Operator Overloading

Operators as members VS non-member

Whether an operator function is implemented as a member function or as a non-member function, *the operator is still used the same way in expressions*. So which is best?

- When an operator function is implemented as a member function, the **leftmost** (or only) **operand must be an object** (or a **reference** to an object) **of the operator's class.**
- If the left operand must be an object of a *different class* or a *fundamental type*, this operator function must be implemented as a non-member function.
- A non-member operator function can be made a **friend** of a class if that function must access private or protected members of that class directly.
- The same operator function cannot be implemented **both** as a member and non-member function:

  Integer Integer::operator+(const Integer& rhs);

  Integer operator+(const Integer& lhs, const Integer& rhs);

  Compiler error: Use of overloaded operator '+' is ambiguous.

- Operator member functions of a specific class are called (implicitly by the compiler) **only when** the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.

Operators as members VS non-member

Another reason why you might choose a non-member function to overload an operator is to enable the operator to be **commutative**.

Suppose we want to support the following expressions for **int** and **Integer** class:

- Integer + Integer
- Integer + int
- int + Integer

Thus, we require the addition operator to be ***commutative.***

- The problem is that the class object must appear on the **left** of the addition operator if that operator is to be overloaded as a **member function.**
- So, we also need to overload the operator as a **non-member function** to allow the int+Integer syntax.
- The "Integer Integer::operator+(const Integer& rhs)" can be member or non-member function.
- The "Integer Integer::operator+(int rhs)" can be member or non-member function.
- The "Integer operator+(int lhs, const Integer& rhs)" should be non-member function and can **simply** call 1) operator**+(Integer lhs, int rhs)** or 2) **"rhs+lhs"** by calling member function "Integer Integer::operator+(int rhs)".

Operators as members VS non-member

```
class Integer {
public:
   Integer(int i=0)
      : _i(i)
   {
      std::cout << "Integer::Integer(" << _i << ") called" << endl;
   }

   Integer operator+(const Integer& rhs) const {
      std::cout << "Integer::operator+(const Integer& rhs) called" << endl;
      return Integer(_i + rhs._i);
   }

private:
   int _i;
};
```

- Note!: **const** function allows “const_Integer + Integer” expression.

```
int main()
{
    Integer i1(1);
    Integer i2(2);
    Integer i3(0);

    i3 = i1 + i2; // OK
    i2 = i1 + 10; // OK
//    i2 = 10 + i1; // Compile Error
}
```

Result:

```
Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

Integer::operator+(const Integer& rhs) called
Integer::Integer(3) called

Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called
```

Operators as members VS non-member

```
class Integer {
   friend Integer operator+(int lhs, const Integer& rhs);
public:
…
   Integer operator+(const Integer& rhs) const {
      std::cout << "Integer::operator+(const Integer& rhs) called" << endl;
      return Integer(_i + rhs._i);
   }
…
};

Integer operator+(int lhs, const Integer& rhs)
{
   std::cout << "operator+(int, const Integer&) called" << endl;
   return rhs + lhs;
}
```

```
int main()
{
   Integer i1(1);
   Integer i2(2);
   Integer i3(0);

   i3 = i1 + i2; // OK
   i2 = i1 + 10; // OK
   i2 = 10 + i1; // OK
}
```

Result:
Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

Integer::operator+(const Integer& rhs) called
Integer::Integer(3) called

Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called

operator+(int, const Integer&) called
Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called

Or we could write only the non-member operator+(Integer, Integer):

```
class Integer {
   friend Integer operator+(const Integer& lhs, const Integer& rhs);
public:
 Integer(int i=0)
      : _i(i)
   {
      std::cout << "Integer::Integer(" << _i << ") called" << endl;
   }
private:
   int _i;
};

Integer operator+(const Integer& lhs, const Integer& rhs)
{
   std::cout << "operator+(Integer, Integer) called" << endl;
   return Integer(lhs._i + rhs._i);
}
```

And all 3 versions will work!

```
int main()
{
    Integer i1(1);
    Integer i2(2);
    Integer i3(0);

    i3 = i1 + i2; // OK
    i2 = i1 + 10; // OK
    i2 = 10 + i1; // OK
}
```

Result:

Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

operator+(Integer, Integer) called
Integer::Integer(3) called

Integer::Integer(10) called
operator+(Integer, Integer) called
Integer::Integer(11) called

Integer::Integer(10) called
operator+(Integer, Integer) called
Integer::Integer(11) called

Operators as members VS non-member

If we know that overriding all the possible combinations of the operator+ will bring **performance benefits**, then we should provide that versions.

For example, see std::string class operator+ functions defined in <string> header file:

1. string operator+ (const string& lhs, const string& rhs);
2. string operator+ (const string& lhs, const char* rhs);
3. string operator+ (const char* lhs, const string& rhs);
4. string operator+ (const string& lhs, char rhs);
5. string operator+ (char lhs, const string& rhs);

For example, if the function-2 of std::string will not be provided, then the string(const char* rhs) constructor will be called and the function-1 will do the + operation.

