the gamedesigninitiative at cornell university

C++: Classes

Classes in C++

Declaration

- Like a Java interface
  - Fields, method prototypes
  - Put in the header file

```
class AClass {
private:  // All privates in group
   int field;
   void helper();

public:  // All publics in group
    AClass(int field);  // constructor
    ~AClass();          // destructor

}; // SEMICOLON!
```

Implementation

- Body of all of the methods
  - Preface method w/ class
  - Put in the cpp file

```
void AClass::helper() {
   field = field+1;
}
AClass::AClass(int field) {
   this->field = field;
}
AClass::~AClass() {
   // Topic of later lecture
}
```

Classes in C++

Declaration

- Like a Java interface
  - Fields, method prototypes
  - Put in the header file

```
class AClass {
private:  // All privates in group
   int field;
   void helper();

public:  // All publics in group
    AClass(int field);  // constructor
    ~AClass();          // destructor

}; // SEMICOLON!
```

Implementation

- Body ... ethods
  - Pre ... class
  - Pu ...

Class name acts like a namespace

```
void AClass::helper() {
   field = field+1;
}
AClass::AClass(int field) {
   this->field = field;
}
AClass::~AClass() {
   // Topic of later lecture
}
```

Stack-Based vs. Heap Based

Stack-Based

- Object assigned to local var
  - Variable is NOT a pointer
  - Deleted when variable deleted
  - Methods/fields with period (.)
- Example:

```
void foo() {
   Point p(1,2,3); // constructor
   ...
   // Deleted automatically
}
```

Heap-Based

- Object assigned to pointer
  - Object variable is a pointer
  - Must be manually deleted
  - Methods/fields with arrow (->)
- Example:

```
void foo() {
   Point* p = new Point(1,2,3);
   ...
   delete p;
}
```

Stack-Based vs. Heap Based

Stack-Based

- Object assigned to local var
  - Variable is NOT a pointer
  - Deleted when variable deleted
  - Methods/fields with dot (.)
- Example:

```
void foo() {
   Point p(1,2,3); // constructor
   ...
   // Deleted automatically
}
```

Heap-Based

- Object assigned to pointer
  - Object variable is a pointer
  - Must be manually deleted
  - Methods/fields with arrow (->)
- Example:

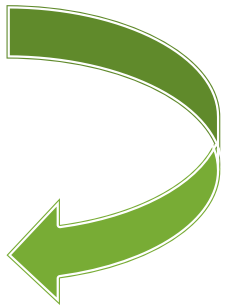
```
void foo() {
   Point* p = new Point(1,2,3);
   ...
   delete p;
}
```

Also if pointer to stack-based

Returning a Stack-Based Object

- Do not need heap to return
  - Can move to calling stack
  - But this must *copy* object
- Need a special constructor
  - Called **copy constructor**
  - Takes *reference* to object
  - C++ calls automatically
- Is this a good thing?
  - Performance cost to copy
  - Cheaper than heap if small

```
Point foo_point(float x) {
    Point p(x, x);
    return p; // Not an error
}
```



Calls

```
Point::Point(const Point& p) {
    x = p.x;
    y = p.y;
    z = p.z;
}
```

Returning a Stack-Based Object

- Do not need heap to return
  - Can move to calling stack
  - But this must *copy* object
- Need a special constructor
  - Called
  - Take
  - C++
- Is this a good thing?
  - Performance cost to copy
  - Cheaper than heap if small

```
Point foo_point(float x) {
    Point p(x, x);
    return p; // Not an error
}
```

```
..Point(const Point& p) {
    x = p.x;
    y = p.y;
    z = p.z;
}
```

What happens when you return a string

Copy vs Move Constructor

Copy Constructor

- **Point(const Point& p)**
  - *Copies* the object p
  - Object p can still be used
- Does not require C++11
- Same as move if
  - Only has primitive fields
  - Has no allocated resources
- **Example**: cugl::Vec2

