Queens College, CUNY, Department of Computer Science
**Object Oriented Programming in C++**
**CSCI 211 / 611**
**Summer 2018**
Instructor: Dr. Sateesh Mane



September 7, 2018

# Polymorphism: Part I

- In this lecture we shall learn about the important concept of **polymorphism**.
- Polymorphism makes essential use of inheritance.

# 1 Introduction

- The term "**polymorphism**" derives from two root words *poly* and *morph*.
- The term "**poly**" means "many" and "**morph**" means to change shape or appearance.
- Hence "**polymorphism**" means the ability of an object to have many different appearances.
- In the context of inheritance, we have seen that an object of a derived class "IS-A" object of the base class.
- Polymorphism takes the concept a step futher.
- It is an important step, and allows some very general/powerful/flexible software to be written.

# 2 Motivation

## 2.1 Basic design

- Let us motivate the idea of polymorphism with an example.
- We wish to calculate the sum of an array of $n$ values $x_i$, $i = 0, \ldots, n-1$, in three ways:

$$S_1 = x_0 + \cdots + x_{n-1}\,,$$
$$S_2 = |x_0| + \cdots + |x_{n-1}|\,,$$
$$S_3 = x_0^2 + \cdots + x_{n-1}^2\,.$$

- We implement the following software design:
  1. We write a base class `SumBase` to hold the array $x$, with accessor and mutator methods.
  2. The base class `SumBase` has a method `double sum()`, which simply returns 0.
  3. We declare three derived classes `SumVec`, `SumAbs` and `SumSq` to override `double sum()` to compute the relevant sums.
- The class declarations are displayed in Sec. 2.2 below.
- A main program to instantiate and use objects of the derived classes is given below.

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

// class declarations

int main()
{
  vector<double> x = {-1, 0, 1, 2};
  SumVec sv;
  SumAbs ab;
  SumSq  sq;
  sv.set(x);
  ab.set(x);
  sq.set(x);
  cout << sv.sum() << "   " << ab.sum() << "   " << sq.sum() << endl;
  return 0;
}
```

- The output is 2  4  6. (This will be significant below.)
- **Analysis and comments are given in Sec. 2.3.**

### 2.2 Class declarations SumBase, SumVec, SumAbs, Sumsq

```
class SumBase {
public:
  vector<double> get() const { return x; }
  void set(vector<double> &v) { x = v; }

  double sum() { return 0; }                  // override in derived class

protected:
  vector<double> x;
};

class SumVec : public SumBase {
public:
  double sum() {
    double s = 0;
    for (int i = 0; i < x.size(); ++i) {
      s += x[i];                                  // sum S1
    }
    return s;
  }
};

class SumAbs : public SumBase {
public:
  double sum() {
    double s = 0;
    for (int i = 0; i < x.size(); ++i) {
      s += abs(x[i]);                             // sum S2
    }
    return s;
  }
};

