Lecture 7

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Inheritance – C++ Primer Chapter 15

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Object-oriented Programming (OOP)

object-oriented programming: Term used to describe programs that use data abstraction, inheritance, and dynamic binding.

Polymorphism: A term derived from a Greek word that means "many forms."

Polymorphism refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

dynamic binding: Delaying until run time the selection of which function to run.

In C++, dynamic binding refers to the run-time choice of which virtual function to run based on the underlying type of the object to which a reference or pointer is bound.

Inheritance

Inheritance: Types related by inheritance share a common interface.

A derived class inherits properties from its base class.

base class: A class that is the parent of another class.

The base class defines the interface that a derived class inherits.

derived class: A derived class is one that shares an interface with its parent class.

Derived classes I

- A derived class can redefine the members of its base and can define new members.
- A derived-class scope is nested in the scope of its base class(es), so the derived class can access members of the base class directly.
- Members defined in the derived with the same name as members in the base hide those base members; in particular, member functions in the derived do not overload members from the base.
- A hidden member in the base can be accessed using the scope operator.

Derived classes II

- **protected access label**: Members defined after a protected label may be accessed by class members and friends and by the members (but not friends) of a derived class.
  - protected members are not accessible to ordinary users of the class.
- **class derivation list**: Used by a class definition to indicate that the class is a derived class.
  - A derivation list includes an optional access level and names the base class.
  - If no access label is specified, the type of inheritance depends on the keyword used to define the derived class.
  - By default, if the derived class is defined with the struct keyword, then the base class is inherited publicly.
  - If the class is defined using the class keyword, then the base class is inherited privately.

Derived classes III

- Derived objects contain their base classes as subobjects
- However, there is no requirement that the compiler lays out the base and derived parts of an object contiguously
- A class must be defined to be used as a base class
- Forward declarations must not include the derivation list
- **immediate (direct) base class**: A base class from which a derived class inherits directly.
  - The immediate base is the class named in the derivation list.
  - Only an immediate base class may be initialized in the derivation list
  - The immediate base may itself be a derived class.
- Example
  - Class Base {}; class D1 : public Base {};
  - Class d1 : public Base; // error!

Types of inheritance

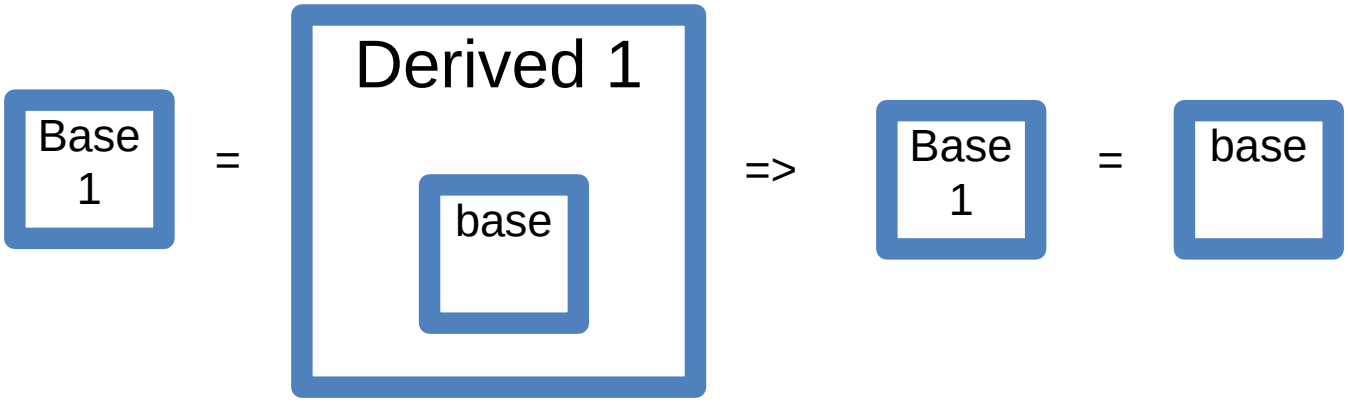
public inheritance: The public interface of the base class is part of the public interface of the derived class.

private inheritance: A form of implementation inheritance in which the public and protected members of a private base class are private in the derived.

protected inheritance: In protected inheritance the protected and public members of the base class are protected in the derived class.

Conversions from Derived to Base I

- Assignment Base = Derived



- The derived portion of the object is "sliced down," leaving only the base portion, which is assigned to the base.

