C++ and Objected Oritend Programming

Yung Yi

Ack

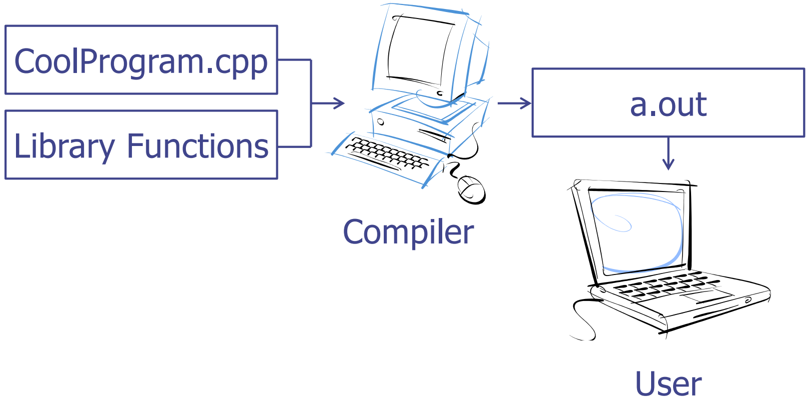
- Big Thanks
  - ✓ These slides are largely borrowed from Prof. Takgon Kim's Slides
- Also, reconfigured, restructured, and added by Prof. Yung Yi

Goals of This Lecture

- Overview of C++ language
  - ✓ At a glance, C++ = C + Class
- Intro to object-oriented (OO) programming
  - ✓ In structured programming, program = a series of functions
  - ✓ In OO programming, program = interaction between objects
  - ✓ OO encourages abstraction
    - Effective in representing a complex problem
  - ✓ OO encourages software reuse
    - Easily reuse classes and their implementation