Move Constructor

- **Point(Point&& p)**
  - *Takes resources* from p
  - Object p not safe to use
- Requires C++11
- Better than copy if
  - Object is a return value
  - Object has fields in heap
- **Example**: cugl::Poly2

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

Method cannot modify the object passed

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

Method cannot modify the object passed

Method cannot modify any object fields

The Many Meanings of const

- In C++, it is common to see something like:

```
const Point& foo(const Point& p) const;
```

Caller cannot modify the object returned

Method cannot modify the object passed

Method cannot modify any object fields

- Believe it or not, these are not the only consts!
  - But these are generally the only ones to use
  - See online tutorials for more

Inlining Method Definitions

- Can implement in .h file
  - Define methods Java-style
  - Will **inline** the methods
- Less important these days
  - Good compilers inline
  - Function overhead is low
- Only two good applications
  - Getters and setters
  - Overloaded operators
  - Use this sparingly

```
class Point {
private:
    float x;
    float y;

public:

    Point(float x, float y, float z);

    float getX() const { return x; }

    void setX(float x) {
        this->x = x;
    }

    ...
};
```

Operator Overloading

- Change operator meaning
  - Great for math objects: +, *
  - But can do any symbol: ->
- Method w/ "operator" prefix
  - Object is always on the left
  - Other primitive or const &
- Right op w/ **friend** function
  - Function, not a method
  - Object explicit 2nd argument
  - Has full access to privates

```
Point& operator*=(float rhs) {
   x *= rhs; y *= rhs; z *= rhs;
   return *this;
}
```

```
Point operator*(const float &rhs) const {
   return (Point(*this)*=rhs);
}
```

```
friend Point operator* (float lhs,
                        const Point& p) {
   return p*lhs;
}
```

Subclasses

- Subclassing similar to Java
  - Inherits methods, fields
  - Protected limits to subclass
- Minor important issues
  - Header must import parent
  - super() syntax very different
  - See tutorials for more details
- Weird C++ things to avoid
  - No **multiple inheritance**!
  - No **private subclasses**

```
class A {
public:
   float x;

   A(float x) { this->x = x; }
   ...
};

class B : public A {
public:
   float y;

   B(float x, float y) : A(x) {
       this->y = y;
   }
   ...
};
```

Subclasses

- Subclassing similar to Java
  - Inherits methods, fields
  - Protected limits to subclass
- Minor important issues
  - Header must import parent
  - super() syntax very different
  - See tutorials for more details
- Weird C++ things to avoid
  - No **multiple inheritance**!
  - No **private subclasses**

```
class A {
public:
   float x;

   A(float
   ...
};

class B : public A {
public:
   float y;

   B(float x, float y) : A(x) {
      this->y = y;
   }
   ...
};
```

Weird things if you make it private

Like Java call to super

C++ and Polymorphism

- Polymorphism was a major topic in CS 2110
  - Variable is reference to interface or base class
  - Object itself is instance of a specific subclass
  - Calls to methods are those implementated in subclass
- **Example**:
  - `List<int> list = new LinkedList<int>();`
  - `list.add(10); // Uses LinkedList implementation`
- This is a major reason for using Java in CS 2110
  - C++ does not *quite* work this way

C++ and Polymorphism

- Cannot change stack object
  - Variable assignment copies
  - Will lose all info in subclass
- Only relevant for pointers
  - C++ uses static pointer type
  - Goes to method for type
- **Why did they do this?**
  - No methods in object data
  - Reduces memory lookup
  - But was it worth it?

```
class A {
public:
   int foo() {return 42;}
};

class B : public A {
public:
   int foo() {return 9000; }
};

B* bee = new B();

x = bee->foo();    // x is 9000

A* aay = (A*)bee;

y = aay->foo();    // y is 42!!!
```