However, directly working with “const char* rhs” in **function-2 will save one string(const char* rhs) constructor call**.

Converting between types

The compiler knows how to perform certain conversions among fundamental types:

```
int i=456;
double d = i;
```

But what about user-defined types?

For example, how to convert std::string to Message object?

Such conversions can be performed with **conversion constructors**—constructors that can be called with a single argument (we'll refer to these as **single-argument constructors**).

Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

Converting between types

```
class Message {
   public:
   Message(std::string s) : _msg(s) {
      std::cout << "Message::Message(std::string) called" << std::endl;
   }
   std::string _msg;
};

void f(Message msg)
{
   std::cout << "f(Message) called: Message._msg = " << msg._msg << std::endl;
}

int main()
{
   std::string str("Hello World");
   f(str);
}
```

Result:
Message::Message(std::string) called
f(Message) called: Message._msg = Hello World

Conversion constructor Message(std::string) was used to *convert* <u>std::string</u> object to <u>Message</u> object.

Converting between types

But how to convert Message object to std::string object?

Conversion operator (also called a cast operator) also can be used to convert an object of one class to another type.

Such a conversion operator must be a **non-static member function**.

Message::operator string() const;

declares an overloaded cast operator function for converting an object of class **Message** into a temporary **string** object.

The operator function is declared **const** because it does *not* modify the original object.

The return type of an overloaded cast operator function **is implicitly the type to which the object is being converted**.

Converting between types

When the compiler sees the expression

static_cast<std::**string**>(**message**)

it generates the call

message.operator **string**()

One of the nice features of cast operators and conversion constructors is that, when necessary, **the compiler can call these functions *implicitly* to create *temporary* objects.**

```
class Message {
  public:
  Message(std::string s) : _msg(s) {
    std::cout << "Message::Message(std::string) called" << std::endl;
  }
  operator string()  const {
    std::cout << "Message::operator string() called" << std::endl;
    return _msg; // Note that the return type is implicitly defined as std::string
  }
  std::string _msg;
};

void f(std::string s)
{
  std::cout << "f(std::string) called: s=" << s << std::endl;
}

int main()
{
  Message msg("Hello World");
  f(msg);
}
```

Result:

```
Message::Message(std::string) called
Message::operator string() called
f(std::string) called: s=Hello World
```

Converting between types

- When a conversion constructor or conversion operator is used to perform an implicit conversion, C++ can apply **only one** implicit constructor or operator function call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator.

- **The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions**.

```
class A {
   public:
...
   operator int () const {
      return _i;
   }
   private:
   int _i;
};
```

```
class Message {
   public:
...
   operator A() const {
      return A(0);
   }
...
};
```

```
void g(int i)
{
   std::cout << "g(int) called: i=" << i << std::endl;
}

int main()
{
   Message msg("Hello World");
   g(msg);
}
```

Compile Error: cannot convert ‘Message’ to ‘int’

- Compiler will **NOT** be able to convert **Message->A->int.**

Explicit Constructors and Conversion Operators

Any constructor that can be called with a single argument and is not declared **explicit** can be used by the compiler to perform an implicit conversion.

The constructor's argument is converted to an object of the class in which the constructor is defined.

The **conversion is automatic -** a cast is not required.

- Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

Explicit Constructors and Conversion Operators

~~explicit~~ Array(int = 10); // default constructor

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

void outputArray(const Array&); // prototype

int main() {
   Array integers1{7}; // 7-element Array
   outputArray(integers1); // output Array integers1
   outputArray(3); // convert 3 to an Array and output Array's contents
}

// print Array contents
void outputArray(const Array& arrayToOutput) {
   cout << "The Array received has " << arrayToOutput.getSize()
      << " elements. The contents are:\n" << arrayToOutput << endl;
}
```

```
The Array received has 7 elements.
The contents are: 0   0   0   0   0   0   0

The Array received has 3 elements.
The contents are: 0   0   0
```

The compiler assumes the constructor is a *conversion constructor* and uses it to convert the argument 3 into a <u>temporary Array object containing three elements</u> (**but we would not want this to be happening**!).

Explicit Constructors and Conversion Operators

explicit Array(int = 10); // default constructor

```
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray(const Array&); // prototype
8
9  int main() {
10    Array integers1{7}; // 7-element Array
11    outputArray(integers1); // output Array integers1
12    outputArray(3); // convert 3 to an Array and output Array's contents
13    outputArray(Array{3}); // explicit single-argument constructor call
14 }
15
16 // print Array contents
17 void outputArray(const Array& arrayToOutput) {
18    cout << "The Array received has " << arrayToOutput.getSize()
19       << " elements. The contents are:\n" << arrayToOutput << endl;
20 }
```

- Line 12: Complie Error: 'void outputArray(const Array &)': cannot convert argument 1 from 'int' to 'const Array &'.
- In line 13 we explicitly create an Array object.

❖ **Always** use the **explicit** keyword on single-argument constructors **unless** they're **intended** to be used as **conversion constructors**.

Explicit Constructors and Conversion Operators

[C++11]: **explicit Conversion Operators**

Just as you can declare single-argument constructors explicit, **you can declare conversion operators explicit** to prevent the compiler from using them to perform implicit conversions.

For example, the prototype

explicit Integer::operator string() const;

declares **Integer**'s **string** cast operator **explicit**, thus requiring you to invoke it explicitly with **static_cast**.

