The C++ Programming Language

CIS 343 - Structure of Programming Languages

A side note: In these slides I have C++ code that creates C-style arrays. While this is *ok*, it is generally considered better to use a `vector` or `array` class from the C++ Standard Library. I have not done so here only because using C style arrays allowed me to easily and quickly illustrate an example with syntax students are already familiar with.

C++ - History

- Created by Bjarne Stroustrup. He started working on it in 1979 calling it "C with Classes". He found a need for this when programming for his Ph.D. thesis in Simula, noting that Simula provided great programming facilities but was too slow, and BCPL was fast but not advanced enough.
- In 1983 the name was changed to C++.

C++ - Paradigm

- C++ allows us to program using the Object Oriented paradigm, but it doesn't impose a restriction that says we *must* use OO. We could just create imperative procedural code.
- This is primarily because C++ is a superset of C (mostly); almost everything legal in C is legal in C++.

C++ - Paradigm

Warning!

- There are C ways of doing things and C++ ways of doing things. These conventions, even if they are not expressly stated, are usually assumed by other programmers.
- For instance, it is perfectly valid for us to use the <stdio.h> library in C++. However, this library does not solve problems "the C++ way". It doesn't use streams for input and output, doesn't have objects, etc. Using it may compile ok but could add confusion for other programmers (or even for you later on!).

So what? I can mix if I want to.

Yes you can, but you probably shouldn't.

Usually the argument I hear goes something like this:

"I know what I'm doing so I will make sure it is safe."

Great. Are you only coding for you? If not, are you going to be around in a decade to continue maintenance on this code?

Are the next prorammers going to be as advanced as you? Will they have had the same education and experiences and therefore be on the lookout for the same mistakes you know to look for?

Likely not. There are lots of things we do in coding to make it easier for future maintenance. This is one of them.

C++ - Paradigm

There are times you have no choice.

- Using existing C libraries.
- Codebase already mixes the two.
- Boss tells you to do it.
- etc.

C++ - Program Structure

We are going to concentrate on Object Oriented C++. Therefore we will structure our programs much like Java code you may have written in the past.

- This is going to be more "academic" C++ than production code.
- A difference between Java is that the **Syntactic Unit** for an object's definition (the class) is contained in a single file.
    - In C++ it *can* be in a single file, but we often split the class into an interface (*.h file), and the implementation (*.cpp file).
    - For our "academic" C++ we won't be putting more than one class in a single file.

C++ - Program Structure

For instance, if we were going to create a car object we might have two files:

- `Car.h` (or `Car.hpp`) which holds the function signatures (the declarations)
- `Car.cpp` that holds the code (or definitions).

`Car.h`

```
#include <string>

class Car {

  public:
    // Constructor with no parameters.
    Car();

    // Constructor with parameters.
    Car(std::string make, std::string model, int year, double mileage);

    // Accessors
    std::string getMake();
    std::string getModel();
    int getYear();
    double getMileage();

    // Setters
    void setMake(std::string make);
    void setModel(std::string model);
    int setYear(int year);
    double setMileage(double mileage);

  private:
    std::string make;
    std::string model;
    int year;
    double mileage;

    // Fake function just to illustrate we can have functions
    // that are private too.
    void doNothing();

};    // Notice the semicolon at the end of the declaration!
```

C++ - The Namespace Operator

Looking at the previous code you may have noticed the "::" operator between std and string. This is called the Namespace Operator (or Scope Resolution Operator); it signifies where a function is defined.

- In C++ we are allowed to separate code into namespaces. If we declare a function of data type inside of a namespace it becomes part of that namespace. That allows us to separate our code into logical groupings. The "::" operator tells us from which namespace a function is being called.
- I like to think of namespaces as neighborhoods the code lives in.
    - "John Smith? Which John Smith - the one in Grandville or the one in Jenison?"

C++ - The Namespace Operator

Namespaces allow us to have functions with the same names, but in different namespaces. In fact, it is not uncommon for programmers to wish to use the same names for data types of functions.

- Consider how many times you've probably made a function called 'min' or 'max'...
- Notice how we define our functions from the `Car.h` file in a `Car.cpp` file. We have to provide a namespace so the compiler knows specifically which functions we are trying to define:

C++ - The Namespace Operator

When using symbols from an imported library, it is best practice to use fully qualified names. For instance:

```
std::string name;
```

instead of simply entering:

```
string name;
```

C++ - The `using` Keyword

We are *able* to set default namespaces with the `using` keyword (doesn't mean we should).

```
using std;

string name;
```

- Again, it is best practice NOT to do this, but you will get a TON of pushback from people if you tell them this.
- All the same 'logic' that people use for mixing C and C++ applies here.

