객체지향 프로그래밍 : 개념 및 언어 (Object-Oriented Programming : Concepts and Languages)

In this talk, I will present

- *the basic concepts of object-oriented programming,*
- *an introduction to C++ with some examples as a case study of object-oriented programming language, and finally*
- *advantages and disadvantages of object-oriented programming*

서강대학교
Sogang University

Table of Contents

I. Introduction

II. Basic Concepts of Object-Oriented Programming

- *2.1 Abstract Data Type*
- *2.2 Object and Message Sending*
- *2.3 Classes and Instances*
- *2.4 (Multiple) Inheritance*
- *2.5 Dynamic Binding and Polymorphism*

III. Object-Oriented Programming Languages

- *3.1 Classifications*
- *3.2 A Case Study : C++*
- *3.3 Analysis*

IV. Summary

I. Introduction

Evolution of Programming Languages

Programming Language Paradigms

- Block Structure, Procedure-Oriented Paradigm
  - Program is a nested set of blocks and procedures
  - Primary Paradigm in the 1960s and 1970s (Algol, Pascal, PL/I, Ada, Modula)
- Object-Based, Object-Oriented Paradigm
  - Program is a collection of interacting objects
  - Simula (67), Smalltalk (70s), Many Languages (80s) (Simula, Smalltalk, C++, Eiffel, CLOS, ..)
- Concurrent, Distributed Programming Paradigm
  - Multiple threads, synchronization, communication
  - fork-join (60s) -> Ada-CSP (70s) -> Linda (CSP, Argus, Actors, Linda, Monitors)
- Functional Programming Paradigm
  - Program is a set of function definitions (rewrite rules)
  - Clear semantics, a lot of implicit parallelisms (LISP, ML, Miranda, Haskel, ..)
- Logic Programming Paradigm
  - Program is a set of theorems (resolution principles)
  - Clear semantics, a lot of implicit parallelisms (Prolog, Parlog, GHC, ..)

Why Object-Oriented Programming ?

- **Natural Modeling of Real-World Problems**
  - Several autonomous entities
  - Simulation systems

자를 때 일 단위로 할 것인가 (procedure)
데이터 단위 (data)

- **Modularity**
  - Data + Procedures
  - Problem decomposition (Software Engineering)
  - Information Hiding (Encapsulation)

Why Object-Oriented Programming ?

- **Software Re-usability**
  - Using Inheritances
  - A lot of useful class libraries (Smalltalk)

- **Parallelism**
  - Each object can be executed in parallel

- **Just a New Programming (Computing) Paradigm !**

II. Basic Concepts of Object-Oriented Programming

- **What is Object-Oriented Programming ?**
  - ***Object-oriented programming*** **is a method of implementation in which programs are organized as** ***cooperative collection of objects*****, each of which represents** ***an instance of some class*****, and whose classes are all member of a hierarchy of classes unites via** ***inheritance relationships***

메시지를 주고 받는다 (indirect subroutine 호출)

- **Object-oriented Programming Paradigm :**
  - **Decide which classes you want**
  - **Provide a full set of operations for each class**
  - **Make commonality explicitly using inheritance**

Abstraction
"Eliminate the Irrelevant,
Amplify the Essential"
Encapsulation
"Hiding the Unnecessary"
Inheritance
"Modeling the Similarity"
Polymorphism
"Same Function Different Behavior"

2.1 Abstract Data Type

- Abstract Data Type
  - ⇔ a data structure that supports both of *encapsulation* and *information hiding*

- Encapsulation
  - data and code that manipulates it are defined together, and that data cannot be separated from or accessed separately from the associated code
  - data is encapsulated within the code
  - only a localized set of procedures directly manipulate the data
  - important for ensuring reliability and modifiability of systems by reducing interdependencies between software components

- Information Hiding
  - it is the principle that states that program should not make assumptions about implementations and internal representations
  - a way of using encapsulation
  - emphasis is on *what* rather than *how*
  - procedure abstraction (subroutine) vs. data abstraction (abstract data type)

- Abstraction helps people to think about what they are doing, whereas encapsulation allows program changes to be reliable with limited effort

Abstract Data Type *Stack*

data oriented programming이지만
밖에서 stack 변수를 접근 할 수 있음

Conventional *Stack*

밖에서 stack 변수를 접근하지 못함
변수와 함수를 묶음(encapsulation)

Old Programming Technique: **Structures**

```
struct queue{
    int a[5];
    int head;
    int tail;
};

struct stack{
    int a[5];
    int top;
}

void main()
{
    struct queue Q;
    struct stack S;

    print( Q.a[2] );
    print( S.a[2] );

    add(S, 3);
    add(Q, 2);
}
```

?

```
void add(queue Z, int x){
    <codes for
    adding queue
    elements here>
}


void remove(queue Z, int x){
    <codes for
    removing queue
    elements here>
}
```

```
void add(stack Z, int x){
    <codes for
    adding stack
    elements here>
}


void remove(stack Z, int x){
    <codes for
    removing stack
    elements here>
}
```

New Programming Technique: **Object-Oriented**

```
void main()
{
    queue Q;
    stack S;

    print( Q.a[2] );
    print( S.a[2] );

    S.add(3);
    Q.add(2);
}
```

```
class stack{
    int a[5];
    int top;

    void add(int x){
        <codes here>
    }


    void remove(int x){
        <codes here>
    }

}
```

```
class queue{
    int a[5];
    int head;
    int tail;

    void add(int x){
        <codes here>
    }

    void remove(int x){
        <codes here>
    }

};
```

2.2 Object and Message Sending

- **What is Object ?**
  - an entity with its private data and methods
    - ⇔ states (instance variable : private data)
    - ⇔ a set of operations (method : procedure handling private data)

```
point : object
x := 0; y := 0 ;
read_x : ^x ; - return value of x
read_y : ^y ; - return value of y
change_x(dx) : x = x + dx ;
change_y(dy) : y = y + dy ;
```

- **Message Sending**
  - data are obtained from an object : by sending message to object
  - a form of indirect procedure call
    - 런타임에 결정(java) ⇔ dynamic vs. static message binding 컴파일 타임에 결정(c)
  - all of actions in object-oriented programming comes from sending messages between objects
  - a *selector* in the message specifies the kind of operation

- **Traditional Programming vs. Object-Oriented Programming**
  - **Traditional Programming**
    - ⇔ a collection of procedures which are independent of data
    - ⇔ function values are completely determined by their arguments being precisely the same for each invocation
    - ⇔ typically procedures act only on certain type of data

  - **Object-Oriented Programming**
    - ⇔ a collection of objects (data + procedure)
    - ⇔ the value returned by an operation on an object may depend on its state as well as its arguments (invocation history)
    - ⇔ finding the correct procedure to execute is handled by the support system of language

Procedural Oriented Programming
Object Oriented Programming
Global Data
Global Data
Function 1
Function 2
Function 3
Local Data
Local Data
Local Data
Data
Functions
Data
Functions
Functions
Data

Java
std:
std1:
std2:
std3:
null
Object (class Student)
name:
test1: 0
test2: 0
test3: 0
(getAverage())
Object (class Student)
name:
test1: 0
test2: 0
test3: 0
(getAverage())
Object (class String)
"Mary Jones"
Object (class String)
"John Smith"

Procedural language
Object-oriented language
Procedure 1
Parameters
Procedure 2
Parameters
Procedure 3
Data passed between procedures
Object 1
Operations
Attributes
Object 2
Operations
Attributes
Object 3
Operations
Attributes
Object encapsulates code and data
Objects interact
Code
Data

