Data Structure — CSci 1200
Lectures 23 and 24
C++ Inheritance and Polymorphism

Review from Lecture 22

- Priority queues
- Binary heap implementation
  - Push and pop (`delete_min`)
  - Heap as a vector
- We will complete our discussion of heaps as vector and of heap sort during Lecture 23.

The Next Two Lectures

- Inheritance is a relationship among classes.
- Example of inheritance: bank accounts.
- Basic mechanisms of inheritance
- Types of inheritance
- Is-A, Has-A, As-A relationships among classes.
- Example of stack inheriting from list
- Polymorphism
- A polymorphic implementation of a `Window` class.

Motivating Example 1

- Consider different types of bank accounts
  - Savings accounts
  - Checking accounts
  - Time withdrawl accounts, which are like savings accounts, except that only the interest can be withdrawn.
- If you were designing C++ classes to represent each of these, what member functions might be repeated among the different classes? What member functions would be unique to a given class?
- To avoid repeating common member functions and member variables, we will create a **class hierarchy**, where the common member functions and variables are placed in a **base class** and specialized ones are placed in **derived classes**.

Motivating Example 2

The window manager on your laptop — on a Mac, a PC or a Linux machine — has a list of every active window. These including browser windows, word processing windows, spreadsheets, chat sesssions, etc. Each window must respond to a series of events:

- Various mouse clicks
- Refresh
- Redraw
- Minimize
- Maximize
- Shutdown

Each of these events generates a function call to one of the objects on the active windows list. These objects are obviously quite different, but they are on the same list. Moreover, the active windows list does not know, at compile time, all the different types of windows that can be created. All of these issues are handled through a combination of class hierarchies and **polymorphism**.

Accounts Hierarchy

We will use the accounts class hierarchy as the main working example. The code is attached

- `Account` is the *base class* of the hierarchy.
- `SavingsAccount` is a *derived* class from `Account`.
- The member variable `balance` in base class `Account` is `protected`, which means
  - It is NOT publically accessible outside the class, but
  - It is accessible in the derived classes.
- `SavingsAccount` has inherited member functions and ordinarily-defined member functions.
- `SavingsAccount` has inherited member variables and ordinarily-defined member variables.
  - If the base class member variables were declared as private, `SavingsAccount` member functions could not access them.
- When using objects of type `SavingsAccount`, the inherited and derived member functions are treated exactly the same — they are not distinguishable. You can see this for the `SavingsAccount` class by looking at the main program.
  - In particular, note the use of `compound`, which is defined as a `SavingsAccount` member function and `get_balance`, which is defined as a `Account` member function, but used with a `SavingsAccount` object.

- `CheckingAccount` is also a derived class from base class `Account`.
- `TimeAccount` is derived from `SavingsAccount`. `SavingsAccount` is its base class and `Account` is its indirect base class.
  - We will discuss `TimeAccount` more below.

Constructors and Destructors

- Constructors of a derived class call the constructor for the base class immediately, before doing ANYTHING else.
  - This can not be prevented. You can only control the form of the constructor call.
  - See the class constructors in each of the account classes.
- The reverse is true for destructors: derived class constructors do their jobs first and then base class destructors are called, automatically.
  - This is generally only a concern for classes with dynamically allocated objects. The objects must be only deleted once, generally by the location in the class hierarchy that allocated them.

Overriding Member Functions in Derived Classes

- A derived class may redefine member functions in the base class.
- The function prototype must be identical, not even the use of `const` can be different.
  - Otherwise, both functions will be accessible, and major confusion will reign!
- Two examples are in class `TimeAccount`: `compound` and `withdraw`.
  - Notice the syntax (in the member function implementation) showing how `compound` calls the `compound` function for its base class.
- Once a function is redefined it is not possible to call the base class function directly from outside the class (only indirectly, as in `compund`, from inside the hierarchy).

Exercise

Create a class hierarchy of 2D geometric objects, such as squares, rectangles, circles, ellipses, etc. How should this hierarchy be arranged? What member functions should be in each class? Based on your hierarchy, what member variables are available in each class?

