Object Oriented Programming by C++

Inheritance

Create a class from an existing class

2017. 8.

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr
Web: http://mobilelab.khu.ac.kr/

Textbook & Copyright

- Textbook: http://python.cs.southern.edu/cppbook/progcpp.pdf
- Sample Codes: https://github.com/halterman/CppBook-SourceCode

Fundamentals of C++ Programming

DRAFT

Richard L. Halterman
School of Computing
Southern Adventist University

July 21, 2017

Preface

Legal Notices and Information

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

- The copyright and this legal notice must appear in any copies of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hard copies:

- The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

Additionally, a hardcopy or a local electronic copy must contain the uniform resource locator (URL) providing a link to the original content so the reader can check for updated and corrected content. The current standard URL is `http://python.cs.southern.edu/cppbook/progcpp.pdf`.

If you are an instructor using this book in one or more of your courses, please let me know. Keeping track of how and where this book is used helps me justify to my employer that it is providing a useful service to the community and worthy of the time I spend working on it. Simply send a message to `halterman@southern.edu` with your name, your institution, and the course(s) in which you use it.

The source code for all labeled listings is available at

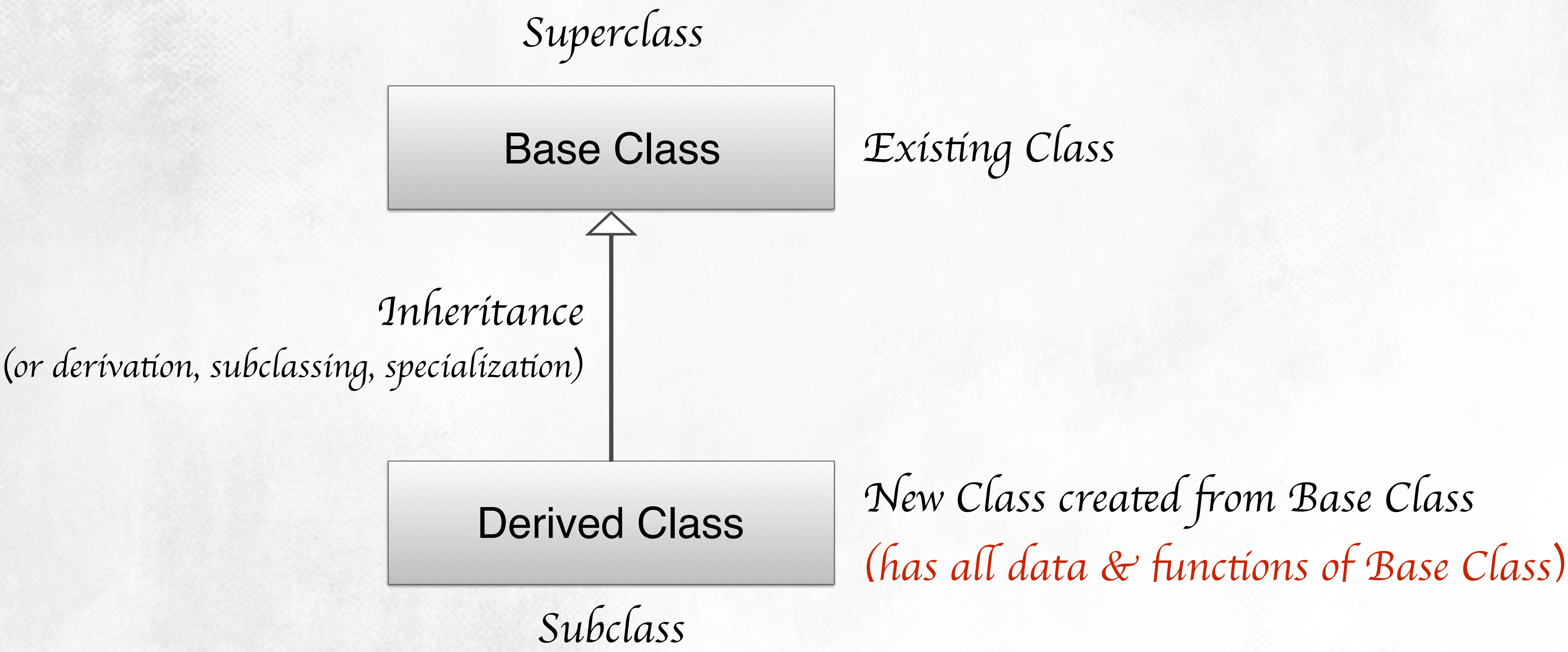
`https://github.com/halterman/CppBook-SourceCode`.

Contents

- ..

Base, Derived and Inheritance

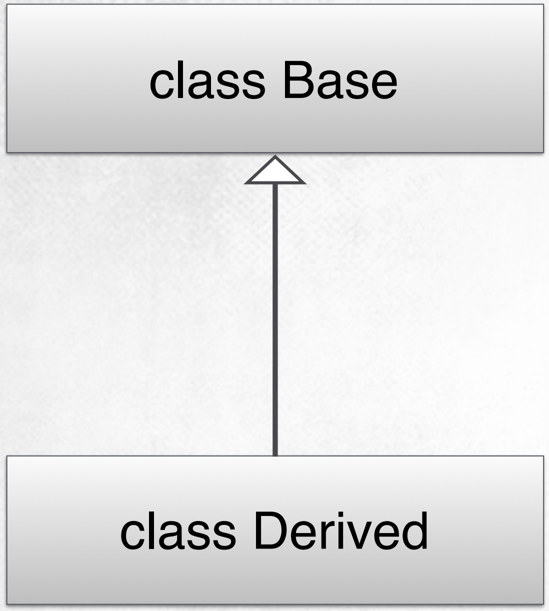
- Class Hierarchy in Inheritance



Base, Derived and Inheritance

- Consequence
  - The *Derived Class* can be treated as if it were an instance of the *Base Class*
  - “is-a” Relationship:
    - ***the Derived Class is-a Base Class***
      - a Derived Class has all data and functions of a Base Class
      - “Every `Employee` is-a `Person`”
    - ***the Base Class is not a Derived Class***
      - a Base Class has no whole data and functions of a Derived Class
      - “Not every `Person` is an `Employee`”

