Intro to C++
We are going to cover a lot in these slides. Though previous knowledge of C is helpful, it is not required to do well in this course.

Please ask questions and come see us afterwards if any concepts are unfamiliar!
Overview

- Hello World
- Basic Memory
- Classes
- Polymorphism
- Standard library
Hello, World!
C++ is not Java or C. It is C++

- C++ is syntactically very similar to C and Java
- However, C++ is not just “C with classes”
- While a lot of things will look familiar, it is important to know how things actually work
- Plenty of stuff will do what you think it does, but plenty won’t

**tl;dr:** Stay tuned!
// HelloWorld.java
public static void main(String [] args) {
    System.out.println("Hello World!");
}
// Prints: Hello World!

// helloworld.cpp
#include <iostream>                // Include directive for inputs and outputs

int main(int argc, char *argv[]) { // Entry to program is with the “main” function
    std::cout << "Hello World!" << std::endl; // Output to standard output (the terminal)
    return 0;                              // Return 0 on success
}
// Prints: Hello World!
New Things

- If you have any experience with Java, C, or almost any other language, most of that looked familiar, right?
- What were the things that didn’t?

- `#include`
- `char *`
- `std`
- `::` and `<<` operators
- `cout` and `endl`
#include

○ Unlike Java, C++ does not have packages or imports
○ Use includes instead
○ Examples:
  - `#include <iostream>` // include a library
  - `#include "headerfile.h"` // include a header file - more on these later!
  - `#include "headerfile.hpp"` // NOTE: file extensions don’t matter
C++ Standard Library (std)

- Namespace is **std**
  - e.g. `std::cout`
  - A namespace is just like a Java package!
  - More on Namespaces later...

- Has all kinds of useful things
  - Strings
  - Vectors (like Java's ArrayList)
  - And more...
Scope Resolution Operator

○ This thing → ::
  - Used like "." in Java for package resolution, just different syntax

○ Used to resolve the scope of an identifier
  - e.g. locate something (function, variable, etc.) in a namespace
  - Say we have two namespaces SpaceA and SpaceB, each with the function myFunc()
  - How can you tell which is which?
  - SpaceA::myFunc() or SpaceB::myFunc()
**std::string**

- Is secretly a C string, but wrapped so it works like more like a Java string
- You can get its size, take substrings, search, etc
  - More info can be found on the [C++ reference website](C++ reference website)
- Also has overload of `+` operator for concatenation
  - `std::string abcd = std::string("abc") + "d";
  - C++ supports operator overloading, it's just like method overloading in Java but for operators
- There are also other types of strings
  - QStrings, char *arrays, etc.
cout and endl

- Available in `<iostream>`  
  ```cpp
  #include <iostream>
  ```
- **std::cout** is how you will print to the command line
  - i.e. the output window in QtCreator, your IDE for this course :)
- `<<` inserts characters into the current output stream
  - Works like string concatenation where the *stream* is the output string
- **std::endl** ends the line and inserts a newline char
- **std::cin** and **std::cerr** also exist
  - **std::cin** reads from stdin
  - **std::cerr** prints to stderr
// print.cpp
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "Print me to the console!" << std::endl;
    std::cerr << "I am an error!" << std::endl;
    return 0;
}

// Output:
Print me to the console! // std::cout prints to the console
I am an error! // std::cerr usually appears in bold red font. Good for debugging!
Basic Memory

https://xkcd.com/138/
Overview of Basic Memory

- Regular Variables
- Pointers
- Automatic vs. Dynamic Storage
  - Smart Pointers
- References
- Array Pointers
Regular Variables

- You can **declare** variables
  - Compiler enters symbol into symbol table
  - Like Java and C, C++ lets you declare variables by their type

- You can **define** (aka initialize) variables
  - Compiler assigns value to symbol
  - Always define your variables before use!!

