Topic 10

INHERITANCE

Intro to Inheritance

OSTREAM & OSTRINGSTREAM

ostream & ostringstream

When we think about our print header function

```
void PrintHeaderToFile(ofstream &oFile, // IN/OUT - output file
                       string    asName, // IN - assignment Name
                       char      asType, // IN - assignment type
                       int       asNum)  // IN - assignment number
{
        oFile << left;
        oFile << "***************************************************\n";
        oFile << "*  Programmed by : Juan Leon\n";
        oFile << "\n*  " << setw(14) << "Student ID"  << ": 7502312";
        …
```

What is different between this and when we output our function to the screen?

`oFile <<` vs `cout <<`

Remember oFile and cout are variables

- Why can't we pass them in as arguments?

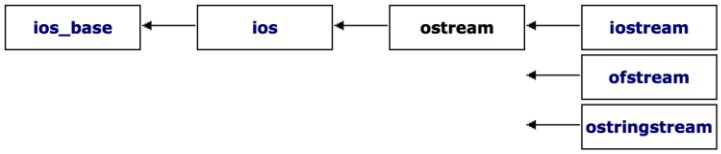
BECAUSE THEY ARE DIFFERENT DATA TYPES!

There are two solutions to writing 2 separate functions

- ostream or
- ostringstream

Output Stream

Output stream datatype can be used to represent different types of output objects such as files, console and output string



We can use an ostream datatype to allow a function to output either to a file or to console (cout)

Ostream

oFile is datatype ofstream
cout is datatype ostream
ofstream is a subtype of ostream

So, we can pass an ofstream variable into an ostream parameter
- But we can't pass an ostream variable into an ofstream parameter
- Why not? ➔ ostream is not a file variable – it can't open or close

We can declare our function like this:

```
void PrintHeader(ostream &output,…
```

And then we have two options for how we can call the function:
- `PrintHeader(cout, …`
- `PrintHeader(oFile, …`

Hence, the calling function decides where the output will go!!!

Ostringstream

Another option is to return the header as a string

```
string PrintHeader(string asName, char asType, int asNum);
```

How can we do that? If we have this in our code:

```
output << "\n*   " << setw(14)
```

Insertion operators and therefore output manipulators only work with output stream variables

The ostringstream datatype solves this
- Acts like a stream
- Easily converts to a string with .str()

You will need to #include <sstream>

Ostringstream (2)

In the function declare an ostringstream variable

```
ostringstream output;
```

Use it as you would an ostream variable

```
output << "\n*   " << setw(14)
```

And return it as a string by using .str()

```
return output.str();
```

This converts the oss to a string

And now we can call it like this:

```
cout << PrintHeader("Functions", 'L', 1);
```

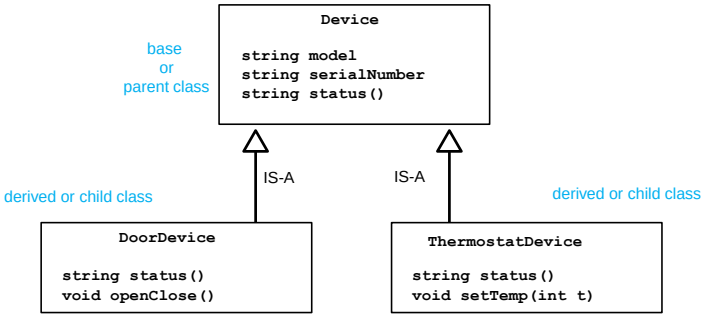
We could have specfied oFile

Inheritance Basics

Inheritance is the process by which a new class, called a derived class, is created from another class, called the base class

- A derived class automatically has all the member variables and functions of the base class
- A derived class can have additional member variables and/or member functions
- The derived class is a child of the base or parent class

Example Inheritance Hierarchy for Home Automation Devices



An object of type `DoorDevice` or `ThermostatDevice` includes functions and variables defined in `Device`, such as `model` and `serialNumber`.

