ECE 2574: Data Structures and Algorithms - Code Reuse

C. L. Wyatt

Today we will learn about code reuse via dynamic polymorphism and compare it to composition.

- Subtyping and Object Hierarchies
- Abstract Base Classes as Interfaces
- Dynamic Casting
- Composition models “has-a”
- Examples

Dynamic Polymorphism

Polymorphism means selecting what code to run based on the type.
Dynamic polymorphism means the selection happens at runtime.
The basic language mechanisms are:

- Inheritance or Derivation
- Virtual Functions, also called dynamic dispatch or runtime dispatch
- Encapsulation via private/public

Inheritance allows us to build one class from another.

Represents an “is-a” relationship. A Circle **is a** Shape.
You specify the *base class* after declaring the *derived class* as

```
class BaseClass {};
class DerivedClass: public BaseClass {};
```

The public part means that the public members of `BaseClass` are inherited and become public members of `DerivedClass`.
Note the above is equavalent to

```
struct BaseClass {};
struct DerivedClass: BaseClass {};
```

since struct members are public by default.

Review: private, protected, and public

- members that are *private* can only be accessed by members in the same class
- members that are *protected* can only be accessed by members in the same class and those derived from it
- members that are *public* can be accessed by any functions

Note, this simplified version ignores friend access

private, protected, and public inheritance

- private inheritance means that the public and protected members of the base class can only be used by the derived class
- protected inheritance means that the public and protected members of the base class can only be used by the derived class and any classes derived from it
- public inheritance means that the public members of the base class can be used by any function

Note, there is no way to inherit the private members of a class.

Summary: private inheritance

So

```
class BaseClass {};
class DerivedClass: private BaseClass {};
```

means that the the public and protected members of `BaseClass` become private members of `DerivedClass`.
This is less common than the others, but can be useful for writing adaptors.

Summary: protected inheritance

```
class BaseClass {};
class DerivedClass: protected BaseClass {};
```

means that the the public and protected members of `BaseClass` become protected members of `DerivedClass`.
This is used for keeping functionality within the tree of objects.

Summary: public inheritance

```
class BaseClass {};
class DerivedClass: public BaseClass {};
```

means that the the public members of `BaseClass` become public members of `DerivedClass`.
This is the most used, and presents the client code with a polymorphic interface.

A graphical view

Inheritance allow you to specify a tree relationship among types

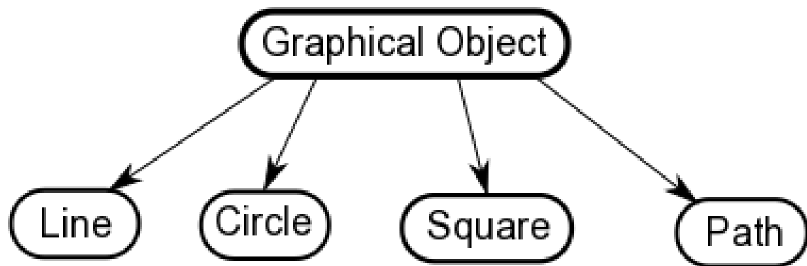

Figure 1: Example

Virtual Functions

Declaring a method virtual means that it can be overridden in the Derived class, but need not be.

Note that you can redefine the method in `DerivedClass` even if it is not marked virtual, but it will not be called when represented as the `BaseClass`. This is a source of much pain. If you intend for a method to be overridden, mark it as virtual.

Note the virtual keyword is not used in the implementation if outside the class definition.

To **force** the derived class to implement the method you make it *pure* as

```
struct BaseClass {
 virtual void aMethod() = 0;
};
```

Take care when overriding that the name and arguments match. The compiler can't catch all the likely mistakes here.

C++ allows multiple inheritance

```
class Base1 {};
class Base2 {};

class Derived: ACCESS Base1, ACCESS Base2
```

where `ACCESS` can be private, protected, or public.
This feature is much maligned because it creates confusion around what gets inherited.
It is useful though for defining an *interface*.

An interface is a base class with only pure virtual methods.

This allows you to express that a class implements that interface explicitly, and the compiler verifies that it at least implements the methods.
When used with multiple inheritance this allows you to express that some parts of an object tree implement a certain interface.
Example: AbstractBag, AbstractStack, AbstractList

Heterogeneous Collections using a Base Type

Since a pointer or reference to a base object can actually refer to a derived object, you can defined containers of mixed type (within the object hierarchy)
See example code

Dynamic Casting

Given a pointer to a base class you can attempt to convert it back to a derived type using `dynamic_cast`.
See code
This works for up-cast (but is unnecessary), down-cast, and sideways-cast.

Some remarks about dynamic polymorphism

- There are many places for bugs to hide
- Don't get carried away with defining object hierarchies.
- Excessive casting is a code smell

Uses of Dynamic Polymorphism

Dynamic Polymorphism is very useful when you want to treat objects (instances of classes) as hierarchical data.

- Graphical user interfaces are ideally suited to polymorphism. They are just trees of widgets. Code to layout and draw widgets should be independent of the specific interface.
- Rendering of dynamic objects, for example in simulations or games, should be independent of the specific objects involved.
- Employees are people that have categories and form departments, units, divisions, etc.

Composition is a major way of modeling has-a relationships

A composite type has member variables that correspond to its components.

```
class Foo
{
  ComponentType component;
}
```

Classic Example: People and Employees

A Person has-a

- name
- age
- address

An Employee is-a Person and has-a

- id
- role
- salary

See code

Classic Example: People, Employees, and Customers

A Person has-a

- name (first/last?)
- age (possibly unknown?)
- address (format?)

An Employee is-a Person and has-a

- id (unique?)
- role (static or dynamic?)
- salary (currency?)

Is a customer always a person?

Prefer Composition to Inheritance

Inheritance is overused and leads to tight coupling.
Composition

- gives the most flexibility with least coupling
- shorter compile times, a member can be a pointer, thus only declared
- less error prone, no private/protected/public

Use inheritance only when you need to implement is-a relationships that require polymorphism.

Example: A stack has-a list.

See code in `stack_as_list`.

Next Actions and Reminders

- Read CH pp. 347-351 on the Sorted List ADT
- Program 3 is due 10/31 by 11:59 pm.