Objected Oriented Programming

The C++ Programming Model



A Simple C++ Program

- Two integer inputs *x* and *y*
- Output their sum

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y and outputs their sum */
int main( ) {
    int x, y;
    std::cout << "please enter two numbers: "
    std::cin >> x >> y;                        // input x and y
    int sum = x + y;                           // compute their sum
    std::cout << "Their sum is " << sum << std::endl;
    return EXIT_SUCCESS                        // terminate successfully
}
```


Abstraction and Abstract Data Type

- Abstraction: depends on what to focus
  - ✓ Procedure abstraction: focuses on operations
  - ✓ Data abstraction: data + operations as one
  - ✓ Object abstraction: data abstraction + reusable sub types (class)
- Abstract data type (ADT)
  - ✓ Definition of a set of data + associated operations
- Implementation of ADT
  - ✓ Data → data structure
    - Stack, Queue, Tree etc.
  - ✓ Operations → manipulation of data structure
    - Stack: push, pop etc.
  - ✓ Error conditions associated with operations

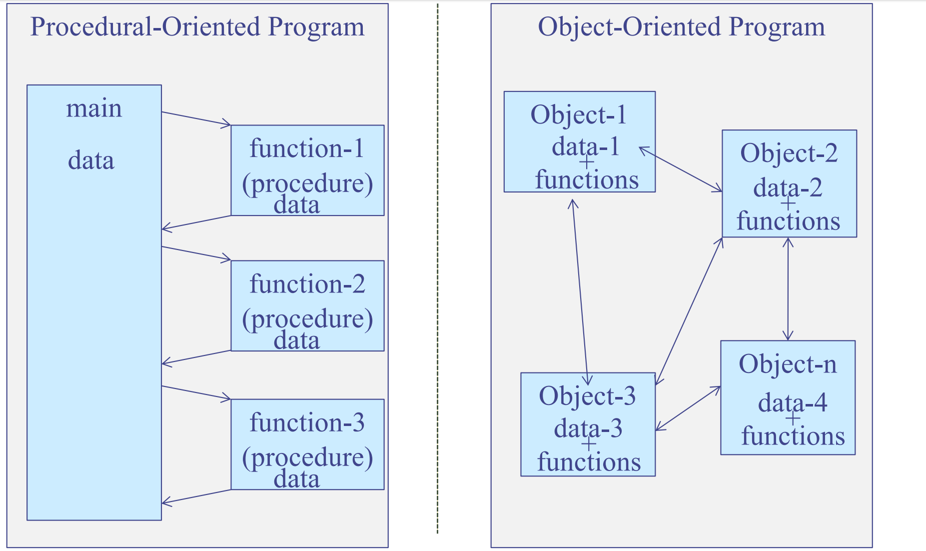
Example of ADT

- Example: ADT modeling a simple stock trading system
  - ✓ The data stored are buy/sell orders
  - ✓ The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - ✓ Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

C & C++ in Abstraction View

- C supports Procedure-Oriented programming
  - ✓ Procedure (function) + data structure
    - Procedure (function) : manipulate data
- C++ supports Object-Oriented programming
  - ✓ Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs.
  - ✓ Data abstract + reusable subtypes with following features
    - Encapsulation, Polymorphism, Inheritance

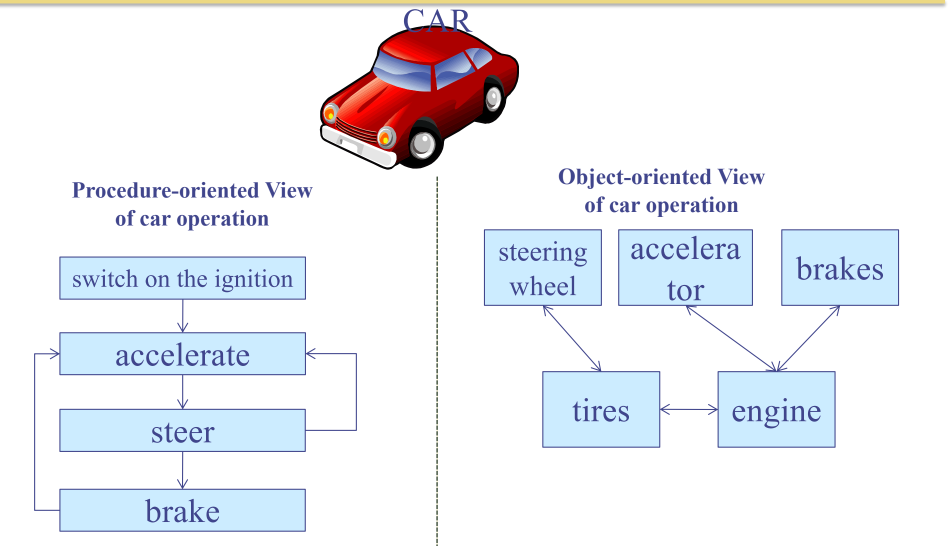
Procedural-Oriented VS. Object-Oriented



data is open to all functions.

Each data is hidden and associated with an object.

Example: PO VS. OO



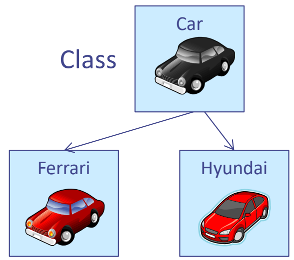
Car = a sequence of functions (procedures)

Car = interaction between components (objects)

What is Object ?

- Class (↔ Type in C)
  - ✓ Defines the abstract characteristics of a thing (object)
    - attributes (data) + behaviors (operations = methods)
- Object (↔ Variable in C)
  - ✓ A pattern (exemplar) of a class
- Instance
  - ✓ The actual object created at runtime
  - ✓ State: the set of values of the attributes of a particular object
- Methods
  - ✓ functions

Class



Object
(Instance at runtime)

Attributes
: color, capacity, max. speed, …

Methods
: accelerate, brake, steer left, steer right, …

C++ Classes

- Similar to structure in C

Class in C++

