C++ Fundamentals

Only what you need to know

www.inl.gov

Idaho National Laboratory

Outline

- Part 1
  - Basic Syntax Review
  - C++ Definitions, Source Code Organization, Building your Code
- Part 2
  - Scope
  - Pointers and References
  - Dynamic Memory Allocation
  - Const-ness
  - Function Overloading
- Part 3
  - Type System
  - Brief Intro to Using Templates
  - C++ Data Structures
  - Standard Template Library Containers
- Part 4
  - Object Oriented Design
  - Classes in C++

Setup

- All of the example code is housed in a GitHub repository
- Copy it to your machine using:
  `git clone https://github.com/friedmud/cpp_tutorial.git`
- There is a PDF of these slides in there
- The default compiler is `g++`
- You can change it using `export CXX=clang++` (or whatever you want)

Typeface Conventions

- Key concepts
- Special attention required!
- `Code`
- `// Comments`
- `int x; // Language keywords`

MOOSE Coding Standards

- Capitalization
  - ClassName
  - methodName
  - _member_variable
  - local_variable
- FileNames
  - src/ClassName.C
  - include/ClassName.h
- Spacing
  - Two spaces for each indentation level
  - Four spaces for initialization lists
  - Braces should occupy their own line
  - Spaces around all binary operators and declaration symbols `+ - * & ...`
- No Trailing Whitespace!
- Documentation for each method (Doxygen keywords)
  - `@param`
  - `@return`
  - `///Doxygen Style Comment`
- See our wiki page for a comprehensive list
  https://hpcsc.inl.gov/moose/wiki/CodeStandards

Part 1

- Basic Syntax Review
- C++ Definitions
- Source Code Organization
- Building your Code

Review: C Preprocessor Commands

- “#” Should be the first character on the line
  - `#include <iostream>`
  - `#include "myheader.h"`
  - `#define SOMEWORD value`
  - `#ifdef, #ifndef, #endif`

- Some predefined Macros
  - `__FILE__`
  - `__LINE__`
  - `__cplusplus`

Review: Intrinsic Data Types

Basic Type	Variant(s)
`bool`	
`char`	`unsigned`
`int`	`unsigned`, `long`, `short`
`float`	
`double`	`long`
`void`[1]	

[1] The "anti-datatype," used e.g. for functions returning nothing

Review: Operators

- Math: `+ - * / % += -= *= /= %= ++ --`
- Comparison: `< > <= >= != ==`
- Logical Comparison: `&& || !`
- Memory: `* & new delete sizeof`
- Assignment: `=`
- Member Access:
  - `->` (Access through a pointer)
  - `.` (Access through reference or object)
- Name Resolution: `::`

Review: Curly Braces { }

- Used to group statements together
- Creates new layer of scope (we’ll get to this)

Review: Expressions

- Composite mathematical expressions:

```
a = b * (c - 4) / d++;
```

- Composite boolean expressions:

```
if (a && b && f()) { e = a; }
```

  - Note: Operators `&&` and `||` use "short-circuiting," so "`b`" and "`f()`" in the example above may not get evaluated.

- Scope resolution operator:

```
t = std::pow(r, 2);
b = std::sqrt(d);
```

- Dot and Pointer Operator:

```
t = my_obj.someFunction();
b = my_ptr->someFunction();
```

Review: Type Casting

```
float pi = 3.14;
```

- C-Style:

```
int approx_pi = (int) pi;
```

- C++ Styles:

```
int approx_pi = int(pi);
int approx_pi = static_cast<int>(pi);
```

Review: Limits to Type Casting

- Doesn't work to change to fundamentally different types

```
float f = (float) "3.14";        // won't compile
```

- Be careful with your assumptions

```
unsigned int huge_value = 4294967295; // ok
int i = static_cast<int>(huge_value); // won't work!
```

Review: Control Statements

- For, While, and Do-While Loops:

```
for (int i=0; i<10; ++i) { }
while (boolean-expression)  { }
do { } while (boolean-expression);
```

- If-Then-Else Tests:

```
if (boolean-expression) { }
else if (boolean-expression) { }
else { }
```

- In the previous examples, `boolean-expression` is any valid C++ statement which results in `true` or `false`. Examples:
  - `if (0) // Always false`
  - `while (a > 5)`