Inheritance Statement



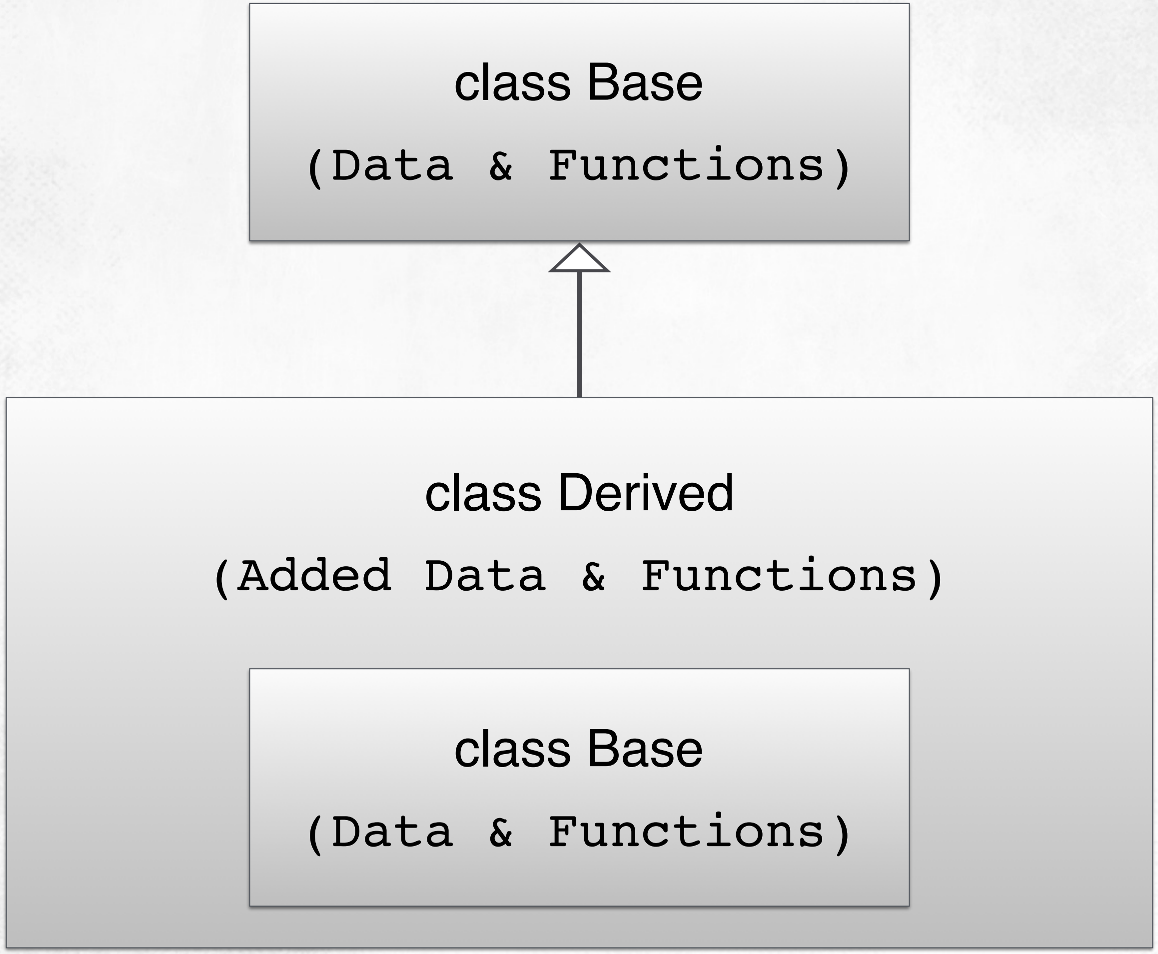

```
class Base {

};

class Derived : public Base {

};
```

Derived Class has all of Base Class



Code Example 1

Derived Class has all of Base Class

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

class Base {
public:
    void f(void);
};

void Base::f(void)
{
    cout << "in function 'Base::f()'\n";
}

class Derived : public Base {
public:
    void g(void);
};

void Derived::g(void)
{
    cout << "in function 'Derived::g()'\n";
}

int main()
{
    Base myB;
    Derived myD;
    myB.f();
    myD.g();
    myD.f();
}
```

```
in function 'Base::f()'
in function 'Derived::g()'
in function 'Base::f()'
```

class Derived has function f() of class Base

Code Example 2

Inheritance Type is Important

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

class Base {
public:
    void f(void);
};

void Base::f(void)
{
    cout << "in function 'Base::f()'\n";
}

class Derived : private Base {
public:
    void g(void);
};

void Derived::g(void)
{
    cout << "in function 'Derived::g()'\n";
}

int main()
{
    Base myB;
    Derived myD;
    myB.f();
    myD.g();
    myD.f();
}
```

```
compile error…
```

private function f() can not be invoked publicly

Inheritance Type

Access Protection of Base Class (1/2)

```cpp
class B {
    // Other details omitted
public:
    void f();
};

void B::f() {
    std::cout << "In function 'f'\n";
}

class D: public B {
    // Other details omitted
public:
    void g();
};
```