Review:
Two Programming Paradigms
Structural (Procedural) PROGRAM
Object-Oriented PROGRAM
FUNCTION
FUNCTION
FUNCTION
Function calls
OBJECT
Operations
Data
OBJECT
OBJECT
Operations
Data
Operations
Data
Messages passing
1

An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

```
class Program
{
  string name;
  int age;

  void setdata(string n,int a)
  {
   name = n;
   age = a;
  }
  void showdata()
  {
Console.WriteLine(name+" "+age);
  }
}
```

Member Variables

Member Functions

Depend on execution of object

```
public static void Main()
   {
    Program obj1,obj2;
    obj1 = new Program();
    obj2 = new Program();

    obj1.setdata("Ram",20);
    obj2.setdata("Shyam",10);

    obj1.showdata();
    obj2.showdata();
   }
```

컴파일 타임이면 static message to method binding
런타임이면 dynamic message to method binding

obj1	obj2
name = Ram age = 20	name = Shyam age = 20

2.3 Classes and Instances

- **What is Class ?**
  - a specification of structure (instance variable), behavior (method), and inheritance (parent) ;
  - a class is a template (cookie cutter) from which objects may be created by "new" or "create" operation
  - objects are created from classes through instantiation
    - ⇔ an object of given class is called an *instance* of that class
  - two kinds of variables
    - ⇔ *Class variable* : a variable stored in the class whose value is shared by all instance of class
    - ⇔ *Instance variable* : a variable for which local storage is available in instances
  - if a class is an object, then class must have a class, called *metaclass*
    - ⇔ class?class

- **Example**
  - **Definition of Class**

```
point : class
Description of instance variables
operations or methods
```

  - **Creation of Object**

```
p1 := make_instance point(0,0)
p2 := make_instance point(1,1)
```

object는 자기 클래스 뿐만 아니라 superclass의 데이터와 operation을 상속받는다
우선 자기 클래스에서 변수, operation을 찾고 없으면 superclass에서 찾는다

Point Class
Class Variables Desc.
Instance Variables Desc.
Shared Class Operations

Class

Object Creation

Instance p1
Private State : (0,0)

Instance p2
Private State : (1,1)

Instance (Object)

2.4 (Multiple) Inheritance

공통적인 걸 한 곳에 모으고 그걸로 새로운 클래스를 만든다

- **Inheritance**
  - a relation ship between classes where one class is the parent (or base) class of another
  - supports refinement and software reuse in Object-Oriented Programming ;
    - ⇔ the ability to inherit state structures and behaviour from an existing class allows the programmer to define new objects in the system not only in terms of existing objects, but also by modifying and mixing the descriptions of *existing classes* (*superclass*)
  - Classification and Specialization
  - Example of Inheritance Hierarchy

- **Example of Class, Inheritance, and Instance**

Inheritance - Example

```
#include <string>
class Person {
 private:
  std::string name;

 public:
  Person(std::string n) : name(n) {}
  void set_name(std::string n);
  std::string get_name();
};
```

```
class Student : public Person {
  private:
    int student_id;
    int year;

  public:
    Student(std::string n, int y, int id) :
      Person(n), year(y), student_id(id) {}
    void set_student_id(int id);
    int get_student_id();
    void set_year();
    int get_year();
};
```

5

5

- **Data Structures Illustrating Concepts Related to OOP**

- **What is Inherited ?**
  - a class inherits instance variable declarations as well as method from its superclass
  - Specialization Method
    - ⇔ Adding : introducing new instance variables and new methods
    - ⇔ Substitution (or Overriding) : class's attributes (variables or methods) may be refined using the superclass as base
    - ⇔ Class Precedence List : accessing closest superclass or ...
- **Inheritance Structures**
  - Hierarchical Inheritance
    - ⇔ classes may inherit only from a single superclass
    - ⇔ most widely used inheritance (Smalltalk)
    - ⇔ simple and efficient, but limited in expressibility
  - Inheritance by Delegation
    - ⇔ each object is responsible for both choosing which messages it will handle, and for choosing an object to handle those messages that it is not prepared to handle
  - Multiple Inheritance
    - ⇔ a class inherits from more than one parent
    - ⇔ increase the sharing

- **Multiple Inheritance**
  - a class inherits from more than one parent
  - increase the sharing
  - a class inherits the union of variables and methods from all its superclasses
  - if there is conflict, then we use a class precedence list to determine precedence for variable description or method (*depth-first up-to-joins*)
- **Advantages of Inheritance**
  - Better Conceptual Modeling
    - ⇔ direct modeling of everyday life
    - ⇔ hierarchical modeling make the program easier to understand
  - Factorization
    - ⇔ describe only once and reuse when needed
  - Stepwise Refinement in Design
    - ⇔ top-down design and verification
  - Polymorphism

- **Multiple Inheritance Example**

- A class can inherit from several other classes.

Most modern languages don't support multiple inheritance!

2.5 Dynamic Method Binding and Polymorphism

- **Dynamic Method Binding**
  - **static message binding :**

    ⇔ **the binding of message to a particular method of an object takes place at compile time (statically typed language)**
  - **dynamic message binding :**

    ⇔ **the binding of message to a particular method of an object takes palce at compile time (untyped languages)**

    ⇔ **a powerful mechanism for supporting polymorphism**

With Dynamic Msg Binding

"draw"

shape

```
class circle : shape {
   ....
   draw() {   ....  }
}
```

```
class triangle : shape {
   ....
   draw() {   ....  }
}
```

```
class rectangle : shape {
   ....
   draw() {   ....  }
}
```

Without Dynamic Msg Binding

"draw"

shape

```
class shape {
   ....
 draw() {
  switch (shape.type) {
    case circle :
            ......
    case triangle :
            ......
    case rectangle :
            ....
   }
}
```

- **Polymorphism (다양성)**
  - ability for operations to operate on more than one type (or class)
  - classification

    c,java 결과 차이 시험

    ⇔ **ad hoc polymorphism** : coercion, operator overloading

    ⇔ **universal polymorphism** : parametric, inclusion (inheritance)

c는 efficiency를 중요시해서 static
interpreter로 구현된건 dynamic

```
int i;
사람 a;
학생 김서강, b;
탤런트 한석규;

b = a; /* ? */

a.쉬어라();
a = 김서강;
a.쉬어라();
a = 한석규;
a.쉬어라();
```

subclass가 superclass를 할당 받을 수 없음

static message to method binding - a는 사람 클래스니까 잠을잔다를 수행
dynamic - 런타임때 - 미팅한다가 수행

dynamic message binding에 의해서 수행되는 내용이 다름 - polymorphism

null 요소를 만드는 이유 - 컴파일 에러를 피하기 위해

III. Object-Oriented Programming Languages

3.1 Classification

- Do they support : Object ? Classes ? Inheritance ?
  - *Object-based Language*
    - ⇔ the class of all language that support object
  - *Class-based Language* 한번 선언하고 여러번 찍어낼 수 있다
    - ⇔ the subclass that requires all objects to belong to a class
  - *Object-oriented Language*
    - ⇔ the subclass that requires classes to support inheritance
      - ⇒ Extending Conventional Languages
        - → C++, Objective C, Object Pascal, Object COBOL, CLOS
      - ⇒ Pure Object-Oriented Languages
        - → Eiffel, Simula, Smalltalk