Public, Private and Protected Inheritance: What is Accessible Where

- Notice the line

```
class Savings_Account : public Account {
```

- This specifies that the member functions and variables from `Account` do not change their *public*, *protected* or *private* status in `SavingsAccount`.

- This is called *public* inheritance.
- *protected* and *private* inheritance are also possible.
  - With protected inheritance, public members becomes protected; other members are unchanged
  - With private inheritance, all member functions and variables become private. This is the default.

Stack Inheriting from List

A simple example of using private inheritance is inheriting a stack from a list.

- Here is the full declaration and definition of the stack:

```
template <class T>
class stack : private std::list<T> {
public:
  stack() {}
  stack( stack<T> const& other ) : std::list<T>( other ) {}
  ~stack() {}
  void push( T const& value ) { this->push_back( value ); }
  void pop() { this->pop_back(); }
  T const& top() const { return this->back(); }
  int size() { return std::list<T>::size(); }
  bool empty() { return std::list<T>::empty(); }
};
```

- Private inheritance hides the `std::list<T>` member functions from the outside world.
- These member functions are still available to the member functions of the `stack<T>` class.
- Finally, note that no member variables are defined — the only member variables needed are in the list class.
- When the stack member function uses the same name as the base class (list) member function, the name of the base class followed by `::` must be provided to indicate that the base class member function is to be used.
- The copy constructor just uses the copy constructor of the base class, without any special designation because the stack object is a list object as well.

Is-A, Has-A, As-A Relationships Among Classes

- When trying to determine the relationship between (hypothetical) classes C1 and C2, try to think of a logical relationship between them that can be written:
  - C1 is a C2,
  - C1 has a C2, or
  - C1 is implemented as a C2

- If writing "C1 is-a C2" is best, for example,

  a savings account is an account

  then C1 should be a derived class (a subclass) of C2.

- If writing "C1 has a C2" is best, for example

  a cylinder has a circle as its base

  then class C1 should have a member variable of type C2.

- In the case of C1 is implemented as a C2, as in the stack is implemented as a list, then C1 should be derived from C2, but with private inheritance. This is by far the least common case!

Introduction to Polymorphism

- Let us consider a simple class hierarchy of "window" objects:
  - The base class is a `Window`
  - Each derived class must handle a series of events, including mouse events, typing events, move events, ...
- Here is how `Window` might look:

```
class Window {
  public:
    Window( )
    virtual void MouseDown( int x, int y );
    virtual void MouseUp( int x, int y );
    virtual void KeyDown( int key_id );
    virtual void SetGeometry( int x, int y, int width, int height );
    virtual void Resize( int width, int height );
    int OriginX() const;
    int OriginY() const;
    int Width() const;
    int Height() const;
    bool IsInside( int x, int y ) const;
    std::string WinName() const { return "Base"; }
    // ....
 protected:
    int x0, y0;          // upper left corner
    int width, height;   // dimensions
};
```

- The key is the label `virtual` before the declaration of these functions.
- A `ChatWindow` object would then also have these functions:

```
class ChatWindow : public Window {
 public:
   ChatWindow( int width, int height );
   void MouseDown( int x, int y );
   void MouseUp( int x, int y );
   void KeyDown( int key_id );
   void Resize( int width, int height );
   std::string WinName() const { return "Chat"; }
   // ....
 private
};
```

- Other window objects, such as a `BrowserWindow` or a `ImageWindow` would have similar declarations.

Virtual Functions — An Example

- If we allocate a `ChatWindow` object, such as

  ```
  ChatWindow * cw = new ChatWindow( 5, 120 );
  ```

  then when the code calls

  ```
  cw->MouseDown( 15, 17 );
  ```

  meaning that the user has pressed the (left, usually) mouse button down while at location 15, 17 relative to the upper left corner of the window, the function called is the `MouseDown` defined in the `ChatWindow`

- The same holds for the call in

  ```
  std::cout << "Win name = " << cw->WinName() << std::endl;
  ```

  which outputs

  ```
  Win name = Chat
  ```

- Now, think about what happens when we change the initial allocation slightly to

  ```
  Window * cw = new ChatWindow( 5, 120 );
  ```

- In this case

  ```
  cw->MouseDown( 15, 17 );
  ```

  proceeds as before, but

  ```
  std::cout << "Win name = " << cw->WinName() << std::endl;
  ```

  outputs

  ```
  Win name = Base
  ```

- The difference occurs because `MouseDown` is *virtual* and `WinName` is not.