HOWEVER

Do. NOT. EVER. Have `using` in a header file.

- This is called "polluting the namespace".
- When someone else imports the header file it applies the `using` syntax to *their* code.
- Seriously, this is the equivalent of letting your dog poop in your neighbors yard. You will be messing with other peoples' code.
- It makes debugging a nightmare.

`Car.cpp`

- This is how we could write the definitions for the functions in the Car.h file (following slide).
- Note that if we didn't have the `Car::` namespace before the funtion names they would not be seen as part of the class.
    - They would instead go in the global namespace.

```cpp
#include <string>

Car::Car(){
  make = "None";
  model = "None";
  year = 1900;
  mileage = 0.0;
}

Car::Car(std::string make, std::string model, int year, double mileage){
  this->make = make;
  this->model = model;
  this->year = year;
  this->mileage = mileage;
}

std::string Car::getMake(){
  return make;
}

std::string Car::getModel(){
  return model;
}
... Other getters ...
void Car::setMake(std::string make){
  this->make = make;
}

void Car::setModel(std::string model){
  this->model = model;
}
... Other setters ...
void Car::doNothing(){

}
```

C++ - Using Objects

C++ uses a similar syntax to Java when creating objects:

```
Car a;        // Note that this actually creates the object!
              // Unlike Java in that respect.
Car b("Porsche", "911", 1972, 37228.3);
```

- Creating in this way puts it on the stack, and the runtime will take care of deleting it when it is no longer needed.
- Objects are deleted with their **destructor**.

C++ - Using Objects

- Once we have an object, we call its functions or access its public members just as we do in Java:

```
int mile = a.getMileage();
```

C++ - Memory Management

- Just like in C, C++ manages memory for static (compile time, on the stack) objects only. Dynamically created memory (run-time, on the heap) must be managed by the programmer.
- When dynamically creating an object, we use the `new` keyword.
- Take note! The `new` keyword always returns a pointer.

```
Student a;      // Creates an object.
                // Static.  Cared for by
                // the system.

Student* b = new Student();   // Creates an object.
                              // On the Heap.  Cared
                              // for by you!  Returns
                              // a *pointer*.
```

Advantages of the second method are that the scope is not limited to the block the object was created in and the object can "live" as long as we want it to.

CAVEAT EXAMPLE!

```
Student getStudent(){
  Student a;
  return a;
}

int main(int argc, char** argv){
  Student b = getStudent();
}
```

What does this do?

C++

- The object was created statically in the block of a function.
- The object is passed back by value (meaning it is copied!).
- This is ok for small objects, but for large ones look out!
- Note that newer C++ standards also have the ability to *move* instead of copy which can work as well, but we won't go into those right now.

C++

Alternative:

```
Student* getStudent(){
  return new Student();
}

int main(int argc, char** argv){
  Student* b;
  b = getStudent();
}
```

This will work; the object is created dynamically (at run-time!). It will exist on the heap, so not inside a function's memory. It will exist after the return. There is no need for this object to be copied; instead the pointer variable will be copied (pointers are always small, so no biggie).

C++ - References

Often though, in C++ we are going to be using references.

- You have had a little experience with references from Java, but it is different in C++.
- In C++ we can choose to **pass-by-reference** which Java cannot do!
- Unfortunately, the designers of C++ decided to use the `&` operator to signify a reference - further lowering the readability of an already hard to read language.
- This means in C++ we have to examine the context of the use of `&`. Is it a reference? Is it the memory address of a variable? Etc.

C++ - References

Warning! We have to remember the memory model we are constrained to use. This means we cannot return a reference to an object created inside of a function.

- The memory that the reference points to does not exist after the function call returns.
- The big problem is that it will *sometimes* work. It depends on whether the stack space was overwritten since the object was created or not.

```
File: ReferenceFromFunction.cpp
/*
 * This may work sometimes, but not other times.
 * You should NEVER do this.
 */

#include <string>
#include <iostream>

class Student {
	public:
		static Student& getNew(){
			Student tmp;
			Student& two = tmp;
			return two;
		}
		std::string name;
		int number;
};

int main(int argc, char** argv){
	Student ira = Student::getNew();
	ira.name = "Ira";
	ira.number = 9;

	std::cout << ira.name << std::endl;
}
```

C++ - The Big Five

(Also called the Rule of Five)

When an object is created in C++, it is given some functions. These functions are the:

- copy constructor
- move constructor
- copy operator=
- move operator=
- destructor

The move operator and move operator= are new as of C++11 so they aren't pertinent to older code. Because of this pre-2011 C++ conversations might refer to "The Big Three" instead.

C++ - The Big Five

Note: It is generally better to use the **Rule of Zero** if you can.

If you don't need dynamic memory, don't use it, and C++ will make the Big Five for you.