- **Declaration != Definition**
# Sample Program

// main.cpp

#include <iostream>

int x = 5; // [type] [name] = [initial value]; , global

int main(int argc, char *argv[]) {
    float pi = 3.1415f; // type name = initial value, local
    std::string s("some string"); // type name constructor arguments
    // ^ more on constructors later

    int y; // y is declared (y exists) but not defined (y has
             // no known value) (default values vary by architecture)

    int z = 100; // z is declared and defined
    return 0;
}

}
Pointers and memory locations

- Variables exist in memory
- Pointers represent memory locations
- Pointers allow you to hold on to the location of the object\(^1\), rather than the object itself
- This is especially helpful when you need to allocate large objects and pass them around

\(^1\)C++ like Java distinguishes between primitives (int, float, bool) and objects. But for sake of convenience, we call all of these objects.
Pointers (now with pictures!)

```c
int x = 10;
int *p;
p = &x;
```

- "&x" gets the address of where x is stored
- "p" now holds the memory location indicated by "&x"
- "p" is not the same as what is stored at the memory location, it just tells you where that location is
- "p = x" would give you a type error

e.g. 0x0023FF74
Pointers (cont’d)

○ To use pointers, you have to use new operators:
  - * and &
○ The first you will probably encounter is *
○ What * means greatly depends on context
  - You’ve already seen it as the multiplication operator (int a = 4 * 3;)
○ When declaring variables, * denotes that the variable is a pointer
  - For example, Car *myPtr is a pointer to an instance of a Car object
  - (Car *myPtr and Car* myPtr are both correct, but first is preferred)
Pointers (now with pictures!)

```
int x = 10;
int *p;
```

- "x" is an object (here, an int) somewhere in memory
- "p" is a pointer to an int

(Not pointing to anything known right now)
Pointers (now with pictures!)

```c
int x = 10;
int *p;
p = &x;
*p = 20;
```

- What if we want to get at what is inside the memory location using “p”?
- Dereference a pointer to access what it points to.
- Now we can change whatever was in that memory location
- While p is unchanged, *p and x are synonyms
Automatic vs. Dynamic Storage

- The code in the previous section was stored using **automatic storage**
  - Automatically allocated variables clean up after themselves ("garbage collection")
  - You don’t need to worry about managing memory
- However, automatic storage is sometimes not enough
Automatic vs. Dynamic Storage Preview

- Use **dynamic storage** to manually allocate memory
- Useful for when
  - You need a lot of memory
  - The object needs to persist longer than a single block of code
- You allocate with **new** and deallocate with **delete**
- Do **NOT** use **malloc/free**
// Example of simple new/delete usage

// Construct a MyClass object with automatic storage
MyClass myClassObject = MyClass();

// Construct a MyClass object and store its memory location in a pointer
MyClass *myClassPointer = new MyClass();

// What’s wrong with this?
MyClass myClassPointer = new MyClass();

// Use . to call member functions (i.e. methods in Java) on an object
myClassObject.baz();

// Use -> to call member functions on a pointer to an object
myClassPointer->baz();

// No need to delete myClassObject

// Delete the object stored at myClassPointer using delete
delete myClassPointer;

// MAKE SURE THAT OBJECTS THAT ARE NEW’D ARE ALWAYS DELETED!!!
// OTHERWISE IT’S A MEMORY LEAK
Smart Pointers (Preview)

○ C++ 11 and newer feature **smart pointers**
  - `#include <memory>`
  - `std::unique_ptr<T>`
  - `std::shared_ptr<T>`

○ Automatic memory management

○ Do NOT call **`new/delete`** unless you absolutely have to
  - `std::unique_ptr<MyClass> x = std::make_unique<MyClass>();`
  - `std::shared_ptr<MyClass> y = std::make_shared<MyClass>();`
  - automatically deleted when they leave scope

○ Use smart pointers as much as possible
  - more info in next C++ help session
  - there is a deduction for using raw pointers unnecessarily but we will be lenient on Brush
References

○ A reference to an object is similar to a pointer, but is of type `Class&` rather than `Class*`
○ Allows two variables to be used for the same object in memory
○ References are created using the `&` operator

```cpp
int x = 5;
int &xref = x;  // x and xref now reference the same int in memory
xref++;
std::cout << x << std::endl;  // prints ‘6’
```

