Main topics of the week:
- C++ Classes
- Method Overloading
- Constructors and Destructors
- Operator Overloading
- Coding a string class

C++ Classes

- Class definition syntax – very similar to Java
- Cornerstone of object oriented language support
- Provides compiler enforced encapsulation
- Requires terminating semicolon
- Just like a C structure definition
- May contain functions (methods) as well as data
- Default is that all methods and data are private
- Access control changed by public or private label
  - Individual methods/data not modified: label affects all lines until next label
- All data is instance data for the class object by default (like Java)
- All methods are instance methods for the class object by default (like Java)
- The keyword static may be used to make data or methods be per-class instead of per-instance. For data this means that there is a single data value, regardless of whether any instances of the class object are created, i.e., the data is per class rather than per object. For methods, this means the methods can be called without a class object. Static methods must not refer to any instance data or methods.
- Complete method definitions may be provided inline (within the class), or the method signature only may be declared. In this case, the method body can be located anywhere outside of the class definition, but the scope qualifier will be required to indicate that the method belongs to the class. For example, if the class name is MyClass, we could have the method foo declared and externally defined:
  ```cpp
class MyClass {
public:
  int value;
  void foo();
};
...
void MyClass::foo() { . . . }
```
- The syntax for class data and method access is the same as for Java, e.g.,
  ```cpp
  MyClass myobj;
  myobj.value = 10;
  myobj.foo();
  ```
- C++ uses new and delete to allocate and de-allocate objects on the heap. These are keywords in the language and so are understood at the compiler level. They may actually use the standard library functions malloc and free, and can be used with primitive types and arrays. However the syntax of new is that it requires a type name, which is most likely a class. The value of the new expression is a pointer to the type given, and this pointer is used to access the object, e.g.,
  ```cpp
  MyClass *p = new MyClass;
  p->value = 10;
  p->foo();
  ```
- Class definitions may be **nested**, but there is no special access for an inner class
  
  - Nesting allows “private” class definitions
  - Friend declarations can permit access

**Method Overloading**
- Each class’ method and data names are in a separate name space, so the same names may be used in different classes
- Within a class, a method is characterized by its signature (return type, name, parameter type list) and the signature must be unique in the class
- Thus we may have methods with same names but different parameter lists (different numbers of parameters or types). This is called method overloading.
- Method overloading can also occur at the global level, or in any namespace.
- Method parameter lists are positional, i.e., order counts
- Method parameters may be defaulted, e.g.,

```c
void foo(int width, int height = 10) {...}; // height is defaulted
foo(20, 30);   // foo is passed 20 and 30
foo(20);       // foo is passed 20 and 10
foo();         // Error: foo requires at least one argument
```

**Constructors and Destructors**
- Constructors are automatically called to initialize an object instance
  - A constructor performs initialization of the object – it does not allocate memory for the object itself – the memory for the object (either on the stack or heap) is there before the constructor begins execution
- A constructor is a method whose name is the same as the class name
- Constructors do not have return values (not even void) since they are not called directly
- The constructor is called implicitly (1) when **new** is used to create an object on the heap and (2) when an automatic variable of a class type is declared within a block (and so is on the stack). Constructors are also called for global variables (which are implicitly static storage class). Constructors may also be called when temporary objects are created in the course of expression evaluation, passing arguments by value, or returning values from a function.
- Arguments (if any) are supplied to the constructor in parentheses after the class name (for new) or the variable name (for automatic or static variables)
- If there is just one argument, the alternative syntax of initialization with the equals sign is permitted for variables
- **new** returns a pointer to the object – the object is created on the heap
- A class does not have to define a constructor – if no constructors are defined, a default constructor method with no arguments and no code is implicitly defined.
- Constructors may be overloaded – distinguished by different argument lists (either number of arguments or type)
- If at least one constructor is defined, there is no implicit constructor with no arguments – it must be explicitly defined if you need it. The only constructors are the ones explicitly defined.
- Constructors may have defaulted arguments like any other method
- Constructors may be defined inline or outside of the class declaration like any other method
- Constructors may be public or private. A private constructor can only be called from a context that has access to private areas of the class (e.g., a method of the class or a friend of the class)
- The constructor does not allocate the space for the object – the space is static, on the stack, or allocated by new. The constructor performs initialization, which might entail the creation of other objects or the allocation of other data space for the object (if the object has pointers to keep track of such space)
- Destructors are called to release resources for an object – they are the cleanup routine
- There can be only one destructor for an object, whose name is tilde (~) followed by the class name, and which has no arguments
- The destructor is called automatically for objects on the stack when they go out of scope
- The destructor is called implicitly when `delete` is applied to an object
- The destructor does not release the object’s space itself – that is done by delete or implicitly by the manipulation of the stack – the destructor is usually responsible for de-allocating space that was explicitly new’d in the code of the constructor.
- There are no arguments to the destructor because it is not explicitly called, and there is no syntax for supplying arguments to the implicit call (unlike the constructor where we have the class name or the variable name and a syntax for supplying parameter values)
- If no destructor is defined, an empty one (empty code body) is implicitly defined
- Remember: constructors are called implicitly when an automatic variable comes into scope and when a heap object is `new`’d. The destructor is called implicitly when an automatic variable goes out of scope and when a heap object is `delete`’d.
- Having the compiler call constructors and destructors implicitly guarantees their consistent use
- C++ has no automatic garbage collection – all heap objects must be dealt with by the program explicitly. However, automatic objects that do allocation can guarantee that deallocation is consistently done by defining constructors and destructors carefully
- Constructors and destructors offer the potential to design classes that handle their own memory management and possibly implement class specific garbage collection schemes (Constructors and destructors can be used to consistently track reference counts)

