Writing new classes

```c
struct point
{
    int x;
    int y;
    double distance_to_origin()
    {
        return sqrt ( (double) x*x + y*y );
    }
};
```

Separating declarations and definitions

```c
// point.h
class Point
{
    int x;
    int y;
    double distance_to_origin();
};

// point.cpp
#include "point.h"
double Point::distance_to_origin()
{
    return sqrt ( (double) x*x + y*y );
}
```
Scopes

- In C, all functions are in the global scope.
- Now, blocks have their own scopes.
- Scopes are all unnamed.

- In C++, classes provide a scope nested inside the global scope.
- Class scopes can be referred to by name using the :: operator.
- Member functions are all within the class scope, and have a local scope.
- Blocks also have a scope.

Where’s the object?

- “Object-style” C function calls make the object explicit:
  ```c
  char* s = "test";
  int len = strlen(s);
  ```
  - Likewise, the function definition has an explicit object:
    ```c
    int strlen( char* str)
    {
      // use argument str here
    }
    ```

- C++ function calls also make the object explicit:
  ```cpp
  std::string s = "test";
  int len = s.length();
  ```
  - C++ function definitions leave object **implicit**:
    ```cpp
    int string::length()
    {
      //use member names here, but where’s the string?
    }
    ```
The “this” pointer

Suppose we have a function that pretty-prints Point objects:

```cpp
prettyPrintPoint(Point* p);
```

We want to use this function from a member function of class Point. How can we do that? Solution: inside any member function there is an implicit local variable named `this`, which is a pointer to the object.

```cpp
void Point::dumpOut()
{
    prettyPrintPoint(this);
}
```

```cpp
Point myPt;
myPt.dumpOut();  // implicitly, this == &myPt
```

The trouble with scopes

- Increasing the nesting of scopes invites shadowing
- Also makes code harder to understand

Some partial solutions:
- Avoid using global scope when possible
- Use a naming convention to indicate the scope of variables
- Also note class naming convention

```cpp
class Point
{
    int m_x;  // Indicate member with m_ prefix
    int m_y;
    ...
};
```
Static data members

- Class members may be declared static
- Static data members belong to the class, not to any object...
- ...so they don’t take up space in objects
- All objects share the same variable, like a global, but in the class scope

class Point
{
    static int sm_xOrigin;
    static int sm_yOrigin;
    int m_x;
    int m_y;
    ...
};
// and, in point.cpp:
int Point::sm_xOrigin = 0;

Static member functions

- Static member functions also belong to the class
- Objects can be used to call these functions (but this is confusing)
- Static member functions can be called without an object
- Static member functions don’t have a this pointer
- And they can’t refer to non-static members in any way.
Static member functions

class Point
{
    static int sm_xOrigin;
    static int sm_yOrigin;
    static void setOrigin( int x, int y );
...
};

void Point::setOrigin( int x, int y )
{
    sm_xOrigin = x;
    sm_yOrigin = y;
}

Calling static member functions

// Correct, but confusing, suggests p is involved somehow
Point p;
p.setOrigin( 0, 0 );

// Better
Point::setOrigin( 0, 0 );
Other things that can go into class scopes

class Point
{
    typedef int coord;
    enum quadrant { FIRST, SECOND, THIRD, FOURTH };
    class silly { ... };
    ...
};

• Using items in class scope:
Point::coord c1 = 4;
Point::quadrant q1 = Point::FIRST;
Point::silly s1;

Digression: Overloading functions

• In C, function names must be unique across all modules.
• In C++, function names may be reused, provided the arguments allow disambiguation.
• Different classes may have the same member function name, since the type of the object allows the compiler to disambiguate.
• Global functions can also be overloaded...
• …as can functions within the same class
• class Point
{                  
    double distance_to_point(point other);
    double distance_to_point(int x, int y);
    ...
};
Resolving overloaded function calls

- Most cases are unambiguous.
- Some ambiguities can be resolved by the compiler.
- If not, the call is an error.
- Resolution tries (in order): Exact match, promotion, conversion

```c
void print(char c);
void print(int i);
void print(char* s);

print('g');      // exact match with char
print(3);        // exact match with int
short x; print(x);  // promote to int
print(3.4);      // convert to int (!)
print("OK");    // exact match with char*
print(NULL);     // exact match with int (!!!)
```

Access control

- Encapsulation means we want to restrict use to a chosen interface
- ...and not allow use based on implementation
- Restrictions should be enforced by compiler...
- …by making violations become compiler errors
- Prevent accidents, not malice
- New keywords: `public` and `private`
Access control declarations

// point.h
class Point
{
private:
    int m_x;
    int m_y;
    void reset_coordinates();
public:
    double distance_to_origin();
};

• Use as many access specifiers as you want, in any order.
• Each one is in effect until the next one, or the end of the declaration
• Specifiers go only in declarations, nowhere else.