```
class class_name {
public:
   // member variables
   int a, b, c;
   …
   // member methods (functions)
   void print(void);
   …
};
```

a collection of types and associated functions

Structure in C

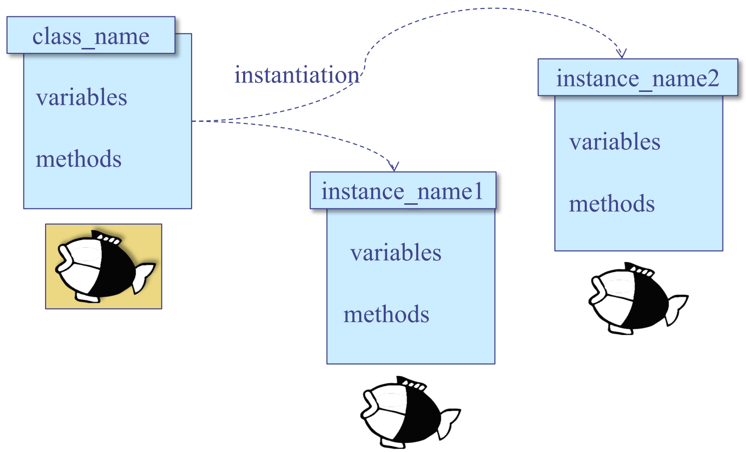
```
struct tag_name {
   type1  member1;
   type2  member2;
   …
   typeN memberN;
};
```

a collection of heterogeneous types

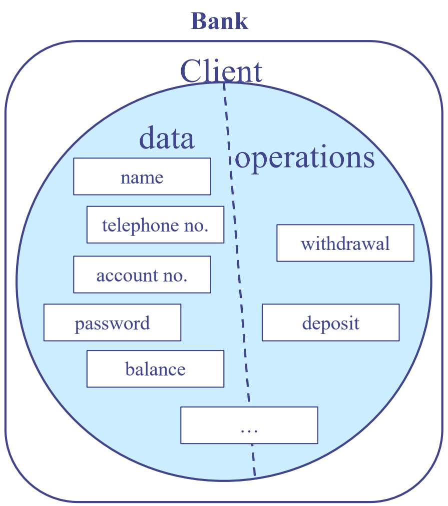
Class Declaration

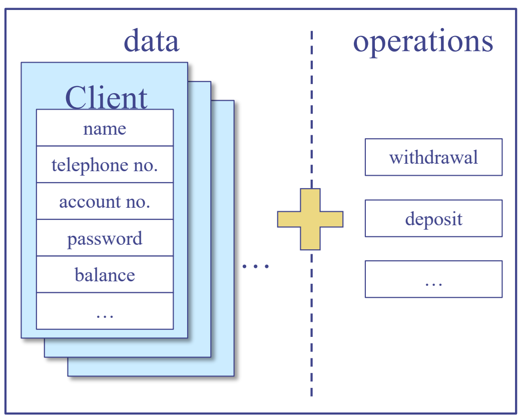
class_name *instance_name1, instance_name2;*

C.f. struct *tag_name struct_variable*, … ;

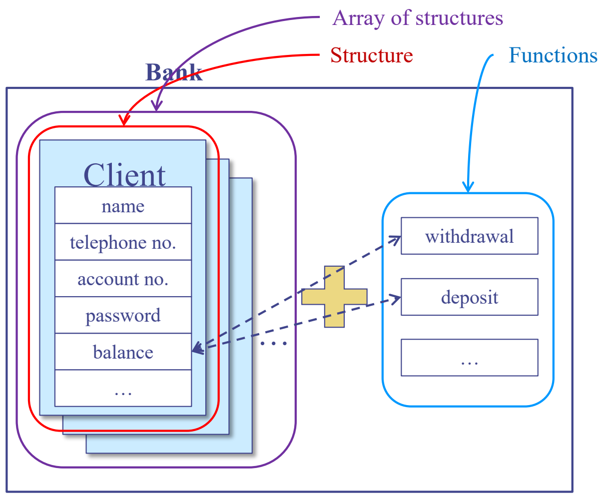


C Style Design (Procedural) (1/2)





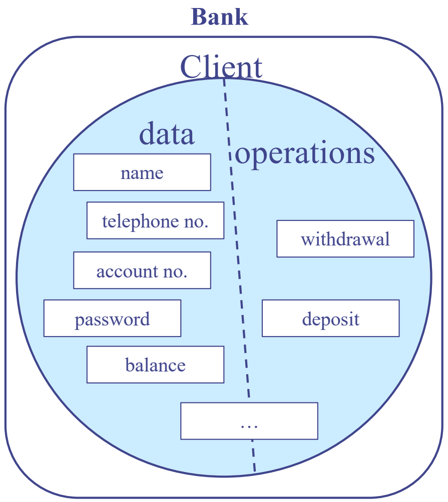
C Style Design (Procedural) (2/2)

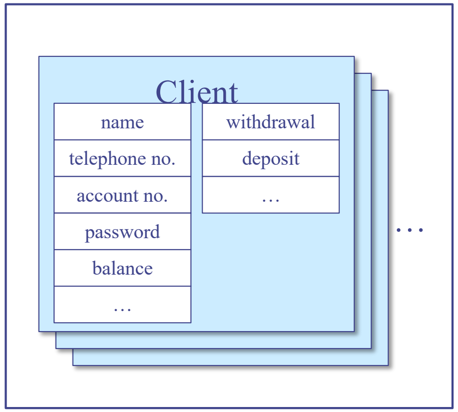



```
struct client {
  char name[MAX];
  char tel[MAX];
  char account[MAX];
  char password[MAX];
  int balance;
};

struct client clients[MAX_NO];

void withdrawal (client &cli, int money);
void deposit (client &cli, int money);
```

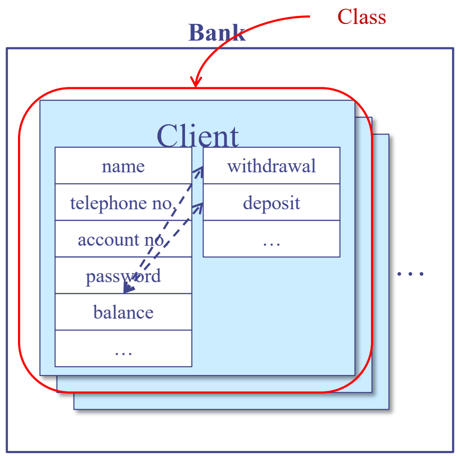
references

C++ Style Design (Object-Oriented) (1/2)





C++ Style Design (Object-Oriented) (2/2)