```
void f(std::string s)
{ … }
…
Integer i1(1);
f(i1); // Compile Error, since Integer::operator string() is explicit
f(static_cast<std::string>(i1)); // OK
```

Overloading the Function Call Operator ()

Overloading the **function call operator ()** is powerful, because functions can take an arbitrary number of comma-separated parameters.

In a customized String class, for example, you could overload this operator to select a **substring** from a String.

The overloaded function call operator **must be a non-static member function** and could be defined as

```
String String::operator()(size_t index, size_t length) const
```

In this case, it should be a **const** member function because obtaining a **substring** should not modify the original String object.

For example:
String s("Hello World");
cout << s(6, 5); // prints "World"

The compiler translates s(6, 5) as follows:

s(6,5) -> s.operator()(6, 5)

➢ Please see [Appendix] for String operator() implementation.

[Appendix] String class example

```
// String.h
// String class definition.
#ifndef STRING_H
#define STRING_H

#include <iostream>
using std::ostream;
using std::istream;

class String
{
   friend ostream &operator<<( ostream &, const String & );
   friend istream &operator>>( istream &, String & );
public:
   String( const char * = "" ); // conversion/default constructor
   String( const String & ); // copy constructor
   ~String(); // destructor

   const String &operator=( const String & ); // assignment operator
   const String &operator+=( const String & ); // concatenation operator
```

```
bool operator!() const; // is String empty?
bool operator==( const String & ) const; // test s1 == s2
bool operator<( const String & ) const; // test s1 < s2

// test s1 != s2
bool operator!=( const String &right ) const
{
   return !( *this == right );
} // end function operator!=

// test s1 > s2
bool operator>( const String &right ) const
{
   return right < *this;
} // end function operator>

// test s1 <= s2
bool operator<=( const String &right ) const
{
   return !( right < *this );
} // end function operator <=
```

```
   // test s1 >= s2
   bool operator>=( const String &right ) const
   {
      return !( *this < right );
   } // end function operator>=

   char &operator[]( int ); // subscript operator (modifiable lvalue)
   char operator[]( int ) const; // subscript operator (rvalue)
   String operator()( int, int = 0 ) const; // return a substring
   int getLength() const; // return string length
private:
   int length; // string length (not counting null terminator)
   char *sPtr; // pointer to start of pointer-based string

   void setString( const char * ); // utility function
}; // end class String

#endif
```

// **String.cpp**

```
#include "String.h" // String class definition

// conversion (and default) constructor converts char * to String
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 )
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString
```

```
// copy constructor
String::String( const String& copy )
   : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   setString( copy.sPtr ); // call utility function
} // end String copy constructor

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor
```

```
// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called" << endl;
   if ( &right != this ) // avoid self assignment
   {
      delete [] sPtr; // prevents memory leak
      length = right.length; // new String length
      setString( right.sPtr ); // call utility function
   } // end if
   else {
      cout << "Attempted assignment of a String to itself" << endl;
   }

   return *this; // enables cascaded assignments
} // end function operator=
```

```
// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory
   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=
```

- char * **strcpy** (char * **destination**, const char * **source**)

Copies the C string pointed by **source** into the array pointed by **destination**, <u>including the terminating null character</u>

```
// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

// Is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;
} // end function operator==

// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<
```

- int **strcmp** (const char * **str1**, const char * **str2**) – compares str1 with str2;

return value	indicates
<0	the first character that does not match has a lower value in ***ptr1*** than in ***ptr2***
0	the contents of both strings are equal
>0	the first character that does not match has a greater value in ***ptr1*** than in ***ptr2***

```
// return reference to character in String as a modifiable lvalue
char & String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
          << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]
```

Returns char& to support indexation and assignment

```
String str("Char");
str[0] = 'c'; // OK
```

```
// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
          << " out of range" << endl;
      exit( 1 ); // terminate program
} // end if

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]
```

Returns char to disable assignment

```
const String str("Char");
   str[0] = 'c'; // Error
```

```
// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr; // delete temporary array
   return tempString; // return copy of the temporary String
} // end function operator()
```

```
// return string length
int String::getLength() const
{
   return length;
} // end function getLength

// overloaded output operator
ostream& operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output; // enables cascading
} // end function operator<<

// overloaded input operator
istream& operator>>( istream &input, String &s )
{
   char temp[ 100 ]; // buffer to store input
   input >> setw( 100 ) >> temp;
   s = temp; // use String class assignment operator
   return input; // enables cascading
} // end function operator>>
```