Conversions from Derived to Base II

- When we call a BaseClass copy constructor or assignment operator on an object of type DerivedClass:
  1. The DerivedClass object is converted to a reference to BaseClass, which means that a BaseClass reference is bound to the DerivedClass object
  2. That reference is passed as an argument to the copy constructor or assignment operator
  3. Those operators use the BaseClass part of DerivedClass to initialize (or assign) the members of the BaseClass on which the constructor or assignment was called
  4. Once the operator completes, the object is a BaseClass. It contains a copy of the BaseClass part of the DerivedClass from which it was initialized or assigned, but the DerivedClass parts of the argument are ignored

Copy Control and Inheritance

- If a derived class explicitly defines its own copy constructor or assignment operator, that definition completely overrides the defaults
- Therefore, copy constructor and assignment operator for inherited classes are responsible for copying and assigning also their base-class members
- The copy constructor cannot, and the assignment operator should not be defined as virtual

Virtual functions I

- **virtual function**: Member function that defines type-specific behavior.
  - Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.
  - Once a function is declared as virtual in a base class it remains virtual in all derived classes

Virtual functions II

- **dynamic type**: Type at run time.
  - Pointers and references to base-class types can be bound to objects of derived type.
  - In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.
- **static type**: Compile-time type.
  - Static type of an object is the same as its dynamic type.
  - The dynamic type of an object to which a reference or pointer refers may differ from the static type of the reference or pointer.

Pure virtual functions

- **pure virtual**: A virtual function declared in the class header using = 0 at the end of the function's parameter list.
  - A pure virtual is one that need not be defined by the class.
  - A class with a pure virtual is an abstract class.
  - If a derived class does not define its own version of an inherited pure virtual, it is abstract as well.
- **abstract base class**: Class that has or inherits one or more pure virtual functions.
  - It is not possible to create objects of an abstract base-class type.
  - Abstract base classes exist to define an interface.
  - Derived classes will complete the type by defining type-specific implementations for the pure virtuals defined in the base.

Virtual destructors

- A derived class destructor automatically invokes the base class destructor
- The root class of an inheritance hierarchy should define a virtual destructor
  - In order to assure proper deleting of pointer members
- If a virtual is called from inside a constructor or destructor, it runs the version defined for the type of the constructor or destructor itself

Class scope under Inheritance

- A derived-class member with the same name as a member of the base class hides direct access to the base-class member
- If the derived class redefines any of the overloaded members, then only the ones redefined in the derived class are accessible through the derived type
- This is the reason why virtuals must have the same prototype on base and derived classes

Name Lookup and Inheritance

- Name lookup happens at compile time and follows the steps:
  1. Determine the static type of the object, reference, or pointer through which the function is called
  2. Look for the function in that class.
     - If it is not found, look in the immediate base class and continue up the chain of classes until either the function is found or the last class is searched.
     - If the name is not found in the class or its enclosing base classes, then the call is an error
  3. Once the name is found, do normal type-checking to see if this call is legal given the definition that was found
  4. Assuming the call is legal, the compiler generates code.
     - If the function is virtual and the call is through a reference or pointer, then the compiler generates code to determine which version is run based on the dynamic type of the object.
     - Otherwise, the compiler generates code to call the function directly

Runtime type identification (RTTI)

- **run-time type identification**: Term used to describe the language and library facilities that allow the dynamic type of a reference or pointer to be obtained at run time.
  - The RTTI operators, typeid and dynamic_cast, provide the dynamic type only for references or pointers to class types with virtual functions.
  - When applied to other types, the type returned is the static type of the reference or pointer.

typeid keyword

- **typeid**: Unary operator that takes an expression and returns a reference to an object of the library type named type_info that describes the type of the expression.
  - When the expression is an object of a type that has virtual functions, then the dynamic type of the expression is returned.
  - If the type is a reference, pointer, or other type that does not define virtual functions, then the type returned is the static type of the reference, pointer, or object.
- **type_info:** Library type that describes a type.
  - The type_info class is inherently machine-dependent, but any library must define type_info with members like name()
  - type_info objects may not be copied.

Dynamic Cast

- **dynamic_cast**: Operator that performs a checked cast from a base type to a derived type.
  - The base type must define at least one virtual function.
  - The operator checks the dynamic type of the object to which the reference or pointer is bound.
  - If the object type is the same as the type of the cast (or a type derived from that type), then the cast is done.
  - Otherwise, a zero pointer is returned for a pointer cast, or an exception is thrown for a cast of a reference.