CSE 303
Lecture 21

Classes and Objects in C++

slides created by Marty Stepp
http://www.cs.washington.edu/303/

C++ classes

- class declaration syntax (in `.h` file):

```
class name {
    private:
        members;

    public:
        members;
};
```

- class member definition syntax (in `.cpp` file):

```
returntype classname::methodname(parameters) {
    statements;
}
```

  - unlike in Java, any `.cpp` or `.h` file can declare or define any class (though the convention is still to put the Foo class in `Foo.h/cpp`)

A class's .h file

```
#ifndef _POINT_H
#define _POINT_H

class Point {
    private:
        int x;
        int y;  // fields

    public:
        Point(int x, int y);   // constructor
        int getX();            // methods
        int getY();
        double distance(Point& p);
        void setLocation(int x, int y);
};

#endif
```

.h file still uses `#ifndef` to guard against multiple inclusion

(many compilers also support an alterative called `#pragma once`)

private/public members are grouped into sections

MUST have a semicolon at end of class, or:

```
Point.cpp:4: error: new types may not be defined in a return type
Point.cpp:4: error: return type specification for constructor invalid
```

A class's .cpp file

```
#include "Point.h"                  // this is Point.cpp

Point::Point(int x, int y) {        // constructor
    this->x = x;
    this->y = y;
}

int Point::getX() {
    return x;
}

int Point::getY() {
    return y;
}

void Point::setLocation(int x, int y) {
    this->x = x;
    this->y = y;
}
```

each member is defined on its own, using :: scope operator to indicate class name

`this` is an unmodifiable <u>pointer</u> to the current object (of type `const Point*`)

works a lot like Java's `this`, but cannot be used to invoke a constructor

using `this->` is optional unless names conflict

Exercise

- Make it so a `Point` can be constructed with no x/y parameter.
  - If no x or y value is passed, the point is constructed at (0, 0).

- Write a `translate` method that shifts the position of a point by a given dx and dy.

Exercise solution

```
// Point.h
    public:
        Point(int x = 0, int y = 0);


// Point.cpp
void Point::translate(int dx, int dy) {
    setLocation(x + dx, y + dy);
}
```

More about constructors

- **initialization list**: alternate syntax for storing parameters to fields
  - supposedly slightly faster for the compiler

```
class::class(params) : field(param), ..., field(param) {
    statements;
}
```

```
Point::Point(int x, int y) : x(x), y(y)  {}
```

- if you don't write a constructor, you get a default () constructor
  - initializes all members to 0-equivalents (0.0, null, false, etc.)
- if your class has multiple constructors:
  - it doesn't work to have one constructor call another
  - but you can create a common init function and have both call it

Constructing objects

- client code creating stack-allocated object:

```
type name(parameters);
```

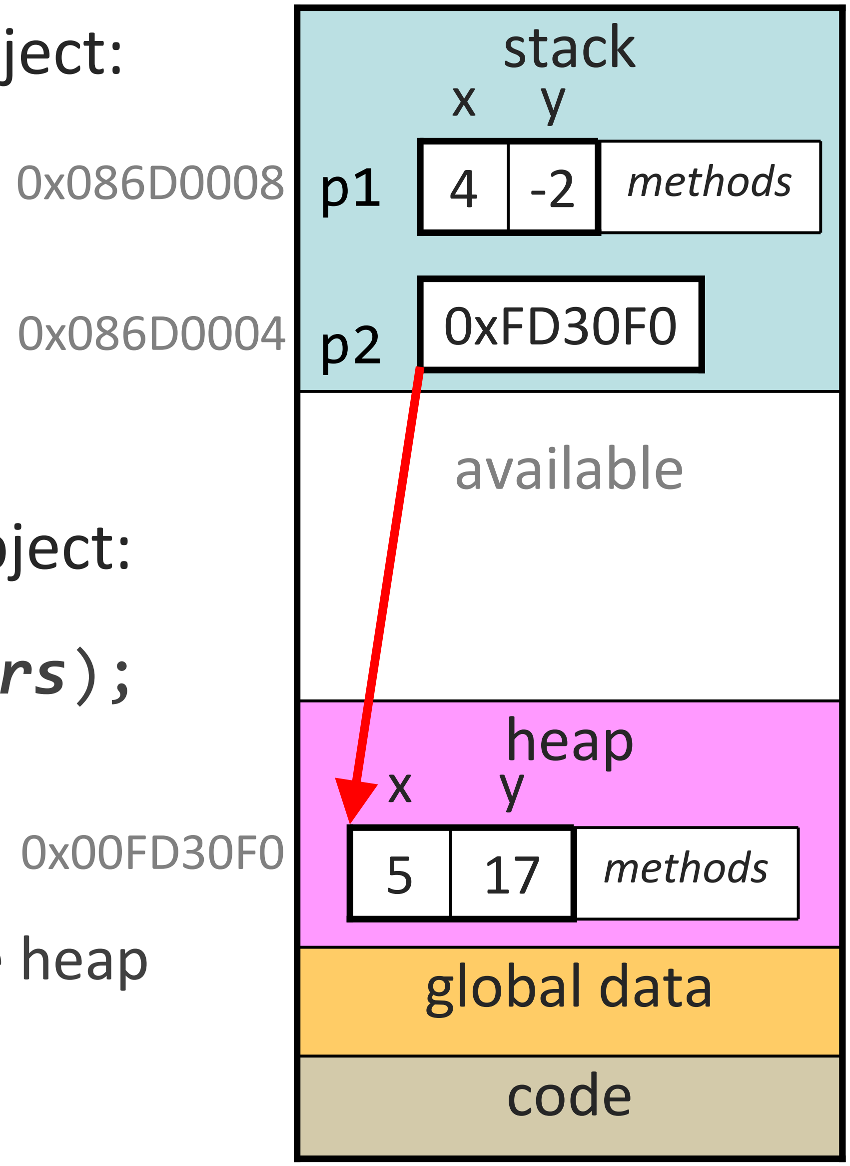
```
Point p1(4, -2);
```

- creating heap allocated (pointer to) object:

```
type* name = new type(parameters);
```

```
Point* p2 = new Point(5, 17);
```