If we omit the word `public` from class D's definition, as in

```cpp
class D: B {
    // Details omitted
};
```

all the public members of B inherited by D objects will be *private* by default; for example, if base class B looks like the following:

```cpp
class B {
public:
    void f();
};

void B::f() {
    std::cout << "In function 'f'\n";
}
```

the following code is not legal:

```cpp
D myD;
myD.f();  // Illegal, method f now is private!
```

This means a client may not treat a D object exacty as if it were a B object. This violates the Liskov Substitution Principle, and the *is a* relationship does not exist.

While this *private inheritance* is useful in rare situations, the majority of object-oriented software design uses public inheritance. C++ is one of the few object-oriented languages that supports private inheritance.

Access Protection of Base Class (2/2)

```
class Base {
};
class Derived : {TYPE} Base {
};
```

Base class member access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
Public	**public** in derived class. Can be accessed directly by any non-**static** member functions, **friend** functions and non-member functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Protected	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

Code Example 3

Access Base's Private Function

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

class Base {
public:
    void f(void);
};

void Base::f(void)
{
    cout << "in function 'Base::f()'\n";
}

class Derived : private Base {
public:
    void g(void);
};

void Derived::g(void)
{
    Base::f();
    cout << "in function 'Derived::g()'\n";
}

int main()
{
    Base myB;
    Derived myD;
    myB.f();
    myD.g();
}
```

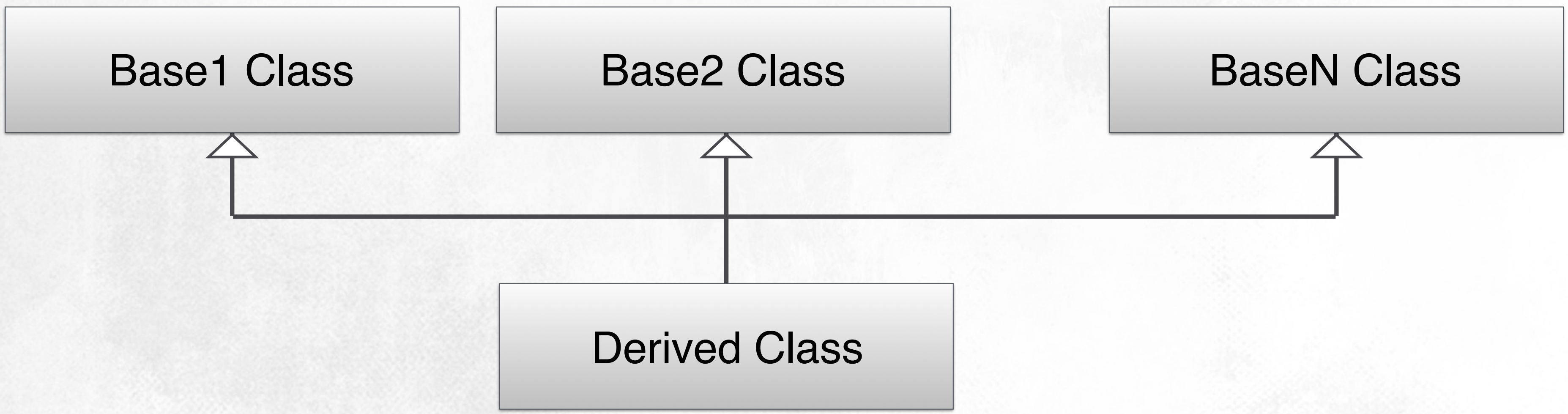
```
in function 'Base::f()'
in function 'Base::f()'
in function 'Derived::g()'
```

private function f() can be invoked privately

Inheritance from Multiple Base Classes

- It's possible in C++
- But, in object-oriented design, multiple inheritance is not as common as single inheritance (one base class)

```
class Derived : public Base1, public Base2, public Base3 {
};
```



Goal of Inheritance

Reusability & Specialization

- Inheritance is a design tool that allows developers to take an existing class and produce a new class that ***provides enhanced behavior or different behavior***.
- The enhanced or new behavior does not come at the expense of existing code; that is, when using inheritance ***programmers do not touch any source code in the base class***.
- Also, developers can ***leverage existing code (in the base class) without duplicating it*** in the derived classes.

Function **Overriding** in the Derived Class

- Specifies that a virtual function (in the Derived class) ***overrides [or replace, re-define]*** another virtual function (in the Base class).

> The `override` keyword was added to the language in C++11. Prior to C++11 when a method in a derived class had the same signature as a virtual method in its base class, the method implicitly overrode its base class version. The problem was that a programmer could intend to override a method in the derived class but get the signature wrong. The resulting method *overloaded* the original method rather than overriding it. If a programmer uses the `override` specifier and uses a signature that does not match the base class version, the compiler will report an error. The `override` specifier provides a way for programmers to explicitly communicate their intentions.
>
> For backwards compatibility the `override` keyword is optional. Its presence enables the compiler to verify that the method is actually overriding a `virtual` method in the base class. Without it, the programmer must take care to faithfully reproduce the signature of the method to override.
>
> The `override` keyword is a *context-sensitive keyword*, meaning it is a keyword only when appearing as it does here in the declaration of a method header. In other contexts it behaves like an identifier.

Function **Overloading** in the Derived Class

- Add more functions with the same name:
  - C++ allows you to specify ***more than one definition for a function name or an operator*** in the same scope, which is called function overloading and operator overloading respectively.
  - An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, ***except that both declarations have different arguments and obviously different definition*** (implementation).
  - When you call an overloaded function or operator, t***he compiler determines the most appropriate definition to use, by comparing the argument types*** you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution

Function **Overriding** Statement

- `virtual`
  - The virtual specifier indicates that the designer of the Base class intends ***for derived classes to be able to customize the behavior of the virtual methods***

- `override`
  - This means ***the exact behavior of these methods (in the Derived class) will be different in some way from their implementation in the Base class***
  - If the Derived class does not overrides the virtual method; it inherits the base method without alteration

Code Example 4

Inherited Texts - Listing 17.2 (1/4)