```
// main.cpp

#include "String.h"
int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \""  << s2
      << "\"; s3 is \"" << s3 << '\"'
      << boolalpha << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 )
      << "\ns2 != s1 yields " << ( s2 != s1 )
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );
```

```
// test overloaded String empty (!) operator
  cout << "\n\nTesting !s3:" << endl;

  if ( !s3 )
  {
     cout << "s3 is empty; assigning s1 to s3;" << endl;
     s3 = s1; // test overloaded assignment
     cout << "s3 is \"" << s3 << "\"";
  } // end if

  // test overloaded String concatenation operator
  cout << "\n\ns1 += s2 yields s1 = ";
  s1 += s2; // test overloaded concatenation
  cout << s1;

  // test conversion constructor
  cout << "\n\ns1 += \" to you\" yields" << endl;
  s1 += " to you"; // test conversion constructor
  cout << "s1 = " << s1 << "\n\n";
```

```
// test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is:\n"
        << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15), is: "
        << s1( 15 ) << "\n\n";

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
```

```
// test copy constructor
  String *s4Ptr = new String( s1 );
  cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

  // test assignment (=) operator with self-assignment
  cout << "assigning *s4Ptr to *s4Ptr" << endl;
  *s4Ptr = *s4Ptr; // test overloaded assignment
  cout << "*s4Ptr = " << *s4Ptr << endl;

  // test destructor
  delete s4Ptr;

  // test using subscript operator to create a modifiable lvalue
  s1[ 0 ] = 'H';
  s1[ 6 ] = 'B';
  cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
     << s1 << "\n\n";