class SumSq : public SumBase {
public:
  double sum() {
    double s = 0;
    for (int i = 0; i < x.size(); ++i) {
      s += x[i] * x[i];                           // sum S3
    }
    return s;
  }
};
```

### 2.3 Comments & criticism

- Let us analyze the above software design.
- The first and most obvious comment is: *why do we need to write three classes?*
  1. We can perform all the calculations using only one class, call it `MiscSum`, by writing three methods, named `sum1()`, `sum2()` and `sum3()`.

     ```
     class MiscSum {
     public:
       double sum1();                          // calculate S1
       double sum2();                          // calculate S2
       double sum3();                          // calculate S3
       // etc
     };
     ```

  2. This is a valid criticism, for such a simple example.
  3. However, what if we want to add a fourth sum, say $S_4 = x_0^4 + \cdots + x_{n-1}^4$?
  4. We would have to add an extra method to `MiscSum` and recompile all the code.
  5. In a large project, a lot of code might have to be recompiled and tested.
- Hence the second comment is the reverse of the first: *why not write three independent classes?*
  1. If we wish to compute a new sum $S_4$, we simply write a new class wihch is independent of the existing classes.
  2. Clearly the existing classes do not need to be recompiled.
  3. This idea also works, but it leads to a lot of duplication of code.
  4. If a new method were added to the project, for example `print()`, in the current design we add it to the base class and it would be accessible to all the derived classes.
  5. If the classes were independent, we would have to copy and paste `print()` in each class.
  6. It is therefore not a good idea to write independent classes.
- These are the advantages of using inheritance:
  1. **We can add new classes to the inheritance tree and the previously defined classes are not affected.**
  2. If we need to make changes to the existing classes, **by putting the changes in the base class the new functionality is available to the full inheritance tree.**
  3. Individual classes can override the new methods for their own purposes, if necessary.
  4. Hence inheritance helps to avoid duplication of code and duplication of functionality.
- ***Nevertheless, we can write a better software design.***

## 3 "IS-A" and sample library function

- We know that a derived class "IS-A" base class.
- A reference/pointer to the base class can bind/point to an object of a derived class.
- Let us do the following.
  1. We add a public method `print()` to the base class `SumBase`.
  2. As stated above, the method `print()` is accessible to all the derived classes.

```
class SumBase {
public:
  vector<double> get() const { return x; }
  void set(vector<double> &v) { x = v; }

  double sum() { return 0; }                  // override in derived class

  void print() const {
    for (int i = 0; i < x.size(); ++i) {
      cout << x[i] << "  ";
    }
    cout << endl;
  }

protected:
  vector<double> x;
};
```

- We also write a function "`library_func`" to call both `print()` and `sum()`.
  1. As the name suggests, it is the beginning of a software library.
  2. It will not work exactly correctly at first, but we shall examine why and improve on it.

```
void library_func(SumBase &sb_ref)              // reference to base class
{
  sb_ref.print();
  cout << "sum = " << sb_ref.sum() << endl;
  cout << endl;
}
```

- **See next page(s).**

### 3.1 Example main program #2

- Here is a main program to use the new function.
- The output of the function call is not completely correct.
  1. For every function call, the function prints the contents of the vector correctly.
  2. **However, the value of the sum is zero every time.**

  $-1 \quad 0 \quad 1 \quad 2$ // correct

  **sum = 0** **// **wrong** sum is always zero**
- The weakness is that the reference `sb_ref` **binds only to the base class part of the input object.**
- Therefore the reference `sb_ref` invokes only the base class version of the method `sum()`.
- The reference `sb_ref` does not know that the derived classes override the method `sum()`.
- We shall correct this weakness in Sec. 4.

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

// class declarations

void library_func(SumBase &sb_ref)              // reference to base class
{
  sb_ref.print();
  cout << "sum = " << sb_ref.sum() << endl;
  cout << endl;
}

int main()
{
  vector<double> x = {-1, 0, 1, 2};
  SumVec sv;
  SumAbs ab;
  SumSq  sq;
  sv.set(x);
  ab.set(x);
  sq.set(x);

  library_func(sv);
  library_func(ab);
  library_func(sq);
  return 0;
}
```

## 4 Polymorphism: virtual functions

- This is where **polymorphism** enters the picture.
- Let us make a peculiar looking but significant change to the method `sum()`.
- We tag `sum()` as a "**virtual**" method.

```
class SumBase {
public:
  vector<double> get() const { return x; }
  void set(vector<double> &v) { x = v; }

  virtual double sum() { return 0; }              // keyword "virtual"

  void print() const {
    for (int i = 0; i < x.size(); ++i) {
      cout << x[i] << "  ";
    }
    cout << endl;
  }

protected:
  vector<double> x;
};
```

- **With this seemingly minor change, the program in Sec. 3.1 works correctly.**
- The program output is now the following:

```
-1  0  1  2
sum = 2                 // correct

-1  0  1  2
sum = 4                 // correct

-1  0  1  2
sum = 6                 // correct
```