- **Genealogy of Object-Based and Object-Oriented Programming Languages**

3.2 A Case Study : C++

- **What is C++ ?**
  - developed by B. Stroustrup's Group at Bell Lab. early 1980
  - based on C (compatible with C, Superset of C)
  - incorporate object-oriented concepts, class, inheritance, etc.
  - extend C with other high-level features such as generic function, reference type, etc.
  - most popular object-oriented programming language
  - emphasize on efficiency and compatibility with C
- **C++ History**

C with class (1981)

AT&T C++ Release 1.0 (1985)

AT&T C++ Release 2.0 (1989)

GNU C++

ANSI C++ Language Committee(1989)

AT&T C++ Release 3.0 (1991)

MS-C/C++ 7.0

Borland C++ 3.1

- **Characteristics of C++**
  - Classes
  - Access Levels

  → **Data Abstraction**

  - Subtyping
  - Virtual Functions

  → **Dynamic Binding**

  - Multiple Inheritance
  - Virtual Base Classes

  → **Inheritance**

  - Template Functions
  - Template Classes
  - Operator Overloading

  → **Polymorphism**

(1) Data Abstraction

constructor
desstructor

C Programming

```
#define MAXSIZE 100

char stack[MAXSIZE];
int top = 0;

push(char x) {
  if ((top+1) == MAXSIZE)
       error("stack is full\n");
  stack[++top] = x;
}

char pop() {
  if (top == 0)
       error("stack is empty\n");
  return(stack[top--]);
}

main() {
  char x, y;
  push('a'); push('b');
  x = pop(); y = pop() ;
  printf("%c, %c \n", x,y);

}
```

global variable

C++ Programming

```
const int MAXSIZE = 100;
class stack {
  private:
    char stack[MAXSIZE];
    int top;
  public:
    stack() {top = 0;}
    void push(char);
    char pop();
};

void stack::push(char x) {
   if ((top+1) == MAXSIZE)
        error("stack is full\n");
   stack[++top] = x;
}

char stack::pop() {
   if (top == 0)
       error("stack is empty\n");
   return(stack[top--]);
}

stack st1;   /* static object creation*/
main() {
  char x, y;
  st1.push('a'); st1.push('b');
  x = st1.pop(); y = st1.pop();
 printf("%c, %c \n", x,y);
}
```

local variable

(2) Operator Overloading

– the same symbol or function name can be used for different meaning

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

class String {
  char* str; int len;
public:
  String(const char*);
  ~String() {delete[] str;}
  char* getString() {return str;}
  String& operator += (String&);
}

String::String(const char* s) {
  len = strlen(s);
  str = new char[len+1];
  strcpy(str,s);
}
```

```
String& String::operator+= (String& s){
  len += s.len ;
  char *p = new char[len+1];
  strcpy(p, str);
  strcat(p, s.str);
  delete str;
  str = p;
  return *this;
}

main() {
  String s1("I am");
  String s2("hungry");
  String s3("and sleepy");
  s1 += s2;
  cout << "The result is";
  cout << s1.getString() << "\n";
}
```

(3) Inheritance, Access Level, and Dynamic Message Binding

- **Access Level**
  - **private members** : accessible only by member functions and friends of the class where they are declared
  - **protected** : like private members, excepts in derived class (subclass)
  - **public** : accessible by any function
- **Access Mode**
  - **public derived class** : the same as the superclass
  - **private derived class** : both the public and protected members of the superclass are private

private으로 선언한 변수가 subclass에서 access할 수 있지만 그 subclass의 subclass에서는 access할 수 없음

```
class A {
  private:
      ….
  protected:
      …..
  public:
      ….
}
```

- **Examples**

```
class employee {
  private:
    static employee* list;
  protected:
    char* name;
    char* dept;
    employee* next;
  public:
    employee(char*, char*);
    print_list();
    virtual print();
}

class manager: public employee {
  protected:
    short level;
  public:
    manager(char*, int, char*);
    print();
};

employee::print_list() {
  for (employee* p=list; p; p->next)
     p->print();
}
```

```
class A {
  public:
    virtual display(int i) {
       printf("in A %d\n", i); }
}

class B {
  public:
    virtual display(double d) {
       printf("in B %d\n", d); }
}

class C : public B, public A {
   public:
     virtual display(int i) {
        A::display(i);
     }
     virtual display(double d) {
        B::display(d);
     }
}
main() {
  C c;
  c.display(13);
  c.display(3.14);
}
```

(3) Built-in Class Libraries

- A lot of built-in class libraries for various applications
  - Borland C++
    - ⇔ Container Class Library
    - ⇔ Object Window Library (OWL)
  - MSC 7.0
    - ⇔ Microsoft Foundation Class (MFC)

3.3 Analysis

- Advantages of Object-Oriented Programming Languages
  - Encapsulation and Data Abstraction
    - ⇔ increase reliability
    - ⇔ help to decouple procedural and representational specification from implementation
  - Dynamic Binding
    - ⇔ increasing flexibility
  - Inheritance
    - ⇔ increase software reusability
- Disadvantages
  - High run-time costs for
    - ⇔ dynamic binding
    - ⇔ message Passing (1.7 times)
  - Implementation is harder
    - ⇔ semantic gap 하드웨어와 프로그램 메커니즘 차이
    - ⇔ software simulation
  - Programmer must learn extensive class libraries
    - ⇔ hard to learn (Smalltalk, ...)

IV. Summary

Object-Oriented Programming
Object
Class
Method Message
Inheritance
Dynamic Binding
Polymorphism

Object-Based
+ *class*
Ada
Actor
Class-Based
Clu
+ *class inheritance*
Object-Oriented
Smalltalk
C++

- **Future Researches**

"Design and Implementation of Parallel C++

Basic Concepts

[1] M. Stefik and D.G. Bobrow, “Object-Oriented Programming : Themes and Variations,” *AI Magazine*, Dec. 1986, pp.40-62.
[2] Peter Wegner, “Dimensions of Object-Based Language Design,” *Proc. of OOPSLA87*, 1987, pp.168-182.
[3] B.L. Horn, *An Introduction to Object-Oriented Programming, Inheritance and Method Combination*, Technical Report, CMU-CS-87-127, CMU.
[4] Bob Hathaway, “comp.object FAQ (Frequently Asked Question)” Internet News Group comp.object (Draft), 1993.
[5] B.P. Pokkunuri, “Object Oriented Programming,” *SIGPLAN Notices*, Vo. 24, No. 11, 1988, pp.96-101.

Object-Oriented DataBase

[1] E. Bertino and L. Martino, “Object-Oriented Database Management Systems : Concepts and Issues,” *IEEE Computer*, Vol. 24, No. 4, April 1991, pp.33-47.
[2] D.H. Fishman, et. al., “Overview of the Iris DBMS,?in *Object-Oriented Concepts, Database, and Applications*, W. Kim and F. Lochovsky (eds), Addison-Wesley, 1989, pp.219-250.
[3] W. Kim, et. al., “Integrating an Object-Oriented Programming System with a Database System,” *Proc. of OOPSLA88*, 1988, pp.142-152.
[4] S. Ahmed, et. al., *A Comparison of Object-Oriented Database Management Systems for Engineering Applications*, Research-Report No. R91-12, IESL90-03, MIT, Dept. of Civil Engineering, May 1991.

Object-Oriented Design

[1] G. Booch, “Object-Oriented Development,” *IEEE Transaction on Software Engineering*, Vol. SE-12, No. 2, Feb. 1986, pp.211-221.
[2] B. Meyer, “Reusability : The Case for Object-Oriented Design,” *IEEE Software*, March 1987, pp. 50-64.
[3] A.I. Wasserman, et. al., “The Object-Oriented Structured Design Notation for Software Design Representation,” I*EEE Computer*, March 1990.

Object-Oriented Languages

[1] J.H. Saunders, “Survey of Object-Oriented Programming Languages,” *Journal of Object-Oriented Programming*, March/April 1989, pp.5-11.
[2] T. Budd, An Introduction to Object-Oriented Programming, Addison-Wesley Publishing, 1991.

Operating System Supports for Object-Oriented Languages

[1] J.A. Marques and P. Guedes, “Extending the Operating System to Support an Object-Oriented Environment,” *Proc. of OOPSLA89*, 1989, pp.113-122.
[2] S. Habert and L. Mosseri, “COOL : Kernel Support for Object-Oriented Environments,” *Proc. of OOPSLA90*, 1990, pp.269-277.
[3] R. Lea, et. al., “POOL-2: An Object-Oriented Support Platform Built Above the Chorus Micro-Kernel,?” *Proc. of Object-Orientation in Operating System*, 1991, pp.68-72.

(Parallel) Implementations of Object-Oriented Language

[1] G. Krasner, “The Smalltalk-80 Virtual Machine,” *BYTE*, August 1981, pp.300-320.
[2] C.B. Duff, “Designing an Efficient Language,” *BYTE*, August 1981, pp.211-224.
[3] S. Krakowiak, et. al., “Generic Object-Oriented Virtual Machine,” *Proc. of Object-Orientation in Operating System*, 1991, pp.73-77.
[4] R.S. Chin and S.T. Chanson, “Distributed Object-Based Programming Systems,” *ACM Computing Survey*, Vol.23, No.1, March 1991, pp.91-124.
[5] C. Chambers, et. al., “An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language based on Prototypes,” *Proc. of OOPSLA89*, pp.49-70.
[6] L.V. Kale and S. Krishnan, *CHARM++: A Portable Concurrent Object-Oriented System Based on C++*, Technical Report-??, University of Illinois, Urbana-Champaign, 1993.

Object-Oriented Computer Hardware

[1] D. Ungar, et. al., “Architecture of SOAR : Smalltalk on a RISC,” *Proc. of 11th Int’l Symposium on Computer Architecture*, 1984, pp.188-197.

Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison Wesley, 2nd Edition, 1997(?).

C++ as a Better C

- C++ extends the C programming language in a number of important ways
- Its features make it more reliable and easier to use than C

- Comment Style : "//"
  - one-line comment
  - everything on a single line after the symbol "//" is treated as a comment

