.
To let this function touch the private members of the class, we must declare that it is a `friend`:

```
class Complex {
  // ...
  friend Complex operator+(const Complex&, const
};
```

Templates 101

Templates are a method of doing metaprogramming: a program that writes a program.

An easy example:

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

What if we need this for `float` and `unsigned`?
Going to get boring and hard to maintain quickly!

Templates 101

Templates are a method of doing metaprogramming: a program that writes a program.

An easy example:

```
template<class T>
T sum(T a, T b) {
  return a+b;
}
```

When you use it later, the compiler will substitute the types you supply for T and then try to compile the template.

```
cout <<"add␣unsigned=" << sum(1U, 4U) << endl;
cout <<"add␣floats=" << sum(1.0f, 4e-2f) << endl;
```

Template classes

You can define a template class - i.e. a template that will produce a class when you instatiate it.
Let's build something useful, like a simple array class.

```
template<class T>
class Array {
  unsigned _size;
  T* _data;
public:
  Array();
  Array(unsigned n);
  ~Array();
  unsigned size() const;
  const T& operator[](unsigned i) const;
  T& operator[](unsigned i);
};
```

Where to put your implementation?

Templates are not executable code - they tell the compiler how to create it. So the definition must be available to the user of your template - i.e. typically in a header file.
You can define the functions in place like:

```
template<class T>
class Array {
public:
  Array() : _size(0), _data(nullptr) {}
};
```

Or at the end of the header (or equivalently in another file that you include at the end of your header)

```
template<class T>
Array<T>::Array(unsigned n) : _size(n) {
  _data = new T[n];
}
```

How to release that memory?

We have acquired some memory in the constructor and at the moment we will leak this.
Typically a class's destructor will do this.
The name of this function is `~Array`

```
template<class T>
Array<T>::~Array() {
  delete[] _data;
}
```

It's important to note that you should never call this directly - the compiler will call it for you when your `Array` objects:

- go out of scope (i.e. local variables)
- are `deleted`
- belong to another object and that object is destructed

RAII / CADRe

A very important pattern in C++ is RAII: resource allocation is instantiation. Also known as constructor acquires, destructor releases.
This odd name is trying to communicate that any resource you have (heap memory in this case) should be tied to the lifetime of an object. So the when the compiler destroys your object it will release the resource (e.g. memory).

```
void do_simulation(Parameters& p) {
  Array<float> work_array(p.problem_size);
  initial_condition(work_array);
  for(int i=0; i<p.timesteps; ++i) {
    do_timestep(work_array);
  }
  write_output(p.outfile, work_array);
}
```

Copying

One thing we've not discussed about this array is what do we want to do about copying?

- We could create a shallow copy, using a simple pointer copy, but then which instance owns the data?
- We could instead do a deep copy of the data each time, but that might be expensive. Maybe we want to disallow implicit copying but allow a user to explicitly copy with a special `copy()` method?

These design decisions should be considered!

Returning an Array

What happens if we compute an array in a function and return it?

```
Array<int> load(const string& fn) {
  auto n = getsize(fn);
  Array<int> ans(n);
  for (auto i=0; i<n; ++i)
    ans[i] = read(fn, i);
  return ans;
}

void user() {
  Array<int> data = load(fn);
}
```

Returning an Array

People assume this is what happens:

```
Array<int> load(const string& fn) {
  auto n = getsize(fn);
  Array<int> ans(n); // allocate
  for (auto i=0; i<n; ++i)
    ans[i] = read(fn, i);
  return ans; // copy to a temporary
}

void user() {
  // Call default c'tor
  Array<int> data = load(fn); // copy tmp->data
                              // destroy tmp
}
```

Returning an Array

Alternative: return a pointer

```
Array<int>* load(const string& fn) {
  auto n = getsize(fn);
  // OMG - new!
  auto ans = new Array<int>(n); //allocate twice
  for (auto i=0; i<n; ++i)
    // UGLY!
    (*ans)[i] = read(fn, i);
  return ans;
}

void user() {
  auto data = load(fn);
 // Better remember to delete this!
}
```

Returning an Array

Alternative: pass an output argument by reference

```
void load(const string& fn, Array<int>& ans) {
  auto n = getsize(fn);
  ans.resize(n); // probably allocate
  for (auto i=0; i<n; ++i)
    ans[i] = read(fn, i);
}

void user() {
  // Have to declare outside factory function :(
  Array<int> data;
  load(fn, data); // No copy :)
  // Regressing to assembly :(
}
```

Returning an Array

Alternative: don't copy - move

Need to define a new move constructor and assignment

```
template<class T>
class Array {
public:
  Array(Array&& other) :
    _size(other._size), _data(other._data) {
    other._data = nullptr;
    other._size = 0;
  }
};
```

The double ampersand indicates an "r-value reference".

The compiler will only use this when the argument is a temporary value that is going to be destructed - you can safely steal its resources.

Returning an Array

Alternative: don't copy - move

```
Array<int> load(const string& fn) {
  auto n = getsize(fn);
  Array<int> ans(n); // Construct
  for (auto i=0; i<n; ++i)
    ans[i] = read(fn, i);
  return ans; // This moves to the temporary
}

void user() {
  Array<int> data = load(fn); // temporary moved
                              // temporary destru
}
```

Returning an Array

Compilers are allowed to do copy elision (even when this may have side effects!) to directly construct the return value in its destination

```
// Compiler adds secret arg
Array<int> load(const string& fn) {
  auto n = getsize(fn);
  Array<int> ans(n); // Alias ans to secret
                     // Construct
  for (auto i=0; i<n; ++i)
    ans[i] = read(fn, i);
  return ans; // No-op
}

void user() {
  // Create empty space for data
```