Constructors and Destructors

Constructors

We can now create classes in C++, but we haven't yet talked about writing constructors. You can write a constructor for a C++ class in much the same way you do for a Java class. Here is the format in the class declaration (the constructor declaration is in red):

```
class Name {
     private:
          //instance variables
     public:
          Name(args);
          //function prototypes
};
```

The implementation of the constructor looks like this:

```
Name::Name(args) {
     //code, usually initializes instance variables
}
```

Just like function implementations, the constructor implementation is considered part of the class. Thus, we can access all class members (including private instance variables) inside the constructor.

Example

Suppose we want to write a class to represent a rectangle. Here's a possible class declaration:

```
class Rectangle {
     private:
          int length;
          int width;
     public:
          Rectangle(int, int);
          int area(void);
          int perimeter(void);
};
```

Notice that the `Rectangle` constructor takes two integers, so it can initialize the two instance variables. Here is the implementation of the `Rectangle` class:

```
Rectangle::Rectangle(int l, int w) {
     length = l;
     width = w;
```

```
}

int Rectangle::area(void) {
     return length*width;
}

int perimeter(void) {
     return 2*length + 2*width;
}
```

Creating objects with constructors
Now that we have constructors, we need to create objects in a slightly different way. Before, we created objects like this:

```
ClassName objectName;
```

which was very similar to creating variables of other types. In fact, this is now how we will create objects with either **no constructor** or with a **no-argument constructor**.

Here is how we will create objects now:

```
ClassName objectName;       //no constructor or no-argument constructor
ClassName objectName(args);     //constructor with arguments
```

For example, here is how we would create a 3x4 rectangle object:

```
Rectangle rect(3, 4);
```

Now we can use this object just like other objects we've created. For example, we can get the area of our rectangle:

```
int area = rect.area();
```

Default constructors
As we have already seen, we can write classes in C++ without providing a constructor. (The same is true in Java.) If you do not provide a constructor, the following constructors will be provided for you:

1) **Default constructor** – takes no arguments and does nothing
2) **Copy constructor** – takes another object of the same type and copies the values of the instance variables

If you provide ANY constructor, NEITHER of these default constructors will be available to you.

Overloading constructors

Just like in Java, we can provide several constructors for a C++ class so long as they accept different argument lists. For example, suppose we wanted to rewrite the Rectangle class so that it had a second constructor with no arguments that created a 5x5 rectangle. Here's how the class declaration and implementation would change:

```
class Rectangle {
     private:
          int length;
          int width;
     public:
          Rectangle(void);
          Rectangle(int, int);
          int area(void);
          int perimeter(void);
};

Rectangle::Rectangle(void) {
     length = 5;
     width = 5;
}

Rectangle::Rectangle(int l, int w) {
     length = l;
     width = w;
}

int Rectangle::area(void) {
     return length*width;
}

int perimeter(void) {
     return 2*length + 2*width;
}
```

Here's how we could create a 3x4 rectangle:

```
Rectangle rect1(3, 4);
```

And here's how we could create a 5x5 default rectangle:

```
Rectangle rect2;
```

(Notice that if we define a no-argument constructor, we DO NOT include the () when we are creating an object.)

We can similarly overload functions in C++ (or C). To do this, the two functions must have the same name but different argument lists. This works exactly like overloading methods in Java.

The "new" keyword

In C, we saw how to create traditional structure objects and pointers to structure objects. To create a pointer to a structure object, we had to first allocate memory (with `malloc`). Then, we used `->` notation to access the struct fields.

We can also create pointers to objects in C++ instead of creating traditional objects. However, we will use the keyword **`new`** to allocate memory instead of the C function **`malloc`**.

Here's how to create a pointer to an object. The first step is to create declare a pointer of that class type:

```
ClassName *objectName;
```

The second step is to create the object itself. This is done using the **`new`** keyword, which **allocates memory** for the object and **calls the constructor**. Here's the format:

```
objectName = new ClassName(args);
```

Notice that creating pointers to objects in C++ is almost the same as creating objects in Java. The only difference is that in C++, the variable is a pointer.

For example, we could create a pointer to a 3x4 rectangle:

```
Rectangle *rectPtr = new Rectangle(3, 4);
```

To access the functions in `Rectangle`, we first need to dereference the pointer (`*`) and then need to use a `.` to get the functions. For example, to get the area we'd do:

```
int area = (*rectPtr).area();
```

Just like we did for structs, we can replace the `*/.` combination with the `->` operator:

```
int area = rectPtr->area();
```

We can also initialize object pointers by setting them equal to the address of another object. For example:

```
Rectangle *rect;
Rectangle r(3, 4);
rect = &r;
```

Now `rect` points to the 3x4 rectangle.

Objects v. pointers
We've now seen how to create objects and how to create pointers to objects. For example:

```
Rectangle rect(3, 4);
Rectangle *rectPtr = new Rectangle(3, 4);
```

`rect` is a 3x4 `Rectangle` object, and `rectPtr` is a pointer to another 3x4 `Rectangle` object. This is confusing because the "objects" we've seen in Java are actually more like pointers to objects than they are like regular objects in C++. In Java, an object is a reference type, which is very similar to a pointer.

Here are some major differences between objects and pointers to objects:

- Only pointers are created with "`new`"
- Regular objects are allocated on the stack
- Pointers to objects are allocated on the heap
- You do not need to release memory for regular objects
- You do need to release memory for pointers to object (see "delete")
- Regular objects are passed BY VALUE (they can't be changed in a function)
- Pointers are passed BY REFERENCE (they can be changed in a function)

Destructors

In Java, the garbage collector frees memory for objects that are no longer in use. C++, just like C, has no garbage collector. Thus we must manually deallocate space for any objects or variables that were allocated dynamically. In this section, we will learn how to release space for object pointers. In the next lecture notes section, we will discuss how to deallocate space for regular variables and arrays.