  // test subscript out of range
  cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
  s1[ 30 ] = 'd'; // ERROR: subscript out of range
  return 0;
} // end main
```

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday
```

```
Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

Object-Oriented Programming: Inheritance

Inheritance

When creating a class, instead of writing completely new data members and member functions, you can specify that the new class should **inherit** the members of an existing class.

This existing class is called the **base class**, and the new class is called the **derived class**.

Other programming languages, such as Java and C#, refer to the base class as the **superclass** and the derived class as the **subclass**.

A derived class represents a more specialized group of objects.

C++ offers **public**, **protected** and **private** inheritance.

Inheritance

With **public** inheritance, every object of a derived class is also an object of that derived class's base class. However, base-class objects are ***not*** objects of their derived classes.

For example, if we have **Vehicle as a base class** and **Car as a derived class**, then all Cars are Vehicles, but not all Vehicles are Cars—for example, a Vehicle could also be a Truck or a Boat.

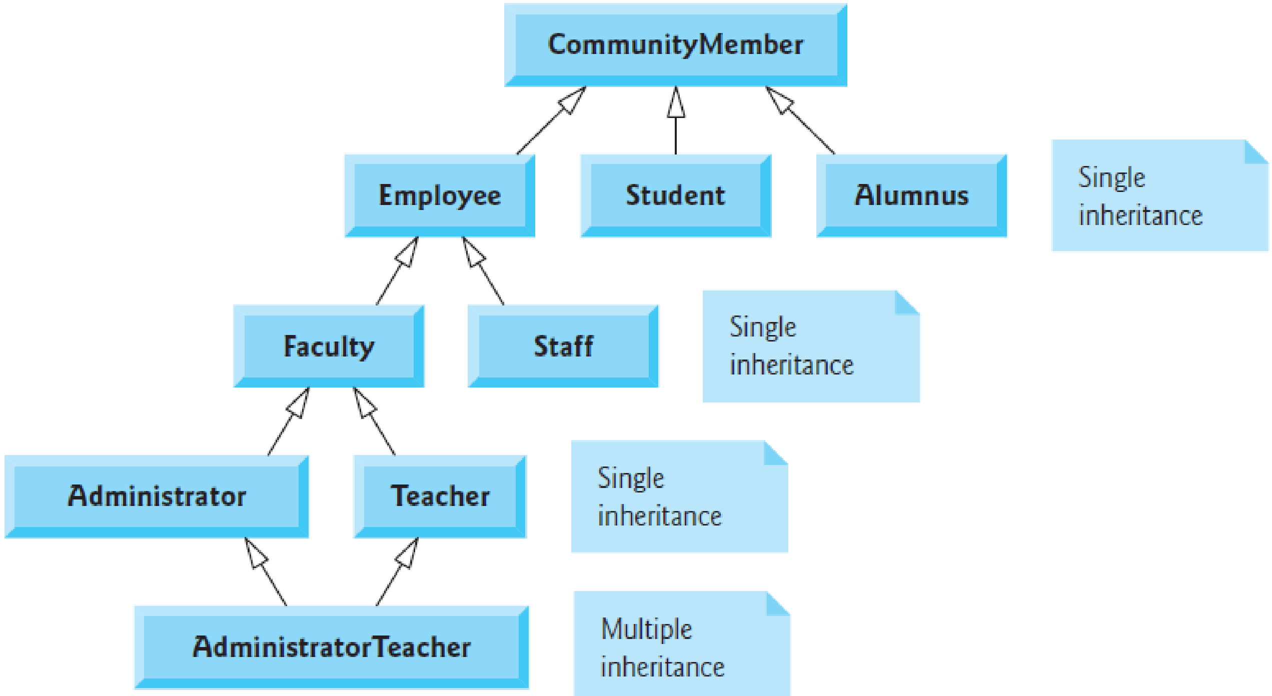
We distinguish between the ***is-a*** **relationship** and the ***has-a*** **relationship**.

The ***is-a*** relationship represents **inheritance**.

In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class—for example, a Car ***is a*** Vehicle, so any attributes and behaviors of a Vehicle are also attributes and behaviors of a Car.

By contrast, the ***has-a*** relationship represents ***composition***. In a *has-a* relationship, an object *contains* one or more objects of other classes as members. For example, a Car has many components—it *has a* steering wheel, *has a* brake pedal, *has a* transmission, etc.

Inheritance

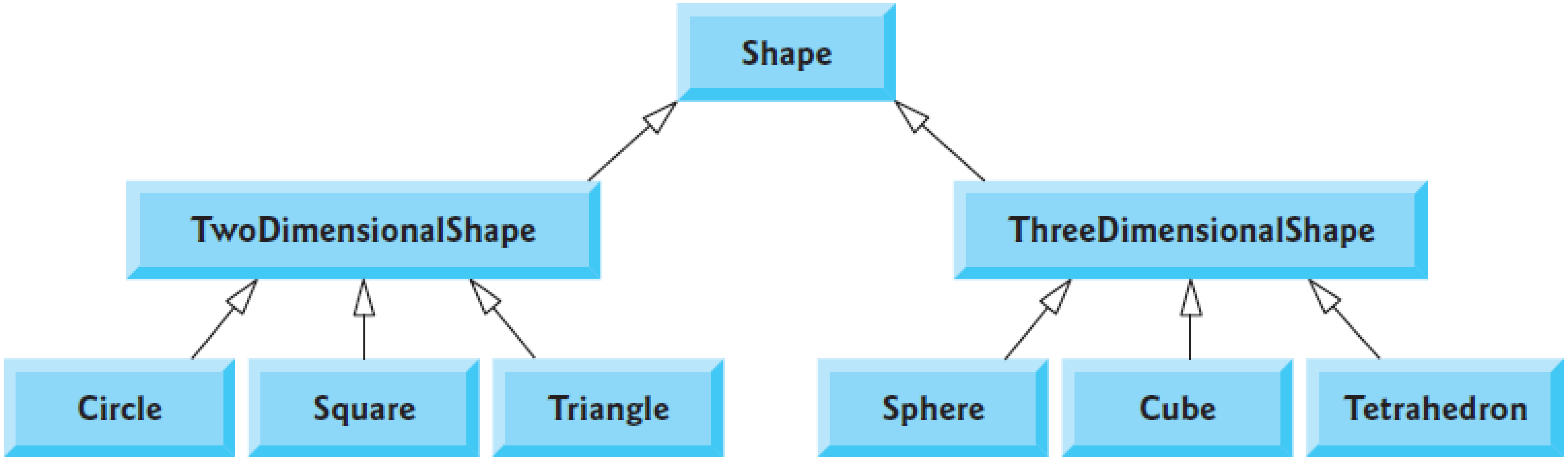


With **single inheritance**, a class is derived from ***one*** base class. With **multiple inheritance**, a derived class inherits simultaneously from ***two or more*** (possibly unrelated) base classes.

Inheritance relationships form **class hierarchies.**

- an Employee *is a* Community-Member
- CommunityMember is the **direct base class** of Employee, Student and Alumnus
- CommunityMember is an **indirect base class** of all the other classes in the diagram
- an AdministratorTeacher *is an* Administrator, *is a* Faculty member, *is an* Employee and *is a* CommunityMember

Inheritance




```
class TwoDimensionalShape : public Shape
```

With all forms of inheritance, private members of a base class are not accessible directly from that class's derived classes, but these private base-class members are still inherited.

With public inheritance, **public** members of the **base** class become **public** members of the **derived** class, **protected** members of the **base** class become **protected** members of the **derived** class.

Through inherited base-class member functions, the derived class can manipulate private members of the base class.

CommissionEmployee class

```cpp
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class

class CommissionEmployee {
public:
    CommissionEmployee(const std::string&, const std::string&,
       const std::string&, double = 0.0, double = 0.0);

    void setFirstName(const std::string&); // set first name
    std::string getFirstName() const; // return first name

    void setLastName(const std::string&); // set last name
    std::string getLastName() const; // return last name

    void setSocialSecurityNumber(const std::string&); // set SSN
    std::string getSocialSecurityNumber() const; // return SSN
```

CommissionEmployee class

