CAAM 519: Computational Science I

Module 7: C++

The C++ programming language

C++ was born as an extension of the C language, initially developed by Bjarne Stroustrup at AT&T Bell Labs starting in 1979. It was initially called "C with Classes", which gives you some idea of its genesis and original vision. In particular, C++ introduces an *object oriented* programming paradigm to C.

Object-oriented programming revolves around "objects", which are like structs (i.e. they hold data in fields), but additionally can contain code (typically called methods). These methods are typically used to access or modify the data stored in the object, meaning that methods are typically how objects interact with the rest of the program. In C++, objects are instances of *classes*, which are embedded inside the type system.

C++ has ballooned beyond this in the intervening years, and now has functional programming, virtual functions, operator overloading, references, templates, exceptions, namespaces, and on and on. But at its core, C++ is (almost, modulo rare corner cases) entirely backwards compatible with C, with new features bolted on.

The very basics

C++ programs are structured similarly to C files. Typically, source files will use the `.cc` or `.cpp` file extensions, while header files will use `.hpp` or just `.h`. Header files will contain function declarations and struct/class definitions (with suitable header guards), and the source files will contain implementations of functions.

To compile a C++ program, you need a C++ compiler. We will use g++, which you can install in your Virtual Box with:

```
> sudo apt install g++
```

Here's a first C++ program for illustrative purposes:

```
#include <cstdio>
#include <iostream>

```

```
int main(void) {
    printf("In C++, you can use printf like in C...\n");
    std::cout << "...or use the new streaming operator syntax." <<
std::endl;
}
```

This example allows to observe a few differences from C.

1. You can omit the file extension for header files when including them.
2. We observe a new streaming operator, `<<`, which is useful in this case for printing. In this code, `std::cout` is an object representing (a stream buffer to) STDOUT, while `std::endl` inserts a newline character and flushes the stream to its target.
3. Finally, we see the syntax for namespacing in C++. Here, `std` is a namespace introduced in the iostream header, and `cout` and `endl` are both objects introduced inside of that namespace. To access them, C++ uses the `::` syntax.

References

C++ introduces a new type of variable called a *reference* for working with data that you do not want to copy when you, for example, pass it as an argument to a function. Instead of a pointer, which makes explicit that you are working with the address where a variable is stored in memory, references create an alias to a variable, meaning that all the code you write working with the reference will look identical to that as if you were working with the original variable. For example,

```
#include <iostream>

int main(void) {
    int i;
    int* ptr_i = &i;
    int& ref_i = i;
    i = 5;
    *ptr_i = 6;
    ref_i = 7;
    std::cout << "i = " << i << std::endl;
}
```

So the syntax is working with references is much cleaner, and they have the same no-copy behavior which is desirable in so many situations. However, references, unlike pointers, must be initialized, and cannot change (i.e. made to point to a different memory address) once initialized.

Since references look so much like regular variables (rather than aliases to memory owned elsewhere), the const annotations are extremely helpful for making sure that your code does not accidentally change data "at a distance." For example, the following is arguably nicer from a syntatic perspective than working with pointers, and does not run the risk of the called function changing the data passed to it as argument:

```
#include <iostream>

typedef struct {
    int n;
} MyStruct;

MyStruct doubleInput(const MyStruct& input) {
    // This will crash at compilation:
    // input.n = 12;
    MyStruct doubled;
    doubled.n = 2 * input.n;
    return doubled;
}

int main(void) {
    MyStruct instance = {12};
    MyStruct doubled = doubleInput(instance);
    std::cout << "doubled value is " << doubled.n << std::endl;
}
```

Overloading

Unlike C, C++ allows you to define multiple functions with the same name. Think of this a bit like defining multiple methods in Julia. So the following is valid C++ code:

```
int fn(int x, double y);
int fn(int x, int y);
double fn(double x, int y);

int main(void) {
    return fn(1, fn(1.0, fn(1, 2)));
}
```

But wait--if overloading is supported, how does the symbol table generated by the compiler look? Remember in C we could very easily recognize the functions we defined in the symbol table because the names were unchanged. In C++, the compiler performs name mangling:

```
> g++ -o overloading.cc
> nm overloading.o

nm overload.o
                 U __Z2fndi   fndi —> fn(double int)
                 U __Z2fnid   fnid —> fn(int double)
                 U __Z2fnii     fnii —> fn(int int)
0000000000000000 T _main
```