Review: Control Statements

```
switch (expression)
{
  case constant1:
    // commands to execute if
    // expression==constant1 ...
    break;
  case constant2:
  case constant3:
    // commands to execute if
    // expression==constant2 OR expression==constant3...
    break;
  default:
    // commands to execute if no previous case matched
}
```

Printing to The Console

- Pull in printing capability: `#include <iostream>`
- Important types:

```
std::cout // Normal output "stream" (stdout)
std::cerr // Error output stream (stderr)
std::endl // "endline" character
```

- Print to console using ”push” or ”left_shift” operator (<<):

```
std::cout << "Stuff!\n";
std::cout << a_variable << std::endl;
```

- Note: output is buffered. `std::endl` “flushes” the buffers.

HelloWorld.C

- Put this in `HelloWorld.C`:

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- Compile using: `cpp_compiler HelloWorld.C -o hello`
- `cpp_compiler` will most likely be either `g++` or `clang++`
- Run using: `./hello`

Declarations and Definitions

- In C++ we split our code into multiple files
  - headers (`*.h`)
  - bodies (`*.C`)

- Note! The extensions are case sensitive!

- Headers generally contain declarations
  - Our statement of the types we will use
  - Gives names to our types

- Bodies generally contain definitions
  - Our descriptions of those types, including what they do or how they are built
  - Memory consumed
  - The operations functions perform

Declaration Examples

- Free functions:

```
returnType functionName(type1 name1, type2 name2);
```

- Object member functions (methods):

```
class ClassName
{
  returnType methodName(type1 name1, type2 name2);
};
```

Definition Examples

- Function definition:

```
returnType functionName(type1 name1, type2 name2)
{
  // statements
}
```

- Class method definition:

```
returnType ClassName::methodName(type1 name1, type2 name2)
{
  // statements
}
```

Function Example: Addition

```
#include <iostream>
int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}



int main()
{
  int z;
  z = addition (5,3);
  std::cout << "The result is " << z << std::endl;
  return 0;
}
```

Addition Cont'd: Separate Definition and Declaration

```
#include <iostream>
int addition (int a, int b);



int main()
{
  int z = addition (5,3);
  std::cout << "The result is " << z << std::endl;
  return 0;
}



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

Compiling, Linking, Executing

- Compile and Link

```
g++ -O3 -o myExample myExample.C
```

- Compile only

```
g++ -O3 -o myExample.o -c myExample.C
```

- Link only

```
g++ -O3 -o myExample myExample.o
```

Compiler/Linker Flags

- Libraries (`-L`) and Include (`-I`) path
- Library Names (`-l`)
  - Remove the leading "lib" and trailing file extension when linking `libutils.so` would link as `-lutils`

```
g++ -I/home/permcj/include       \
    -L/home/permcj/lib -lutils \
    -Wall -o myExec myExec.o
```

Recall Addition Example

```
#include <iostream>
int addition (int a, int b);  // will be moved to header

int main()
{
  int z = addition (5,3);
  std::cout << "The result is " << z << std::endl;
  return 0;
}

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

Header File (add.h)

```
#ifndef ADD_H    // include guards
#define ADD_H

int addition (int a, int b); // Function declaration

#endif  // ADD_H
```

- Headers typically contain declarations only

Source File (add.C)

```
#include "add.h"

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

Driver Program (main.C)

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

int main()
{
  int z = addition(5,3);
  std::cout << "The result is " << z << std::endl;
  return 0;
}
```

Compiling the "Addition" Example

1. `g++ -g -c -o add.o add.C`
2. `g++ -g -c -o main.o main.C`
3. `g++ -g -o main main.o add.o`

- The `-c` flag means compile only, do not link
- These commands can be stored in a Makefile and executed automatically with the `make` command

Make

- `make` is a UNIX command that uses Makefiles to build an executable from code
- A Makefile is a list of dependencies with rules to satisfy those dependencies
- The rules specify how to turn code into libraries / executables
- Important: `make` can take a `-j` argument to specify the number of simultaneous compile processes (normally the number of processor cores in your box):

```
make -j 4
```