○ Why use references instead of pointers?
  - Generally, references are simpler and “safer” (next slide)
○ More on references in the next help session
// References vs. Pointers Example
#include <iostream>

void incrementPointer(int *x) {
    (*x)++;  // x++ would increment the actual memory address
}

void incrementReference(int &x) {
    x++;  // Wow! So simple. So beautiful.
}

void incrementConstReference(const int &x) {
    x++;  // Error! Can’t modify const reference (more on const later)
}

int main(int argc, char *argv[]) {
    int a = 10, b = 20, c = 30;
    incrementPointer(&a);
    incrementReference(b);
    incrementConstReference(c);
    std::cout << a << " " << b << std::endl;  // 11 21
    return 0;
}
Array Pointers

- In C++, arrays can be managed and manipulated using pointers
- Some syntax:

```cpp
double emptyArray[10]; // an empty double array with 10 slots
int staticArray[3] = { 1, 2, 4 }; // statically allocated array
int num = staticArray[2]; // num = 4

int *dynamicArray = new int[10]; // common in OpenGL - dynamic array
dynamicArray[0] = 1;
int num2 = dynamicArray[0]; // num2 = 1
delete[] dynamicArray; // make sure to use array deletion!
// no need to delete emptyArray or staticArray

- Multidimensional arrays do exist, but it is better to instead make a 1D array and index into it as if it had higher dimensions