Take a moment to see if you can decipher the name mangling scheme. As you might intuit, this name mangling can make it difficult to compile projects that contain both C and C++ object files. Fortunately, C++ has a mechanism for exposing "unmangled" names:

```
int foo(int x) {
    return 2 * x;
}

extern "C" {
int bar(int x) {
    return 2 * x;
}
}
```

```
> g++ -c unmangled.cc
> nm unmangled.o
nm unmangled.o
0000000000000000 T __Z3fooi
0000000000000020 T _bar
```

Classes

Classes in C++ are structs (i.e. data in fields) with functions (typically called *methods*) also attached to them. Typically, these methods will do something with the data stored in the class instance: i.e. allow external code to access it or change it.

Defining new classes is similar to C code we've seen before to define structs:

```
class Vector {
 public:
  int length_;
  double* vals_;
};

int main(void) {
  Vector vec;
  vec.length_ = 3;
}
```

The big difference here is the appearance of the public annotation. In C++, member fields can be have three levels of access permissions:

- public: External code can access or set (e.g. via `vector.length = 10`).
- private: Only member methods can access or set data (i.e. no external code). The default.
- protected: The only external code that touch fields are subclasses (or things explicitly labeled "friend"s).

So, if you have a class for which you label all fields as private, you can completely control external access to your data via class methods.

Remember, the main distinction between classes and structs is that a class has member functions. It might look a little like this:

```
#include <iostream>

class Container {
 public:
  void SetValue(int x);
  int GetValue(void);

 private:
  int value_;
};

void Container::SetValue(int x) {
  value_ = x;
}

int Container::GetValue(void) {
  return value_;
}

int main(void) {
  Container x;
  x.SetValue(12);
  std::cout << "Value is " << x.GetValue() << std::endl;
}
```

Here the public interface is two methods, which get and set the value stored inside the class. The actual member field itself is private, so can only be used via these methods. The class definition body contains method declarations (in the public block), and the actual definitions appear outside the class body. To make clear that the functions are class methods, the `Container::member_method` syntax is used. Note that the fields are available inside the method bodies with no special annotations; this motivates the trailing underscores in the field names, to make them visually distinct from other local variables.

Finally, note the syntax for calling member methods is the dot, just like the syntax for accessing fields.

Constructors and destructors

You can write methods that are called whenever an object is created (constructor) or goes out of scope and gets destroyed (destructor). For example, you can use these to safely allocate and free memory stored inside a class.

destructors are automatically run when the class goes out of scope i.e. the block of code which calls the class is exited

```
#include <cstdlib>
#include <iostream>

class MyClass {
 public:
  // Constructor
  MyClass() {
    ptr_ = (int*) malloc(sizeof(int));
  }

  // Overloaded constructor with other argument signatures
  explicit MyClass(int value) {
    ptr_ = (int*) malloc(sizeof(int));
    *ptr_ = value;
  }
  // Destructor
  ~MyClass() {
    std::cout << "Freeing MyClass!" << std::endl;
    free(ptr_);
    ptr_ = nullptr;
  }

  int GetValue(void) const {
    return *ptr_;
  }

 private:
  int* ptr_;
};

int main(void) {
  MyClass x;
  std::cout << "x value = " << x.GetValue() << std::endl;

  MyClass y(12);
  std::cout << "y value = " << y.GetValue() << std::endl;

  // Interesting case: The following is a compilation warning:
  // MyClass z();
}
```

Notice here that our old friend `NULL` from C has been replaced by `nullptr`. The first was a macro that the C preprocessor expanded to some value; the second is an actual keyword in the language.

Notice also the explicit keyword that appeared in our new constructor with a single argument. This is a command to the compiler that this method must be called explicitly in code; it cannot be called as the result of an implicit conversion.

Finally, the `const` appearing in `GetValue` indicates that the class method does not alter the associated `MyClass` instance.

