Comp151

Inheritance: Abstract Base Class

ABC Example: personal_asset.hpp

- Let's design a system for maintaining our assets: stocks, bank accounts, real estate, horses, cars, yachts, etc.
- Each asset has a net worth (value), we would like to be able to make listings and compute total net worth.

```
class Personal_Asset {
public:
    Personal_Asset(const Date& purchase_date);

    virtual double compute_net_worth() const;  // What is asset’s current net worth?
    virtual bool is_insurable() const;         // Can this asset be insured?
    void set_purchase_date(const Date& d);

private:
    Date purchase_date;
};
```

ABC Example: bank_asset.hpp

- There are different kinds of assets, and they are all derived from Personal_Asset, e.g.

```
class Bank_Account_Asset : public Personal_Asset
{
public:
   // ...
   virtual double compute_net_worth() const { return balance; }
private:
   double balance;
   double interest_rate;
};
```

ABC Example: asset_fcn.cpp

- There can be other classes of assets such as Car_Asset, Stock_Asset, House_Asset, etc.
- To compute the total asset value for an array of assets:

```
double compute_total_worth(const Personal_Asset* assets[], int size)
{
   double total_worth = 0.0;
   for (int i = 0; i < size; ++i) {
      total_worth += assets[i]->compute_net_worth();  // virtual function call
   }
   return total_worth;
}
```

- Things must be arranged so that this will work for any combination of assets of different kinds.

ABC Example: asset_base.cpp

- But now we have to implement the methods of the base class Personal_Asset:

```
Personal_Asset::Personal_Asset(const Date& date)
    : purchase_date(date) { }

void Personal_Asset::set_purchase_date(const Date& date) {
    purchase_date = date;
}

double Personal_Asset::compute_net_worth() const {
    /* return what ??? */
}
```

- How should we implement compute_net_worth( )? It depends completely on the type of the asset. There is no “standard way” of doing it – no meaningful “default method” to compute net worth!

ABC Example: compute net_worth()??

- The truth is: <u>It makes no sense to have objects of type Personal_Asset</u>.
- Such an object has only a purchase date, but otherwise no meaning. It is not a bank account, not a car, not a house – it is too general (too abstract) to be used.
- We cannot implement the compute_net_worth( ) method in the base class Personal_Asset as the information needed to implement it is missing.
- However, we do not want to remove the method, because that would make it impossible to write a function that depends on polymorphism, such as compute_total_worth( ).

Solution: Abstract Base Class (ABC)

- The solution is to make Personal_Asset an abstract base class (or ABC for short):

```
class Personal_Asset {
public:
    Personal_Asset(const Date& purchase_date);

    virtual double compute_net_worth() const = 0;  // What is asset’s current net worth?
    virtual bool is_insurable() const;             // Can this asset be insured?
    void set_purchase_date(const Date& d);

private:
    Date purchase_date;
};
```

- compute_net_worth( ) has become a pure virtual function or pure virtual method.
- Any class that has one or more pure virtual methods is an ABC.

Abstract Base Class (ABC)

- An ABC has two properties:

  1. There cannot be objects of that type.

     ```
     Personal_Asset pa("2000.01.07");                  // error
     Bank_Account_Asset baa(“2002.01.01", 0.0);  // ok
     ```

  2. Derived classes are responsible for implementing the pure virtual methods.

- If a derived class (for instance, Securities_Asset) does not implement the pure virtual methods, then the derived class is also abstract, and there cannot be objects of that type (but it can be used as a base class itself, for instance for Stocks_Asset, Bonds_Asset, etc.)

Interface reuse

- *“An abstract base class provides a uniform interface to deal with a number of different derived classes.”*
  - A base class contains what is <u>common</u> about several classes.
  - If the only thing that is common is the <u>interface</u>, then the base class is a “<u>pure interface</u>”, called an <u>ABC</u> in C++.
  - We discussed before that code reuse is a major advantage of inheritance. With <u>pure virtual functions</u> we do not directly reuse code, but create an interface that can be reused by derived classes.
  - Interfaces are the soul of object-oriented programming. They are the most effective way of separating use and implementation of objects. The user [i.e., compute_total_worth( )] only knows about the abstract interface, while we can have many objects that implement this interface in different ways.
  - In C++, an ABC serves a similar purpose as a Java “interface”.

Final Remark

- Pure virtual functions are inherited as pure virtual functions unless the derived class implements the function.
- An abstract base class cannot be used
  - as an argument type (called by value)
  - as a function return type (returned by value)
  - as the type of an explicit conversion
- However, pointers and references to an ABC can be declared.
- Calling a pure virtual function from the constructor of an ABC is undefined – DON'T do that.

Example: “Do”s and “Don’t”s

```
Personal_Asset x(" 2002.01.01 ");  // Error: can't create objects of ABC
Personal_Asset f1() { … }  // Error: Can't return ABC objects
void f2(Personal_Asset x) {… }  // Error: Can't CBV with ABC objects
Bank Account_Asset y(“2002.01.01", 0.0);  // Ok!
Personal_Asset* passet = &y;  // Ok!
Personal_Asset& rasset = y;  // Ok!
Personal_Asset* f3(const Personal_Asset& x) {…}  // Ok!
```