Makefile (Don't try to copy from slides)

```
METHOD ?= opt

appname := myapp
full_appname := $(appname)-$(METHOD)

CXX := clang++
CXXFLAGS := -std=c++11

srcfiles := $(shell find . -name "*.C")
objects  := $(patsubst %.C, %-$(METHOD).o, $(srcfiles))

all: $(full_appname)

$(appname)-opt: $(objects)
	$(CXX) $(CXXFLAGS) $(LDFLAGS) -g -O3 -o $(full_appname) $(objects) $(LDLIBS)

$(appname)-dbg: $(objects)
	$(CXX) $(CXXFLAGS) $(LDFLAGS) -g -o $(full_appname) $(objects) $(LDLIBS)

%-opt.o: %.C
	$(CXX) -c $(CXXFLAGS) $(LDFLAGS) -g -O3 -MMD -MP -MF $@.d -MT $@ $< -o $@ $(LDLIBS)

%-dbg.o: %.C
	$(CXX) -c $(CXXFLAGS) $(LDFLAGS) -g -MMD -MP -MF $@.d -MT $@ $< -o $@ $(LDLIBS)

clean:
	rm -f *.o
	rm -f $(objects)
	rm -f $(full_appname)
	rm -rf *.d
	rm -f *~*

-include $(patsubst %, %-opt.o.d, $(basename $(srcfiles)))
```

Part 2

- Scope
- Pointers and References
- Dynamic Memory Allocation
- Const-ness
- Function Overloading

Scope

- A scope is the extent of the program where a variable can be seen and used.
  - local variables have scope from the point of declaration to the end of the enclosing block { }
  - global variables are not enclosed within any scope and are available within the entire file
- Variables have a limited lifetime
  - When a variable goes out of scope, its destructor is called
- Dynamically-allocated (via `new`) memory *is not* automatically freed at the end of scope

“Named” Scopes

- `class` scope

```
class MyObject
{
public:
  void myMethod();
};
```

- `namespace` scope

```
namespace MyNamespace
{
  float a;
  void myMethod();
}
```

Scope Resolution Operator

- “double colon” :: is used to refer to members inside of a named scope

```
// definition of the "myMethod" function of "MyObject"
void MyObject::myMethod()
{
  std::cout << "Hello, World!\n";
}

MyNamespace::a = 2.718;
MyNamespace::myMethod();
```

- Namespaces permit data organization, but do not have all the features needed for full encapsulation

Assignment (Prequel to Pointers and Refs)

- Recall that assignment in C++ uses the “single equals” operator:

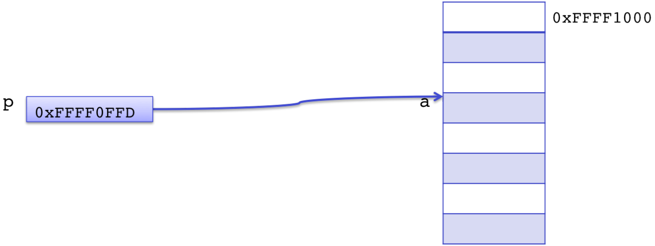
```
a = b; // Assignment
```

- Assignments are one of the most common operations in programming

- Two operands are required
  - An assignable location on the left hand side (memory location)
  - An expression on the right hand side

Pointers

- Pointers are a native type just like an `int` or `long`
- Pointers hold the location of another variable or object in memory



Pointer Uses

- Pointers are useful in avoiding expensive copies of large objects
  - Ex: Functions are passed pointers to large objects, rather than the objects themselves

- Pointers also facilitate shared memory
  - Ex: One object “owns” the memory associated with some data, and allows others objects access through a pointer

Pointer Syntax

- Declare a pointer

```
int *p;
```

- Use the “address-of” operator to initialize a pointer

```
int a;
p = &a;
```

- Use the “dereference” operator to get or set values pointed-to by the pointer

```
*p = 5;                     // set value of "a" through "p"
std::cout << *p << "\n"; // prints 5
std::cout <<  a << "\n"; // prints 5
```

Pointer Syntax, Cont'd

```
int a = 5;
int *p;       // declare a pointer
p = &a;       // set 'p' equal to address of 'a'
*p = *p + 2;  // get value pointed to by 'p', add 2,
              // store result in same location

std::cout <<  a << "\n";  // prints 7
std::cout << *p << "\n";  // prints 7
std::cout <<  p << "\n";  // prints an address (0x7fff5fbfe95c)
```