"The rule of three"

Let's modify our example code above ever so slightly. First, to make things a bit cleaner, let's move our class to its own header and source files. In `MyClass.h`:

```
#ifndef _MY_CLASS_H
#define _MY_CLASS_H
class MyClass {
 public:
  // Constructors
  MyClass();
  explicit MyClass(int value);

  // Destructor
  ~MyClass();

  int GetValue(void) const;

 private:
  int* ptr_;
};
#endif
```

And in `MyClass.cc`:

```
#include "MyClass.h"
#include <cstdlib>
#include <iostream>

MyClass::MyClass() {
  ptr_ = (int*) malloc(sizeof(int));
 }

// Overloaded constructor with other argument signatures
MyClass::MyClass(int value) {
  ptr_ = (int*) malloc(sizeof(int));
  *ptr_ = value;
}

MyClass::~MyClass() {
  std::cout << "Freeing MyClass!" << std::endl;
  free(ptr_);
  ptr_ = nullptr;
}

int MyClass::GetValue(void) const {
  return *ptr_;
}
```

And finally in `driver.cc`:

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

int main(void) {
  MyClass x;
  std::cout << "x value = " << x.GetValue() << std::endl;

  MyClass y(12);
  std::cout << "y value = " << y.GetValue() << std::endl;

  MyClass z = y;
  std::cout << "z value = " << z.GetValue() << std::endl;
}
```

Compiling and running this gives:

```
> g++ MyClass.cc driver.cc -o myclass
> ./myclass
x value = 0
y value = 12
z value = 12
Freeing MyClass!
Freeing MyClass!
myclass(16710,0x1156415c0) malloc: *** error for object 0x7fc572c02a80:
pointer being freed was not allocated
myclass(16710,0x1156415c0) malloc: *** set a breakpoint in
malloc_error_break to debug
[1]    16710 abort      ./myclass
```

What happened here? When I define and initialize z, the compiler will call the default copy constructor for MyClass, attempting to copy the value y to the new value z. When it does this, it just copies the fields in a "shallow" manner, so it copies the pointer address value directly between the two. But then when the destructor for z is called after the destructor to y was called, the memory stored at that address has already been freed!

To fix this, we should add a new copy constructor method. It could look something like this:

```
MyClass::MyClass(const MyClass& copy_from) {
  ptr_ = (int*) malloc(sizeof(int));
  *ptr_ = copy_from.GetValue();
}
```

Remember we are constructing a new instance here, so we will have to allocate the requisite memory on the heap.

So this fixes that problem. But what about the following modification to `driver.cc`:

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

int main(void) {
  MyClass x;
  std::cout << "x value = " << x.GetValue() << std::endl;

  MyClass y(12);
  std::cout << "y value = " << y.GetValue() << std::endl;

  MyClass z = y;
  std::cout << "z value = " << z.GetValue() << std::endl;

  z = x;
  std::cout << "z value = " << z.GetValue() << std::endl;
}
```

```
> g++ MyClass.cc driver.cc -o myclass
> ./myclass
x value = 0
y value = 12
z value = 12
z value = 0
Freeing MyClass!
Freeing MyClass!
Freeing MyClass!
myclass(19111,0x105d285c0) malloc: *** error for object 0x7fb7a4402a70:
pointer being freed was not allocated
myclass(19111,0x105d285c0) malloc: *** set a breakpoint in
malloc_error_break to debug
```

```
[1]    19111 abort      ./myclass
```

What's going on here? Well, the compiler is calling the default *copy assignment operator*, which is again performing a shallow copy of the fields, meaning that the destructor for z is trying to free the same slot of memory already freed by the destructor of x.

How to fix this one? By adding a new copy assignment operator method:

```
MyClass& MyClass::operator=(const MyClass& copy_from) {
  *ptr_ = copy_from.GetValue();
  return *this;
}
```

We're copying one pre-existing instance into another here, so no need to initialize the memory like in the constructor.

Here we see the `this` keyword for the first time. This is a special pointer, available within a class method, which points to the address where the current instance is stored in memory. Here we only use it in a conventional manner, but there will be other occasions where you will actually need this to make your code run correctly.

These past two examples have been an illustration of the *rule of threes*. This rule of thumb states that if you implement a non-default version of a destructor, copy constructor, or copy assignment operator, you should implement all three. Otherwise, you run the risk of suffering bizarre crashes like we saw above.

Inheritance

Like Julia, C++ has a type system which supports inheritance, or defining behavior in a hierarchical manner. In Julia, this was through subtyping abstract classes. In C++, the behavior is more complex.