Virtual Functions

- Consider a pointer `p` to an object in a class hierarchy:

  ```
  ptr_type * p = new obj_type( ... );
  ```

  where `ptr_type` is either the same as `obj_type` or above it in the same class hierarchy.

- Now consider function, `f`, defined in that class hierarchy, then

  - If `f` is defined in the usual way, as a non-virtual function, then `p->f( ... )` calls the version of `f` defined for `ptr_type`, whereas
  - If `f` is defined as a `virtual` function at the level of `ptr_type` (and therefore below it in the hierarchy), then `p->f( ... )` calls the version of `f` defined for `obj_type`

- This explains the difference in the example above.
- It is important that `f` have exactly the same signature in all cases. Even a missing or extraneous `const` creates a different signature.

A Polymorphic List of Windows

The significance of using virtual functions can be seen in the following

- Suppose we have a list of pointers to windows:

```
std::list< Window* > active_windows;
```

- We can fill this list with pointers to specific windows:

```
Window * w = new ChatWindow;
w->SetGeometry( 15, 125, 200, 350 );
active_windows.push_back( w );
w = new ImageWindow();
w->SetGeometry( 300, 600,  480, 620 );
active_windows.push_back( w );
//  ... creation of many more window.
```

- When an event handler object receives a mouse-down signal, with mouse location (x, y) relative to the coordinates of the screen, it must assign the event to one window. For example, we might write

```
for ( std::list<Window*>::iterator p = active_windows.begin();
      p != active_windows.end(); ++p )
  {
    if ( (*p)->IsInside(x,y) )
      {
         (*p)->MouseDown( x - (*p)->OriginX(),
                          y - (*p)->OriginY() );
         return;
      }
  }
return;   // mouse outside all windows, so return silently
```

- While this has dramatically over-simplified the situation, this works because we are able to use polymorphism to store all the pointers to the different window types on the same list. The event handler does not need to know what types of windows it is working with.

Exercise

Here is an exercise to explore the mechanism of virtual functions and polymorphism. Something like it could appear on the final exam.

```
#include <iostream>
using namespace std;

class Base {
 public:
  Base() {}
  virtual void A() { cout << "Base A\n"; }
  void B() { cout << "Base B\n"; }
};

class One : public Base {
 public:
  One() {}
  void A() { cout << "One A\n"; }
  void B() { cout << "One B\n"; }
};

class Two : public Base {
 public:
  Two() {}
  void A() { cout << "Two A\n"; }
  void B() { cout << "Two B\n"; }
};

int main()
{
  Base* a[3];
  a[0] = new Base;
  a[1] = new One;
  a[2] = new Two;
  for ( unsigned int i=0; i<3; ++i )
    {
      a[i]->A();
      a[i]->B();
    }
  return 0;
}
```

Course Summary

- Approach any problem by studying the requirements carefully, playing with hand-generated examples to understand them, and then looking for analogous problems that you already know how to solve.
- The standard library offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming program.
  - Just think how much harder some of the homework problems would have been without generic container classess.

- When choosing between algorithms and between container classes (data structures) you should consider
  - efficiency,
  - naturalness of use, and
  - ease of programming.

  All three are important
- Use classes with well-designed public member functions to encapsulate sections of code.
- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
  - copy constructor,
  - assignment operator, and
  - destructor.

  Work hard to get these correct.
- When testing and debugging:
  - Test one function and one class at a time.
  - Figure out what your program actually does, not what you wanted it to do.
  - Always find the first mistake in the flow of your program and fix it before considering other apparent mistakes
  - Use small examples and boundary conditions when testing.
- Above all, remember the excitement and satisfaction of developing a deep, computational understanding of a problem and turning it into a program that realizes your understanding flawlessly.