- *What has happened?*
- **See next page(s).**

## 5 Keyword "`virtual`"

- The keyword "**virtual**" makes a world of difference.
- The keyword **virtual** tells the compiler that a "reference to the base class" is actually bound to an object which **may belong to a derived class.**
- If a method is tagged as virtual, **the compiler searches through the inheritance tree until it finds the correct override.**
- The same concept applies also to pointers: if a pointer (to the base class) calls a virtual method, the compiler searches through the inheritance tree until it finds the correct override.
- Notice something that was not explicitly stated above:
  1. We tagged the keyword "virtual" **only to the method `sum()` in the base class.**
  2. We did not tag the method `sum()` in the derived classes as virtual.
  3. **If a method in a class is tagged as virtual, then the same method is automatically virtual in all the derived classes above it in the inheritance tree.**
  4. Hence by tagging `sum()` as virtual in the base class, it is automatically virtual in all the derived classes.

## 6 Why `virtual`?

- The word "virtual" signifies that something is not real.
- That is exactly correct.
- In our example, just because `sb_ref` is a reference to the base class, that does not mean the base class version of `sum()` will be invoked.
- The method `sum()` is virtual.
- We do not know which version of `sum()` will really be invoked.
- During program execution, the function call "`sb_ref.sum()`" may invoke many different versions of `sum()`.

# 7 Library function

- Let us examine the "library function" again, with the method `sum()` declared as virtual.

```
void library_func(SumBase &sb_ref)              // reference to base class
{
  sb_ref.print();
  cout << "sum = " << sb_ref.sum() << endl;
  cout << endl;
}
```

- What would happen if we wrote a new derived class to implement the sum $S_4$?

  1. *The above library function will work correctly.*
  2. Because the method `sum()` was declared virtual in the base class, it is automatically virtual in every derived class.
  3. Hence the above function will work correctly, without modification.
  4. The compiler will search and find the correct override of the method `sum()`.

- **Therefore the above function can be placed in a software library.**

  1. Users of the library are told what the base class `SumBase` is.
  2. (Actually the users are only told what the public part of `SumBase` is.)
  3. The library manual states that `sum()` is a virtual function.
  4. The users write classes which derive from `SumBase`, and override `sum()` for their purposes.
  5. They call the library function and it prints the correct output, using their derived class.

## 8 Polymorphism and interfaces

- We see now the great flexibility and generality which polymorphism opens up to us.
- We can write a complete software library using polymorphism.
    1. We write a collection of base classes.
    2. We declare virtual functions in our base classes.
    3. We write complete working code using entirely base class pointers and/or references.
    4. The term "complete working code" means not only the function bodies of the base classes, but also library functions such as the above example.
- *And why is the above library polymorphic?*
    1. Users write derived classes, which inherit from our collection of base classes.
    2. **We do not know what the derived classes will be.**
    3. Different users will (in general) write different derived classes.
    4. When we write the library, we only know about our own collection of base classes.
    5. However, in reality the pointers and references in our library are polymorphic: they will act correctly for a wide variety of derived classes, and as stated above, we do not know what those derived classes will be.
- **This is polymorphism:**
  **Our library is written entirely using our collection of base classes, but nevertheless it will work for an unknown variety of user-defined derived classes.**
- The pointers and references in our library take on many appearances: they are polymorphic.

