Inheritance Details

Modified from Sections 15.1, 15.2

Private is Private

- A member variable (or function) that is private in the parent class is not accessible to the child class
  - The parent class member functions must be used to access the private members of the parent
  - This code would be illegal:

```
void HourlyEmployee::print_check( )
{
        net_pay = hours * wage_rage;
}
```

    - net_pay is a private member of Employee!

The protected Qualifier

- protected members of a class appear to be private outside the class, but are accessible by derived classes
  - If member variables *name*, *net_pay*, and *ssn* are listed as protected (not private) in the Employee class, this code, illegal on the previous slide, becomes legal:

```
void HourlyEmployee::print_check( )
{
        net_pay = hours * wage_rage;
}
```

Programming Style

- Using protected members of a class is a convenience to facilitate writing the code of derived classes.
- Protected members are not necessary
  - Derived classes can use the public methods of their ancestor classes to access private members
- Many programming authorities consider it bad style to use protected member variables

Redefinition of Member Functions

- When defining a derived class, only list the the inherited functions that you wish to change for the derived class
  - The function is declared in the class definition
  - HourlyEmployee and SalariedEmployee each have their own definitions of print_check
- Display 15.8 (1-2) demonstrates the use of the derived classes defined in earlier displays.

Display 15.8 (1/2)

Using Derived Classes (*part 1 of 2*)

```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using std::cout;
using std::endl;
using namespace employeessavitch;

int main( )
{
    HourlyEmployee joe;
    joe.set_name("Mighty Joe");
    joe.set_ssn("123-45-6789");
    joe.set_rate(20.50);
    joe.set_hours(40);
    cout << "Check for " << joe.get_name( )
         << " for " << joe.get_hours( ) << " hours.\n";
    joe.print_check( );
    cout << endl;

    SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
    cout << "Check for " << boss.get_name( ) << endl;
    boss.print_check( );

    return 0;
}
```

The functions set_name, set_ssn, set_rate, set_hours, *and* get_name *are inherited unchanged from the class* Employee.
The function print_check *is redefined.*
The function get_hours *was added to the derived class* HourlyEmployee.

Display 15.8 (2/2)

Back Next

Using Derived Classes (*part 2 of 2*)

Sample Dialogue

```
Check for Mighty Joe for 40 hours.


_____________________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars
_____________________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820
_____________________________________________________

Check for Mr. Big Shot


_____________________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars
_____________________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
_____________________________________________________
```

Redefining or Overloading

- A function **redefined** in a derived class has the same number and type of parameters
  - The derived class has only one function with the same name as the base class
- An overloaded function has a different number and/or type of parameters than the base class
  - The derived class has two functions with the same name as the base class
    - One is defined in the base class, one in the derived class

Function Signatures

- A function signature is the function's name with the sequence of types in the parameter list, not including any const or '&'
  - An overloaded function has multiple signatures
- Some compilers allow overloading based on including const or not including const

Access to a Redefined Base Function

- When a base class function is redefined in a derived class, the base class function can still be used
  - To specify that you want to use the base class version of the redefined function:

```
HourlyEmployee sally_h;
sally_h.Employee::print_check( );
```

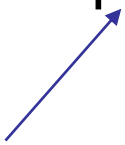
Inheritance Details

- Some special functions are, for all practical purposes, not inherited by a derived class
  - Some of the special functions that are not effectively inherited by a derived class include
    - Constructors
    - Destructors
    - The assignment operator
    - Friendship of the base class

Derived Class Constructors

- A base class constructor is not inherited in a derived class
  - The base class constructor can be invoked by the constructor of the derived class
  - The constructor of a derived class begins by invoking the constructor of the base class in the initialization section:

```
HourlyEmployee::HourlyEmployee : Employee( ),
wage_rate( 0), hours( )
{ //no code needed }
```



Any Employee constructor could be invoked

Default Initialization

- If a derived class constructor does not invoke a base class constructor explicitly, the base class default constructor will be used
- If class B is derived from class A and class C is derived from class B
  - When a object of class C is created
    - The base class A's constructor is the first invoked
    - Class B's constructor is invoked next
    - C's constructor completes execution

Copy Constructors and Derived Classes

- If a copy constructor is not defined in a derived class, C++ will generate a default copy constructor
  - This copy constructor copies only the contents of member variables and will not work with pointers and dynamic variables
  - The base class copy constructor will not be used

The Copy Constructor

- This code segment shows how to begin the implementation of the derived class copy constructor for a derived class:

```
Derived::Derived(const Derived& object)
                              :Base(object), <other initializing>
{…}
```

  - Invoking the base class copy constructor sets up the inherited member variables
    - Since object is of type Derived it is also of type Base

Destructors and Derived Classes

- A destructor is not inherited by a derived class
- The derived class should define its own destructor

Destructors in Derived Classes

- If the base class has a working destructor, defining the destructor for the derived class is relatively easy
  - When the destructor for a derived class is called, the destructor for the base class is automatically called
  - The derived class destructor need only use delete on dynamic variables added in the derived class, and data they may point to

Destruction Sequence

- If class B is derived from class A and class C is derived from class B…
  - When the destructor of an object of class C goes out of scope
    - The destructor of class C is called
    - Then the destructor of class B
    - Then the destructor of class A
  - Notice that destructors are called in the reverse order of constructor calls

Operator = and Derived Classes

- If a base class has a defined assignment operator = and the derived class does not:
  - C++ will use a default operator that will have nothing to do with the base class assignment operator

The Assignment Operator

- In implementing an overloaded assignment operator in a derived class:
  - It is normal to use the assignment operator from the base class in the definition of the derived class's assignment operator
  - Recall that the assignment operator is written as a member function of a class

The Operator = Implementation

- This code segment shows how to begin the implementation of the = operator for a derived class:

```
Derived& Derived::operator= (const Derived& rhs)
{
    Base::operator=(rhs);
    // …
}
```

  - This line handles the assignment of the inherited member variables by calling the base class assignment operator
  - The remaining code would assign the member variables introduced in the derived class