The `status()` function can be *overridden.* If a `DoorDevice` object is treated like a `Device` object, then calling `status()` will invoke `DoorDevice`'s status() function, not `Device`'s `status()` function. This is necessary when the `Device` class doesn't know what to return as a status and only the derived classes can return the information.

Employee Classes

To design a record-keeping program with records for salaried and hourly employees…

- Salaried and hourly employees belong to a class of people who share the property "employee"
- A subset of employees are those with a fixed wage
- Another subset of employees earn hourly wages

All employees have a name and SSN

- Functions to manipulate name and SSN are the same for hourly and salaried employees

A Base Class

We will define a class called Employee for all employees

- The Employee class will be used to define subclasses for hourly and salaried employees

```
class Employee
{
public:
        Employee();
        Employee(string the_name,
                 string the_ssn);

        string get_name() const;
        string get_ssn() const;
        double get_net_pay() const;

        void   set_name(string new_name);
        void   set_ssn(string new_ssn);
        void   set_net_pay(double new_net_pay);
        void   print_check() const;
private:
        string name;
        string ssn;
        double net_pay;
};
```

```
//This is the file: employee.cpp
//Implementation file for Base Class Employee
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"

using namespace std;

Employee::Employee(): name("No name yet"), ssn("No number yet"), net_pay(0)
{}

Employee::Employee(string the_name,
                   string the_number): name(the_name), ssn(the_number), net_pay(0)
{}

string Employee::get_name() const
{
        return name;
}

string Employee::get_ssn() const{
{
        return ssn;
}
```

```
double Employee::get_net_pay() const
{
        return net_pay;
}

void Employee::set_name(string new_name)
{
        name = new_name;
}

void Employee::set_ssn(string new_ssn)
{
        ssn = new_ssn;
}

void Employee::set_net_pay(double new_net_pay)
{
        net_pay = new_net_pay;
}

void Employee::print_check() const
{
        cout << "\nError: print_check FUNCTION CALLED FOR AN \n"
             << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
             << "Check with the author of the program about this bug.\n";
        exit(1);
}
```

Function `print_check`

Function **`print_check`** will have different definitions to print different checks for each type of employee

- An **`Employee`** object lacks sufficient information to print a check
- Each derived class will have sufficient information to print a check

Class `HourlyEmployee`

HourlyEmployee is derived from **Class Employee**

- **HourlyEmployee** inherits all member functions and member variables of **Employee**
- The class definition begins

```
class HourlyEmployee : public Employee
```

Shows that **`HourlyEmployee`** is derived from class **`Employee`**

- **HourlyEmployee** declares additional member variables:
  - **`wage_rate`**
  - **`hours`**

```
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H
#include <string>
#include "employee.h"
using namespace std;

class HourlyEmployee: public Employee
{
public:
        HourlyEmployee();
        HourlyEmployee(string the_name,
                       string the_ssn,
                       double the_wage_rate,
                       double the_hours);

        void   set_rate(double new_wage_rate);
        double get_rate() const;
        void   set_hours(double hourse_worked);
        double get_hours() const;
        void   print_check();
private:
        double wage_rate;
        double hours;
};
#endif //HOURLYEMPLOYEE_H
```

Inherited Members

A derived class inherits all the members of the parent class

- The derived class does not re-declare or re-define members inherited from the parent, except…
  - The derived class re-declares and re-defines member functions of the parent class that will have a different definition in the derived class
  - The derived class can add member variables and functions

Implementing a Derived Class

Any member functions added in the derived class are defined in the implementation file for the derived class