## 9 Notes on the use of the keyword "`virtual`"

- Data members cannot be tagged as virtual.
- Static methods cannot be tagged as virtual.
  1. Think about it: a static method belongs to the class, not to an object.
  2. Hence there is no concept of "object of a derived class" for a static method.
- **Constructors cannot be virtual.**
- **The destructor can be tagged as virtual. This is a very important fact that we shall discuss later.**
- **Overloaded operators can be tagged as virtual.**
  1. The assignment operator can be tagged as virtual, which can lead to difficulties.
  2. Is is safer not to mess with the assignment operator.
- A `virtual` method can be tagged as `const`.
- A `virtual` method can be public, private or protected.
- Virtual methods can internally call non-virtual methods.
- Non-virtual methods can internally call virtual methods.
- **Static methods cannot call virtual methods.**
  1. For that matter static methods cannot call non-virtual non-static methods either.
  2. Static methods can only call other static methods.
- The "`this`" pointer is always available in a virtual method (because a virtual method is non-static) and always points to the correct object in the inheritance tree.
- *Can all the non-static methods of a class be declared virtual?*
  Almost . . . not the constructors.
- This is perhaps a bit strange, but is legal in C++. The privacy level of a virtual function does not have to be the same in all the classes of the inheritance tree.
  1. A virtual function which is public in the base class can be declared as public/private/protected in a derived class.
  2. *The virtual function does not have to be public in the derived class.*
  3. A virtual function which is protected in the base class can be declared as public/private/protected in a derived class.
  4. **This means that a virtual function which is protected in the base class can be made public in a derived class.**

## 10 Non-virtual methods

- If a method is non-virtual, the compiler does not search the inheritance tree for an override.
- In the "`library_func`" example, the method `print()` is not virtual.
- If a derived class overrides `print()`, the reference `sb_ref` will only call the `print()` method in the base class and not the one in the derived class.
- **Hence the keyword `virtual` is selective: the compiler searches the inheritance tree only if a method is virtual.**

## 11 Why not declare every class method as virtual?

- It is a matter of quality control.
- If we write a library where every method in our base classes is virtual, the users can essentially employ our library to do almost anything.
- The users can override all of our base class methods and our library becomes meaningless.
- Sometimes, there are functions where it is important that the computations must be done in a particular way. This may not necessarily be for programming reasons. It may be for legal reasons.
- Hence we write a non-virtual function to perform the relevant computations.

## 12 Virtual function table (advanced topic not for examination)

- In the inheritance tree, it is easy to work downwards from a derived class to the base class, to figure out the appropriate override to use for a class method.
- ***However, for a virtual function, the complier must go in the opposite direction.***
    1. The compiler is given a pointer or a reference **to the base class.**
    2. The compiler is told that a method is virtual.
    3. *How does the complier go UP the inheritance tree, to determine the appropriate override?*
- Remember that when we write a polymorphic software library, we do not know what the derived classes will be. The derived classes do not exist yet.
- If a class has virtual functions, the compiler creates a **virtual function table.**
    1. We shall not analyze the contents or operation of the virtual function table.
    2. When a derived class is written, which overrides a virtual function, an entry is made in the virtual function table.
    3. This is obviously something which must happen at run time, not at compile time, since we compile our library without knowing anything about the derived classes.
- **The existence of the virtual function table has the consequence that the memory allocation to create an object is not equal to the sum of the sizes of all the class data members.**
    1. The compiler needs to allocate memory for the virtual function table also.
    2. **That is why the size of an object should always be determined by calling the "`sizeof`" operator.**
    3. The compiler automatically generates the "`sizeof`" operator for every user-defined class, which correctly takes into account the virtual function table.

# 13 WARNING

- **Never call a virtual function in a constructor or destructor.**
- This is serious mistake that some programmers make.
- Suppose `CVF` ("constructor virtual function") is a base class.
- Suppose furthermore the constructor of `CVF` calls a virtual function.

```
class CVF {
public:
  CVF() { f(); }
  virtual void f();
  // etc
};
```

- When an object of a derived class is instantiated, the base class constructor is called first.
  1. The problem with the virtual function call is that the object is still under construction.
  2. **Hence the "derived class" part of the object has not been created yet.**
  3. **Hence the virtual function table is still incomplete.**
  4. Hence the compiler (actually the run-time system) invokes the wrong virtual function, with possibly unpredictable and/or dangerous consequences.
- Calling a virtual function in a constructor is a **serious programming error.**
- Similarly, calling a virtual function in a destructor is also a serious programming error.
- *Do not mess with the virtual function table when an object is being created or destroyed.*