```
    void setGrossSales(double); // set gross sales amount
    double getGrossSales() const; // return gross sales amount

    void setCommissionRate(double); // set commission rate (percentage)
    double getCommissionRate() const; // return commission rate

    double earnings() const; // calculate earnings
    std::string toString() const; // create string representation
private:
    std::string firstName;
    std::string lastName;
    std::string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
};

#endif
```

```cpp
#include <iomanip>
#include <stdexcept>
#include <sstream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(const string& first,
   const string& last, const string& ssn, double sales, double rate) {
   firstName = first; // should validate
   lastName = last; // should validate
   socialSecurityNumber = ssn; // should validate
   setGrossSales(sales); // validate and store gross sales
   setCommissionRate(rate); // validate and store commission rate
}

// set first name
void CommissionEmployee::setFirstName(const string& first) {
   firstName = first; // should validate
}

// return first name
string CommissionEmployee::getFirstName() const {return firstName;}

// set last name
void CommissionEmployee::setLastName(const string& last) {
   lastName = last; // should validate
}

// return last name
string CommissionEmployee::getLastName() const {return lastName;}
```

```
// set social security number
void CommissionEmployee::setSocialSecurityNumber(const string& ssn) {
   socialSecurityNumber = ssn; // should validate
}

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const {
   return socialSecurityNumber;
}

// set gross sales amount
void CommissionEmployee::setGrossSales(double sales) {
   if (sales < 0.0) {
      throw invalid_argument("Gross sales must be >= 0.0");
   }

   grossSales = sales;
}

// return gross sales amount
double CommissionEmployee::getGrossSales() const {return grossSales;}

// set commission rate
void CommissionEmployee::setCommissionRate(double rate) {
   if (rate <= 0.0 || rate >= 1.0) {
      throw invalid_argument("Commission rate must be > 0.0 and < 1.0");
   }

   commissionRate = rate;
}
```

```cpp
// return commission rate
double CommissionEmployee::getCommissionRate() const {
   return commissionRate;
}

// calculate earnings
double CommissionEmployee::earnings() const {
   return commissionRate * grossSales;
}

// return string representation of CommissionEmployee object
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision
   output << "commission employee: " << firstName << " " << lastName
      << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate;
   return output.str();
}
```

```
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

int main() {
   // instantiate a CommissionEmployee object
   CommissionEmployee employee{"Sue", "Jones", "222-22-2222", 10000, .06};

   // get commission employee data
   cout << fixed << setprecision(2); // set floating-point formatting
   cout << "Employee information obtained by get functions: \n"
      << "\nFirst name is " << employee.getFirstName()
      << "\nLast name is " << employee.getLastName()
      << "\nSocial security number is "
      << employee.getSocialSecurityNumber()
      << "\nGross sales is " << employee.getGrossSales()
      << "\nCommission rate is " << employee.getCommissionRate() << endl;

   employee.setGrossSales(8000); // set gross sales
   employee.setCommissionRate(.1); // set commission rate
   cout << "\nUpdated employee information from function toString: \n\n"
      << employee.toString();

   // display the employee's earnings
   cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
}
```

```
Employee information obtained by get functions:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information from function toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10

Employee's earnings: $800.00
```

BasePlusCommissionEmployee class

We want to create the class BasePlusCommissionEmployee which in addition to the commissionRate parameter, has also baseSalary.

We have two options:

1. Just copy-and-paste the code of CommisionEmployee class and modify it according to BasePlusCommissionEmployee class.
2. **Inherit BasePlusCommissionEmployee from CommisionEmployee** and add BasePlusCommissionEmployee specific functionality.

Let’s discuss these two options.

```
class BasePlusCommissionEmployee {
public:
   BasePlusCommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0, double = 0.0);

   void setFirstName(const std::string&); // set first name
   std::string getFirstName() const; // return first name

   void setLastName(const std::string&); // set last name
   std::string getLastName() const; // return last name

   void setSocialSecurityNumber(const std::string&); // set SSN
   std::string getSocialSecurityNumber() const; // return SSN

   void setGrossSales(double); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate(double); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setBaseSalary(double); // set base salary
   double getBaseSalary() const; // return base salary

   double earnings() const; // calculate earnings
   std::string toString() const; // create string representation
private:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
   double baseSalary; // base salary
};
```

```cpp
// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string& first, const string& last, const string& ssn,
   double sales, double rate, double salary) {
   firstName = first; // should validate
   lastName = last; // should validate
   socialSecurityNumber = ssn; // should validate
   setGrossSales(sales); // validate and store gross sales
   setCommissionRate(rate); // validate and store commission rate
   setBaseSalary(salary); // validate and store base salary
}

// set base salary
void BasePlusCommissionEmployee::setBaseSalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Salary must be >= 0.0");
   }

   baseSalary = salary;
}

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const {
   return baseSalary;
}

// calculate earnings
double BasePlusCommissionEmployee::earnings() const {
   return baseSalary + (commissionRate * grossSales);
}
```