```
// The computation of circumference and area of circle
#include <iostream.h>

const float pi = 3.14159; // pi accurate to six places
const int true = 1;

inline float circum(float rad) {return (pi*2*rad);}
inline float area(float rad) {return (pi*rad*rad);}

main() {
  float r;
  while (true) {
    cout << "\n Enter radius: "  // prompt for input
    cin >> r;
    cout << "\n Area is " << area(r);
    cout << "\n Circumference is " << circum(r) << endl;
  }
  return(0);
}
```

`<<` : *put to*
`>>` : *get from*
`endl` : *new line and flush*

- **Avoiding the Preprocessor : *inline* and *const***
  - **inline** 타입체킹까지 function call overhead 없이 호출
    - ⇔ a request to the compiler that the function be compiled without function call overhead
    - ⇔ inline *vs.* macro
      - ⇒ `#define SQ(X) X*X /* macro */`
      - ⇒ `SQ(a+b) ==> a+b*a+b`
      - ⇒ type checking

inline 함수의 경우
컴파일 단계(Compiling)에서 처리가 이루어짐
일반적인 함수들을 위해 실행하는 타입 검사가 그대로 적용됨.

macro 확장자의 경우
전처리 단계(Preprocessing)에서 치환이 이루어짐
대치 시 어떠한 타입 검사도 이루어지지 않는다.

  - **const**
    - ⇔ a type specifier
    - ⇔ a variable declared as *const* cannot have its value changed

```
const false = 0;                // implicit type is int
const double e = 2.71828;       // natural logarithm base
const int M_size = 100;         // used in array declaration
const* p = &M_size;             // a pointer to a constant int
char* const s = "abcd";         // a constant pointer to char

const double pi=3.141592;
const double *d_p1 = &pi;       // legal: pi is an lvalue
const double *d_p2 = &3.1       // illegal : 3.1 is not an lvalue
pi = 3.141596 ;                 // illegal because pi is nonmodifiable
```

- **Declaration**
  - C++ allows declarations to be intermixed with executable statements

```
for (int i=0; i,52; ++i) {
  int k = rand()%52;
  card t = d[i] ;
  d[i] = d[k] ;
  d[k] = t;}
```

- **Scope Resolution Operator** : "::"
  - *static scoping rule* : a name in an inner block hides the outer block or external use of the same name
  - however, when used in form *::variable*, it allows access to the externally named variable

```
#include <iostream.h>
int i = 1 ;        // external i

main() {
  int i = 2;     // re-declares i locally
  {
    cout << "enter inner block\n"
    int n = i;
    int i = 3;
    cout << i << "i <> ::i" << ::i << "\n";
    cout << "n = " << n << "\n";
  }
  cout << "enter outer block\n"
  cout << i << "i <> :: i" << ::i << endl;
}
```

Output

```
enter inner block
3 i <> ::i 1
n = 2
enter outer block
2 i <> ::i 1
```

- **Function Prototyping** parameter type checking
  - by *explicitly* listing the type and number of arguments, strong type checking and assignment-compatible conversions are possible in C++
  - Example

```
double sqrt(double x);
void make_str(char*, int);
void print(const *char s);     // s is not modified
int printf(char* format, ...) // variable number of arguments
```

  - C prototyping vs. C++ prototyping

in C

```
double sqrt() ;
main() {
 ...
 printf("%f is sqrt of 4\n",
                    sqrt(4));
 ...
}
```

output

```
0 is sqrt of 4
```

in C++

```
double sqrt(double) ;
main() {
 ...
 printf("%f is sqrt of 4\n",
                    sqrt(4));
 ...
}
```

output

```
2.0 is sqrt of 4
```

- **Reference Declarations and Call-by-Reference**
  - declare the identifier to be an alternative name for an object specified in an initialization of reference
  - Example

```
int n ;
int& nn = n ; // nn is an alternative name for n
double a[10];
double& last=a[9]; // last is an alias for a[9]
const char& newline='\n' ;
```

  - it allows C++ to have call-by-reference argument directly

```
int greater(int&, int&) ;
mani() {
 int a = 5;
 int b = 4;
 ...
 greater(a,b);
 printf("a=%d, b=%d\n", a, b);
}
```

output

```
a=4, b=5
```

```
int greater(int& x, int& y)
{
  if (x > y) {  // exchange
     int temp = a;
     a = b;
     b = temp;
     return(1);
  }
  else
     return(0);
}
```

- **Default Arguments** 끝에 default parameter를 써야함
  - *a formal parameter* can be given a default argument
    ⇔ this is usually a constant that occurs frequently when the function is called

```
int mult(int n, int k=2) // k=2 is default
{
  if (k==2)  return (n*n);
  else       return (mult(n,k-1) * n);
}
main() {
  ..
  mul(i+5)   // compute (i+5)*(i+5)
  mult(i+5,3) // compute (i+5)^3
}
```

  - *only trailing parameters* of a function can have default values

```
void foo(int i, int j=7);              // legal
void goo(int i=3, int j);              // illegal
void hoo(int i, int j=3, int k=3);     // legal
void moo(int i=1, int j=2, int k=3);   // legal
void noo(int i, int j=2, int k);       // illegal
```

- **Overloading Function** 이름은 같지만 return 타입이 다름
  - the term *overloading* refers to using the same name for multiple meaning of an operator or function
  - the meaning selected depends on the types and number of arguments used by the operator or function
  - Example