- Definitions are not given for inherited functions that are not to be changed

```
#include <string>
#include <iostream>
#include "hourlyemployee.h"
using namespace std;

HourlyEmployee()::HourlyEmployee(): Employee(), wage_rate(0), hours(0){}

HourlyEmployee()::HourlyEmployee(string the_name,
                                 string the_number,
                                 double the_wage_rate,
                                 double the_hours)
                                 :Employee(the_name, the_number),
                                  wage_rate(the_wage_rate), hours(the_hours){}

void HourlyEmployee::set_rate(double new_wage_rate)
{
        wage_rate = new_wage_rate;
}

double HourlyEmployee::get_rate() const
{
        return wage_rate;
}
```

```
void HourlyEmployee::set_hours(double hours_worked)
{
        hours = hours_worked;
}

double HourlyEmployee::get_hours() const
{
        return hours;
}

void HourlyEmployee::print_check()
{
        set_net_pay(hours * wage_rate);

        cout << "\n_____________________________________________\n";
        cout << "Pay to the order of " << get_name() << endl;
        cout << "The sum of " << get_net_pay() << " Dollars\n";
        cout << "\n_____________________________________________\n";
        cout << "Check Stub: NOT NEGOTIABLE\n";
        cout << "Employee Number: " << get_ssn() << endl;
        cout << "Hourly Employee. \nHours worked: " << hours;
             << " Rate: " << wage_rate << " Pay: " << get_net_pay() << endl;
        cout << "_____________________________________________\n";
}
```

Class SalariedEmployee

The class **SalariedEmployee** is also derived from Employee

- Function `print_check` is redefined to have a meaning specific to salaried employees
- **SalariedEmployee** adds a member variable **salary**

```
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

class SalariedEmployee : public Employee
{
public:
    SalariedEmployee();
    SalariedEmployee(string the_name,
                     string the_ssn,
                     double the_weekly_salary);

    double get_salary() const;
    void   set_salary(double new_salary);
    void   print_check();
private:
    double salary; //weekly
};
#endif
```

```
#include <iostream>
#include <string>
#include "salariedemployee.h"
using namespace std;

SalariedEmployee::SalariedEmployee() : Employee(), salary(0)
{}

SalariedEmployee::SalariedEmployee(string the_name,
                                   string the_number,
                                   double the_weekly_salary)
                                   :Employee(the_name, the_number), salary(the_weekly_salary)
{}

double SalariedEmployee::get_salary() const
{
    return salary;
}

void SalariedEmployee::set_salary(double new_salary)
{
    salary = new_salary;
}
```

```
void SalariedEmployee::print_check()
{
        set_net_pay(salary);

        cout << "\n_____________________________________________\n";
        cout << "Pay to the order of " << get_name() << endl;
        cout << "The sum of “ << get_net_pay() << " Dollars\n";
        cout << "\n_____________________________________________\n";
        cout << "Check Stub: NOT NEGOTIABLE\n";
        cout << "Employee Number: " << get_ssn() << endl;
        cout << “Salaried Employee. \nRegular Pay: " << salary;
        cout << "_____________________________________________\n";
}
```

Parent and Child Classes

Recall that a child class automatically has all the members of the parent class

The parent class is an ancestor of the child class

The child class is a descendent of the parentclass

The parent class (**Employee**) contains all the code common to the child classes
- You do not have to re-write the code for each child

Derived Class Types

An hourly employee is an employee

- In C++, an object of type **HourlyEmployee** can be used where an object of type **Employee** can be used
- An object of a class type can be used wherever any of its ancestors can be used
- An ancestor cannot be used wherever one of its descendants can be used

Derived Class Constructors

A base class constructor is not inherited in a derived class

- Base class constructor can be invoked by the constructor of the derived class
- 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(0){}
```

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

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:

```
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 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**

```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using namespace std;

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;
}
```

```
Check for Might 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
  - Destructors
  - Copy constructors
  - The assignment operator

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

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

Destructors and Derived Classes

A destructor is not inherited by a derived class

The derived class should define its own destructor

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

The Copy Constructor

Implementation of the derived class copy constructor is much like that of the assignment operator:

```
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 in Derived Classes

If the base class has a working destructor, defining the destructor for the defined 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