```
class client {
  char name[MAX];
  char tel[MAX];
  char account[MAX];
  char password[MAX];
  int balance;
  void withdrawal (int money);
  void deposit (int money);
};
```

member variables are not required

```
client clients[MAX_NO];
```

"struct" can be omitted in C++

In C++, structure is a class with all members public.
struct s { , , , } ≡ class s {public: , , , }

Example: Class

```
#include<iostream>
#define MAX 10
using namespace std;

class record{
public:
  char name[MAX];
  int course1, course2;
  double avg;
  void print(void) {
    cout << name << endl;
    cout << "course1 = " << course1
      << ", course2 = " << course2 << endl;
    cout << "avg = " << avg << endl;
  }
};
```

Access specifier

member variables

member function

```
int main( ) {
  record myrecord;
  myrecord.name = "KIM JH";
  myrecord.course1 = 100;
  myrecord.course2 = 90;
  int sum = myrecord.course1 +
            myrecord.course2;
  myrecord.avg = ((double) sum) / 2;
  myrecord.print( );
  return 0;
}
```

instantiation

referencing public member variables

member function call

```
result>
KIM JH
course1 = 100, course2 = 90
avg = 95
```


Definition of Member Functions

whole code in same file
ex) "record.cpp"

```
class record{
public:
  char name[MAX];
  int course1, course2;
  double avg;
  void print(void) {
    cout << name << endl;
    cout << "course1 = " << course1
      << ", course2 = " << course2
<< endl;
    cout << "avg = " << avg << endl;
  }
};
```

declaration & definition

"record.h"

```
class record{
public:
  char name[MAX];
  int course1, course2;
  double avg;
  void print(void);
};
```

declaration

definition:
always after declaration

"record.cpp"

```
void record::print(void) {
  cout << name << endl;
  cout << "course1 = " << course1
    << ", course2 = " << course2 << endl;
  cout << "avg = " << avg << endl;
}
```

- don't miss #include "record.h" in "record.cpp"

Member Variables & Functions

```
#include<iostream>
#define MAX 10
using namespace std;

class record{
public:
    char name[MAX];
    int course1, course2;
    double avg;
    void print(void) {
        cout << name << endl;
        cout << "course1 = " << course1
            << ", course2 = " << course2
<< endl;
        cout << "avg = " << avg << endl;
    }
};
```

always must reference member variables with instance name

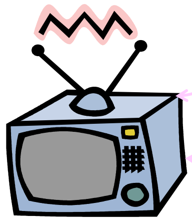
member function

```
int main( ) {
    record myrecord;
    myrecord.name = "KIM JH";
    myrecord.course1 = 100;
    myrecord.course2 = 90;
    int sum = myrecord.course1 +
                myrecord.course2;
    myrecord.avg = ((double) sum) / 2;
    myrecord.print( );
    return 0;
}
```

can reference member variables without class name inside member functions

Encapsulation

- Encapsulation conceals the functional details defined in a class from external world (clients).
  - ✓ Information hiding
    - By limiting access to member variables/functions from outside
  - ✓ Operation through interface
    - Allows access to member variables through interface
  - ✓ Separation of interface from implementation
    - Similar to Stack data type and implementation (Lecture 11)



no necessity for knowing inside

encapsulation

clients (users) only know interface.

Encapsulation in C++

Class in C++

```
class class_name {
public:
    int a, b, c;
    …
    void print(void);
    …
private:
    …
protected:
    …
} ;
```

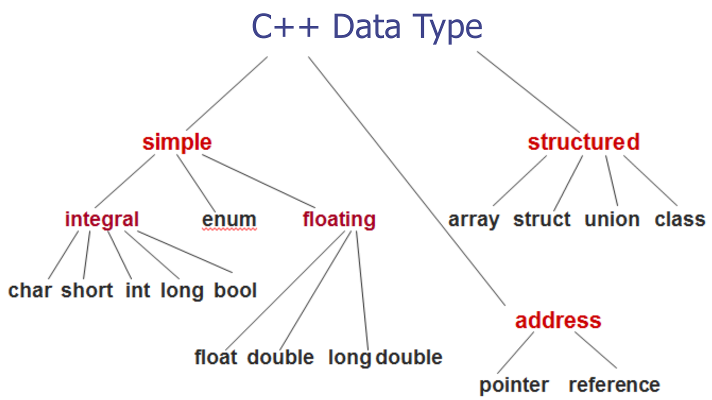
Access specifier

Interfaces : open outside

Encapsulation: hide inside

Basic Features (Mostly same as C)

C++ Data Types



Fundamental Types

- Basic data types
  - ✓ bool Boolean value, either true or false
  - ✓ char Character
  - ✓ short Short integer
  - ✓ int Integer
  - ✓ long Long integer
  - ✓ float Single-precision floating-point number
  - ✓ double Double-precision floating-point number
  - ✓ enum User-defined type, a set of discrete values
  - ✓ void The absence of any type information

Declaration of a Variable

- We can provide a definition, or initial value
- Without definition, initial value is zero
- Variable names may consist of any combination of letters, digits, or the underscore (_) character, but the first character cannot be digit
- ex)