Derived classes

In C++, inheritance can occur between two classes which each have distinct member fields and methods. One class (`derived_class`) can be derived from another (`base_class`) using the following syntax:

```
class derived_class : type_of_inheritance base_class {
  // class definition
};
```

Here, `type_of_inheritance` can be public, private (the default), or protected. This access descriptor indicates how "closed off" the base_class members should be in the derived class. According to this permission level, derived_class inherits all public and protected variables and methods from base_class besides a) constructors, b) destructors, c) assignment operators, and 4) friend methods/operators.

Moreover, C++ has multiple inheritance!

```
class derived_class : type_of_inheritance_1 base_class_1,
type_of_inheritance_2 base_class_2 {
  // class definition
};
```

For a more concrete example, let's consider a simple 2D shape class, and its inheritors, first in `shape2d.h` put:

```
class Shape2D {
 public:
  Shape2D(void) {}
  Shape2D(double w, double h) : width_(w), height_(h) {}
  void SetWidth(double w) { width_ = w; }
  void SetHeight(double h) { height_ = h; }
  double GetWidth(void) const { return width_; }
  double GetHeight(void) const {return height_; }

 protected:
  double width_ = 0.0;
  double height_ = 0.0;
};

class Rectangle : public Shape2D {
 public:
  double GetArea(void) { return width_ * height_; }
};
```

no constructor initialized so the defualt constructor must be used

```
class Triangle : public Shape2D {
 public:
  Triangle(double w, double h) : internal_field_(12), Shape2D(w, h) {}
  double GetArea(void) { return 0.5 * GetWidth() * GetHeight(); }
 private:
   int internal_field_;
};
```

Then in `driver.cc` put:

```
#include <iostream>
#include "shape2d.h"

int main(void) {
  // Can create an instance of Shape2D
  Shape2D shape;
  shape.SetWidth(3.0);
  shape.SetHeight(1.0);
  // No way to compute area...

  // How to create a Rectangle instance? The following doesn't work...
  // Rectangle rect(3.0, 1.0);
  // Need to do:
  Rectangle rect;
  rect.SetWidth(3.0);
  rect.SetHeight(1.0);
  std::cout << "rectangle width is " << rect.GetWidth() << std::endl;
  std::cout << "rectangle area is " << rect.GetArea() << std::endl;

  Triangle tri(3.0, 1.0);
  std::cout << "triangle width is " << tri.GetWidth() << std::endl;
  std::cout << "triangle area is " << tri.GetArea() << std::endl;
}
```

A few things of note here.

- We have two constructors for `Shape2D`. The first doesn't do anything. For the second, we see new syntax. Immediately after the constructor syntax is a colon. After this colon, you may initialize some or all of the fields of the class using the field types constructors. After this is a regular block of code contained in curly brackets, where you can put more complicated initialization instructions.
- Starting in C++ 11, you can specify default initialization values for member fields. We see this in the field annotations for `Shape2D`.
- Since the fields of `Shape2D` are protected, they are accessible to any derived classes (e.g. `Rectangle` and `Triangle`).

- As such, when computing the area you can either use the fields directly (like `Rectangle` does) or use the getter functions (like `Triangle` does).
- We added a custom constructor to `Triangle`; otherwise, it's not possible to create and initialize an instance in one go. We use the new field initialization syntax. Notably, you can use the same syntax to initialize any base classes (see the call to the `Shape2D` constructor).

Virtual functions and inheritance

The example above does a good job illustrating the utility of polymorphism, or the use of a single interface for multiple instances of different types. We've already seem polymorphism in action with function overloading (single function name as interface, different types of arguments). We'll see another, templating, in a bit (we saw similar concepts in Julia). A third type of polymorphism in C++ is subtyping, which we can illustrate with a modification of `driver.cc` from above:

```cpp
#include "shape2d.h"
#include <iostream>

int main(void) {
  Shape2D shape;
  shape.SetWidth(3.0);
  shape.SetHeight(1.0);

  Rectangle rect;
  rect.SetWidth(3.0);
  rect.SetHeight(2.0);

  Triangle tri(3.0, 1.0);

  Shape2D* ptr_shape;
  ptr_shape = &rect;
  std::cout << "First height is " << ptr_shape->GetHeight() << std::endl;
  ptr_shape = &tri;
  std::cout << "Second height is " << ptr_shape->GetHeight() << std::endl;
}
```

Note that we created a pointer to a `Shape2D`, but then the compiler allowed us to use that to point to instances of its derived classes. We can call the `GetHeight` method, since this is a member method for `Shape2D`, so the compiler knows that the method will be available. But what if we want to call `GetArea` instead? That seems like it should be possible, since both `Rectangle` and `Triangle` have the correct method implemented. However, `Shape2D` does not, since it's not possible to provide a meaningful answer for an arbitrary shape.