Pointers Are Powerful But Unsafe

- On the previous slide we had this:

```
p = &a;
```

- But we can do almost anything we want with `p`!

```
p = p + 1000;
```

- Now what happens when we do this?

```
*p;    // Access memory at &a + 1000
```

References to the Rescue

- A reference is an alternative name for an object [Stroustrup]
  - Think of it as an alias for the original variable

```
int a = 5;
int &r = a;  // define and initialize a ref
r = r + 2;

std::cout <<  a << "\n";  // prints 7
std::cout <<  r << "\n";  // prints 7
std::cout << &r << "\n";  // prints address of a
```

References Are Safe

- References cannot be modified

```
&r = &r + 1;    // won't compile
```

- References never start out un-initialized

```
int &r;      // won't compile
```

- Note that class declarations may contain references
- If so, initialization must occur in the constructor!
- We will see an example later on. . .

Summary: Pointers and References

- A pointer is a variable that holds a memory address to another variable
- A reference is an alternative name for an object [Stroustrup]
  - Can't create a reference without an existing object

Summary: Pointers and References

```
int b = 23
int c = 19;
```

- Pointers

```
int *iPtr;       // Declaration
iPtr = &c;
int a = b + *iPtr;
```

- References

```
int &iRef = c;     // Must initialize
int a = b + iRef;
```

Calling Conventions

- What happens when you make a function call

```
result = someFunction(a, b, my_shape);
```

- If the function changes the values inside of `a`, `b` or `my_shape`, are those changes reflected in my code?

- Is this call expensive? (Are arguments copied around?)

- C++ by default is “Pass by Value” (copy) but you can pass arguments by reference (alias) with additional syntax

Swap Example (Pass by Value)

```
void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}

int i = 1;
int j = 2;
swap (i, j);                     // i and j are arguments
std::cout << i << " " << j;     // prints 1 2
                                 // i and j are not swapped
```

Swap Example (Pass by Reference)

```
void swap(int &a, int &b)
{
  int temp = a;
  a = b;
  b = temp;
}

int i = 1;
int j = 2;
swap (i, j);                    // i and j are arguments
std::cout << i << " " << j;    // prints 2 1
                                // i and j are properly swapped
```

Dynamic Memory Allocation

- Why do we need dynamic memory allocation?
  - Data size specified at run time (rather than compile time)
  - Persistence without global variables (scopes)
  - Efficient use of space
  - Flexibility

Dynamic Memory in C++

- "`new`" allocates memory
- "`delete`" frees memory
- Recall that variables typically have limited lifetimes (within the nearest enclosing scope)
- Dynamic memory allocations do not have limited lifetimes
  - No automatic memory cleanup!
  - Watch out for memory leaks
  - Should have a "delete" for every "new".

Example: Dynamic Memory

```
int a;
int *b;

b = new int; // dynamic allocation, what is b's value?

a = 4;
*b = 5;
int c = a + *b;

std::cout << c;  // prints 9
delete b;
```

Example: Dynamic Memory Using References

```
int a;
int *b = new int;     // dynamic allocation
int &r = *b;          // creating a reference to newly created variable

a = 4;
r = 5;
int c = a + r;

std::cout << c;  // prints 9
delete b;
```

Const

- The `const` keyword is used to mark a variable, parameter, method or other argument as constant
- Typically used with references and pointers to share objects but guarantee that they won't be modified

```
{
  std::string name("myObject");
  print(name);
  ...
}

void print(const std::string & name)
{
  // Attempting to modify name here will
  // cause a compile time error

  ...
}
```

Function Overloading

In C++ you may reuse function names as long as they have different parameter lists or types. A difference only in the return type is not enough to differentiate overloaded signatures.

```
int foo(int value);
int foo(float value);
int foo(float value, bool is_initialized);
...
```

This is very useful when we get to object "constructors".