```
short  n;
int    octalNumber = 0400;
char   newline_character = '\n';
long   BIGnumber = 314159265L;
short  _aSTRANGE__1234_variABlE_NaMe;
```


Characters: char

- Typically 8-bit
- Literal
  - ✓ A constant value appearing in a program
  - ✓ Enclosed in single quotes
  - ✓ A backslash (\ ) is used to specify a number of special character literals

'\n'	newline	'\t'	tab
'\b'	backspace	'\r'	return
'\0'	null	'\''	single quote
'\"'	double quote	'\\'	backslash

Integers: short, int, long

- Short int, (plain) int, long int
- Decimal numbers
  - ✓ ex) 0, 25, 98765, -3
- Suffix “l” or “L” indicate a long integer
  - ✓ ex) 123456789L
- Prefix “0” indicates octal constants
  - ✓ ex) 0400 (256)
- Prefix “0x” indicates hexadecimal constants
  - ✓ ex) 0x1c (28)

Floating Point: float, double

- Floating point literals
  - ✓ ex) 3.14159, -1234.567, 3.14E5, 1.28e-3
- Default is double type
- Suffix “f” or “F” indicate float
  - ✓ ex) 2.0f, 1.234e-3F

Enumerations: enum

- A user-defined type that can hold any of a set of discrete values
- Once defined, enumerations behave much like an integer type
- Each element of an enumeration is associated with an integer value
- ex)

```
enum Color {RED, GREEN, BLUE};   //RED=0, GREEN=1, BLUE=2
enum Mood {HAPPY=3, SAD=1, ANXIOUS=4, SLEEPY=2};

Color skycolor = BLUE;
Mood myMood = SLEEPY;
```


Pointers

- Pointer holds the value of an memory address
- The type T* denotes a pointer to a variable of type T
  - ✓ ex) int*, char*
- The ‘address-of’ operator, ‘&’, returns the address of a variable
- Dereferencing
  - ✓ Accessing the object addressed by a pointer
  - ✓ Done by * operator

Pointers

- ex)

```
char ch = 'Q';
char* p = &ch;         // p holds the address of ch
cout << *p;            // outputs the character 'Q'
ch = 'Z';              // ch now holds 'Z'
cout << *p;            // outputs the character 'Z'
```

- Null pointer points to nothing
- Void type pointer can point to a variable of any type
- Cannot declare a void type variable

Arrays

- A collection of elements of the same type
- Index references an element of the array
- Index is a number from 0 to N-1
- ex)

```
double  f[3];          // array of 3 doubles: f[0], f[1], f[2]
double* p[10];         // array of 10 double pointers: p[0], … , p[9]
f[2] = 25.3;
p[4] = &f[2];          // p[4] points to f[2]
cout << *p[4];         // outputs "25.3"
```


Arrays

- Two-dimensional array
  - ✓ An "array of arrays"
  - ✓ ex) int A[15][30]
- Initializing
  - ✓ ex)

```
int  a[4] = {10, 11, 12, 13};     // declares and initializes a[4]
bool b[2] = {false, true};        // declares and initialize b[2]
char c[] = {'c', 'a', 't'};       // declares and initialize c[3]
                                  // compiler figures the size of c[]
```


Pointers and Arrays

- The name of an array can be used as a pointer to the array's initial element and vice versa
- ex)

```
char c[] = {'c', 'a', 't'};
char *p = c;                       // p point to c[0]
char *q = &c[0];                   // q also points to c[0]
cout << c[2] << p[2] << q[2]       // outputs "ttt"
```

C-Style Structure

- Storing an aggregation of elements which can have different types
- These elements called "member" or "field", is referred to by a given name
- ex)

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string      name;           // possible value: "John Smith"
    MealType  mealPref;       // possible value: VEGETARIAN
    bool        isFreqFlyer;    // possible value: true
    string      freqFlyerNo;    // possible value: "293145"
};
```