What we want is some way to stipulate that the `GetArea` method is part of the `Shape2D` interface. We don't want to provide a default implementation, but we would like to enforce that each derived class provides an implementation. This is where virtual functions come in handy.

Virtual functions allow the compiler to dynamically (i.e. at runtime) determine which code to run. You can provide an implementation in the base class, or leave one empty if you a default behavior doesn't make sense. If you want to force derived classes to add an implementation of the virtual function, you can use the (kind of ugly) `= 0` syntax in the virtual function declaration. This is a pure virtual function.

We can modify `shape2d.h` to read

```
class Shape2D {
 public:
  Shape2D(void) {}
  Shape2D(double w, double h) : width_(w), height_(h) {}
  void SetWidth(double w) { width_ = w; }
  void SetHeight(double h) { height_ = h; }
  double GetWidth(void) const { return width_; }
  double GetHeight(void) const {return height_; }
  virtual double GetArea(void) = 0;

 protected:
  double width_ = 0.0;
  double height_ = 0.0;
};

class Rectangle : public Shape2D {
 public:
  double GetArea(void) { return width_ * height_; }
};

class Triangle : public Shape2D {
 public:
  Triangle(double w, double h) : Shape2D(w, h) {}
  double GetArea(void) { return 0.5 * GetWidth() * GetHeight(); }
};
```

Here we stipulate that `GetArea` is a virtual function, and that any derived classes must implement this exact method.

Virtual functions are the way to implement abstract classes a la Julia. They require an implementation of the virtual function to compile correctly, and no default is provided by the abstract class; hence, it must be subtyped to be used. In other words, the previous version of the driver.cc code will fail to compile, since Shape is now an abstract class and thus cannot be instantiated:

```
#include "shape2d.h"
#include <iostream>

int main(void) {
  // Can no longer instantiate a Shape, since it is an abstract class
  // Shape2D shape;

  // Instantiating a Rectangle is still fine, though, since GetArea
  // is implemented
  Rectangle rect;
  rect.SetWidth(3.0);
  rect.SetHeight(2.0);
  // Same for Triangle
  Triangle tri(3.0, 1.0);

  Shape2D* ptr_shape;
  ptr_shape = &rect;
  std::cout << "First height is " << ptr_shape->GetHeight() << std::endl;
  ptr_shape = &tri;
  std::cout << "Second height is " << ptr_shape->GetHeight() << std::endl;
}
```

Here's another interesting example. Consider the following code:

```
#include <iostream>

class A {
 public:
  double GetValue(void) { return 1; }
};

class B : public A {
  double GetValue(void) { return 2; }
};

int main(void) {
  A* ptr_A;
  B b;
  ptr_A = &b;
  std::cout << "GetValue() = " << ptr_A->GetValue() << std::endl;
}
```

What value gets printed?

```
> g++ teaser.cc
> ./a.out
GetValue() = 1
```

How could we get the opposite behavior? Be declaring the `GetValue` method in class `A` virtual.

Templated functions and classes

Templates are permitted in C++ that allow another form of polymorphism by using the same code to describe the behavior for different types. Here's a simple example:

```
#include <iostream>

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

int main(void) {
  int int_val = add<int>(2, 3);
  std::cout << "2 + 3 = " << int_val << std::endl;

  double dbl_val = add<double>(2.1, 3.0);
  std::cout << "2.1 + 3.0 = " << dbl_val << std::endl;
}
```

You specify the template placeholder using `template <typename T>`; here, we use the placeholder `T` in the return type, and twice in the argument list.

Templates can also appear in class definitions:

```
#include <iostream>

template <typename T> class Wrapper {
 public:
  T GetValue(void) { return value_; };
  void SetValue(T x) { value_ = x; }

 private:
  T value_;
};

int main(void) {
  Wrapper<int> wrapper;
  wrapper.SetValue(3);
  std::cout << "Value is " << wrapper.GetValue() << std::endl;
}
```

Question: can you define the templated method `GetValue` outside of the class definition block?