Part 3

- Type System
- Brief Intro to Using Templates
- C++ Data Structures
- Standard Template Library Containers

Static vs Dynamic Type systems

- C++ is a “statically-typed” language
- This means that “type checking” is performed during compile-time as opposed to run-time
- Python is an example of a “dynamically-typed” language

Static Typing Pros and Cons

- Pros
  - Safety - compilers can detect many errors
  - Optimization - compilers can optimize for size and speed
  - Documentation - The flow of types and their uses in expression is self documenting
- Cons
  - More explicit code is needed to convert ("cast") between types
  - Abstracting or creating generic algorithms is more difficult

Using Templates

- C++ solves the problem of creating generic containers and algorithms with "templates"

- The details of creating and using templates are extensive, but little basic knowledge is required for simple tasks

```cpp
template <class T>
T getMax (T a, T b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Using Templates

```
template <class T>
T getMax (T a, T b)
{
  return (a > b ? a : b); // "ternary" operator
}

int i = 5, j = 6, k;
float x = 3.142; y = 2.718, z;
k = getMax(i, j);        // uses int version
z = getMax(x, y);        // uses float version

k = getMax<int>(i, j);   // explicitly calls int version
```

Compiler Generated Functions

```
template <class T>
T getMax (T a, T b)
{
  return (a > b ? a : b);
}

// generates the following concrete implementations
int getMax (int a, int b)
{
  return (a > b ? a : b);
}

float getMax (float a, float b)
{
  return (a > b ? a : b);
}
```

Template Specialization

```cpp
template<class T>
void print(T value)
{
  std::cout << value << std::endl;
}

template<>
void print<bool>(bool value)
{
  if (value)
    std::cout << "true";
  else
    std::cout << "false";
  std::cout << std::endl;
}
```

```cpp
int main()
{
  int a = 5;
  bool b = true;

  print(a); // prints 5

  print(b); // prints true
}
```

C++ Standard Template Library (STL) Data Structures

vector	
list	
map	multimap
set	multiset
stack	
queue	priority_queue
deque	
bitset	
unordered_map	
unordered_set	

Using the C++ Vector Container

```cpp
#include <vector>

int main()
{
  // start with 10 elements
  std::vector<int> v(10);

  for (unsigned int i=0; i<v.size(); ++i)
    v[i] = i;
}
```

Using the C++ Vector Container

```
#include <vector>

int main()
{
  // start with 0 elements
  std::vector<int> v;

  for (unsigned int i=0; i<10; ++i)
    v.push_back(i);
}
```

Using the C++ Vector Container

```
#include <vector>

int main()
{
  // start with 0 elements
  std::vector<int> v;
  v.resize(10);  // creates 10 elements

  for (unsigned int i=0; i<10; ++i)
    v[i] = i;
}
```

More features

- Containers can be nested to create more versatile structures

```
std::vector<std::vector<Real> > v;
```

- To access the items:

```
for (unsigned int i=0; i < v.size(); ++i)
  for (unsigned int j=0; j < v[i].size(); ++j)
    std::cout << v[i][j];
```

Part 4

- Object Oriented Design
  - Data Encapsulation
  - Inheritance
  - Polymorphism

- Classes in C++
  - Syntax
  - Constructors, Destructors

Object-Oriented Definitions

- A “class” is a new data type.
- Contains data and methods for operating on that data
  - You can think of it as a “blue print” for building an object.
- An “interface” is defined as a class’s publicly available “methods” and “data”
- An “instance” is a variable of one of these new data types.
  - Also known as an “object”
  - Analogy: You can use one “blue-print” to build many buildings. You can use one “class” to build many “objects”.

Object Oriented Design

- Instead of manipulating data, one manipulates objects that have defined interfaces
- Data encapsulation is the idea that objects or new types should be black boxes. Implementation details are unimportant as long as an object works as advertised without side effects.
- Inheritance gives us the ability to abstract or "factor out" common data and functions out of related types into a single location for consistency (avoids code duplication) and enables *code re-use*.
- Polymorphism gives us the ability to write *generic algorithms* that automatically work with derived types.

Encapsulation (Point.h)

```
class Point
{
public:
  // Constructor
  Point(float x, float y);

  // Accessors
  float getX();
  float getY();
  void setX(float x);
  void setY(float y);

private:
  float _x, _y;
};
```

Constructors

- The method that is called explicitly or implicitly to build an object
- Always has the same name as the class with no return type
- May have many overloaded versions with different parameters
- The constructor body uses a special syntax for initialization called an initialization list
- Every member that can be initialized in the initialized list - should be
  - References have to be initialized here