```
// return string representation of BasePlusCommissionEmployee object
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision
   output << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate
      << "\nbase salary: " << baseSalary;
   return output.str();
}
```

BasePlusCommissionEmployee class

This *copy-and-paste approach* is error prone and time consuming.

- Copying and pasting code from one class to another can spread many physical copies of the same code and can spread errors throughout a system, creating a code-maintenance nightmare. To avoid duplicating code (and possibly errors), **use inheritance**, rather than the “copy-and-paste” approach, in situations where you want one class to “**absorb**” the data members and member functions of another class.

- With inheritance, the **common** data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only in the base class—derived classes then inherit the changes. *Without inheritance*, changes would need to be made to all the source-code files that contain a copy of the code in question.

```
8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9
10 class BasePlusCommissionEmployee : public CommissionEmployee {
11 public:
12    BasePlusCommissionEmployee(const std::string&, const std::string&,
13       const std::string&, double = 0.0, double = 0.0, double = 0.0);
14
15    void setBaseSalary(double); // set base salary
16    double getBaseSalary() const; // return base salary
17
18    double earnings() const; // calculate earnings
19    std::string toString() const; // create string representation
20 private:
21    double baseSalary; // base salary
22 };
```

BasePlusCommissionEmployee inherits all the members of class CommissionEmployee, except for the **constructor**—each class provides its own constructors that are specific to the class. (**Destructors**, too, are not inherited.)

Notice that we **#include the base class's header in the derived class's header** (line 8). There are 3 reasons to include:

1. For the derived class to use the base class's name in line 10, we must tell the compiler that the base class exists—the class definition in CommissionEmployee.h does exactly that.
2. The compiler uses a class definition to determine the *size* of an object of that class.
3. To allow the compiler to determine whether the derived class uses the base class's inherited members properly. For example, to determine that the data members being accessed by the derived class are private in the base class.

```
9   // constructor
10  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string& first, const string& last, const string& ssn,
12     double sales, double rate, double salary)
13     // explicitly call base-class constructor
14     : CommissionEmployee(first, last, ssn, sales, rate) {
15     setBaseSalary(salary); // validate and store base salary
16  }
17
18  // set base salary
19  void BasePlusCommissionEmployee::setBaseSalary(double salary) {
20     if (salary < 0.0) {
21        throw invalid_argument("Salary must be >= 0.0");
22     }
```

The constructor (lines 10–16) introduces **base-class initializer syntax** (line 14), which uses a member initializer to pass arguments to the base-class constructor.

- C++ requires that a **derived-class constructor call its base-class constructor** to initialize the base-class data members that are inherited into the derived class.
- The compiler would issue an error if BasePlusCommissionEmployee's constructor did not invoke class CommissionEmployee's constructor **explicitly**— in this case, C++ attempts to invoke class CommissionEmployee's default constructor implicitly, but the class does not have such a constructor.
- Compiler provides a default constructor with no parameters in any class that **does not explicitly include a constructor**. However, CommissionEmployee does explicitly include a constructor, so a default constructor is not provided.

```cpp
// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string& first, const string& last, const string& ssn,
   double sales, double rate, double salary)
   // explicitly call base-class constructor
   : CommissionEmployee(first, last, ssn, sales, rate) {
   setBaseSalary(salary); // validate and store base salary
}

// set base salary
void BasePlusCommissionEmployee::setBaseSalary(double salary) {
   if (salary < 0.0) {
      throw invalid_argument("Salary must be >= 0.0");
   }
```

- When a derived-class constructor calls a base-class constructor, <u>the arguments passed to the base-class constructor must be consistent</u> with the number and types of parameters specified in one of the <u>base-class constructors</u>; otherwise, a compilation error occurs.

- In a derived-class constructor, invoking base-class constructors and initializing member objects explicitly <u>in the member initializer list prevents **duplicate initialization**</u> in which a default constructor is called, then data members are modified again in the derived-class constructor’s body.

```
   baseSalary = salary;
}

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const {
   return baseSalary;
}

// calculate earnings
double BasePlusCommissionEmployee::earnings() const {
   // derived class cannot access the base class's private data
   return baseSalary + (commissionRate * grossSales);
}

// returns string representation of BasePlusCommissionEmployee object
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision

   // derived class cannot access the base class's private data
   output << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate
      << "\nbase salary: " << baseSalary;
   return output.str();
}
```

Compilation Errors from the Clang/LLVM Compiler in Xcode 7.2

```
BasePlusCommissionEmployee.cpp:34:25:
   'commissionRate' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:34:42:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:42:55:
   'firstName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:10:
   'lastName' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:43:54:
   'socialSecurityNumber' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:44:31:
   'grossSales' is a private member of 'CommissionEmployee'
BasePlusCommissionEmployee.cpp:45:35:
   'commissionRate' is a private member of 'CommissionEmployee'
```

The compiler generates errors because base-class CommissionEmployee's data members are **private**.
C++ rigidly enforces restrictions on accessing private data members, so that *even a derived class cannot access the base class's private data.*

The error can be prevented by the following two ways

- By using the **get member functions** inherited from class CommissionEmployee.
- By using **protected** data.

Protected access specifier

A **base class's public members** are accessible within its body and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that class or one of its derived classes, including in derived classes.

A **base class's private members** are accessible only within its body and to the friends of that base class.

A **base class's protected members** can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes ***derived*** from that base class.

Objects of a derived class also can access **protected** members in *any* of that derived class's ***indirect*** base classes.

To enable class BasePlusCommissionEmployee to *directly access* CommissionEmployee data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as **protected** in the base class.

