# C++ Primer

# Structs and Functions

```
struct foo {
   // data members
};

void bar();             // function declaration

void bar() {…}          // function definition
```

# Classes with Methods

```
class foo {
public:
   void bar();              // method declaration
};

void foo::bar() {}     // method definition
```

# Scope

| | |
|---|---|
| `public` | Accessible from anywhere |
| `private` | Accessible only from within the class |
| `protected` | Accessible only from within the class and any derived class |

# Scope

```
class foo {
public:
    int accessible_to_everyone;
private:
    int accessible_to_just_foo;
};
```

# Pass-by-Reference

In C, you passed all parameters by value. The values specified in the arguments of a function call were copied. You had to use pointers in order to change a variable's value.

In C++, you can additionally pass parameters by **reference**.

# Pass-by-Reference

Suppose I want to change the value of some `int` variable:

```
void foo(int *c) {  // pass by value
    *c = 1;
}
void bar(int &c) {  // pass by reference
    c = 1;
}
```

# Pass-by-Reference

You can access the variables directly through pass by reference.

**It is very important to note that you can only pass variables!** For example, you can't use the `bar` function on the previous page and call `bar(1)`.

# const Methods

The `const` keyword can restrict which instances of a class can call a specific method.  This prevents certain methods from changing the data of a constant object.

# const Methods

Suppose in class `foo` we have the functions:

```
void foo::bar();
void foo::baz() const;
```

`foo f1` can call `bar()` and `baz()`.

`const foo f2` can only call `baz()`.

# this

In C++, `this` is a pointer to an instance of the class, hence "this."

# this

Suppose we have the following class:

```
class Pair {
public:
   Pair(int x, int y);
   int x, y;
};
```

# this

The function takes arguments of the same name as the class members.  Use `this` to specify which one to use.

```
Pair::Pair(int x, int y) {
    this->x = x;
    this->y = y;
}
```

# this

You can also use `this` even if the names are different, although it is not necessary.

```
Pair::Pair(int a, int b) {
    this->x = a;
    this->y = b;
}
```

# Constructors and Destructors

When an object is created, the **constructor** is called.  This will initialize the object.

When the scope of an object is exited, the **destructor** is automatically called on the object.  This will perform any needed cleanup.

# Constructors and Destructors

```
class foo {
    int * _a;
public:
    foo();              // empty constructor
    foo(int *a);        // another constructor
    ~foo();             // destructor
};
```

# Constructors and Destructors

You can initialize members of the class such as below

```
foo::foo() : _a(NULL) {}
foo::foo(int * a) : _a(a) {}

foo::~foo() { if (_a) delete _a; }
```

# Operator Overloading

Suppose we have a class `vec3` that represents a mathematical 3D vector.

It makes sense for us to add two `vec3` objects, but to C++, it doesn’t directly understand how to add two arbitrary object types.

**Operator overloading** lets us define a series of operations on objects.

# Operator Overloading

```
vec3 operator+(const vec3& v);


vec3 vec3::operator+(const vec3& v) {
   return vec3(_x+v.x, _y+v.y, _z + v.z);
}
```

Addition is just one of the several operators you can overload.

# Templates

In C, `void *` is used to handle arbitrary types.

In C++, we can define a **template** type to take the place of any type.

# Templates

This is how a template function is declared:

```
template <typename T>
void swap(T& a, T& b) {
    T c = a;
    a = b;
    b = c;
}
```

# Templates

This is how templates are used in code:

```
int a = 2, b = 3;
char c = ‘c’, d = ‘d’;
swap<int>(a, b);
swap<char>(c, d);
```

# Templates

Since a template can be used for any arbitrary type, C++ compiles a separate object file for each type used.

Template classes should reside only in .h files, not split into .h and .cc files.