Fixing C++ Polymorphism

- Purpose of virtual keyword
  - Add to method in base class
  - Says "will be overridden"
- Use optional in subclass
  - Needed if have subsubclass
  - Or if not further overridden
- Hard core C++ users hate
  - Causes a performance hit
  - Both look-up and storage
  - But not a big deal for you

```
class A {
public:
    virtual int foo() {return 42;}
};

class B : public A {
public:
    int foo() override {return 9000; }
};

B* bee = new B();

x = b->foo();    // x is 9000

A* aay = (A*)bee;

y = a->foo();    // y is 9000
```

Templates: Like Generics But Not

Usage

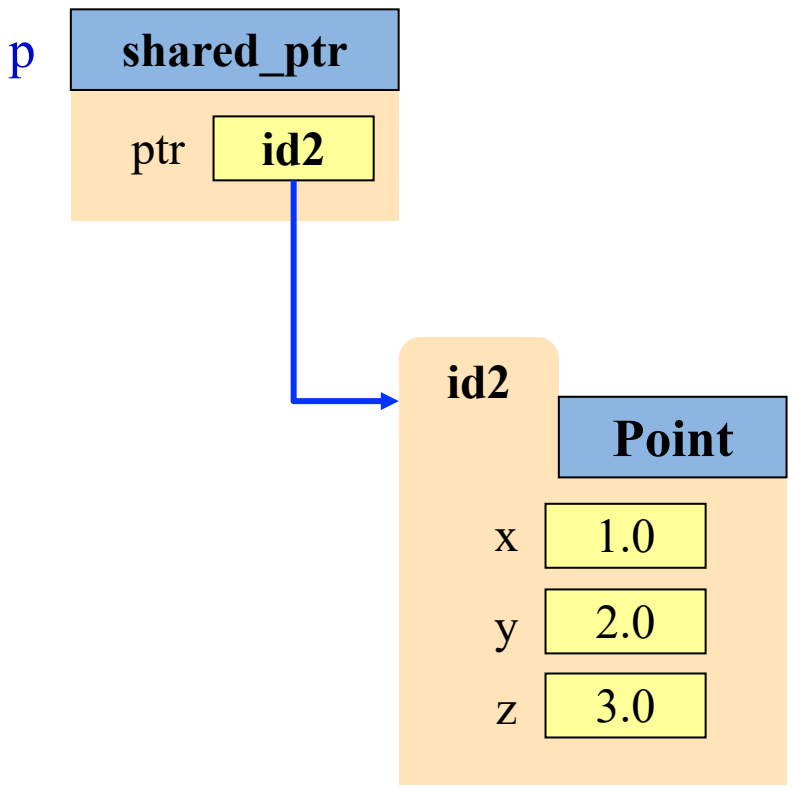
- Class has type parameter <>
  - Add type at allocation time
  - `v = new std::vector<int>();`
- Required in the C++ STL
  - `std::vector`, `std::deque`
  - `std::unordered_map`
- Also in our asset manager
  - Associate a loader with type
  - `amgr->attach<Font>(loader);`

Definition

- Preface class with template
- ```
  template <class T>
  class A{
     T x
     const T& getX() { return x;}
     void setX(T v)    { x = v;}
  };
  ```
- No .cpp file! Only .h
  - Import header to use class
  - Compiled at instantiation
  ```

Application: Smart Pointers

- Class that holds a pointer
  - Tracks the pointer usage
  - Can delete pointer for you
  - Access pointer with `get()`
- Type is *templated* type
  - `std::shared_ptr<Point>`
  - `std::shared_ptr<Font>`
- **This requires C++11**
  - Which you should use…
  - Check your IDE settings



Smart Pointers and Allocation

Heap Allocation

```
void func() {
  Point* p = new Point(1,2,3);
  ...
  delete p;
}
```

- Must remember to delete
- Otherwise will *memory leak*

Smart Pointer