```
   Point::Point(float x, float y):
// Point has no base class, if it did, it
// would need to be constructed first
_x(x),
_y(y)
   {}  // The body is often empty
```

Point Class Definitions (Point.C)

```
#include "Point.h"

Point::Point(float x, float y):  _x(x), _y(y) { }

float Point::getX() { return _x; }
float Point::getY() { return _y; }
void Point::setX(float x) { _x = x; }
void Point::setY(float y) { _y = y; }
```

- The data is safely encapsulated so we can change the implementation without affecting users of this type

Changing the Implementation (Point.h)

```
class Point
{
public:
  Point(float x, float y);
  float getX();
  float getY();
  void setX(float x);
  void setY(float y);

private:
  // Store a vector of values rather than separate scalars
  std::vector<float> _coords;
};
```

New Point Class Body (Point.C)

```
#include "Point.h"

Point::Point(float x, float y)
{
  _coords.push_back(x);
  _coords.push_back(y);
}

float Point::getX() { return _coords[0]; }
float Point::getY() { return _coords[1]; }
void Point::setX(float x) { _coords[0] = x; }
void Point::setY(float y) { _coords[1] = y; }
```

Using the Point Class (main.C)

```
#include "Point.h"

int main()
{
  Point p1(1, 2);
  Point p2 = Point(3, 4);
  Point p3;     // compile error, no default constructor

  std::cout << p1.getX() << "," << p1.getY() << "\n"
            << p2.getX() << "," << p2.getY() << "\n";
}
```

Outline Update

- Object Oriented Design
  - Data Encapsulation
  - Inheritance
  - Polymorphism
- Classes in C++
  - Syntax
  - Constructors, Destructors

A More Advanced Example (Shape.h)

```
class Shape {
public:
  Shape(int x=0, int y=0): _x(x), _y(y) {}  // Constructor
  virtual ~Shape() {}       // Destructor
  virtual float area()=0;  // Pure Virtual Function
  void printPosition() const;    // Body appears elsewhere

protected:
  // Coordinates at the centroid of the shape
  int _x;
  int _y;
};
```

The Derived Classes (Rectangle.h)

```
#include "Shape.h"

class Rectangle: public Shape
{
public:
  Rectangle(int width, int height, int x=0, int y=0):
    Shape(x,y),
    _width(width),
    _height(height)
  {}
  virtual ~Rectangle() {}
  virtual float area() { return _width * _height; }

protected:
  int _width;
  int _height;
};
```

The Derived Classes (Circle.h)

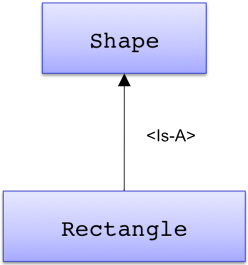
```
#include "Shape.h"

class Circle: public Shape
{
public:
  Circle(int radius, int x=0, int y=0):
    Shape(x,y), _radius(radius) {}
  virtual ~Circle() {}
  virtual float area() { return PI * _radius * _radius; }

protected:
  int _radius;
};
```

Is-A

- When using inheritance, the derived class can be described in terms of the base class
  - A `Rectangle` “is-a” `Shape`



- Derived classes are “type” compatible with the base class (or any of its ancestors)
  - We can use a base class variable to point to or refer to an instance of a derived class

```
Rectangle rectangle(3, 4);
Shape & s_ref = rectangle;
Shape * s_ptr = &rectangle;
```

Writing a generic algorithm

```
// create a couple of shapes
Rectangle r(3, 4);
Circle c(3, 10, 10);

printInformation(r);   // pass a Rectangle into a Shape reference
printInformation(c);   // pass a Circle into a Shape reference

...

void printInformation(const Shape & shape)
{
  shape.printPosition();
  std::cout << shape.area() << '\n';
}

// (0, 0)
// 12
// (10, 10)
// 28.274
```

Homework Ideas

1. Implement a new `Shape` called `Square`. Try deriving from `Rectangle` directly instead of `Shape`. What advantages/disadvantages do the two designs have?

2. Implement a `Triangle` shape. What interesting subclasses of `Triangle` can you imagine?

3. Add another constructor to the `Rectangle` class that accepts coordinates instead of height and width.