```cpp
double *not2DArray = new double[width * height];
valAtXY = not2DArray[row * width + col]; // index into row-major 2D grid
```
Mapping from 2D (row, col) to 1D (index):

\[ \text{index} = \text{row} \times \text{width} + \text{col} \]
So what’s this “segfault” thing?

- A very common error you will learn to hate
- “Segfault” is short for segmentation fault
- It often occurs when attempting to access memory that was not allocated, or which had already been deleted
  ```
  Car *p; // should point to a Car, but nothing is there!
  p->drive(); // *CRASH*
  ```
- What makes segfaults even more insidious is that in some cases, what looks like it should trigger a segfault doesn’t, and you could get errors later in the program
  - In the above example, if `drive()` didn’t access any member variables, the code would just keep running...
Classes
Classes - Intro

○ C++ is an object oriented language much like Java
○ A class is a collection of **member variables** and **member functions**
○ Classes are instantiated as objects
Overview of C++ Classes

- Syntax
- Access Modifiers
- Constructors
- Initializer Lists
- Destructors
- Structs
- Namespaces
Syntax

- C++ classes are laid out similarly to Java classes
- On the next slide is an example C++ class
// Example class definition in a header. Also, everything on a line after two slashes is a comment.

class Foo { // class [name of class]

public: // Access modifier
    Foo(int x); // Constructor
    ~Foo(); // Destructor

    int getX(); // Member function declaration
    void setX(int x); // Member function declaration
    void incrementY(); // Member function declaration

private: // Access modifier
    int m_x; // Member variable (Convention: always begin with m or m_)
    Car *m_car; // Member variable

}; // Semicolon that you will not forget
Syntax

- You’ll notice that the member functions are not filled in.
- C++ classes are split into two files, the header file (.h) and the source file (.cpp).
- Think of this as separating the declaration from the definition.
- Member functions are *almost always* implemented in the source file.
  - Source code in header files will be recompiled by every file that includes them: unnecessarily long build times!
Access Modifiers

○ Like Java, C++ has access modifiers
○ These keywords define who is able to access the functions and variables that follow:
  - **public**: The class, derived classes, and “anyone outside” the class (other classes, functions, etc.)
  - **protected**: Only the class and derived classes
  - **private**: Only the class
○ Access is determined by the most recent modifier
// Example cpp file - the implementation of Foo

#include "Foo.h" // Include preprocessor directive. More on this later.

Foo::Foo(int x) : // Constructor - used to initialize all class resources
    m_x(x), // Initializer list
    m_car(new Car())
{
}

Foo::~Foo() // Destructor - used to free all class resources
{
    delete m_car;
}

void Foo::setX(int x) { // Double colon (the scope resolution operator)
    // It’s not the same as single colon (as seen in the constructor above).
    m_x = x;
}

// Other definitions omitted
Constructors

○ Constructors are special functions that are called whenever an object is initialized
  - Classes may have multiple constructors, like in Java
○ Use them to initialize all of the object’s state
○ The object should be useable after construction
  - Having an “init()” function that needs to be called after the constructor means you’ve done something wrong
  - More on why this is important in the Intermediate C++ Help Session
Constructors - Default Constructors

○ Every class that does not explicitly define a constructor has a *default constructor* implicitly defined
  - `Foo();` // Note: No arguments

○ Explicitly defining a constructor will remove the implicitly define default one
  - Defining `Foo(int x)` removes implicit `Foo()`
  - You can still explicitly define `Foo()`, too
Initializer Lists

- Member variables are initialized in initializer lists, not the constructor body!
  - By the time the body executes, every member variable has already been constructed
  - If you don’t explicitly initialize a member variable, its default (empty) constructor is implicitly called
    - This will cause a compilation error if that variable has no default constructor!
// Foo.h

class Foo {
public:
    Foo(); // Default (no arguments) constructor
    Foo(int x, float q, Bar b); // Some other constructor

private:
    int m_x; // Member variables
    float m_q;
    Bar m_bar; // Some other object
};

// Foo.cpp

Foo::Foo() :
    m_x(0), // Alternatively, you can use some default values rather
    m_q(3.15f), // than constructor arguments to initialize member variables
    m_bar(Bar(4, 1, "some string")) // Here, Bar is some class that takes some arguments
    // in its own constructor
{
}
// Foo.h

class Foo {

public:
    Foo(); // Default (no arguments) constructor
    Foo(int x, float q, Bar b); // Some other constructor

private:
    int m_x; // Member variables
    float m_q;
    Bar m_bar; // Some other object
};

// Foo.cpp

Foo::Foo(int x, float q, Bar b) :
    m_x(x), m_q(q), m_bar(b)
{
    m_x = 5; // m_x is already initialized by now
            // It is now being reassigned
}
    // You want to avoid initializing things here
Watch out!

○ Leaving member variables uninitialized is a bad idea
○ Leaving this task to the implicitly defined default constructor can also be bad
○ Consider these examples:
// Bar.h

class Bar {

public:
    Bar(int a, int b, std::string label);  // Constructor explicitly defined.
    // No more default (empty) constructor

private:
    int m_a;
    int m_b;
    std::string m_label;

};
// Foo.h
class Foo {
public:
    Foo();
private:
    Bar m_bar;       // Some other object that has no default constructor
};

// Foo.cpp with error
Foo::Foo() { // Error! By default, Bar’s default constructor is called when no other constructor is specified. But Bar has no default constructor!
}

