COEN 171

Lecture 8: Classes

Program Structuring

- Programs need to be structured somehow.
  - Structuring helps understanding and reduces maintenance.
- Let’s look at three common approaches:
  - Procedures;
  - Modules;
  - Classes.
- These three approaches are evolutionary, but complementary.
  - That is, they should all be used together.

Procedures

- A procedure is a way of grouping code together to form an abstraction.
  - The procedure can be used again and again.
  - Its behavior is specified through its declaration.
- Procedures only offer simple control abstractions.
  - How the procedure does its job is abstracted away.
  - But what the procedure operates on is not.
  - The data representation is known throughout the program.
- All high-level languages support procedures.
  - In some, they are called functions or static methods.

Modules

- A module is a collection of procedures, variables, and types.
  - All can be hidden within the module.
  - This allows procedures to share variables.
- Modules provide a public and a private view.
  - The interface is public; the implementation is private.
- Pascal supports procedures and variables declared within procedures.
  - But a collection of procedures cannot share otherwise hidden variables and it has no support for hidden types.

Modules in C

- C has some support for modules.
  - Any global name declared `static` in C is local to the file.
- Thus, we can have any number of functions or global variables that are visible only within the file.
- Any type declared within the file is local to that file.
  - However, we can still use the name of the type outside.
- All of these must be present in one source file.
  - There is no native way to extend this idea across files.
  - We could start names with an underscore as a convention.

Example: Shared Variables

- Consider a random number generator in C.

```
static unsigned long seed = 1;

void srand(unsigned int val) {
   seed = val;
}

int rand(void) {
   seed = seed * 1103515245 + 12345;
   return seed >> 16;
}
```

Limited Types

- Modules should offer more than just control abstractions; they should offer type abstractions.
- Modules use the idea of a **limited type**.
    - Only the name of the type is visible outside the module.
    - Its implementation is hidden from the clients of the module.
- Limited types are possible only if the language does not use strict structural equivalence of types.
- They are often implemented as pointer types.
    - A pointer's size is the same regardless of the underlying type, so the compiler knows how much space to allocate.

Limited Types in C

- C uses opaque pointer types as limited types.
  - An **opaque type** is one whose implementation is hidden.
  - The size and structure are unknown to the client.
- An opaque type itself cannot be declared as we do not know how much space is required.
- But we can declare a pointer to an opaque type.
  - We can declare a pointer to a `struct foo` without knowing what a `struct foo` looks like.
  - Recall that C uses name equivalence for structures.

Example: Opaque Types

- Let’s look at a set abstract data type in C.
- The header file, `set.h`, might look like:

```
struct set *createSet(void);
int addElement(struct set *sp, int elt);
```

- Whereas the source file, `set.c`, might look like:

```
# include "set.h"

struct set {
    unsigned count;
    …
};
```

Limitations

- Since limited types are usually implemented using pointer types, they have some serious limitations.
  - A function must be called to explicitly allocate space.
  - Equality and assignment, including call by value, are based on pointer semantics.
- Consider strings in C: `typedef char *string`;
  - We still need to explicitly allocate space.
  - We still need to use `strcmp` to compare strings.
  - Passing a string to a function does not make a copy of it.

Classes

- A **class** corresponds to a type.
- Classes should have the same abilities as built-in types such as integers.
  - In other words, classes should be first-class types.
- Classes should support information hiding by having both public and private members.
- An **object** is an instance of a class.
  - Classes are abstract; objects are concrete.
  - Operations are performed on objects, not classes.

Classes in Java

- Note that Java uses “class” for everything.
  - A type is a class.
  - A module is a class.
  - A namespace is a class.
  - A class, in the object-oriented sense, is a class.
- Even though Java has a `Math` class that provides operations, it is really just a namespace.
- When we talk about “classes,” we mean classes in the object-oriented sense.