```cpp
class CommissionEmployee {
public:
   CommissionEmployee(const std::string&, const std::string&,
      const std::string&, double = 0.0, double = 0.0);

   void setFirstName(const std::string&); // set first name
   std::string getFirstName() const; // return first name

   void setLastName(const std::string&); // set last name
   std::string getLastName() const; // return last name

   void setSocialSecurityNumber(const std::string&); // set SSN
   std::string getSocialSecurityNumber() const; // return SSN

   void setGrossSales(double); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate(double); // set commission rate
   double getCommissionRate() const; // return commission rate

   double earnings() const; // calculate earnings
   std::string toString() const; // return string representation
protected:
   std::string firstName;
   std::string lastName;
   std::string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
};
```

Protected access specifier

Inheriting protected data members **slightly improves performance**, because we can directly access the members without incurring the overhead of calls to *set* or *get* member functions.

Using protected data members creates two serious problems:

1. The derived-class object **does not have to use a member function** to set the value of the base class's protected data member. An invalid value can easily be assigned to the protected data member.
2. Derived-class member functions are more likely to be written so that they **depend on the base-class implementation**. Derived classes should depend **only on the base-class services** (i.e., nonprivate member functions) and not on the base-class implementation.

- ✓ You should be able to change the base-class implementation while still providing the same services to derived classes.
- ✓ It's appropriate to use the protected access specifier when a **base class should provide a service** (i.e., a non-private member function) **only to its derived classes and friends**.
- ✓ Declaring base-class data members private (as opposed to declaring them protected) enables you to change the base-class implementation without having to change derived-class implementations.

Using private data and set/get methods

We now re-examine our hierarchy once more, this time using *best software engineering practices*.
We **replace the protected data by private data** and access them **through set/get methods**.

```
// constructor
CommissionEmployee::CommissionEmployee(const string &first,
   const string &last, const string &ssn, double sales, double rate)
   : firstName(first), lastName(last), socialSecurityNumber(ssn) {
   setGrossSales(sales); // validate and store gross sales
   setCommissionRate(rate); // validate and store commission rate
}
```

```cpp
// calculate earnings
double CommissionEmployee::earnings() const {
   return getCommissionRate() * getGrossSales();
}

// return string representation of CommissionEmployee object
string CommissionEmployee::toString() const {
   ostringstream output;
   output << fixed << setprecision(2); // two digits of precision
   output << "commission employee: "
      << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber()
      << "\ngross sales: " << getGrossSales()
      << "\ncommission rate: " << getCommissionRate();
   return output.str();
}
```

If we decide to change the data member **names**, the earnings and toString definitions will *not* require modification - only the definitions of the *get* and *set* member functions that directly manipulate the data members will need to change.

These changes occur solely within the base class—no changes to the derived class are needed.

❖ Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, today's optimizing compilers perform many optimizations **implicitly** (such as **inlining** set and get member-function calls). **You should write code that adheres to proper software engineering principles, and leave optimization to the compiler.**

```cpp
// calculate earnings
double BasePlusCommissionEmployee::earnings() const {
   return getBaseSalary() + CommissionEmployee::earnings();
}

// return string representation of BasePlusCommissionEmployee object
string BasePlusCommissionEmployee::toString() const {
   ostringstream output;
   output << "base-salaried " << CommissionEmployee::toString()
      << "\nbase salary: " << getBaseSalary();
   return output.str();
}
```

Note the syntax used to invoke a redefined base-class member function from a derived class — place the **base-class name** and the **scope resolution operator (::)** before the base-class member-function name.

This member-function invocation is a good software engineering practice: **we avoid duplicating the code** and **reduce code-maintenance problems**.

❖ When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the **::** operator prefixed with the name of the base class when referencing the base class's member function causes **infinite recursion**, because the derived-class member function would then call itself.

Inheritance

So, what is the good software engineering design with inheritance?

- Declaring base-class data members private (as opposed to declaring them protected) enables you to change the base-class implementation without having to change derived-class implementations.

- Declare the base-class data members as **private** and access them through set/get methods **everywhere**.

- It's appropriate to use the protected access specifier when a base class should provide a service (i.e., a non-private member function) only to its derived classes and friends.

Types of Inheritance

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.