Objects and Classes (Chapter 2)

Object-oriented (OO) design and programming is a way of thinking about programs as a collection of interacting "objects", each of which models some computational entity, real or imaginary.

OO programming has taken-off because there is the common perception among software engineers that this style of designing and implementing programs has practical benefit. There are several OO programming languages, all with a basic set of common abstraction features, and each with its own unique set of features and nuances.

C++ is widely used because it is seen to be a practical (meaning efficient) programming language to use for software engineering. However, the language has a complex set of features, and the ill-defined interaction of those features has required several years of compiler development to implement the full language specification.

Independent of an OO language like C++, OO programming dictates a style that embodies software engineering principles that are generally accepted as "good" by some measure of goodness. The principles are *abstraction*, *encapsulation* and *information hiding*, which dictate how programs are constructed.

The "goodness measure" is usually an economic one. Software engineers, like other engineers, are concerned with the quality and reliability of software and the cost of developing and maintaining a software product. Engineers want high quality and reliability, and low costs for development and maintenance. Unfortunately, as a profession, software engineers haven't yet achieved these goals.

C++ provides language mechanisms, like the class, that enable programmers to utilize in practice the principles of encapsulation and information hiding.

Abstraction, Encapsulation and Information Hiding

Abstraction is difficult to define precisely, but in the OO context, it means having a language mechanism that gives you the ability to define a conceptual structure that represents only the computationally interesting aspects of some external entity, real or otherwise, that is being modelled.

Encapsulation is a grouping of data and the operations that manipulate that data into an aggregate data structure. Encapsulation achieves *spatial locality* of two forms: *conceptual locality* and *run-time locality.* Conceptual locality helps our brain deal with complexity by allowing us to put related data and operations together in the same syntactic structure, and run-time locality means a machine can then do things efficiently with our defined structures.

Information hiding is concerned with the *scope,* or visibility, of parts of an implementation. The key realization of information hiding in C++, as we will learn throughout this course, is the ability to separate the *interface* of an abstraction representing an object from its implementation.

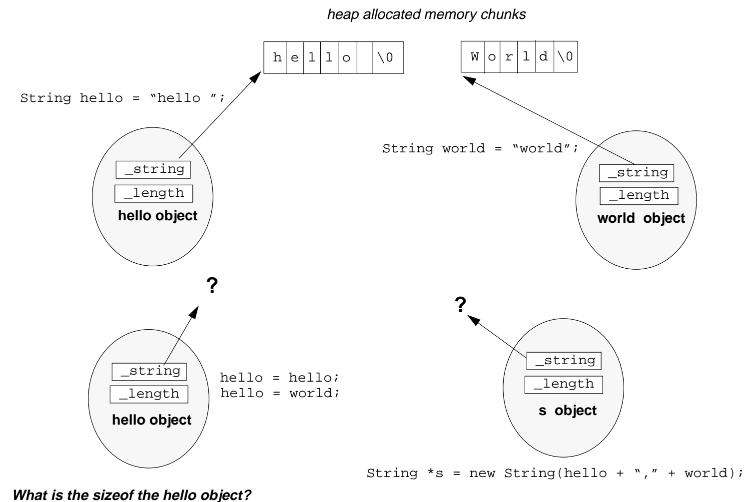
The interface of an object defines how it is to be used and may only give hints as to the implementation. The implementation is separated from the interface using either *scope control* mechanism, or by defining separate interface abstraction and implementation abtractions, and then relating them using other language mechanisms.

The ability to cleanly separate the definition of an interface from an implementation for that interface, and the ability to have multiple implementations for the same interface is perhaps its most important contribution to date of the OO paradigm. **Classes**, **templates**, and **inheritance** are the compile-time language mechanisms that allow you to define and compose abstract interfaces and encapsulated implementations in a "type safe" manner.

Objects are defined using classes

```
class String {
private:     // "private" means instance variables are hidden from non-member access

     char* _string;    // leading '-' is just a stylistic convention
     int   _length;

public:   // data & methods visible to clients and subclasses

     // inline class constructors
     String() : _string(NULL), _length(0) {}// an "initializer list"

     String(const String& s) : _string(NULL), _length(0) {// "copy" constructor
          _length = s._length;
          if (_length > 0) {
            _string = new char[s._length + 1];
            strcpy (_string, s._string);
          }
     }

     String(const char* s) : _string(NULL), _length(0) {
        if (s != NULL) {
          _length = strlen(s);
          _string = new char[_length + 1];
          strcpy (_string, s);
        }

     ~String() { delete [] _string; _string = NULL; _length = 0; }

     String& operator+ (const String&); // concatenation operator
};
```


Constructors and Assignment



Object Instantiation

Just as with the primitive built-in-types, objects are instantiated statically, automatically (on the stack), or dynamically (in the heap)

```
// constructors for external/static objects are executed at program start-up and
// destructed on exit from the program

const String version = "1.0";// global, but immutable because it is const

static const String rcsid = "$Header$";// local to this compilation unit

contrived()
{
     // local object constructors are implicitly called on entry to a scope

     String s;// invokes String() constructor

     String hello = "Hello";     // invokes String::String(const char*) constructor

     String helloWorld = hello + ", world";   // calls hello's operator+(const String&)

     String *str = new String(hello);// calls String::String(const String&) constructor
     ...
     delete str;  // invokes String::~String() destructor

} // on exit from procedure scope, String::~String() is implicitly called for local
  // objects s and hello
```


Accessors and Mutators

```
class String {
private:

     char* _string;
     int   _length;

public:
     ...

     // "accessor" methods do not modify an object's instance variables

     int length() const { return _length; }

     // "mutator" methods modify an object's instance variables

     void capitalize();

     // overloaded operators make objects appear as "builtin" types */

     String& operator+(const String& s) { ...; return *this; }

     // type conversion operators "convert" an object to a related type

     operator const char*() const { return _string; }

     // friend functions can access private/protected data elements

     friend ostream& operator<<(ostream& stream, const String& s);
};
```


Member functions (except static members functions) are passed an implicit first argument, called the **this** pointer. The **this** pointer associates an instance of a class with the set of methods defined for that class. That is to say, it *binds together* the data and methods of an object at run-time.

```
void contrived()
{
     String s = "Hello, World";
     ...
     int len = s.length();  // invokes String::length() using implicit this == &s
     ...
}
```

How would you do a similar with a struct (public class)?

```
struct String {   // all data is public and there are no member methods

     char* _string;
     int   _length;
};

int string_length(struct String* s) { return s->_length; }

void contrived()
{
     String s = { "Hello, World", 12 };
     ...
     int len = string_length(&s);
     ...
}
```

Q: Which approach is "better"? Why?

Separation of Interface and Implementation

Golden Rule: Separate the interface astraction from the implementation abstraction

```
class StringRep {// A class for implementing a string representation using a char*
private:
     char* _string;
     int _length;
public:
     StringRep () : _string(NULL), _length(0) {}
     int length() { return _length; }
     void copy(const StringRep& r);
     void copy(const char*);
};

class String {
     StringRep _rep;   // could use a StringRep* instead
public:
     String() {}
     String(const String& s) { _rep.copy(s._rep); }
     String(const char* s) { _rep.copy(s); }

     String& operator=(const String& s) {   // overloaded assignment operator
       if (&s != this)
          copy(s._rep);
       return *this;
     }

     int length() { return _rep.length(); }
};
```