```
#include <string>
#include <iostream>

// Base class for all Text derived classes
class Text {
    std::string text;
public:
    // Create a Text object from a client-supplied string
    Text(const std::string& t): text(t) {}

    // Allow clients to see the text field
    virtual std::string get() const {
        return text;
    }

    // Concatenate another string onto the
    // back of the existing text
    virtual void append(const std::string& extra) {
        text += extra;
    }
};
```

```
plain
<<fancy>>
FIXED
-------------------------
plainA
<<fancy***A>>
FIXED
-------------------------
plainAB
<<fancy***A***B>>
FIXED
```

Code Example 4

Inherited Texts - Listing 17.2 (2/4)

```
// Provides minimal decoration for the text
class FancyText: public Text {
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
public:
    // Client supplies the string to wrap plus some extra
    // decorations
    FancyText(const std::string& t, const std::string& left,
              const std::string& right, const std::string& conn):
    Text(t), left_bracket(left),
    right_bracket(right), connector(conn) {}

    // Allow clients to see the decorated text field
    std::string get() const override {
        return left_bracket + Text::get() + right_bracket;
    }

    // Concatenate another string onto the
    // back of the existing text, inserting the connector
    // string
    void append(const std::string& extra) override {
        Text::append(connector + extra);
    }
};
```

```
plain
<<fancy>>
FIXED
-------------------------
plainA
<<fancy***A>>
FIXED
-------------------------
plainAB
<<fancy***A***B>>
FIXED
```

Code Example 4

Inherited Texts - Listing 17.2 (2/4+)

```
// Provides minimal decoration for the text
class FancyText: public Text {
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
public:
    // Client supplies the string to wrap plus some extra
    // decorations
    FancyText(const std::string& t, const std::string& left,
              const std::string& right, const std::string& conn):
    Text(t), left_bracket(left),
```

We want to assign the constructor's first parameter, t, to the inherited member *text*, but *text* is private in the base class. This means the FancyText constructor cannot initialize it (member data *Text::text* in base class) directly. Since the constructor of its base class knows what to do with this parameter, the first expression in the constructor initialization list (the part between the : and the {):

```
 ... Text(t) ...
```

explicitly calls the base class constructor, passing it t.

```
};
```

Code Example 4

Inherited Texts - Listing 17.2 (3/4)

```
// The text is always the word FIXED
class FixedText: public Text {
public:
    // Client does not provide a string argument; the
    // wrapped text is always "FIXED"
    FixedText(): Text("FIXED") {}

    // Nothing may be appended to a FixedText object
    void append(const std::string&) override {
        // Disallow concatenation
    }
};
```

```
plain
<<fancy>>
FIXED
-------------------------
plainA
<<fancy***A>>
FIXED
-------------------------
plainAB
<<fancy***A***B>>
FIXED
```

Code Example 4

Inherited Texts - Listing 17.2 (4/4)

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "***");
    FixedText t3;
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
    std::cout << "-------------------------\n";
    t1.append("A");
    t2.append("A");
    t3.append("A");
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
    std::cout << "-------------------------\n";
    t1.append("B");
    t2.append("B");
    t3.append("B");
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
}
```

```
plain
<<fancy>>
FIXED
-------------------------
plainA
<<fancy***A>>
FIXED
-------------------------
plainAB
<<fancy***A***B>>
FIXED
```

Base, Derived and Inheritance

- Consequence
  - The *Derived Class* can be treated as if it were an instance of the *Base Class*
  - “is-a” Relationship:
    - *the Derived Class is-a Base Class*
      - ***a Derived Class has all data and functions of a Base Class***
        - ***thus, a Derived class object can fill the Base class object***
      - “Every `Employee` is-a `Person`”
    - *the Base Class is not a Derived Class*
      - ***a Base Class has no whole data and functions of a Derived Class***
        - ***thus, a Base class object can not fill the Derived class object***
      - “Not every `Person` is an `Employee`”

Code Example 5

only `main()` modified from Listing 17.2

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "::");
    std::cout << t1.get() << "  " << t2.get() << ‘\n’;
    t1 = t2; // copy Derived class object to Base class object
    std::cout << t1.get() << "  " << t2.get() << '\n';
}
```