Access control rules (simple version)

• Member functions can access any other members (functions or data)
• Non-members (i.e. clients) can access only public members
• Overload resolution happens before access checking
Access control examples

Point p;
p.m_x = 4;  // illegal, m_x is private
int i = p.m_x // illegal
int* ip = &(p.m_x); // illegal

p.reset_coordinates(); // illegal, fn is private
double d = p.distance_to_origin();  // Ok, public

Point::distance_to_origin()
{
    // Access to private members m_x,m_y allowed
    // because we are in a member function.
    return sqrt ( (double) m_x*m_x + m_y*m_y );
}

Access control examples, cont.

•Member access includes members of other objects of our class

Point::distance_to_point(point other)
{
    // Access to private members m_x,m_y allowed
    // because we are in a member function.
    double x_diff = m_x - other.m_x;
    double y_diff = m_y - other.m_y;
    return sqrt( x_diff*x_diff + y_diff*y_diff );
}
struct vs. class

• “Default” access control is public for structs, private for classes.

Common practice is to:
• tag *everything* in classes,
• use structs as if using C (i.e. forget tags, everything public, no functions),
• if you find your structs need functions, they should probably become classes

Making exceptions - friends

• Sometimes two classes need to work together
• Sometimes a function needs to be global, but also needs to have access to internal data

class Point
{
    friend class screen;
    friend void printPoint(Point* p);
    friend void circle::setRadius(Point* p);
    ...
};
Rules of friendship

- Friends get full access to private members
- Friendship is a one-way street (i.e. friendship is not symmetrical)
- The friend of my friend is still a stranger to me (i.e. friendship is not transitive)

Suggestions:
- Prefer making functions friends rather than classes
- Declare friends at the top of class declarations
- Don’t make friends lightly

OOP Philosophy

Encapsulation is a Good Thing

- Makes code more modular by supporting abstraction
- Decreases time spent maintaining code
- Increases likelihood of reusability
- Makes code easier to understand
Object Oriented Programming in C++

Assignment 1 -- Due Feb. 18

Download the hw1.cpp driver file from the website (it implements main())

Part 1

1. Create a class `IntStack` to keep a stack of integers. The initial implementation of the stack should be as an array of 100 integers.
2. Write member function `void reset()` which will reset the stack, i.e. make it empty.
   Write member functions `void push (int n)` and `int pop()`.
3. Overload push to take in an array of integers and push them onto the stack: `void push (int a[], size_t array_size)` . Push a[0] first and then a[1] and so forth until array_size.
4. Overload pop to return n integers in an array: `void pop (int a[], size_t n)` . So if n = 2, then a[0] = top of stack, and a[1] = next on stack.
   The effect of the following code should be to reverse the integers in arr:
   ```
   int arr[5] = { 1, 2, 3, 4, 5};
   push (arr, 5);
   pop (arr, 5);
   ```
5. Write member fn. `bool is_full()` to return if the stack is full or not. Write member fn. `bool is_empty()` to return if the stack is empty or not.
6. Write a global function `void printStack (IntStack*)` which will print the integers on the stack to cout. Print each integer on a separate line. Make it a friend of `IntStack` so it can access the stack without popping.

Part 2

7. Modify the internal implementation of `IntStack` so that it dynamically allocates the array when it is constructed. Add a default constructor that allocates an array of size 100. Add a destructor that deallocates this memory.
8. Write a second constructor that takes a single integer specifying the stack size. Assume that memory allocation will succeed no matter how large the requested size. If the requested initial size is 0, it should still be possible to push values onto the stack. Note that instead of writing a second constructor, you can instead provide just one constructor with a default argument; it's up to you.
9. Write a member fn. `void doubleCapacity()` which will double the size of the internal array and keep all the things on the stack intact. Do not worry about handling out of memory conditions. Write members `size_t size()` and `size_t capacity()` which return the current number of items on the stack and the current size of the internal array.
10. Modify `push()` so that it automatically doubles the size of the stack if necessary to hold the item being pushed.
Please do not check for boundary/error conditions (like trying to pop off an empty stack). Later on in this course we will address C++ error handling techniques. Your project should consist of three files: instack.h, intstack.cpp and hw1.cpp. You will not write hw1.cpp. Rather, download it from the above link and add it to your project. Do not modify this file!

Hand in your final IntStack source code (we don't need hw1.cpp). Don't hand in any code from earlier steps.

**Grading Policy for Assignment 1**

This assignment is worth 12 points, as follows:

- produces correct output = 3
- no memory leaks = 2
- no bugs undetected by driver = 1
- code reuse = 1
- code clarity/efficiency = 4 (code reads easily; implementations are short and to the point)
- grader discretion = 1