```
double average(const int a[],int size) {
  int sum=0;
  for (int i=0; i<size; i++) {
    sum = sum + a[i]; // int arithmetic
  }
  return((double) sum/size);
}
double average(const double a[],int size){
  double sum=0.0;
  for (int i=0; i<size; i++) {
    sum = sum + a[i]; // double arith
  }
  return(sum/size);
}
double average(const int a[], double b[],
               int size) {
  double sum=0.0;
  for (int i=0; i<size; i++) {
    sum = sum + a[i] + b[i]; //double arith
  }
  return(sum/size);
}
```

the compiler chooses the function with matching types and arguments

```
main() {
 int w[5]={1,2,3,4,5};
 double x[5]={1.1,2.1,3.1
              4.1,5.2}

 cout << average(w,5);
 cout << average(x,5);
 cout << average(w,x,5);

 return(0);
}
```

- **Free Store Operators `new` and `delete`**
  - the unary operator `new` and `delete` are available to manipulate *free store*
    - ⇔ free store is a system-provided memory pool for objects whose lifetimes are directly managed by the programmer
    - ⇔ replace the standard library functions `malloc`, `calloc`, `free`
  - Example

memory leak - 잘 돌아가다 죽음. new을 얼마나 많이 하는가

```
int* prt_i, *v;
double (*)[N] q;
ptr_i = new int(5); // allocate and initialize, so *ptr_i is 5
v = new int[40] ; // allocate a vector of 40 integers, v == &v[0]
q = new double[n][N]; // allocate an n by N vector of integer,
                      // q == &q[0][0]
```

```
main() {
  int *data ; int size ;

  cout << "\nEnter array size:"; cin >> size;
  data = new int[size];
  for (int j=0; j< size; j++) {
    cout << (data[j] = j) << "\t";
  cout << endl;

  delete []data;
  data = new int[size];
  for (int j=0; j< size; j++) {
    cout << (data[j] = j) << "\t";
  }
}
```

print different values???

- **Stack Example -1**

```
class stack {
 private:
   char s[max_len];
   int top;
   enum{EMPTY=-1, FULL=max_len-1};
 public:
   void reset() {top = EMPTY;}
   void push(char c) {top++; s[top]=c;}
   void pop() {return(s[top--]);}
   void top_of() {return(s[top]);}
   void empty() {return (top==EMPTY);}
   void full() {return (top==FULL);}
}
```

```
main() {
  statck ss;
  char str[40] =
        {"Sogang Univ!"};

  cout << str << "\n";
  ss.reset();
  while(str[i])
     if (!ss.full())
          ss.push(str[i++]);
  while (!ss.empty())
     cout << ss.pop();
  cout << "\n";
}
```

destructor는 Parameter가 없음 overloading이 안됨

output

```
Sogang Univ!
!vinU gnagoS
```

- **Stack Example -2**

```
class stack {
 private:
   char *s;
   int top, max_len;
   enum{EMPTY=-1, FULL=max_len-1};
 public:
   void stack() {max_len=100;
                 s = new char[max_len];
                 top = EMPTY;}
   void stack(int size) {
                 max_len=size;
                 s = new char[max_len];
                 top = EMPTY;}
   void ~stack() {
                delete []s;
   }
   void push(char c) {
                top++; s[top]=c;}
   void pop() {
                return(s[top--]);}
   void top_of() {
                return(s[top]);}
   void empty() {
                return (top==EMPTY);}
   void full() {
                return (top==FULL);}
}
```

overloaded constructor

```
main() {
  statck ss, tt(20);
  char str[40] =
        {"Sogang Univ!"};

  int i=0;
  cout << str << "\n";
  while(str[i])
     if (!ss.full())
          ss.push(str[i++]);
  while (!ss.empty())
     cout << ss.pop();
  cout << "\n";
}
```

output

```
Sogang Univ!
!vinU gnagoS
```

C++ as an Object-Oriented Programming Language

data oriented에서 시작된 oop
procedure에서 데이터를 관리하자 - adt
c++에서 classs는 adt를 형성하는 것

- **Classes and Abstract Data Type**
  - a class provides the means for implementing a user-defined data type and associated functions and operators
  - class can be used to to implement an ADT

```
#include <string.h>
#include <iostream.h>
const int max_len = 255;

class string {
  public:  // univeral access
    void assign(const char* st) {
      strcpy(s, st); len = strlen(st);
    }
    int length() {
      return(len);
    }
    void print() {
      cout << s <<
      "\nLength: " len << "\n";
    }
  private:
    char s[max_length];
    int len ;
}
```

밖에서 보임

밖에서 안보임(추상화)

```
main() {
  string one, two;
  char three[40]={"Sogang Univ."};
  one.assign("Dept. of CS");
  two.assign(three);
  cout << three;
  cout << "\nLength:" <<
        strlen(three) << endl ;
  if (one.length()<= two.length())
      one.print();
  else
      two.print();
}
```

output

```
Sogang Univ.
Length: 12
Dept. of CS
Length: 11
```

- **static Member** 데이터 중에서 static으로 선언된 것

  ⇔ a data member that is declared *static* is *shared* by all variables of that class and is stored uniquely in one place

  ⇒ nonstatic data members are created for each instance of the class

  ⇔ since a static member is independent of any particular instance, it can be accessed in the form *class_name*::identifier

```
class str {
  public:      member function
    static int how_many;   // declaration
    void print();
    void assign(const char*);
    ...
  private:     member variable
    char s[100];
}
```

static이 없으면 s1,s2,s3가 각각 how_many 멤버를 가짐
근데 static으로 선언하면 모두가 공유되게 선언됨

```
main() {

  str s1, s2, s3, *p;
  str::how_many = 3;
  ...
  str t ;
  t.how_many++;
  ...
  p = new str;
  p->how_many++;
  ...
  delete p;
  str::how_many--;
}
```

- **nested class**

```
char c;                    // external scope ::c
class X {
  public:
    class Y{
      public:
        void foo(char e) { ::c = X::c = c = e;}
      private:
        char c; // X::Y::c
    }
  private:
    char c;                // X::c
};
```

c 에서는 struct 안에 struct를 선언하는 게 안됨
근데 c++은 class를 nested하게 선언할 수 있다

- **Constructor and Destructor**
  - **Constructor**
    - ⇔ a member function whose job is to initialize a variable of its class
    - ⇔ is invoked anytime an object of its associated class is created
  - **Destructor**
    - ⇔ a member function whose job is to deallocate or finalize a variable of its class
    - ⇔ is called implicitly when an automatic objects goes out of scope
  - **new**
    - ⇔ allocates the appropriate amount of memory to store this type from free store and returns the pointer value that addresses this memory

```
class string {
 public:   // univeral access
  string() {len=255; s = new char[255];}
  string(int n) { s = new char[n+1]; len = n;}
  string(const char* p) {
     len = strlen(p) ; s = new char[len+1];
     strcpy(s, p); }
  ~string() { delete []s; }
  void assign(const char* st) {
     strcpy(s, st); len = strlen(st);}
  int length() { return(len);}
  void print() {
     cout << s << "\nLength: " len << "\n";}
 private:
    char *s;
    int len ;
}
```

세개의 constructor가 overloading

```
main() {
   string a,b(10);
   string c("Sogang");
   ...
}
```

param이 없는 기본이 수행

param이 int인 것을 수행

param이 const char* 인 게 수행