```
void func() {
  shared_ptr<Point> p;
  p = make_shared<Point>(1,2,3);
  ...
}
```

- Deletion is not necessary
- Sort-of garbage collection

Smart Pointers and Allocation

Heap Allocation

```
void func() {
  Point* p = new Point(1,2,3);
  ...
  delete p;
}
```

- Must remember to delete
- Otherwise will *memory leak*

Smart Pointer

```
void func() {
  shared_ptr<Point> p
                    t>(1,2,3);
}
```

- Deletion is not necessary
- Sort-of garbage collection

More on this in Memory Lectures

Typecasting and Smart Pointers

Normal Pointers

```
B* b;    // The super class
A* a;    // The subclass
```

Acceptable:

```
b = new B();
a = (A*)b;
```

Better:

```
b = new B();
a = dynamic_cast<A*>(b);
```

Smart Pointers

```
shared_ptr<B> b;    // Contains B*
shared_ptr<A> a;    // Contains A*
```

Bad:

```
b = make_shared<B>();
a = (shared_ptr<A>)b;
```

Good:

```
b = make_shared<B>();
a = dynamic_pointer_cast<A>(b);
```

Typecasting and Smart Pointers

Normal Pointers

```
B* b;   // The super class
A* a;   // The subclass
```

Acceptable:

```
b =
a =
```

Better:

```
b = new B();
a = dynamic_cast<A*>(b);
```

Smart Pointers

```
shared_ptr<B> b;   // Contains B*
shared_ptr<A> a;   // Contains A*
```

Bad:

```
                                    )b;
```

Polymorphism is messy on Smart Pointers

Good:

```
b = make_shared<B>();
a = dynamic_pointer_cast<A>(b);
```

Closures: C++ Lambda Functions

- **Type**: std::function<T>
  - Type is function signature
  - Allows function in variable
  - **Example Declaration**:
    std::function<void(int)> a;
- Important for callbacks
  - **Example**: Collision listener
  - See WorldController class
- **This requires C++11**
  - Which you should use…
  - Check your IDE settings

Variable Capture Rules

```
int x = 0;

std::function<int(int)> a = [=](int y)
      { return x+y; };


std::function<int(int)> b =[&](int y)
      { return x+y; };

x = 5;

int y = a(4);
int z = b(4);
```

Closures: C++ Lambda Functions

- **Type**: std::function<T>
  - Type is function signature
  - Allows function in variable
  - **Example Declaration**:
    std::function<void(int)> a;
- Important for callbacks
  - **Example**: Collision listener
  - See WorldController class
- **This requires C++11**
  - Which you should use…
  - Check your IDE settings

Variable Capture Rules

```
int x = 0;

std::function<int(int)> a = [=](int y)
    { return x+y; };

std::function<int(int)> b =[&](int y)
    { return x+y; };

x = 5;

int y = a(4);
int z = b(4);
```

free variable

Closures: C++ Lambda Functions

- **Type**: std::function<T>
  - Type is function signature
  - Allows function in variable
  - **Example Declaration**:
    std::function<void(int)> a;
- Important for callbacks
  - **Example**: Collision listener
  - See WorldController class
- **This requires C++11**
  - Which you should use…
  - Check your IDE settings

Variable Capture Rules

```
int x = 0;

std::function<int(int)> a = [=](int y)
    { return x+y; };

std::function<int(int)> b =[&](int y)
    { return x+y; };

x = 5;

int y = a(4);   // Value is 4
int z = b(4);   // Value is 9
```

copies x

free variable

references x

Summary

- C++ has a lot of similarities to Java
  - Java borrowed much of its syntax, but "cleaned it up"
- Memory in C++ is a lot trickier
  - Anything allocated with `new` must be deleted
  - C++ provides many alternatives to avoid use of `new`
- Classes in C++ have some important differences
  - Can be copied between stacks if written correctly
  - C++ supports operator overloading for math types
  - C++ needs special keywords to support polymorphism