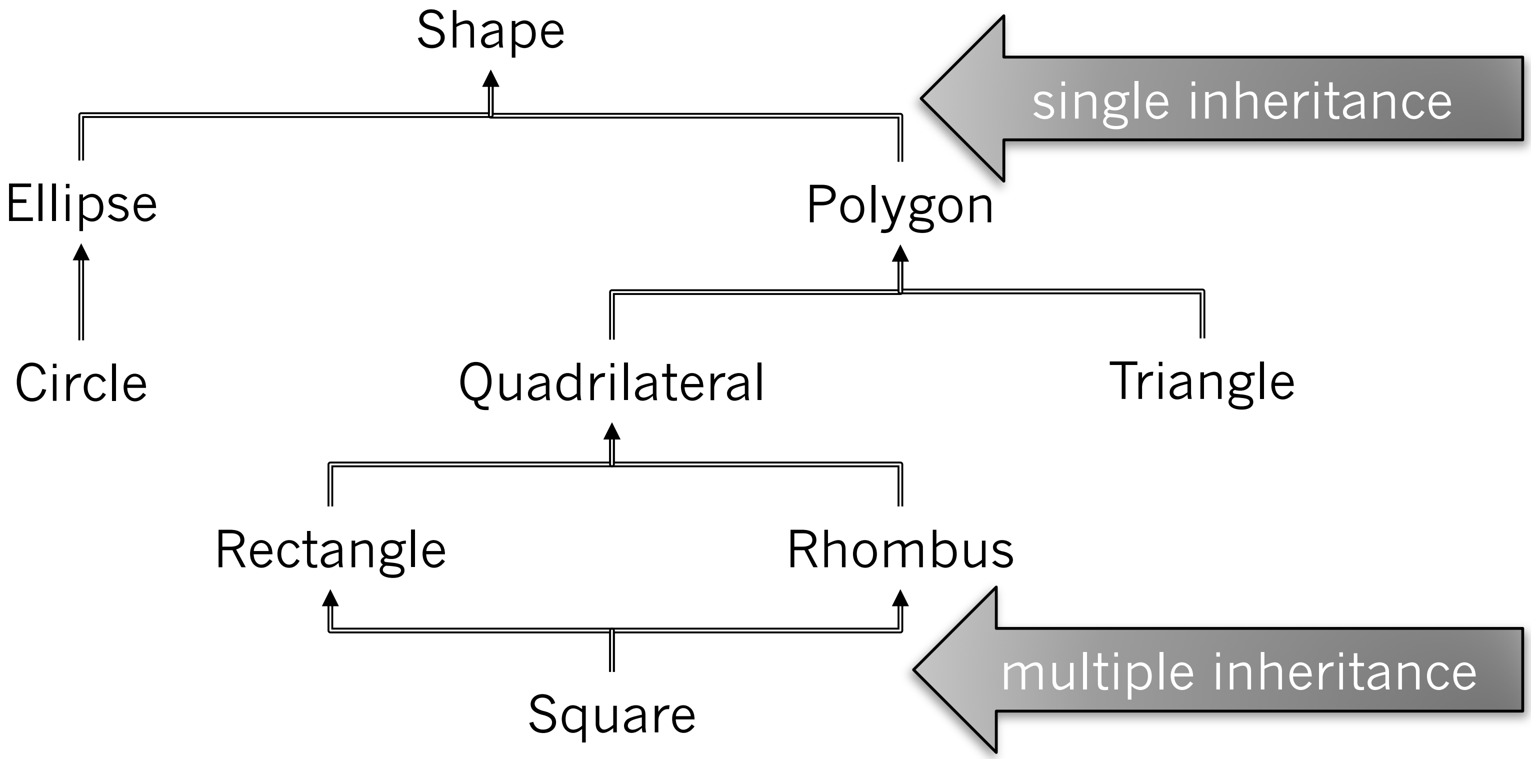
Classes in C++

- C++ classes are a generalization of structures.
  - In C, a structure can contain only data.
  - In C++, a class can contain both data and operations.
  - A `class` is simply a `struct` in which all members are private by default.
  - A `struct` is simply a `class` in which all members are public by default.
- Just as we can have structures and pointers to structures in C, we can do the same with classes.
  - In Java, a `class` is always a reference type.
  - In C#, a `class` is a reference type; a `struct` is a value type.

Inheritance

- **Inheritance** is the mechanism of basing a class upon another class, retaining similar information.
  - The operations and data of the base class are available to the new, derived class.
  - The base class is also called the parent or superclass, and the derived class is also called the child or subclass.
- Many languages have three levels of access control:
  - **Private** members are only accessible in the class itself;
  - **Protected** members are also accessible to derived classes;
  - **Public** members are accessible by anyone.

Example: Inheritance



- Every polygon is a shape, but not all shapes are polygons.
- Anything true of a shape is also true of a polygon.
- Any function of a shape is also a function of a polygon.

Interface vs. Implementation

- We need to distinguish two types of inheritance:
  - Inheritance of **interface** in which only declarations are inherited;
  - Inheritance of **implementation** in which definitions are inherited.
- Multiple inheritance of interface is not problematic.
  - There are no conflicting definitions.
- A class is required to implement all interface functions before it can be instantiated.
  - Otherwise, the class is an **abstract class**.

Inheritance in Java

- Does Java allow multiple inheritance? Yes and no.
  - Java allows single inheritance of implementation.
  - It allows multiple inheritance of interface.
- A class that is completely abstract is an interface.
  - A class that is partially abstract (has at least one definition) is declared `abstract` in Java.
- If a class does not implement all functions in an interface, it cannot be instantiated.
  - The code will simply not compile.
  - You can think of it as a contract not being fulfilled.

Inheritance in C++

- Does C++ allow multiple inheritance? Yes.
    - C++ allows multiple inheritance of both interface and implementation.
- How does C++ resolve any ambiguity of which inherited function or variable to use?
    - It requires the programmer to explicitly qualify any ambiguous reference with the name of the parent class.
- We'll look at the problems with multiple inheritance when we examine C++ in detail.

Summary

- We looked at three ways of structuring a program:
  - Procedures, which provide only control abstraction;
  - Modules, which provide both a public and private view of their collection of procedures, variables, and types;
  - Classes, which are first-class types.
- Classes should support public and private views.
- Classes should support inheritance (the reuse of variables and functions from a parent class).
  - Java and C++ both support multiple inheritance, though in different ways.