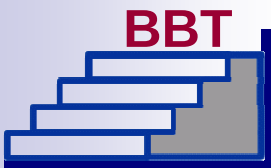


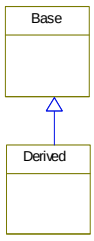
CPP

Inheritance

Template

Class inheritance

- class inheritance derive new classes from old ones, with the derived class inheriting the properties, including the methods
- the old class is called a base class.
- Here are some things you can do with inheritance:
  - You can add functionality to an existing class.
  - You can add to the data that a class represents.
  - You can modify how a class method behaves.



Inheritance

The key points about constructors for derived classes

- The base class object is constructed first.
- The derived class constructor should pass base class information to a base class constructor via a member initializer list.
- The derived class constructor should initialize those data members that were added to the derived class.

Member Initializer Lists

- A constructor for a derived class can use the initializer-list mechanism to pass values along to a base-class constructor.

```
derived::derived(type1 x, type2 y) : base(x,y) // initializer list
{
    ...
}
```

- The member initializer list can be used only with constructors.

Inheritance—An Is-a Relationship

- C++ has three varieties of inheritance: **public, protected, and private**.
- Public inheritance is the most common form, and it models an is-a relationship. This is shorthand for saying that an object of a derived class should also be an object of the base class.

Polymorphic Public Inheritance

- Polymorphic is the way a particular method behaves will depend upon the object that invokes it.
- There are two key mechanisms for implementing polymorphic public inheritance:
  - Redefining base class methods in a derived class
  - Using **virtual** methods

```
Class name{
virtual return method(arg…);
…
}
```

Static Binding

- Interpreting a function call in the source code as executing a particular block of function code is termed binding the function name.
- **static binding** (or early binding) takes place during compilation is called.

Dynamic Binding

- **dynamic binding** (or late binding) is ,for virtual functions ,the decision of which function to use can't be made at compile time. Therefore, the compiler has to generate code that allows the correct virtual method to be selected as the program runs.
- Dynamic binding in C++ is associated with methods invoked by pointers and references.

Pointers and References Type Compatibility

- Converting a derived-class reference or pointer to a base-class reference or pointer is called upcasting, and it is always allowed for public inheritance without the need for an explicit type cast.
- Converting a base-class pointer or reference to a derived-class pointer or reference, is called downcasting, and it is not allowed without an explicit type cast.

Virtual function mechanism

- Each object has its size increased by the amount needed to hold an address (virtual table address).
- For each class, the compiler creates a table (an array) of addresses of virtual functions.
- For each function call, there's an extra step of going to a table to look up an address.

Virtual things

- Constructors
  - Constructors can't be virtual. A derived class doesn't inherit the base class constructors
- Destructors
  - Destructors should be virtual unless a class isn't to be used as a base class.
- Friends
  - Friends can't be virtual functions because friends are not class members, and only members can be virtual functions.
- Redefine
  - If you redefine a function in a derived class, it doesn't just override the base class declaration with the same function signature. Instead, it hides all base class methods of the same name, regardless of the argument signatures.

Rules of redefine

- if you redefine an inherited method, make sure you match the original prototype exactly. One exception is that a return type that is a reference or pointer to a base class can be replaced by a reference or pointer to the derived class.
- if the base class declaration is overloaded, redefine all the base class versions in the derived class

Access Control-protected

- The **protected** keyword is like private in that the outside world can access class members in a protected section only by using public class members.
- Members of a derived class can access protected members of a base class directly, but they cannot directly access private members of the base class.

Abstract Base Classes (ABC)

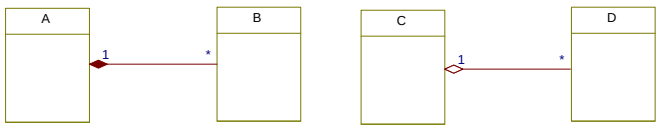
- **Pure virtual function** is an unimplemented function. A pure virtual function has = 0 at the end of its declaration.
- An **ABC** describes an interface using a least one pure virtual function. you can't create an object of that class.
- classes derived from an ABC use regular virtual functions to implement the interface in terms of the properties of the particular derived class.