```
// ADT Conversion
x = float(i) ; // C++ function notation
x = (float) i ;
```

```
// automatical type conversion from char* to string
string::string(const char* p) { /* constructor */
  len = strlen(p);
  s = new char[len+1];
  strcpy(s, p);
}

.............
string s;
char* logo = "Sogang Univ.";
s = string(logo) ; // perform conversion then assign
s = logo // implicit invocation of conversion
```

s를 생성

새로운 object 생성 param으로 char* 넘어감
이 생성된 object를 아까 생성한 s에 복사

타입이 맞지 않는 할당이지만 C++는 logo가 char* 타입이고 s가 string object니까
만약 string의 생성자가 char*를 받은 게 있다면 그 constructor를 실행시킨다

- **Overloading**
  - refers to the practice of giving several meanings to an operator or a function
  - the meaning selected depends on the types of the arguments used by the operator or function

```
class string {
    ...
    void print() {
      cout << s <<
      "\nLength: " len << "\n";
    }
    void print(int n) {
      for (int i=0;i<n;i++){
        print();
      }
    }
    ...
}
```

```
main() {
    string three;
    three.print();
    three.print(9);
    three.print(-2);
}
```

쓰는 이유 : 언어에서 +는 양옆에 숫자밖에 못 오는데,
행렬 더하기 같은 걸 하고 싶은데 + 기호를 쓰고 싶을 때 operator overloading으로 + 연산함수를 정의

함수 이름을 그 language의 연산자로 하겠다

  - *operator* overloading and *friend* function
    - ⇔ `operator` : precedes the operator token and replaces what would otherwise be a function name in a function declaration
    - ⇔ `friend` :
      - ⇒ the keyword `friend` gives a function access to the private members of a class variable
      - ⇒ a friend function is not a member of the class but has the privileges of function in the class in which it is declared

```
#include <string.h>
#include <iostream.h>
const int max_len = 255;

class string {
 public:  // univeral access
  void assign(const char* st) {
     strcpy(s, st); len = strlen(st);}
  int length() {
     return(len);}
  void print() {
     cout << s <<
     "\nLength: " len << "\n";}
  friend string operator+(const string& a,
                          const string& b);
 private:
    char s[max_length];
    int len ;
}

string operator+(const string& a,
                 const string& b){
  string temp;
  temp.assign(a.s);
  temp.len = a.len + b.len;
  if (temp.len < max_len)
     strcat(temp.s, b.s);
  else
     cerr << "Max length exceeded
              in concatenation.\n";
  return(temp);
}
void print(const char* c) {
  cout << c << "\nlength: " <<
          strlen(c) << "\n";
}
```

이 함수는 이 class의 member function은 아니지만 private data를 access할 수 있다 data 추상화에는 맞지 않음

not a member function of string class,but can access len variable

```
main() {
 string one, two, both;
 char three[40]={"Sogang Univ."};

 one.assign("Dept. of CS");
 two.assign(three);
 print(three);

 if (one.length()<=two.length())
      one.print();
 else
      two.print();
 both = one + two ;
 both.print();
 return(0);
}
```

friend로 선언 안하면 one.+(two)일케 써야함

operator의 매개변수가 해당 object class와 동일하면 해당 object class의 멤버에 접근할 수 있다

output

```
Sogang Univ.
length: 12
Dept. of CS
Length: 11
Dept. of CSSogang Univ.
Length:23
```

이거 시험 빵꾸 채우기

```
// a safe vect with [] overloaded
#include <iostream.h>
#include <stdio.h>

class vec {
  public:
    vect();
    vect(int n);
    vect(const vect& v);
    vect(const int a[], int n);
    ~vect() {delete []p;}
    int ub() const {return (size-1);}
    int& operator[](int i) const;
  private:
    int* p;
    int size;
}

vect::vect() {
  size = 10; p = new int[size];
}

vect::vect(int n) {
  if (n <= 0) {
    cerr<<"illegal vect size:"<<n<<"\n";
    exit(1);
  }
  size = n ; p = new int[n];
}

vect::vect(const int a[], int n) {
  if (n <= 0) {
    cerr<<"illegal vect size:"<<n<<"\n";
    exit(1);
  }
  size = n ; p = new int[size] ;
  for (int i=0; i<size; i++) p[i] = a[i];
}

vect::vect(const vect& v) {
  size = v.size ; p = new int[size];
  for (int i=0 ; i<size; i++)
        p[i] = v.p[i]; /* IS IT OK ?? */
}
```

class안에 prototyping 하고 밖에서 정의

```
int& vect::operator[](int i) const {
  if (i<0 || i>ub()) {
    cerr << "illegal vect index:"
         << i << "\n";
    exit(0);
  }
  return(p[i]);
```

자기랑 같은 object가 매개변수로 오니까 access할 수 있음

```
vect& vect::operator=(const vect& v) {
  int s = (size<v.size)?size:v.size;
  if (v.size != size)
    cerr << "copying different size"
         << size << " and " << v.size;
  for (int i=0; i<s; i++)
    p[i] = v.p[i];
  return(*this);
}

vect vect::operator+(const vect& v) {
  int s = (size<v.size)?size:v.size;
  vect sum(s);
  if (v.size != size)
    cerr << "adding different size"
         << size << " and " << v.size;
  for (int i=0; i<s; i++)
    sum.p[i] = p[i] + v.p[i];
  return(sum);
}
```

..... a.[](1) : a 객체의 []함수에 1을 매개변수로 줌

```
vect a(10), b(5);
a[1] = 5; a[12] = b[4]+3;
a = b        // a, b are type vect
a = b = c ; // a,b,c are type vect
a = vect(data, DSIZE) // data[DSIZE]
a = b+a;
a = b + (c = a) + d ;
```

```
// friend function

class vect {
  public:
    friend vect mpy(const vect& v, const matrix& m);
    ...
  private:
    int*  p;
    int size;
};

class matrix {
  public
    friend vect mpy(const vect& v, const matrix& m);
    ...
  private:
    int ** p;
    int s1, s2;
};

vect mpy(const vect& v, const matrix& m) {
  if (v.size != m.s1) { // incorrect sizes
    cerr << "multiply failed - size incorrect" << v.size
         << " and " << m.s1 << "\n";
    exit(1);
  }
  // use privilleged access to p in both classes
  vect ans(m.s2)
  int i, j;
  for (i=0; i<=m.ub2;i++) {
    ans.p[i] = 0;
    for (j=0; j<=m.ub1; j++) ans.p[i] += v.p[j] * m.p[i][j];
  }
  return(ans);
}
```

vect, matrix object의 멤버를 다 접근할 수 있음

a-b-c 객체 순으로 inherit 될 때
c를 생성하면 a,b,c, 순서로(general superclass부터) constructor가 생성된다
destructor는 c,b,a 순으로(specialize class) 한다

- **Inheritance**
  - many types are variants of one another, and it is frequently tedious and error prone to develop new code for each
  - deriving a new class from an existing one called *base class* : *inheritance*

    ⇔ the base class can be *added* to or *altered* to create the derived class

```
enum support {ta, ra, fellowship, other} ;
enum year {fresh, soph, junior, senior, grad};

class student {
  public:
    student (char* nm, int id, double g, year x);
    void print();
  private:
    int student_id;
    double gpa ; year y; char name[30];
}

class grad_student: public student {
  public:
    grad_student(char *nm, int id, double g, year x,
                 support t, char *d, char *th);
    void print();
  private:
    support s;
    char dept[10];
    char thesis[80];
}
```