```
plain   <<fancy>>
fancy   <<fancy>>
```

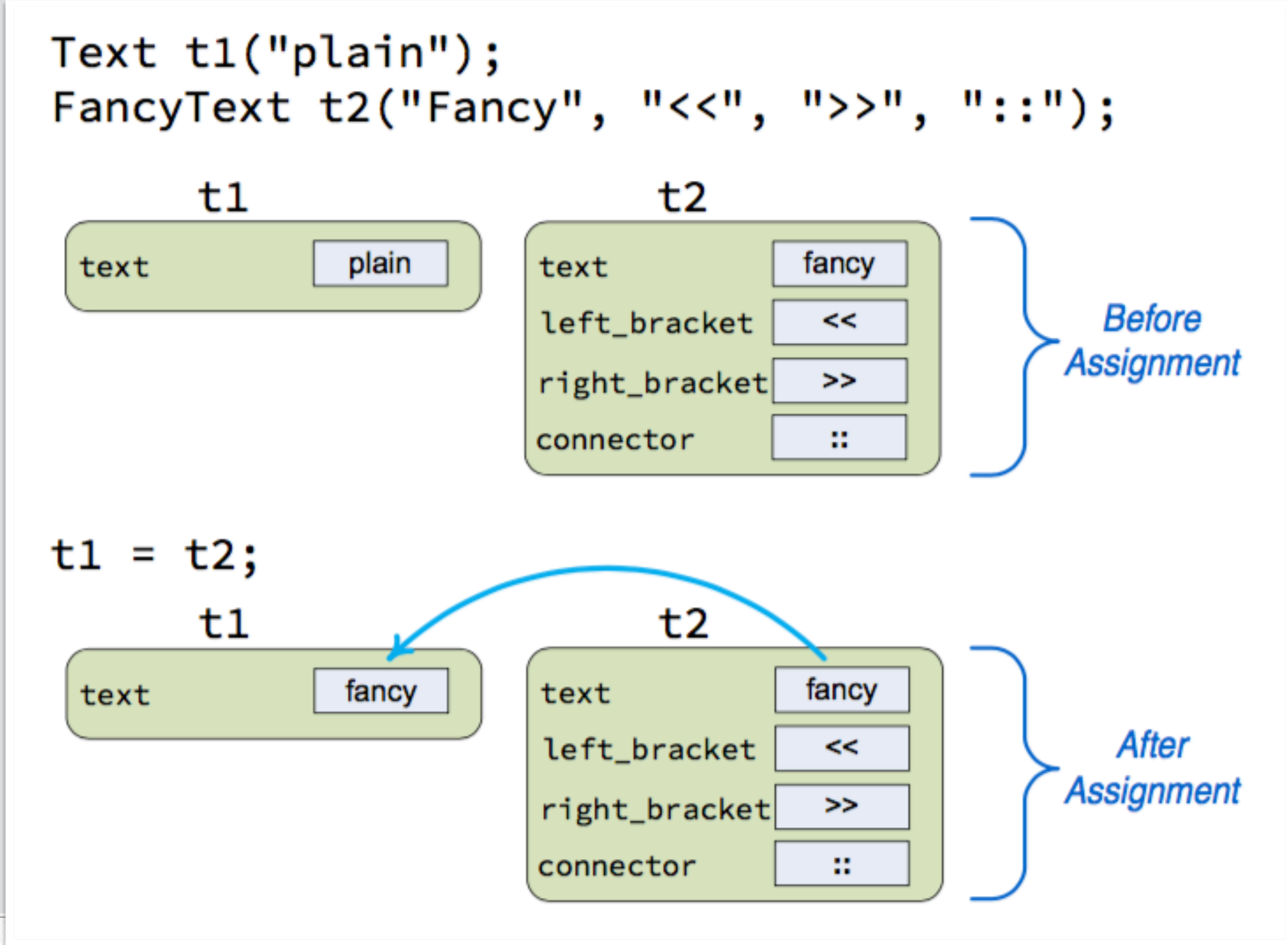
Code Example 5

only main( ) modified from Listing 17.2

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "::");
    std::cout << t1.get() << "  " << t2.get() << '\n';
    t1 = t2; // copy Derived class object to Base class object
    std::cout << t1.get() << "  " << t2.get() << '\n';
}
```

Object Slicing

losing derived's information during copy.



Code Example 6

only `main()` modified from Listing 17.2

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "::");
    std::cout << t1.get() << "  " << t2.get() << '\n';
    t2 = t1;
    std::cout << t1.get() << "  " << t2.get() << '\n';
}
```