- Let C++ do all the work for you it can!
- It is highly optimized.
- Use primitives as much as possible.
- There are classes for managing data and memory for you. Leverage them!

C++ - The Big Five

The copy constructor is called whenever we pass an object as the parameter to the constructor of the same type.

- It allows us to create a new object that is a copy of a prior one.

```
Student a;
a.gpa = 2.34;
a.number = 9293;
Student b(a);           // Student b is now a copy of a
```

C++ - The Big Five

- But... what happens if we have an object like this:

```
Student {
  int number;
  float gpa;
  float* grades;
};

Student a;
Student b(a);
```

File: ShallowCopy.cpp

```cpp
#include <iostream>

class Student {
        public:
                int number;
                float gpa;
                float* grades;
};

int main(int argc, char** argv){
        Student a;
        a.grades = new float[50];
        a.grades[0] = 100;
        Student b(a);
        b.grades[0] = 46;
        std::cout << "Student a got a " << a.grades[0] << std::endl;
}
```

C++ - The Big Five

We have a problem... During the copy we copied the pointer.

- The pointer holds the memory location of a's grades.
- Anytime we change a, it will change b's grades as well.
- They will both be forced to have the same grades!

C++ - The Big Five

This is called a **shallow copy**.

- We don't want just the values to be copied.
- They both need memory areas for their own grades. This is what we call a **deep copy**.

The Big Five

Since our `Student` object uses dynamic memory allocation we are going to rewrite (at least some of) The Big Five.

- The default ones won't work for us because our object needs some custom behavior.
- However remember this is for demonstration purposes. We could have used a `vector` or other data structure that does this for us.

File: DeepCopy.cpp

```cpp
#include <iostream>

class Student {
       public:

               // Because we are overriding the constructor,
               // we need to provide a new default one as well.
               Student(){

               }
               // Copy constructor
               Student(const Student& other){
                       // The first two fields we copy
                       this->number = other.number;
                       this->gpa = other.gpa;

                       // This is a pointer though... if we
                       // copy we get the address.  We
                       // need a new memory area for it.
                       this->grades = new float[50];

                       // Now copy the old values in.
                       for(int i=0; i<50; i++){
                               this->grades[i] = other.grades[i];
                       }
               }

               int number;
               float gpa;
               float* grades;
};

int main(int argc, char** argv){
       Student a;
       a.grades = new float[50];
       a.grades[0] = 100;
       Student b(a);
       b.grades[0] = 46;
       std::cout << "Student a got a " << a.grades[0] << std::endl;
}
```

C++ - The Big Five

We used the copy constructor to create a new object that is a copy of a previous one.

- Sometimes, we wish to set an existing object equal to another existing object.
- For this, we use the copy `operator=`.

C++ - The Big Five

There are multiple ways of writing a copy operator=.

- The generally accepted "best" way is to use the Copy-and-Swap method.
- This method works by passing a copy of the original object in as a parameter to the function, instead of passing a reference.
- Then, we steal the copy's data.

```
    /*
     * Copy Operator=
     * Pass copy of object in (will call
     * the copy constructor!).  Then
     * steal its data.
     */
    Student& operator=(Student other){
            // Swap our old data for the
            // good new data.
            std::swap(this->number, other.number);
            std::swap(this->gpa, other.gpa);
            std::swap(this->grades, other.grades);

            // Return ourself!
            return *this;
   }
```

C++ - The Big Five

Why in the world don't we just copy the same way we did for the copy constructor?

- Code duplication is bad.
- This way, we leverage the copy code from the constructor.
- If we ever need to change the copy code, we only change it in one place.

C++ - The Big Five

Additionally, here we are only swapping small data types (ints, etc.).

- We could have just copied them.
- But, if they were large data types (collections, etc.) swapping is much faster than copying.

C++ - The Big Five

C++ 2011 introduced Move Semantics.

- This is based on the idea that it is much quicker for the computer to move an object than to create a new one.
- When moving the memory is already made and setup. Copying will call a copy constructor.
- Moves are often used for temporary objects (such as a return from a function).

C++ - The Big Five

Makes a lot of sense; if we create a new object in a function, and then try to pass that object back as a return value, it gets copied (via its copy constructor).

- If it is a complicated object to create this is very, very slow.
- If we are already creating it in the function, then let's just move the one we created instead.

```cpp
        /*
         * Move Constructor
         * Notice the &&.
         */
        Student(Student&& other){
                std::cout << "Move Constructor called." << std::endl;
                if (this != &other){
                        // Steal the other object's data
                        this->number = std::move(other.number);
                        this->gpa = std::move(other.gpa);

                        // Steal the other object's grades
                        this->grades = std::move(other.grades);

                        // Set the other object's pointer to nullptr.
                        // Otherwise, when it dies it will take
                        // our stolen data with it.
                        other.grades = nullptr;
                }
        }
```

C++ - The Big Five

- We don't usually need to call the Move Operator=, since C++ can recognize whether a temporary (movable) object is being passed in as a parameter to the operator= we already have.
- It will pick the correct constructor on its own.