### Copy Constructors

- Copy constructors are a special type of constructor
- Signature is one argument: a `reference` to a `constant` object of the `same type`

```cpp
class MyClass {
    MyClass( const MyClass & original ) { . . . }
}
```

Atkins – Spring 2010 - 3 - CIS 330 C/C++ and Unix
Copy constructors are called when
- An object is **initialized from another object of the same type**:
  
  ```
  MyClass object1; // Default constructor with no args
  MyClass object2 = object1; // Initialize as copy of object1
  MyClass object3(object1); // Alternative syntax - still copy of object1
  ```

- An object is **passed to a function by value**:
  ```
  void foo(MyClass arg) { ... }
  ...
  MyClass object1;
  foo(object1); // Copy constructor makes copy of object1 on foo stack
  ```

- An object is **returned from a method by value**:
  ```
  MyClass foo() { MyClass tmp; ... ; return tmp; }
  ...
  MyClass object1 = foo(); // Copy constructor makes copy of tmp for foo return
 // Also invokes copy constructor to copy copy of tmp to object1
  ```

If no copy constructor is defined, the compiler supplies a default copy constructor that does a pure data copy of the object to be copied.

But, the default copy constructors does recursive copy constructor calls for data fields of class.

Compiler may optimize away some unnecessary copy constructor calls for transient values.

**The keyword this**
- The keyword **this** is only meaningful in a non-static class method.
- **this** is a self reference – its type is **pointer** to object of the class.
- Thus, **this** is the address of the object itself.
- **this** can be de-referenced with the * operator to get the object rather than the address.
- Use **this** when you need to refer to the object explicitly (e.g., to obviously distinguish from another object of the same type).
- The keyword const after a method signature (before the body) means that **this** is a pointer to a **constant** object, and so effectively means that the method is immutable with respect to the object.

**Operator Overloading**
- Some operators for built-in types are already overloaded, e.g.,
  - if i and j are int’s, x and y are doubles, the / in i/j is a different operation than the / of x/y
  - The assignment operator = is overloaded – it is dependent on the types being assigned
  - The + of i+j is different from p+i when p is a pointer
  - Behavior of operator (and whether it is allowed) is dependent on the types of the operands.
- C++ allows the programmer to define operators for user defined types, i.e., for objects of classes defined by the user.
- All operators (unary and binary) can be defined for class types.
Binary operators are dependent on the types of both operands (which don’t have to be the same type). For different types, order can make a difference, too.

Operators can be defined to do whatever you want

You can’t define new syntax, e.g., you can’t define the operator **

An operator definition is just a function
  - For a global (non-member) function, the operands are the arguments. If the operator is unary, it has one argument; if binary, it has two arguments.
  - For a member function, the first operand is the object (this) and the second operand is the single argument to the function (if the operator is binary). If the operator is unary, there will be no second operand, and thus no argument to the function, just the implicit object via this.

Operator definition is nothing more than syntactic sugar for functional notation

Operators should only be defined for a class when there is intuitive meaning

The assignment operator is the most commonly defined, and in fact is defined by default as bitwise copy (like the copy constructor)
  - This allows class specific copying semantics. Assignment is typically a combination of the actions of a copy constructor and the destructor.

The array operator is often defined for classes that have some notion of indexing, such as a vector class or a string class

Operators can be immutable methods (constant), e.g., +

Operators can change an object, e.g., ++

Comparison operators are commonly defined for objects that can be compared

The syntax of operator definition is just a function definition, where the function name is the keyword operator followed by the symbolic form of the operator

The iostream library defines the operators << and >> for all primitive types when they are combined as the second operand with an ostream or istream, respectively

It is common to define input and output operators to extend the iostream type safe IO to user defined types

Reference types are useful in operator definitions to avoid construction of temporaries. In fact, operator overloading is the reason references were added to the language.

References are essential for an array operator to be able to return an lvalue

Cast operators may be defined as well. In this case, the name of the operator is the type name being cast to, and that is also the return type.

Coding a string class

Simple class to encapsulate a C character array string

Constructor for empty string as well as construction from C string

Copy constructor

Assignment, from another string as well as a C string

Cast operator to produce a C string

output operator to print a string

Array operator to get characters of String, both for reading and writing

Length member function

Concatenation operator