```
illegal - compile error
```

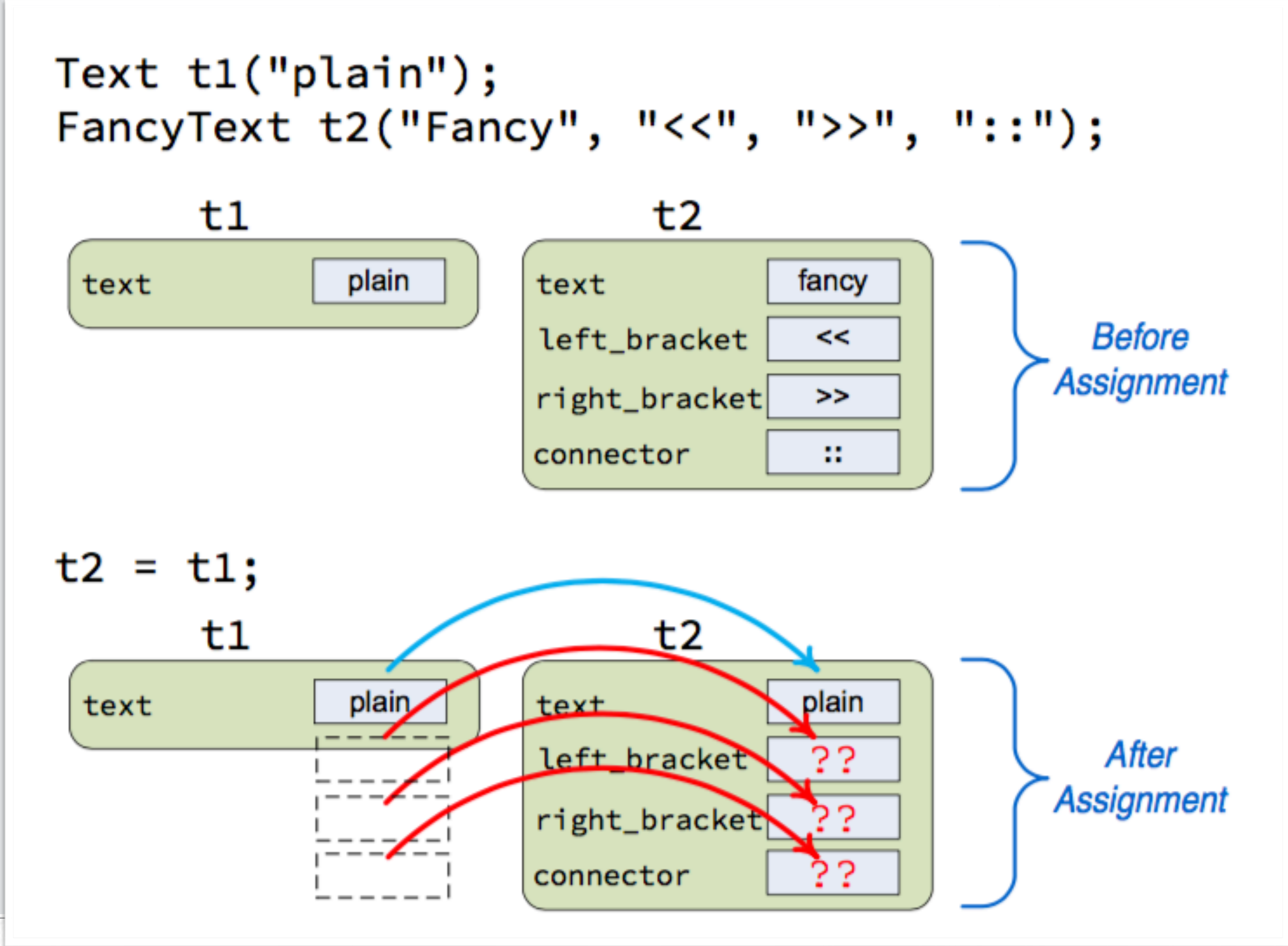
Code Example 6

only `main()` modified from Listing 17.2

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "::");
    std::cout << t1.get() << "  " << t2.get() << '\n';
    t2 = t1;
    std::cout << t1.get() << "  " << t2.get() << '\n';
}
```

illegal

base class instance to a derived class variable



Code Example 7

only `main()` modified from Listing 17.2

```
int main() {
    Text t1("plain");
    FixedText t3;
    std::cout << t1.get() << "   " << t3.get() << '\n';
    t1 = t3;
    std::cout << t1.get() << "   " << t3.get() << '\n';
}
```

```
plain   FIXED
FIXED   FIXED
```

Remind

is-a Relationship

It always is legal to assign a derived class instance to a variable of a base type. This is because a derived class instance *is a* specific kind of base class instance. In contrast, it never is legal to assign a base class instance to a variable of a derived type. This is because the *is a* relationship is only one directional, from a derived class to its base class.

It always is legal to assign a derived class instance to a variable of a base type. This is because a derived class instance *is a* specific kind of base class instance. In contrast, it never is legal to assign a base class instance to a variable of a derived type. This is because the *is a* relationship is only one directional, from a derived class to its base class.

It always is legal to assign a derived class instance to a variable of a base type. This is because a derived class instance *is a* specific kind of base class instance. In contrast, it never is legal to assign a base class instance to a variable of a derived type. This is because the *is a* relationship is only one directional, from a derived class to its base class.

Code Example 8

Static Binding

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

class Base {
public:
    void f(void) {
        cout << "in function 'Base::f()'\n";
    }

    virtual void vf(void) {
        cout << "in function 'Base::vf()'\n";
    }
};

class Derived : public Base {
public:
    void f(void) {
        cout << "in function 'Derived::f()'\n";
    }

    void vf(void) override {
        cout << "in function 'Derived::vf()'\n";
    }
};

int main()
{
    Base myB;
    Derived myD;

    myB.f();
    myB.vf();
    myD.f();
    myD.vf();
}
```

```
in function 'Base::f()'
in function 'Base::vf()'
in function 'Derived::f()'
in function 'Derived::vf()'
```

Code Example 9

Dynamic Binding (1/3)

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

class Base {
public:
    void f(void) {
        cout << "in function 'Base::f()'\n";
    }

    virtual void vf(void) {
        cout << "in function 'Base::vf()'\n";
    }
};

class Derived : public Base {
public:
    void f(void) {
        cout << "in function 'Derived::f()'\n";
    }

    void vf(void) override {
        cout << "in function 'Derived::vf()'\n";
    }
};

int main()
{
    Base *p;
    ____ myB;
    ____ myD;

    p = &myB;
    p->f();
    p->vf();

    p = &myD;
    p->f();
    p->vf();
}
```

```
// Let's GUESS!!!!
```