```
enum year {fresh, soph, junior, senior, grad};
class student {
  public:
    student (char* nm,int id,double g year x);
    void print() const;
  protected:
    int student_id; double gpa;
    year y; char name[30];
};

enum support{ta, ra, ga, fellowship, other};
class grad_student : public student {
  public:
    grad_student(char* nm, int id, double g,
       year x, support t, char *d, char* th);
    void print() const;
  protected:
    support s; char dept[10]; char thesis[80];
};

student::student(char* nm, int id, double g,
     year x): student_id(id), gpa(g), y(x) {
  stcpy(name, nm);
}

grad_student::grad_student(char* nm, int id,
   double g,year x,support t,char *d,char* th):
   student(nm,id,g,x), s(t) {
  strcpy(dept,d); strcpy(thesis, th);
}

student:: print() const {
  cout << "\n" << name << ", " << studnet_id
       << ", " << y << gpa << endl;
}

grad_student::print() const {
  student::print();
  cout << dept << ", " << s << "\n" << thesis
       << endl ;
}
```

```
// Test pointer conversion rule

main() {
  student s("Nang", 821102,
            3.412, fresh),
          *ps = &s ;
  grad_studnet gs("Kim", 811102,
    2.523, grad, ta, "Computer",
    "on PC"), *pgs;

   // student::print
   ps->print() ;

   ps = pgs = &gs;

   // student::print
   // static mgs binding
   ps->print();


   // grad_student::print
   pgs->print();
   return(0);
}
```

a pointer whose type is pointer to base class can point to objects having the derived class type

parent의 class의 Pointer는
subclass의 object를 가리킬 수 있다

b를 만들 때 a의 어떤 constructor를 실행시킬지 알려줄 수 있다

superclass의 포인터가 자길 inherit하는 객체를 포인팅해도 되지만

subclass의 포인터가 superclass를 포인팅하면 안됨

static message to method binding - 컴파일 타임
ps가 gs객체를 포인팅하고 있지만 컴파일 타임 때 ps는 student class 타입에 바인딩되어
s객체의 print를 한다

multiple inheritance의 문제 - 양쪽에서 받을 때 어떤걸 선택할 건가

```
// Multiple Inheritance
class tools{
  public:
    tools(char*);
    int cost();
    ...
};

class parts {
  public:
    parts(char*);
    int cost();
    ..
};
class labor {
  public:
    labor(char*);
    int cost();
    ...
};

class plans: public tools, public parts, public labor {
  public:
    plans(int m): tools("lathe"), parts("widget"), labor(m), a(m) {
      ...
    }
  tot_cost() {return (parts::cost() + labor::cost());}
  ...
}
```

특별한 케이스에는 dynamic message to method를 서포트해줌

- **Polymorphism**
  - ***virtual function*** **allows run-time selection from a group of functions overridden within a type hierarchy (dynamic msg binding)**
  - **abstract base class**

superclass에서 virtual로 선언된 함수를 subclass에서 override하고 pointer를 통해서 호출할 때

```
// virtual function selection
#include <iostream.h>

class B {
  public:
    int i;
    virtual void print_i() { cout << i << "inside B\n";}
};
class D : public B {
  public:
    void print_i() {cout << i << "inside D\n";}
};
main() {
  B b;
  B* pb = &b;
  D f;
  f.i = 1 + (b.i = 1);
  pb->print_i()    // call B::print_i()
  pb = &f;         // points at a B object
  pb->print_i();   // call D::print_i();
  return(0);
}
```

superclass pointer는 하위 클래스를 가리킬 수 있음

b.print_i()는 B를 실행

output

```
1 inside B
2 inside D
```

b.i =1

pb->print_i()는 B class의 함수가 호출 됨

B int i ———————b

pb

근데 pb가 f를 포인팅하면, B의 print_i()가 virtual로 선언되어 있고 D에서 print_i를 override할 때
pb가 f를 포인팅하면 D의 함수가 실행됨

만약 virtual로 선언 안돼있으면 override안되고 pb가 f를 가리켜도 superclass인 B가 실행됨

D int i ———————f

f.i = 2

```
// shape is an abstract base class
class shape {
  public:
    virtual double area()=0;
};

class rectangle: public shape {
  public:
    rectangle(double h,double w):
                        height(h),width(w) {}
    double area() { return(height*width);}
  private:
    double height, width;
}
class circle : public shape {
  public:
    circle(double r): radius(r) { }
    double area() {return(3.14*radius*radius);}
  private:
    double radius;
}
class square : public rectangle {
  public:
    square(double h) : rectagle(h,h) { }
    double area() { return(rectangle::area());}
}
```

만약 이것을
정의하지 않으면??

body가 null인 class
abstract base class

superclass에 subclass assign 가능

```
b = a ;
b.area();
```

컴파일에러 null이라

Client Code

```
...
shape *ptr_shape;
...
cout << "area =" <<
   ptr_shape->area();
...
```

superclass의 포인터

ptr_shape = &a

circle의 area가 선언됨
superclass의 virtual함수를 override

Client Code

```
shape* p[N];
...
for (int i=0;i<N;i++){
  tot_area +=
        p[i]->area();
}
```

rect, circle, square 등이 실행됨

the client code does not need to change if new shapes are added to the system

만약 shape의 area 함수를 정의하지 않으면
밑에 subclass에서 area를 overriding하는게 컴파일 에러
java라면 실제 포인팅하는 걸 쫓아가서 실행(항상 dynamic message binding)

• Overloading, Overriding, and Dynamic Method Binding

```
class B {
  public:
    virtual foo(int);
    virtual foo(double);
};

class D : public B {
  public:
    foo(int);
};

main() {

  D  d;
  B  b, *pb = &d ;

  b.foo(9);     // B::foo(int)
  b.foo(9.5);   // B::foo(double)
  d.foo(9);     // D::foo(int)
  d.foo(9.5);   // D::foo(int)
  pb->foo(9);   // D::foo(int)
  pb->foo(9.5);  // B::foo(double)
```

the declaration of an identifier in a scope hides all declarations of that identifiers in outer scopes (static scoping rule), and a base class is outer scope of any class drived from it.

d.B::foo(5.2) // B::foo(double)

d의 foo 가 overriding하는 것은 b의 foo(int)
그래서 b.foo(double)가 실행되는 건 아닌가? 생각했는데
c++에서 static scope의 coercion을 적용해서 9.5를 9로 만들어서 d.foo(int)가 실행됨

static scoping rule, and coercion

dynamic msg binding

D에 double을 인수로 받는 함수가 없으니 superclass B의 것을 실행

B foo(int), foo(double) ——————— b
pd
D foo(int) ——————— d
override

- **Templates**
  - provide *parametric polymorphism*
    - ⇔ allows the same code to be used with respect to *different types*, where the type is a parameter of the code body
  - the template is used to generate different actual class when class T is substituted for with an actual type
  - a stack container class as a parameterized type

```
template <class LALA>
class stack {
  public:
    stack(int size=100):max_len(size) {
      s=new LALA[SIZE];top=EMPTY;}
    ~stack(){delete []s;}
    void reset { top = EMPTY;}
    void push(LALA c) {s[++top]=c;}
    void pop{return(s[top--]);}
    boolean empty() {
       return(boolean(top==EMPTY));}
    boolean full() {
       return{boolean(top==max_len-1);}
  private:
    enum {EMPTY = -1};
    LALA* s;
    int max_len;
    int top;
}
```

```
main() {

  // 100 element char stack
  stack<char> stk_ch ;

  // 200 element char* stack
  stack<char*> stk_str(200);