ABC UML

Inheritance and Dynamic Memory Allocation

- If the derived class does not itself use dynamic memory allocation, you needn't take any special steps.
- If the derived class does also use dynamic memory allocation, you do have to define an explicit destructor, copy constructor, and assignment operator for the derived class

Classes with Object Members



composition aggregation

- Modeling has-a relationships is to use composition or containment; that is, to create a class composed of members that are objects of another class
- constructors use member-initializer-list syntax to initialize member objects.

Private Inheritance

- Private inheritance is the second means of implementing the has-a relationship
- With private inheritance, public and protected members of the base class become private members of the derived class.

Containment or Private Inheritance?

- In general, use containment to model a has-a relationship.
- Use private inheritance
  - if the new class needs to access protected members in the original class
  - if it needs to redefine virtual functions.

Protected Inheritance

- It uses the keyword protected when listing a base class:
- Public and protected members of a base class become protected members of the derived class.

Redefining Access with using

- use a **using**- declaration (like those used with namespaces) to announce a particular base class member can be used by the derived class, even though the derivation is private.
- using-declaration just uses the member name—no parentheses, no function signatures, no return types.
- The using-declaration approach works only for inheritance and not for containment.

Multiple Inheritance

- Multiple inheritance (MI) describes a class that has more than one immediate base class.
- Public multiple inheritance should express an is-a relationship.

Virtual Base Classes

- Virtual base classes allow an object derived from multiple bases that themselves share a common base to inherit just one object of that shared base class
- Example:

```
class Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

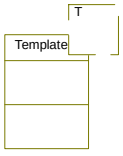
- If you want to use something other than the default constructor for a virtual base class, you need to invoke the appropriate base constructor explicitly.

Virtual Base Classes and Dominance

- If a class inherits two or more members (data or methods) of the same name from different virtual base classes, and one name dominates all others, it can be used unambiguously without a qualifier.
- **Dominance rule:**
  - A name in a derived class dominates the same name in any ancestor class, direct or indirect.

Class Templates

- Templates provide parameterized types, that is, the capability of passing a type name as an argument to a recipe for building a class or a function
  - template <class *Type*>
  - template <typename *Type*> // newer choice



Parameterized class

Default Type Template Parameters

- you can provide default values for type parameters:
  template <class T1, class T2 = int> class Topo {...};
- you can't provide default values for function template parameters.
- you can provide default values for non-type parameters for both class and function templates.

Template Specializations

- Implicit Instantiations
  - declare one or more objects indicating the desired type, and the compiler generates a specialized class definition using the recipe provided by the general template:
  - Example: ArrayTb<int, 100> stuff; // implicit instantiation
- Explicit Instantiations
  - The compiler generates an explicit instantiation of a class declaration when you declare a class using the keyword template and indicating the desired type or types
  - Example: template class ArrayTb<String, 100>;

Specializations –cont.

- Explicit Specializations
  - The explicit specialization is a definition for a particular type or types that is to be used instead of the general template.
  - A specialized class template definition has the following form:

template <> class *Classname*<specialized-type-name> { ... };

Partial Specializations

- partial specializations
  - partially restrict the generality of a template. A partial specialization, for example, can provide a specific type for one of the type parameters
  - Example:

```
// general template
    template <class T1, class T2> class Pair {...};
// specialization with T2 set to int
    template <class T1> class Pair<T1, int> {...};
```

Member Templates

- Template can be a member of a structure, class, or template class

Templates As Parameters

- You've seen that a template can have type parameters, such as typename T, and non-type parameters, such as int n. A template also can have a parameter that is itself a template.

Template Classes and Friends

- Nontemplate Friend Functions to Template Classes
- Bound Template Friend Functions to Template Classes
- Unbound Template Friend Functions to Template Classes