Code Example 9

Dynamic Binding (2/3)

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

class Base {
public:
    void f(void) {
        cout << "in function 'Base::f()'\n";
    }

    virtual void vf(void) {
        cout << "in function 'Base::vf()'\n";
    }
};

class Derived : public Base {
public:
    void f(void) {
        cout << "in function 'Derived::f()'\n";
    }

    void vf(void) override {
        cout << "in function 'Derived::vf()'\n";
    }
};

int main()
{
    Base *p;
    Base myB;
    Derived myD;

    p = &myB;
    p->f();
    p->vf();

    p = &myD;
    p->f();
    p->vf();
}
```

```
in function 'Base::f()'
in function 'Base::vf()'
in function 'Base::f()'
in function 'Derived::vf()'
```

Static binding is relatively easy to understand: the method to execute depends on the declared type of the variable upon which the method is invoked.

*"I don't care what's in MEMORY now, and *p is just a Base class"*

Code Example 9

Dynamic Binding (3/3)

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

class Base {
public:
    void f(void) {
        cout << "in function 'Base::f()'\n";
    }

    virtual void vf(void) {
        cout << "in function 'Base::vf()'\n";
    }
};

class Derived : public Base {
public:
    void f(void) {
        cout << "in function 'Derived::f()'\n";
    }

    void vf(void) override {
        cout << "in function 'Derived::vf()'\n";
    }
};

int main()
{
    Base *p;
    Base myB;
    Derived myD;

    p = &myB;
    p->f();
    p->vf();

    p = &myD;
    p->f();
    p->vf();
}
```

```
in function 'Base::f()'
in function 'Base::vf()'
in function 'Base::f()'
in function 'Derived::vf()'
```

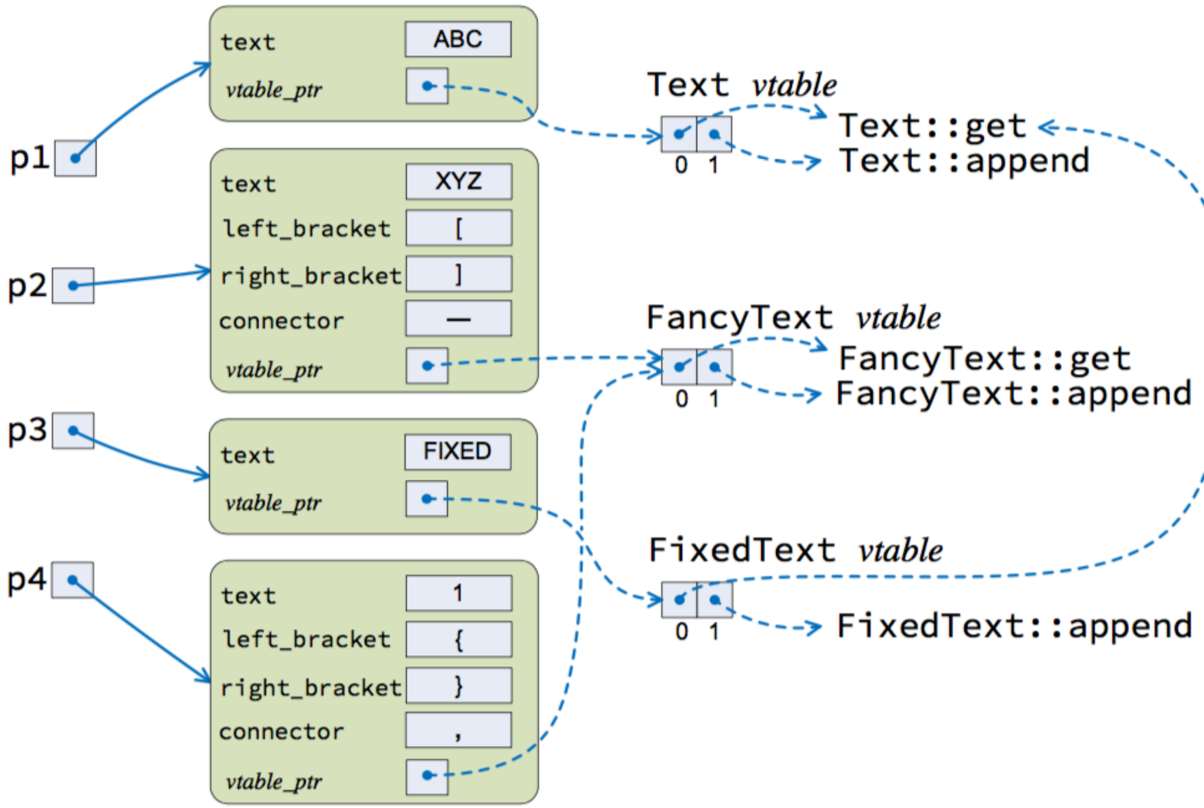
In the case of a ***virtual method*** invoked via a pointer, the running program, not the compiler, determines exactly which code to execute. The process is known as ***dynamic binding*** or ***late binding***.

"What's in MEMORY now?"

vtable and virtual function

- To track the right virtual function to invoke, dynamics binding manages (pointer to function) table for virtual functions - vtable

```
Text *p1, *p2, *p3, *p4;
p1 = new Text("ABC");
p2 = new FancyText("XYZ", "[", "]", "-");
p3 = new FixedText;
p4 = new FancyText("1", "{", "}", ",");
```



FINALLY, WE ARRIVED !!!