C-Style Structure

- This defines a new type called Passenger
- Declaration and initialization
  - ✓ ex)

```
Passanger pass = { "John Smith", VEGETARIAN, true, "293145" }
```

- Member selection operator
  - ✓ struct_name.member
  - ✓ ex)

```
pass.name = "Pocahontas";         // change name
pass.mealPref = REGULAR;          // change meal preference
```

- This is just for backward-compatibility
- ``Class" is much more powerful

References

- An alternative name for an object (i.e., alias)
- The type T& denotes a reference to an object of type T
- Cannot be NULL
- ex)

```
string author = "Samuel Clemens";
string &penName = author;       // penName is an alias for author
penName = "Mark Twain";         // now author = "Mark Twain"
cout << author;                 // outputs "Mark Twain"
```


Constants

- Adding the keyword const to a declaration
- The value of the associated object cannot be changed
- ex)

```
const double PI = 3.14159265;
const int CUT_OFF[] = {90, 80, 70, 60};
const int N_DAYS = 7;
const int N_HOURS = 24*N_DAYS;          // using a constant expression
int counter[N_HOURS];                   // constant used for array size
```

- Replace "#define" in C for the definition of constants

Typedef

- Define a new type name with keyword typedef
- ex)

```
typedef char* BufferPtr;        // type BufferPtr is a pointer to char
typedef double Coordinate;      // type Coordinate is a double

BufferPtr p;                    // p is a pointer to char
Coordinate x, y;                // x and y are of type double
```


Dynamic Memory Allocation

Dynamic Memory and ‘new’ Operator

- Create objects dynamically in the ‘free store’
- The operator ‘new’ dynamically allocates the memory from the free store and returns a pointer to this object
- Accessing members
  - ✓ pointer_name->member
  - ✓ (*pointer_name).member
  - ✓ Same as how to access a member in C Struture
- The operator ‘delete’ operator destroys the object and returns its space to the free store

Dynamic Memory and ‘new’ Operator

- ex)

```
Passenger *p;
//...
p = new Passenger;                // p points to the new Passenger
p->name = “Pocahontas”;           // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = “NONE”;
//...
delete p;                         // destroy the object p points to
```

Example: Operators for Dynamic Allocation

C

Functions

void * malloc (size_t *size*)
void * calloc (size_t *nmemb*, size_t *size*)
void free(void **ptr*);

C++

Operators

new *data_type*
new *data_type*[*size*]
delete *scalar_variable*;
delete [] *array_variable*;

returns a pointer addressing the 1st element of the array

Ex) To allocate a char

C

```
char *cptr;
cptr = (char *) malloc(sizeof(char));
…
free(cptr);
```

C++

```
char *cptr = new char;
…
delete cptr;
```

Ex) To allocate an integer array of 100 elements

C

```
int *iptr;
iptr = (int *) calloc(100, sizeof(int));
…
free(iptr);
```

C++

```
int *iptr = new int[100];
…
delete [] iptr;
```


Questions

- How to dynamically allocate "array of pointers"?
- How to declare two dimensional matrix (i.e., matrix) and dynamically allocate its space?
- You can use your own method, but you can also use 'vector' class in STL library

Memory Leaks

- C++ does not provide automatic garbage collection
- If an object is allocated with new, it should eventually be deallocated with delete
- Deallocation failure can cause inaccessible objects in dynamic memory, memory leak

Strings in C++

Strings

- C-style strings
  - ✓ A fixed-length array of characters that ends with the null character
  - ✓ This representation alone does not provide many string operations (concatenation, comparison,...)
- STL strings
  - ✓ C++ provides a string type as part of its “Standard Template Library” (STL)
  - ✓ Should include the header file “<string>”

- STL: Standard Template Library
  - ✓ Collection of useful, standard classes and libraries in C++

STL Strings

- Full name of string type is “std::string”
  - ✓ We can omit the “std::” prefix by using the statement “using std::string” (see “namespaces” later)
- Features
  - ✓ Concatenated using + operator
  - ✓ Compared using dictionary order
  - ✓ Input using >> operator
  - ✓ Output using << operator

C	C++
array of char types	string class
library functions	member functions of string class
relatively difficult, but many sources	easy

STL Strings

- ex)

```
#include <string>
using std::string;
//...
string s = "to be";
string t = "not " + s;              // t = "not to be"
string u = s + " or " + t;          // u = "to be or not to be"
if (s > t)                          // true: "to be" > "not to be"
    cout << u;                      // outputs "to be or not to be"
```


STL Strings

- Appending one string to another using += operator
- Indexed like arrays
- The number of characters in a string s is given by s.size()
- Conversed to C-style string by s.c_str() which returns a pointer to a C-style string

STL Strings

- ex)

s = "John";	// s = "John"
int i = s.size();	// i = 4
char c = s[3];	// c = 'n'
s += " Smith";	// s = "John Smith"
char *p = s.c_str();	// p is a C-style string

- Other C++ STL operations are providing
  - ✓ ex) extracting, searching, replacing,...