Suppose we've created a class with a dynamic array as an instance variable. For example:

```
class Vector {
     private:
          int *nums;
          int length;
          int pos;
     public:
          Vector(int);
          boolean add(int);
          void print(void);
};

Vector::Vector(int size) {
     //We'll learn a different way next time
     nums = malloc(size*sizeof(int));

     length = size;
```

```
        pos = 0;
}

boolean Vector::add(int val) {
        if (pos == length) return false;

        nums[pos++] = val;
        return true;
}

void Vector::print(void) {
        int i;
        for (i = 0; i < pos; i++) {
                cout << nums[i] << endl;
        }
}
```

Further suppose we've created a pointer to a `Vector` object:

```
Vector *v = new Vector(3);
v->add(1);
v->add(2);
v->add(3);
v->print();
```

...and now we're done using it. We now that we want to deallocate the space needed for the object. However, we also want to deallocate the space needed for the dynamic array instance variable. To handle the second need, C++ created the *destructor*, which allows us to deallocate space for any instance variables. The declaration for a destructor looks like:

```
~ClassName();
```

(A destructor cannot take any arguments and cannot be overloaded.) Here's how we would redo the `Vector` class declaration to include a destructor:

```
class Vector {
        private:
                int *nums;
                int length;
                int pos;
        public:
                Vector(int);
                ~Vector();
                boolean add(int);
                void print(void);
};
```

We've already said that we want our destructor to deallocate space for our dynamic array. So, here's the implementation of the `Vector` destructor:

```
Vector::~Vector() {
      free(nums);
}
```

As a rule of thumb, your destructor should include a call to `free` for every call to `malloc` in your class. (Later, this will become a call to `delete` for every call to `new` in your class.)

Destructors and objects

We've seen how to write destructors, but we haven't seen how to use them to deallocate memory. This section will focus on deallocating memory for normal objects (non-pointers), and the next section will focus on deallocating memory for object pointers (created with `new`).

If we create a normal object, space for that object is reserved on the stack before the program starts. Consequently, space for the object is released when the program finishes. Since normal objects are not allocated dynamically, we don't need to do anything special about releasing the memory.

However, we do need to release any memory that was allocated dynamically by the object itself. This can be done by calling the destructor. However: **you should never explicitly call the destructor**.

Instead, the destructor for a normal object is automatically called when that object's scope of existence ends. For example, consider this function:

```
void createVect(void) {
      Vector v(2);
      v.add(1);
      v.add(2);
      v.print();
}
```

As soon as the `createVect` function finishes, the scope of the `Vector v` is also complete. The destructor for `v` is called automatically at the end of this function.

Conclusion: write a destructor for all class that allocate dynamic memory. However, **if you only create normal objects then the destructor for these objects will be called automatically** – you don't need to do anything extra.

The "delete" keyword

You will have to do something to manage the memory for pointers to objects – the destructor is not called automatically. If you're done using a pointer to an object, you should do:

```
delete pointerName;
```

This statement will do two things:

- Call the destructor for the object
- Release the memory needed for the object

Here's an example:

```
void createVectPtr(void) {
     Vector *v = new Vector(2);
     v->add(1);
     v->add(2);
     v->print();

     //we're done using v
     delete v;
}
```

Later, we will see that the "`delete`" keyword will be used to release all dynamic memory in C++.

The "static" keyword

The "`static`" keyword in Java allowed you to create class variables and methods that could be accessed using the class name instead of the object name. These variables/methods are the same across all object instances.

Example

C++ also has a `static` keyword that yields a similar behavior. You can declare a variable or function to be `static` in the class declaration. For example:

```
class Math {
     public:
          static double PI = 3.14;
          static double square(double);
};
```

If we want to implement a `static` function, we do not reuse the `static` keyword in the implementation:

```
double Math::square(double num) {
     return num*num;
}
```

Then, we can access static variables and methods using just the class name. For example:

```
//Computes the area of a circle with radius 2
double area = Math.PI * Math.square(2.0);
```

Static in C

There is also a `static` keyword in C that behaves a bit differently than static in C++ and Java. Static variables in C are only visible inside the function they're declared in, but they maintain their value across function calls. For example:

```
int count(void) {
     static int total = 0;
     total++;
     return total;
}

int main() {
     printf("count: %d\n", count());
     printf("count: %d\n", count());
     printf("count: %d\n", count());

     return 0;
}
```

The variable `total` is only visible inside the `count()` function. However, it is only initialized to 0 on the first function call. After that, `total` retains its value from previous function calls. The print statements in `main` will display:

1
2
3

which shows that `total` is not getting reset with each call.

The "this" keyword

In Java, you can use the "`this`" keyword to access variables and methods inside this object. You can do something similar in C++.

In C++, "`this`" is a **pointer** to this object. For example, we could redo the `Rectangle` class to use "`this`" when initializing the instance variables:

```
class Rectangle {
     private:
          int length;
```

```
            int width;
      public:
            Rectangle(int, int);
            int area(void);
            int perimeter(void);
};

Rectangle::Rectangle(int length, int width) {
      this->length = length;
      this->width = width;
}

int Rectangle::area(void) {
      return length*width;
}

int perimeter(void) {
      return 2*length + 2*width;
}
```

Notice that we must use the -> notation with "`this`" because `this` is a pointer.