// Foo.cpp fixed
Foo::Foo() :
    m_bar(Bar(4, 1, "some string")) // Much better
{
}
/Alternate Foo.h
class Foo {
    // No explicit constructors = implicit default constructor
private:
    Bar m_bar;  // Some other object that \textbf{has no default constructor}
};

// Now the implicit default constructor is built in the background
// \textbf{Note:} This is showing default behavior, not C++ code written by you

// Alternate Foo.cpp

Foo::Foo() :
    m_bar(Bar())  // \textbf{Error!} The default constructor calls the default constructors of
{  // each member object. But if a member object has no default
}  // constructor, this is impossible!
Initializer Lists - Recap

- Don’t rely on the implicit default constructor to get things right in the background
- Always initialize member variables in initializer lists

**tl;dr:** Make your own constructors, use initializer lists for everything
Destructors

○ Each class has one and only one destructor, which takes the form of \(~\text{ClassName}()\)
  - From the example, \(~\text{Foo}()\)

○ Destructors are responsible for freeing any resources allocated by the class

○ Destructors are called when an object is deleted or falls out of scope
  - More on this later
Structs

- Structs are a holdover from C
  - Side note: C++ is (except for a few edge cases) a superset of C

- When to use structs?
  - For POD: Plain Old Data
    - Container for heterogeneous data
    - (By convention) no constructor, destructor, member functions, etc.
    - Breaks usual class design in favor of memory layout
  - C compatibility (library/legacy code)
Structs

- C++ has structs in addition to classes
- The only functional difference between classes and structs is that **classes default to private accessibility** and **structs default to public accessibility**
Namespaces

- Alice makes a class called Foo
- Bob makes a class called Foo
- Alice and Bob join code bases
  - Now there are two different classes called Foo
  - Naming conflicts!
  - Code is broken!
Namespaces

- Namespaces solve naming conflicts
- Very similar to Java packages
- Provide mechanism for protecting against naming collisions
namespace Alice {
    class Foo {
        // ...
    };
}

namespace Bob {
    class Foo {
        // ...
    };
}

int main(int argc, char *argv[]) {
    Alice::Foo alicesFoo; // Use the scope resolution operator (::) to access
    Bob::Foo bobsFoo;     // the contents of the namespace.
}
Namespaces

○ You may have namespaces within namespaces
  - Big companies do this - they have lots of code
  - You won’t need to do this, although some of our support code uses it

○ Things in the C++ Standard Library are in the namespace std
  - Examples: std::cout, std::vector
Namespaces

○ But what if I don’t want to type out that namespace every time?
○ Keyword **using** to the rescue
○ The using directive works like *import* in Java
  - Everything in that namespace is now available inside your current scope
// Previous declarations omitted for brevity

using namespace Alice;  // Like so

int main(int argc, char *argv[]) {
  Foo alicesFoo;          // Resolves to Alice::Foo
  Bob::Foo bobsFoo;       // Still need to access Bob’s namespace explicitly
}
Namespaces

○ **Warning:** Never use keyword *using* in a header file!
  - This will open up the namespace into every file the header gets included in
  - If this header goes in other headers, the problem blows up

○ Generally safe to use in `.cpp` files
○ Can also be used in functions
Polymorphism
Polymorphism Overview

- **Class inheritance**
- **Virtual functions**
  - Constructors/Destructors (again)
- **Pure virtual functions**
- **Class Inheritance**
Class Inheritance

○ A class can be *derived* from another class in order to extend or implement its functionality
  - example: *Ferrari* derives from *Car*
  - generally: “Derived” derives from “Base”

○ Derived classes can access base member functions and variables, just like in Java!
Class Inheritance

○ For now, we will only cover public inheritance (child only inherits public parent)
  - private inheritance is much less common and shouldn't be used in this course
  - nobody knows what protected inheritance is for

○ Here’s an example:
// Base.h
class Base {
public:
    Base(int x);
    int getX();
private:
    int m_x;
};

// Derived.h
class Derived : public Base {   // Derived publicly derives from Base
public:
    Derived (int x, float y);
    void increaseY();
private:
    float m_y;
};
// Base.cpp
Base::Base(int x) : // Base’s constructor
    m_x(x) // m_x initialized in initializer list
{
}

int Base::getX() { // Member function
    return m_x;
}

// Derived.cpp
Derived::Derived(int x, float y) : // Derived’s constructor:
    Base(x), // First initializes Base through its constructor! (Like Java’s super.)
    m_y(y) // Then initializes its own member variables
{
}

void Derived::increaseY() { // Member function
    m_y++;
}
Class Inheritance

○ Base classes are constructed before derived classes
  - watch out for implicit default constructors here as well!
○ Destructors are called in the reverse order
○ Next: Access modifiers in public inheritance
  - should be very familiar
// Base.h
class Base {
public:
    void foo();
protected:
    void bar();
private:
    bool dodge();
    double m_x;
};

// Derived.cpp
void Derived::someFunction() {
    foo();      // Okay - foo() is public
    bar();      // Okay - bar() is protected and visible to derived classes
    dodge();   // Error! dodge() is private to Base
    m_y = m_x;  // Error! m_x is private to Base
}
Virtual functions

○ The base class marks overridable functions with keyword `virtual`
○ Non-virtual functions cannot be overridden
○ Use the `override` keyword in the derived class to signify that you’re overriding a virtual function
  - **NOTE:** using keyword override is optional. However, we will see that it is very good practice!
Virtual functions

○ Can we re-implement member functions?
  - you can override them completely or partially (if you choose to call the base class’s method using **Base::**)  