In programming languages and type theory, polymorphism (from Greek πολύς, polys, "many, much" and μορφή, morphē, "form, shape") is the provision of a single interface to entities of different types.
[1] A polymorphic type is one whose operations can also be applied to values of some other type, or types.

Example Code

A *vector* is a collection of *homogeneous* elements.

But, with inheritance (actually dynamic binding), not only can `texts` hold pointers to simple `Text` objects, it also simultaneously can hold pointers to `FixedText` and `FancyText` objects.

Listing 17.4: polymorphicvector.cpp

```cpp
#include <string>
#include <vector>
#include <iostream>

// Base class for all Text derived classes
class Text {
    std::string text;
public:
    Text(const std::string& t): text(t) {}
    virtual std::string get() const {
        return text;
    }
};

// Provides minimal decoration for the text
class FancyText: public Text {
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
public:
    FancyText(const std::string& t, const std::string& left,
              const std::string& right, const std::string& conn):
         Text(t), left_bracket(left), right_bracket(right),
         connector(conn) {}
    std::string get() const override {
        return left_bracket + Text::get() + right_bracket;
    }
};

// The text is always the word FIXED
class FixedText: public Text {
public:
    FixedText(): Text("FIXED") {}
};

int main() {
    std::vector<Text *> texts { new Text("Wow"),
                                new FancyText("Wee", "[", "]", "-"),
                                new FixedText,
                                new FancyText("Whoa", ":", ":", ":") };
    for (auto t : texts)
        std::cout << t->get() << '\n';
}
```

```
Wow
[Wee]
FIXED
:Whoa:
```

Summary

A polymorphic method in C++ requires four key ingredients:

1. The method must appear in a class that is part of an inheritance hierarchy.
2. The method must declared `virtual` in the base class at the top of the hierarchy.
3. Derived classes override the behavior of the inherited virtual methods as needed.
4. Clients must invoke the method via a pointer to an object, not directly through the object itself.

Opened Data and Codes, Only to Derived

- **protected:**
  - Data or a method, is inaccessible to all code outside the class,
  - Except for code within a derived class
  - For the derived classes, it looks like public member data of Base class
  - For the outside, it is hidden

Pure Virtual Function & Abstract Class

Listing 17.5: shape.h

```
#ifndef SHAPE_H_
#define SHAPE_H_

/*
 *  Shape is the base class for all shapes
 */
class Shape {
public:
    // Longest distance across the shape
    virtual double span() const = 0;
    // The shape's area
    virtual double area() const = 0;
};

#endif
```

"***Assignment to zero***" of the virtual function means:

function is not defined at here, but the ***derived classes should override*** it in the future.

Use Visual Studio !! Goorm can't support user defined header file

What is this and meaning of `#include`?

- Header files allow you to make the interface (in this case, the `class Text`) visible to other .cpp files, while keeping the implementation (in this case, `class Text`'s member function bodies) in its own .cpp file.

Use Visual Studio !! Goorm can't support user defined header file

Code Example for File Separation (1/3)

```
#include <string>
#include <vector>
#include <iostream>

// Class declare
class Text {
    std::string text;
public:
    Text(const std::string& t);
    std::string get() const;
};

// Class definition
Text::Text(const std::string& t)
{
    this->text = t;
}

std::string Text::get() const
{
    return this->text;
}

// Client codes for class Text
int main() {
    std::vector<Text *> texts { new Text("Wow") };
    for (auto t : texts)
        std::cout << t->get() << '\n';
}
```

main.cpp

Use Visual Studio !! Goorm can't support user defined header file

Code Example for File Separation (2/3)

- Step.0 Separates class Text declaration and definition statements
- Step.1 Create new header file with appropriate name and '.hpp' file extension
  - Example: text.h for class Text
- Step.2 Cut & Paste ***class Text declaration code*** into the new header file
- Step.3 Defined conditional compile statement to avoid duplicated inclusion
  - Example: #ifndef text_hpp, #define text_hpp, #endif
- Step.4 Create new source file with appropriate name and '.cpp' file extension
  - Example: text.cpp for class Text
- Step.5 Cut & Paste ***class Text definition code*** into the new source file
- Step.6 Add #include statement for a new header file into a new source file
  - Example: #include "text.hpp" at the first line of "text.cpp"
- Step.7 Add #include statement for a new header file into a main() file
  - Example: #include "text.hpp" at the first line of "main.cpp"

Use Visual Studio !! Goorm can't support user defined header file

Code Example for File Separation (3/3)

```
#include <string>
#include <vector>
#include <iostream>
#include "text.hpp"

// Client codes for class Text
int main() {
    std::vector<Text *> texts { new Text("Wow") };
    for (auto t : texts)
        std::cout << t->get() << '\n';
}
```

main.cpp

```
#ifndef text_hpp
#define text_hpp

#include <stdio.h>
#include <iostream>
#include <string>

// Class declare
class Text {
    std::string text;
public:
    Text(const std::string& t);
    std::string get() const;
};

#endif /* text_hpp */
```

text.hpp

```
#include "text.hpp"

// Class definition
Text::Text(const std::string& t)
{
    this->text = t;
}

std::string Text::get() const
{
    return this->text;
}
```

text.cpp

Use Visual Studio !! Goorm can't support user defined header file

Code Review

Listing 17.5-17.17

Polymorphism Application

Object Oriented Programming by C++

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: http://mobilelab.khu.ac.kr/