C Style String to C++

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

main() {
  char cstyle[] = "KKIST";
  string cppstyle;

  cppstyle = cstyle;

  cppstyle[1] = 'A';

  cout << "cstyle = " << cstyle << endl;
  cout << "cppstyle = " << cppstyle << endl;
}
```

Result>
cstyle = KKIST
cppstyle = KAIST

C++ Style String to C (1/2)

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

main() {
  string cppstyle = "KAIST";
  const char *cstyle;

  cstyle = cppstyle.c_str();

  cout << "cstyle = " << cstyle << "\n";
  cout << "cppstyle = " << cppstyle << "\n";
}
```

return value : const char *
∴cannot modify a string

Result>
cstyle = KAIST
cppstyle = KAIST

C++ Style String to C (2/2)

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

main() {
  string cppstyle = "KKIST";
  char* cstyle = new char [ cppstyle.size() + 1];

  strcpy( cstyle, cppstyle.c_str() );

  cstyle[1] = 'A';

  cout << "cppstyle = " << cppstyle << "\n";
  cout << "cstyle = " << cstyle << "\n";

  delete[] cstyle;
}
```

can modify a string

Result>
cppstyle = KKIST
cstyle = KAIST

Scope, Namespace, Casting, Control Flow

Local and Global Variables

- Block
  - ✓ Enclosed statements in {...} define a block
  - ✓ Can be nested within other block
- Local variables are declared within a block and are only accessible from within the block
- Global variables are declared outside of any block and are accessible from everywhere
- Local variable hides any global variables of the same name

Local and Global Variables

- ex)

```
const int cat = 1;              // global cat

int main () {
    const int cat = 2;          // this cat is local to main
    cout << cat;                // outputs 2 (local cat)
    return EXIT_SUCCESS;
}
int dog = cat;                  // dog = 1 (from the global cat)
```


Scope Resolution Operator (::)

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

int  x;

int main()
{
      int  x;
      x = 1;
      ::x = 2;

      cout << "local x = " << x << endl;
      cout << "global x = " << ::x << endl;

      return 0;
}
```

local x hides global x

assign to global x

result>
local x = 1
global x = 2

Namespaces: Motivation

- Two companies A and B are working together to build a game software "YungYung"
- A uses a global variable
  - ✓ struct Tree {};
- B uses a global variable
  - ✓ int Tree:
- Compile? Failure
- Solution
  - ✓ A: struct Atree {}; B: int BTree; → dirty, time consuming, inconvenient
- Let's define some "name space"
- Very convenient in making "large" software

Namespaces

- A mechanism that allows a group of related names to be defined in one place
- Access an object x in namespace group using the notation group::x, which is called its fully qualified name
- ex)

```
namespace myglobals {
    int cat;
    string dog = "bow wow";
}

myglobals::cat = 1;
```


The Using Statement

- Using statement makes some or all of the names from the namespace accessible, without explicitly providing the specifier
- ex)

```
using std::string;            // makes just std::string accessible
using std::cout;              // makes just std::cout accessible

using namespace myglobals;    // makes all of myglobals accessible
```


Example : Namespace

```
#include <iostream>
namespace IntSpace{
    int data;
    void add(int n){ data += n; }
    void print(){ std::cout << data << std::endl; }
}
namespace DoubleSpace{
    double data;
    void add(double n){ data += n; }
    void print(){ std::cout << data << std::endl; }
}
int main()
{
    IntSpace::data = 3;
    DoubleSpace::data = 2.5;
    IntSpace::add(2);
    DoubleSpace::add(3.2);
    IntSpace::print();
    DoubleSpace::print();
    return 0;
}
```

same variable name is allowed in different namespaces

```
result>
5
5.7
```

Traditional C-Style Casting

```
int     cat = 14;
double  dog = (double) cat;        // traditional C-style cast
double  pig = double(cat);// C++ functional cast


int     i1 = 18;
int     i2 = 16;
double  dv1 = i1 / i2;                    // dv1 = 1.0
double  dv2 = double(i1) / double(i2);    // dv2 = 1.125
double  dv3 = double( i1 / i2);           // dv3 = 1.0
```


Static Casting (to give “warning”)

```
double  d1 = 3.2;
double  d2 = 3.9999;
int     i1 = static_cast<int>(d1);   // i1 = 3
int     i2 = static_cast<int>(d2);   // i2 = 3
```