  - e.g. **Base::myFunction()**  
  - **Base::** is a special namespace that automatically invokes the parent class

Note: **Base** is the name of the superclass, not a special keyword like **super** in Java. If the name of your superclass is **Tree**, you can invoke its methods in the subclass by using **Tree::myFunction()**.
// Base.h
class Base {  
public:  
    virtual void foo();  
protected:   
    void bar();  
private:  
    bool dodge();   
    double m_x;  
};

// Derived.cpp
void Derived::foo() {
    foo();  // Okay - calls this function
    Base::foo();  // Okay - calls Base's foo() function! (like super.foo() in Java)
}

void Derived::someFunction() {
    m_y = m_x;  // Error! m_x is private to Base!
    foo();  // Okay - foo() is public
    bar();  // Okay - bar() is protected and visible to derived classes
    dodge();  // Error! dodge() is private to Base!
}
// Base.h
class Base {
public:
    virtual void wumbo();    // Virtual function. Can be overridden
    void mini();              // Nonvirtual function. Can’t be overridden
};

// Derived.h
class Derived : public Base {
public:
    void wumbo() override;    // No error - this overrides Base’s void wumbo()
    void wumbo();             // No error - override is optional. But you should still use it because...
    bool wumbo(int x) override; // Error! wumbo matches no declaration in Base. Thanks keyword override!
    bool wumbo(int x);        // No error but tricky! You may think you’ve overridden Base’s void wumbo(),
                              // however this wumbo is different, now you have two! This isn’t what you want
    void mini();              // Watch out! Nonvirtual can’t be overridden. See next slide
};
// Base.h
class Base {
public:
    void mini(); // Nonvirtual function. Can’t be overridden
}

// Derived.h
class Derived : public Base {
public:
    void mini(); // Watch out! Nonvirtual can’t be overridden!
}

// main.cpp
int main( int argc, char *argv[] ) {
    Base *b = new Base();
    Derived *d = new Derived();
    Base *sneaky = new Derived(); // sneaky is a Derived, which is a Base

    b->mini(); // Calls Base’s mini()
    d->mini(); // Calls Derived’s mini()
    sneaky->mini(); // Calls Base’s mini()!
}
Constructors/Destructors (again)

○ When you have a derived class:
  - You must call the base constructor explicitly
  - The destructor of the base class **MUST** be marked virtual
    - If it isn’t, you will get a memory leak
    - There is a deduction for this!

○ Conversely, if a class is not derived from:
  - Its destructor should NOT be marked virtual
    - Unnecessarily marking things virtual adds a lot of overhead
    - There is a deduction for this too!
Pure virtual functions

○ Pure virtual functions need to be implemented by a Derived class
○ Classes with pure virtual functions cannot be instantiated
○ Think of them like abstract classes in Java
○ Pure virtual functions are declared virtual and followed by “= 0”

  e.g. `virtual void foo(...args...) = 0;`

*Base classes can technically provide implementations of pure virtual functions, but they can only be called from derived classes. Don’t worry about this.
// Base.h
class Base {
public:
    virtual void wumbo() = 0; // Pure virtual function
};

// Derived.h
class Derived : public Base {
public:
    void wumbo() override; // Derived’s version of wumbo
}

// Derived.cpp
void Derived::wumbo() { // Implementation
    // Do stuff
}
How to: Polymorphism

○ Prefer **composition** over **inheritance**
  ○ In general, it is better to use composition rather than inheritance
  ○ Makes for cleaner, more manageable code
  ○ Easier to maintain in the long run
  ○ Less debugging for you :)

○ Next is an example where composition makes more sense than inheritance...
Inheritance

// Tree.h
class Tree {
    void grow();
};

// Enemy.h
class Enemy {
    void attack();
};

// Entities.h
class WillowTree : public Tree {
    // What do we inherit?
    ...
};

class Goblin : public Enemy {
    // ...
};

class WhompingWillow : // What do we inherit?

Composition

// Entities.h
class WillowTree {
    Tree* m_treeComponent;
};

class Goblin {
    Enemy* m_enemyComponent;
};

class WhompingWillow {
    Tree* m_treeComponent;
    Enemy* m_enemyComponent;

    void growAndAttack() {
        m_treeComponent->grow();
        m_enemyComponent->attack();
    }
};
For Brush it is up to you to decide whether you want to use **Inheritance** or **Composition**, either is fine with us. However, we do not want to see four separate member variables for the four separate brushes all calling the same methods. You should be taking advantage of polymorphism in Brush.
Summary: Converting from Java

○ Here’s a recap of importance concepts:
  - Inheritance: `class Derived : public Base`
  - Allow function re-implementation in derived:
    - use the `virtual` keyword
  - Be explicit when re-implementing functions:
    - use the `override` keyword
  - Force implementation in derived: `virtual ... = 0;`
  - Avoid using multiple inheritance
    - use composition instead
    - more in the Intermediate C++ Help Session...
Standard Library
Overview of Standard Library

- The C++ Standard Library
- Standard Headers
- Using the Standard Library
- `<iostream>`
- `<vector>`
- `<cmath>`
The C++ Standard Library

- C++ comes with a standard library of functions, objects, and constants to make calculation, storage, threading, timing, and other useful operations possible
- Every standard library element is held in the `std` namespace
- Standard library functions can be accessed by including their header files
Standard Library Headers

- Library headers are included with
  
  ```
  #include <iostream>
  ```
  
  - Notice the substitution of "" for <>
  - This is important as it alerts the preprocessor to look in system, rather than local, headers!
Using the Standard Library

- Access standard library elements by including a standard header calling `std::` with the name of the element