Answer: yes, though you may run into compilation errors if you include it in another source file. The implementation of `GetValue` would look like

```
template <typename T> T Wrapper<T>::GetValue(void) { return value_; }
```

Templates don't have to be types! You can plug in other values as templates as well. The caveat here is that the type parameter has to be either a literal, or a const variable. An example:

```
#include <iostream>

template <typename T, int N> T add_N(T a) {
  return a + N;
}

int main(void) {
  // Use a literal type parameter
  double d = add_N<double, 5>(3.1);
  std::cout << "d is " << d << std::endl;
  const int n = 5;
  int i = add_N<int, n>(3);
  std::cout << "i is " << i << std::endl;
}
```

Templates can also take default arguments:

```
#include <iostream>

template <typename T, int N=0> T add_N(T a) {
  return a + N;
}

int main(void) {
  // Use a literal type parameter
  double d = add_N<double>(3.1);
  std::cout << "d is " << d << std::endl;
}
```

Forward declarations

Here's a brain teaser: How do you make the following circular dependencies work:

In `ClassA.h`:

```
#include "ClassB.h"
class A {
 public:
  B b;
};
```

In `ClassB.h`:

```
#include "ClassB.h"
class B {
 public:
  A a;
};
```

In `driver.cc`:

```
#include "ClassA.h"
#include "ClassB.h"

int main(void) {
  A a;
  B b;
  B.a = a;
  A.b = b;
}
```

Three things should change here.

1. Use header guards to avoid multiple class definitions.
2. Use a forward declaration of the classes so that you can use the identifier `A` in the definition of `B`, without requiring `A` to be defined first (and vice versa).
3. Use pointers in the fields.

In `ClassA.h`:

```
#ifndef _CLASS_A_H
#define _CLASS_A_H
class B;
class A {
 public:
  B* ptr_b;
};
#endif
```

In `ClassB.h`:

```
#ifndef _CLASS_B_H
#define _CLASS_B_H
class A;
class B {
 public:
  A* ptr_a;
};
#endif
```

In `driver.cc`:

```
#include "ClassA.h"
#include "ClassB.h"

int main(void) {
  A a;
  B b;
  a.ptr_b = &b;
  b.ptr_a = &a;
}
```

Note that, with these forward declarations, we don't even have to include the header for `ClassA` in the header for `ClassB`, and vice versa.

Dynamic memory in C++

In C, we managed memory dynamically on the heap using the standard library functions `malloc` and friends. This memory management scheme is still available in C++. However, the language also offers you alternative interface using new language keywords. Some advantages of the new approach:

- You can allocate and initialize single values, if desired.
- It will call the appropriate constructors or destructors for whatever objects you allocate on the heap.

To allocate a single object on the heap, do

```
T* ptr = new T;
```

where `T` is some type (e.g. int). This is analogous to `malloc`.

To free the memory once done, do

```
delete ptr;
```

You can also initialize the single value directly upon definition:

```
int* ptr_int = new int(4);
double* ptr_dbl = new double(-3.2);
std::cout << "int value is " << *ptr_int << std::endl;
std::cout << "double value is " << *ptr_dbl << std::endl;
delete ptr_int;
delete ptr_dbl;
```

As mentioned above, the new keyword will use whatever constructor methods are available.

```
#include <iostream>

class A {
 public:
  A(void) : value_(12) {}
  A(int x) : value_(x+1) {}
  int value_;
};

int main(void) {
  A* ptr_a = new A;
  std::cout << "First value is " << ptr_a->value_ << std::endl;
  A* ptr_b = new A(3);
  std::cout << "Second value is " << ptr_b->value_ << std::endl;
}
```

To allocate a block of memory for multiple objects of some type, use the square bracket syntax:

```
int* ptr = new int[4]; \\ array of 4 integers
delete[] ptr;
```

Note that delete also has the square brackets appear; don't mix and match the keywords with and without brackets!

What about multidimensional arrays? Here are two ways to allocate a matrix, first discontiguously in memory:

```
const int M = 5;
const int N = 4;

double** A = new double*[M];
for (int i = 0; i < M; ++i) {
  A[i] = new double[N];
}
for (int i = 0; i < M; ++i) {
  delete[] A[i];
}
delete[] A;
```

and then contiguously in memory:

```
const int M = 5;
const int N = 4;

double** A = new double*[M];
A[0] = new double(M * N);
for (int i = 1; i < M; ++i) {
  A[i] = A[0] + i * N;
}
delete[] A[0];
delete[] A;
```

Exceptions

Exceptions are another new C++ features that allows you to handle errors in a sophisticated manner. You can "throw" an exception when a problem occurs, but also "catch" potential errors and attempt to repair them (or crash gracefully).