The C++ Standard Library

- C++ has a **very** powerful and efficient library of utility code. It includes code for the following:
- Containers - arrays, lists, queues, maps, stacks, sets, vector, etc.
- General - container algorithms, time functions, function objects, iterators, memory management functions, eceptions, etc.
- Localization - encapsulation and conversion of locale information.
- Strings - strings and regular expressions.
- Streams - for input and output.
- Language Support - exceptions, limits, typeinfo, etc.
- Threads - threads, mutexes, etc.
- Numerics - random numbers, numeric operations, complex number classes, etc.

C++ - The Standard Library

- The Standard Library should be used as much as possible. It is incredibly robust and efficient.
- If you need code for a common purpose (sorting, searching, specific container types) check here first.

C++ - Best practices

The following slides offer a small list of best practices for writing safe and portable C++ code.

We will start with **iterators**.

C++ - Best practices

- Looping over collections is a very common task. However, not every collection type can be looped over with an integer index. For instance, consider a map. If the index is a string or other object than an int means nothing.
- Iterators are objects that point to an object in a collection. Most collections have a .begin() and .end() function to the a pointer to the first or last element.
- We move to the next element with the increment operator. Some iterators are bidirectional and use the decrement operator to go back an element as well.
- Access to the current element is provided by the '*' operator on the iterator.

C++ - Best practices

```
std::vector<int> nums;

... some code ...

// Probably use 'auto' on the type below
// if writing for production

std::vector<int>::iterator it = nums.begin();
while(it != nums.end()){
        std::cout << *it << std::endl;
        it++;
}
```

C++ - Best practices

- C++ added to the meaning of the `auto` keyword with the 2011 standard.
- `auto` can be used where we wish to have the compiler infer a type.
- In general it should not be used for everything; if you know you are using an `int` or `double`, use those.

C++ - Best practices

- The biggest problem with imperative programming is Side Effects (modifying some variables' state that is not local to the function).
- You should use `const` as much as possible. It helps other people understand your intentions for the code, it protects your data from accidentally being changed and it helps the compiler help you better.
- Here are some examples:

C++ - Best practices

This is a const member function (imagine that this is a getter for a Student class):

```cpp
int& Student::getName() const {
        return name;
}
```

Written this way the function cannot make changes to member variables. If code attempts to, you will get a compiler error.

C++ - Best practices

This is a function that takes a parameter as a const reference:

```
void interestingFunction( const MyBox& obj ){

...

}
```

This code cannot modify the object passed to it. This is a VERY heavily used idiom in C++ - for good reason. It helps to prevent **Side Effects**.

C++ - Best practices

With pointers, we can either mark the pointer const (meaning the pointer's value can't change), or we can mark the data the pointer points to as const:

```cpp
// Can't change the data it points to here
double x = 42;
const double* y = &x;

// Can't change the address to something else later:
int z = 42;
int * const w = &z;
```

C++ - Best practices

The Standard Library requires const iterators for const collections so that the collections; data cannot be mutated accidentally. So you may see something like this:

```
std::vector<double> nums;
...

for( std::vector<double>::const_iterator it = nums.begin(); it != nums.end; ++i){
        ...
}
```

Here the iterator cannot modify the data, only read it.

C++ - Operator Overloading

C++ allows us to create custom operators for our objects.

- `a + b`, `a - b`, `+a`, `-a`, `a * b`, `a / b`, `a % b`, `++a`, `a++`, `--a`, `a--`
- `a == b`, `a != b`, `a > b`, `a < b`, `a >= b`, `a <= b`, `a <=> b`
- `!a`, `a && b`, `a || b`
- `~a`, `a & b`, `a | b`, `a ^ b`, `a << b`, `a >> b`
- `a = b`, `a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`, `a &= b`, `a |= b`, `a ^= b`, `a <<= b`, `a >>= b`
- `a[b]`, `*a`, `&a`, `a -> b`, `a->*b`
- `new`, `new type[n]`, `delete`, `delete[] a`, `(type)a`
- And more...

Miscellaneous Things You Should Know

... but we don't have time for.

- Smart Pointers
- Templates
- Initializer Lists
- Lambda Expressions

Final Thoughts

There is SO much more to C++. An entire semester couldn't do it justice. It is a language that must be used (and used often) to truly be good at it.

Using best practices and keeping up with language changes certainly will help you in the long run.