```cpp
#include <string>
#include <iostream>

std::string message = std::string("abc"); // new string
std::cout << message << std::endl; // prints out "abc"
```
Vectors (i.e. ArrayLists)

- **std::vector** is similar to a Java's ArrayList object
- **std::vector** are dynamically sized, and can grow as you add elements, etc
  - more performant to declare with initial size if known to avoid repeatedly resizing array ([] instead of push_back)
- **std::vector** is a templated class, which is similar to a Java Generic
  
  ```cpp
  std::vector<T> vector;
  std::vector<int> intVec; // a vector of ints!
  std::vector<MyClass> objVector; // a vector of MyClass!
  ```

**NOTE:** Use vectors over arrays when possible!
```cpp
#include <vector>

std::vector<int> intVector; // create an empty int vector  // []
intVector.reserve(2);       // reserve memory for 2 ints (always reserve if you can)
intVector.push_back(5);    // append 5 to the vector  // [5]
intVector.push_back(7);    // [5, 7]
intVector.at(0) = 1;       // replace the 0th element with 1  // [1, 7]
intVector[0] = 2;          // can also use brackets to access  // [2, 7]
int val = intVector.at(1); // val = 7  // [2, 7]
intVector.clear();         // empty the vector  // []
```

Items can also be inserted, deleted, and swapped in a vector by using `std::vector::iterator`.
cmath

○ `<cmath>` is the C++ Standard Library math package.
○ It contains useful elements such as:
  - `abs(number)` // absolute value
  - `exp(power)` // e^power
  - `pow(base, power)` // base^power
  - `sqrt(number)` // square root
  - `sin(angle), cos(angle), tan(angle)` // as well as inverse and hyperbolic
  - `ceil(number), floor(number)` // round up, round down
  - `round(number)` // round up or down based on .5 cutoff
  - `M_PI` // 3.14159265358979323846264338327950...
○ And that’s just the basics :)  
Standard C++ Deductions

- Don't lose points!
- [https://cs.brown.edu/courses/cs123/docs/Standard%20Deductions.pdf](https://cs.brown.edu/courses/cs123/docs/Standard%20Deductions.pdf)
Questions?