  - in Java, all objects are allocated on the heap
  - in Java, all variables of object types are references (pointers)



A client program

```
// use_point.cpp
// g++ -g -Wall -o use_point Point.cpp use_point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point p1(1, 2);
    Point p2(4, 6);
    cout << "p1 is: (" << p1.getX() << ", "
         << p1.getY() << ")" << endl;       // p1 is: (1, 2)
    cout << "p2 is: (" << p2.getX() << ", "
         << p2.getY() << ")" << endl;       // p2 is: (4, 6)
    cout << "dist : " << p1.distance(p2) << endl;
    return 0;                               // dist : 5
}
```

Client with pointers

```
// use_point.cpp
// g++ -g -Wall -o use_point Point.cpp use_point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point* p1 = new Point(1, 2);
    Point* p2 = new Point(4, 6);
    cout << "p1 is: (" << p1->getX() << ", "
         << p1->getY() << ")" << endl;      // p1 is: (1, 2)
    cout << "p2 is: (" << p2->getX() << ", "
         << p2->getY() << ")" << endl;      // p2 is: (4, 6)
    cout << "dist : " << p1->distance(*p2) << endl;
    delete p1;                              // dist : 5
    delete p2;   // free
    return 0;
}
```

Stack vs. heap objects

- which is better, stack or pointers?
  - if it needs to live beyond function call (e.g. returning), use a pointer
  - if allocating a whole bunch of objects, use pointers

- "primitive semantics" can be used on objects
  - stack objects behave use primitive value semantics (like ints)

- `new` and `delete` replace `malloc` and `free`
  - `new` does all of the following:
    - allocates memory for a new object
    - calls the class's constructor, using the new object as `this`
    - returns a pointer to the new object
  - must call `delete` on any object you create with `new`, else it leaks

Implicit copying

Why doesn't this code change p1?

```
int main() {
    Point p1(1, 2);
    cout << p1.getX() << "," << p1.getY() << endl;
    example(p1);
    cout << p1.getX() << "," << p1.getY() << endl;
    return 0;
}

void example(Point p) {
    p.setLocation(40, 75);
    cout << "ex:" << p.getX() << "," << p.getY() << endl;
}

// 1,2
// ex:40,75
// 1,2
```

Object copying

- a stack-allocated object is *copied* whenever you:
  - pass it as a parameter `foo(p1);`
  - return it `return p;`
  - assign one object to another `p1 = p2;`

- the above rules do not apply to pointers
  - object's state is still (shallowly) copied if you dereference/assign

```
*ptr1 = *ptr2;
```

- You can control how objects are copied
  by redefining the = operator for your class (ugh)

Objects as parameters

- We generally don't pass objects as parameters like this:

```
double Point::distance(Point p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

  - on every call, the entire parameter object p will be copied
  - this is slow and wastes time/memory
  - it also would prevent us from writing a method that modifies p

References to objects

- Instead, we pass a reference or pointer to the object:

```
double Point::distance(Point& p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

  - now the parameter object p will be shared, not copied
  - are there any potential problems with this code?

const object references

- If the method will not modify its parameter, make it `const`:

```
double Point::distance(const Point& p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

  - the distance method is promising not to modify p
    - if it does, a compiler error occurs
    - clients can pass `Point`s without fear that their state will be changed
  - which of these lines would be legal inside `distance`?

```
Point p2 = p;
Point& p3 = p;
```

const methods

- If the method will not modify the object itself, make it `const`:

```
double Point::distance(const Point& p) const {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

  - a `const` after the parameter list signifies that the method will not modify the object upon which it is called (`this`)
    - helps clients know which methods are / aren't mutators
    - helps compiler optimize method calls
  - a `const` reference only allows `const` methods to be called on it
    - we could call `distance` on p, but not `setLocation`

const and pointers

- **`const`** `Point* p`
  - p points to a `Point` that is `const`; cannot modify that `Point`'s state
  - can reassign p to point to a different `Point` (as long as it is `const`)

- `Point*` **`const`** `p`
  - p is a constant pointer; cannot reassign p to point to a different object
  - can change the `Point` object's state by calling methods on it

- **`const`** `Point*` **`const`** `p`
  - p points to a `Point` that is `const`; cannot modify that `Point`'s state
  - p is a constant pointer; cannot reassign p to point to a different object

(This is not one of the more beloved features of C++.)

Pointer, reference, etc.?

- How do you decide whether to pass a pointer, reference, or object?

- Some design principles:
  - Minimize the use of object pointers as parameters.
    (C++ introduced references for a reason. They are safer and saner.)
  - Minimize passing objects by value, because it is slow, it has to copy the entire object and put it onto the stack, etc.
  - In other words, pass objects as references as much as possible.
    - Though if you really want a copy, pass it as a normal object.
  - Objects as fields are usually pointers (why not references?).
  - If you are not going to modify an object, declare it as `const`.
  - If your method returns a pointer/object field that you don't want the client to modify, declare its return type as `const`.