  // 100 element complex stack
  stack<complex> stk_cmplx(100);

  ...
}

reverse( char* str[], int n) {
  stack<char*> stk(n);
  for (int i=0;i<n;i++)
    stk.push(str[i]);
  for (i=0;i<n;i++)
    str[i] = stk.pop()
}
```

- **Exceptions**
  - the exception is handled by invoking the appropriate handler selected from a list of handlers found immediately after the handler's `try` block
  - an exception is raised by using the `throw` expression
  - not an extension for OOP

```
// stack constructor with exception
stack::stack(int n) {
  if (n<1) throw(n)
  p = new char[n];
  if (p == 0) throw("FREE STORE EXHAUSTED");
}

void g() {
  ...
  try {
    stack a(n), b(m) ;
    ...
  }
  catch(int n) { ... } // an incorrect size
  catch(char* err) { ... } //free store exhaustion
```

5. 오른쪽과 같은 C++프로그램을 이용하여 아래의 물음에 답하라. (15점)
(a) 이 프로그램에 error가 2개 있다 어느 부분이 왜 error인가 설명하라. (5점)
(b) Error 문정을 comment 처리한 후 프린트 되는 결과는 무엇인가 ? (5점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ? (5점)

```
#include <iostream.h>
class One {
     private: int i ;
     public:
       One() { i = 4;}
       virtual int value() {
           i = Two::value() + 1;
           return(i);}
} ;
class Two : public One {
        private: int i ;
        public:
           Two() { i = 5;}
           int value() {
              i = One::value() + 7;
              return(i);}
} ;
main() {
 Two x, *y ; One &a = x ; One *b ; One c;
 b = &x ;y = &c ; c = x;
 cout << "a.value():" << a.value() << "\n"
   << "b->value():" << b->value() << "\n"
   << "c.value():" << c.value()   << "\n";
}
```

superclass에서 subclass 함수를 호출함

subclass의 포인터 y가 superclass 객체 c를 포인팅하고 있음

x랑 a를 똑같이 만듦(alias)

5. 오른쪽과 같은 C++프로그램을 이용하여 아래의 물음에 답하라. (15점)
(a) 이 프로그램에 error가 2개 있다 어느 부분이 왜 error인가 설명하라. (5점)
(b) Error 문정을 comment 처리한 후 프린트 되는 결과는 무엇인가 ? (5점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ? (5점)

```
#include <iostream.h>
class One {
     private: int i ;
     public:
       One() { i = 4;}
       virtual int value() {
           i = Two::value() + 1;
           return(i);}
} ;
class Two : public One {
        private: int i ;
        public:
           Two() { i = 5;}
           int value() {
              i = One::value() + 7;
              return(i);}
} ;

main() {
 Two x, *y ; One &a = x ; One *b ; One c;
 b = &x ;y = &c; c = x ;
 cout << "a.value():" << a.value() << "\n"
   << "b->value():" << b->value() << "\n"
   << "c.value():" << c.value()   << "\n";
}
```

reference type 도 포인터와 같이 virtual로 선언된 superclass의 함수를 overriding 함

virtual로 선언된 superclass의 함수를 override해서 subclass에서 실행

x.value() ??

5. 오른쪽과 같은 C++프로그램을 이용하여 아래의 물음에 답하라. (15점)
(a) 이 프로그램에 error가 2개 있다 어느 부분이 왜 error인가 설명하라. (5점)
(b) Error 문정을 comment 처리한 후 프린트 되는 결과는 무엇인가 ? (5점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ? (5점)

```
#include <iostream.h>
class One {
     private: int i ;
     public:
       One() { i = 4;}
       virtual int value() {
           i = Two::value() + 1;
           return(i);}
} ;
class Two : public One {
        private: int i ;
        public:
           Two() { i = 5;}
           int value() {
              i = One::value() + 7;
              return(i);}
} ;
main() {
 Two x, *y ; One &a = x ; One *b ; One c;
 b = &x ;y = &c ; c = x ;
 cout << "a.value():" << a.value() << "\n"
   << "b->value():" << b->value() << "\n"
   << "c.value():" << c.value()   << "\n";
}
```

here

One c로 선언되어 One에 static message to method binding이 되었기 때문에 c=x로 할당해도 여전히 method는 One이 실행

(a) 이 프로그램에 error가 2개 있다 어느 부분이 왜 error인가 설명하라. (5점)
(정답)
1) `i = Two::value() + 1;  //*` One에서는 subclass인 Two가 보이지 않는다. (inner scope)
2) `y = &c  //` 서브클래스 포인터인 y가 superclass 객체인 c를 포인트하지 못한다.

(b) Error 문정을 comment 처리한 후 프린트되는 결과는 무엇인가 ? (5점)
(정답)

```
jhnang@cspro:~/testcode$ ./a.out
a.value():11
b->value():11
c.value():4
jhnang@cspro:~/testcode$
```

(b) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ? (5점)

virtual이 없으면 dynamic message to method binding이 안되어 One께 실행된다

```
jhnang@cspro:~/testcode$ ./a.out
a.value():4
b->value():4
c.value():4
jhnang@cspro:~/testcode$
```

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
           this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
  A a, *ap ; B b;
  ap = &b ;
  a.test() ;
  b.test() ;
  ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
            this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
   A a, *ap ; B b;
   ap = &b ;
   a.test() ;
   b.test() ;
   ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ?

a.test() ; ➔ ??

b.test() ; ➔ ??

ap->test() ; ➔ ??

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
            this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
   A a, *ap ; B b;
   ap = &b ;
   a.test() ;
   b.test() ;
   ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ?

a.test() ; ➔ 198

b.test() ; ➔ 187

ap->test() ; ➔ 187

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
            this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
   A a, *ap ; B b;
   ap = &b ;
   a.test() ;
   b.test() ;
   ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ?

a.test() ; ➔ 198

b.test() ; ➔ 187

ap->test() ; ➔ 187

a.aa(55) ➔ ??
b.aa(55) ➔ ??
a=b; b.aa(55) ➔ ??
ap->aa(55) ➔ ??

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
            this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
   A a, *ap ; B b;
   ap = &b ;
   a.test() ;
   b.test() ;
   ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)
(c) 만약 virtual 키워드를 없애면 프린트되는 결과는 무엇인가 ?

a.test() ; ➔ 198

b.test() ; ➔ 187

ap->test() ; ➔ 187

a.aa(55) ➔ 154
b.aa(55) ➔ 143
a=b; a.aa(55) ➔ 154
ap->aa(55) ➔ 143

7. 다음과 같은 C++ 프로그램에 대하여 아래의 물음에 답하라. (15점)

```
class A {
  private:
      int i ;
  public:
    A() { i = 99; }
    virtual test() {
            this->aa(i); }

    virtual aa(int k) {
       printf("%d", i+k);}
}
```

```
class B : public A {
  private:
   int i ;
  public:
   B() { i = 88;}
   aa(int k) {
     A::i = 19 ;
    printf("%d", i+k);}
}
```

```
main()
  A a, *ap ; B b;
  ap = &b ;
  a.test() ;
  b.test() ;
  ap->test() ;
}
```

(a) 이 프로그램은 error가 있다. 어느 문장이 왜 error인지 설명하라. (5점)
(b) 위의 error 문장을 comment처리하고 난 후에 이 프로그램에 의하여 print되는 결과를 써라. (10점)

(정답)

(a) A::i = 19 /* A 클래스에서 i는 private member이다. */ (5점)

(b) Class A : 198 (3점)
Class B : 187 (3점)
Class B : 187 (4점)