To throw an exception, use the keyword:

```
throw "This is my exception;
throw 12;
```

You can bundle up blocks of code in "try/catch" statements to "try" out code, and "catch" exceptions that might be raised while executing it:

```
#include <iostream>

int main(void) {
  try {
    int i = -1;
    if (i < 0) {
        throw i;
    }
  } catch(int e) {
    std::cout << "Exception encountered: " << e << std::endl;
    return 1;
  }
  std::cout << "Should not be reached." << std::endl;
  return 0;
}
```

You can have multiple catch blocks for different types of exceptions, and those blocks don't have to terminate the program if you don't want:

```
#include <iostream>

int main(void) {
  try {
    int i = -1;
    if (i < 0) {
        throw -1.0;
    }
  } catch(int e) {
    std::cout << "Exception encountered: " << e << std::endl;
    return 1;
  } catch(double e) {
    std::cout << "Exception encountered: " << e << std::endl;
    std::cout << "Not fatal; will continue." << std::endl;
  }
  std::cout << "Now reachable" << std::endl;
  return 0;
}
```

As the code might suggest, there are a host of exception types built into the language that categorize different types of errors (e.g. memory allocation failure, invalid arguments, and so on).

If an exception is not caught by a "catch" block, it will terminate the program by default:

```
#include <iostream>

int main(void) {
  throw 12;
}
```

```
> g++ exceptions.cc
> ./a.out
libc++abi.dylib: terminating with uncaught exception of type int
[1]    93577 abort      ./a.out
```

The standard template library

C++ comes with a number of handy code baked in, in the *standard template library* (STL). It contains a host of data structures, utility functions, algorithms, iterators, and so on that might be useful for your code. I will focus only on a few items in the STL that we will need for the homework; know that this is a very small sampling of what is at your disposal.

std::vector

STL includes a handy built-in vector class. They can be dynamically resized, and all the memory is handled for you.

```
#include <vector>
#include <iostream>

int main(void) {
  std::vector<double> vec; // = {}
  std::vector<double> vec_2(3, -4.0); // = {-4.0, -4.0, -4.0}
  std::vector<double> vec_3(vec_2); // = a copy of {-4.0, -4.0, -4.0}
  std::cout << "vec.size() = " << vec.size() << std::endl;
  std::cout << "vec_2[2] = " << vec_2[2] << std::endl;
  vec_2[1] = 1.3;
  std::cout << "vec_2[1] = " << vec_2[1] << std::endl;
  double& elem = vec_2.at(2);
  elem = 3.1;
  std::cout << "vec_2[2] = " << vec_2[2] << std::endl;
  vec.push_back(3.2);
  std::cout << "vec[0] = " << vec[0] << std::endl;
  vec.clear();
  std::cout << "vec.size() = " << vec.size() << std::endl;
  vec_2.pop_back();
  std::cout << "vec_2.back() = " << vec_2.back() << std::endl;
}
```

std::pair

This templated class stores pairs of elements in an ordered fashion:

```
#include <iostream>

int main(void) {
  std::pair<int,double> p1(3, 4.2);
  std::cout << "p1 = (" << p1.first << ", " << p1.second << ")" <<
std::endl;
  auto p2 = std::make_pair<double, bool>(3.1, false);
  std::cout << "p2 = (" << p2.first << ", " << p2.second << ")" <<
std::endl;
}
```

Two other new things of note here: first, the built in bool class. Second, the auto keyword, introduced in C++ 11, which instructs the compiler to "fill in" a complicated type it can otherwise infer (e.g. from the return type of a function). It's useful for avoiding typing out complicated type names (like above), but use sparingly!

std::numeric_limits

The numeric limits library has a bunch of handy functions for working with the limits of what can be represented with finite precision numeric types. For example, we can use it to work with floating point infinite values:

```
#include <limits>
#include <iostream>

int main(void) {
  double infinity = std::numeric_limits<double>::infinity();
  std::cout << "Double infinity is " << infinity << std::endl;
  int largest_int = std::numeric_limits<int>::max();
  std::cout << "Largest int is " << largest_int << std::endl;
}
```