Implicit Casting

```
int     i = 3;
double  d = 4.8;
double  d3 = i / d;        // d3 = 0.625 = double(i) / d
int     i3 = d3;           // i3 = 0 = int(d3)
               // Warning! Assignment may lose information
```


Control Flow: If Statement

```
if (<boolean_exp>)
        <true_statement>
[else if (<boolean_exp>)
        <else_if_statement>]
[else
        <else_statement>]
```

Control Flow: Switch Statement

```
char      command;
cin >> command;

switch (command) {
    case 'I' :
        editInsert();
        break;
    case 'D' :
        editDelete();
        break;

    case 'R' :
        editReplace();
        break;
    default :
        cout << "Error\n";
        break;
}
```


Control Flow: While & DO-While

```
while (<boolean_exp>)
        <loop_body_statement>


do
        <loop_body_statement>
while (<boolean_exp>)
```


Control Flow: For Loop

```
for ([<initialization>];[<condition>];[<increment>])
        <body_statement>
```


Functions, Overloading, Inline function

Functions

```
bool evenSum (int a[], int n); // function declaration
int main() {
        const int listLength = 6;
        int list[listLength] = {4, 2, 7, 8, 5, 6};
        bool result = evenSum(list, listLength);  // call the function
        if (result) cout << "even sum.\n";
        else         cout << "odd sum.\n";
        return EXIT_SUCCESS;
}
bool evenSum (int a[], int n){ //function definition
        int sum = 0;
        for (int i = 0; i < n; i++) sum += a[i];
        return (sum %2) == 0;
}
```


Function Overloading

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

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

In C, you can't use the same name for different functions

C++ allows multiple functions with the same name: the right function is determined at runtime based on argument types

Function Overloading

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

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

Type matching

Type matching

In C, you can't use the same name for multiple function definitions

C++ allows multiple functions with the same name as long as argument types are different: the right function is determined at runtime based on argument types

Polymorphism

- Allow values of different data types to be handled using *a uniform interface*.
- One function name, various data types
  - ✓ Function overloading
- Merit
  - ✓ improve code readability
- Ex.

C	abs ( )	labs ( )	fabs ( )
	int	long int	floating point
C++	abs ( )		
	int	long int	floating point

Resolving an Overloaded Function Call

Precedence for function calls using arg type

1. An exact match
2. A match through promotion
3. A match through application of a type conversion

Implicit type conversion by widening

(char → short → int → long → float → double)

Implicit type conversion by narrowing
+ Explicit type conversion

```
void WhichOne ( float f );     // exact match
void WhichOne ( double d );   // promotion
void WhichOne ( int c );        // type conversion
int main( ) {
    WhichOne (3.5f );
    return 0;
}
```

Type Casting in C++

In C, *(type_name) expression*
In C++,
(i) the same as in C or
(ii) *type_name* may be used as if function name with argument *expression*.
ex: (int) 1.5 → int (1.5)   is ok in C++.

Default Arguments (1/2)

default values

```
#include<iostream>
using namespace std;
int calcCubeVolume(int width = 1, int height = 1, int depth = 1);

int main ( ) {
  cout << "[def, def, def] " << calcCubeVolume( ) << endl;

  cout << "[2, def, def] " << calcCubeVolume(2) << endl;

  cout << "[2, 2, def] " << calcCubeVolume(2, 2) << endl;

  cout << "[2, 2, 2] " << calcCubeVolume(2, 2, 2) << endl;
  return 0;
}

int calcCubeVolume(int width, int height, int depth) {
  return (width * height * depth);
}
```

All default values are used.

First arg. overrides the default value.

First two args. overrides the default values.

All args. overrides the default values.

result>
[def, def, def] 1
[2, def, def] 2
[2, 2, def] 4
[2, 2, 2] 8

Default Arguments (2/2)

Default arguments may be provided for trailing arguments only.

```
int calcCubeVolume(int width = 1, int height = 1, int depth = 1);   ( O )
int calcCubeVolume(int width, int height = 1, int depth = 1);        ( O )
int calcCubeVolume(int width, int height, int depth = 1);             ( O )
int calcCubeVolume(int width = 1, int height = 1, int depth);        ( X )
int calcCubeVolume(int width = 1, int height, int depth = 1);        ( X )
```

```
int calcCubeVolume(int = 1, int = 1, int = 1);
```

Argument names can be omitted in prototype.

Default Args vs. Function Overloading

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

int calcCubeVolume(int width = 1, int height = 1, int depth = 1) {
  return (width * height * depth);
}

void calcCubeVolume( ) {
  cout << "No argument!" << endl;
}

int main ( ) {
  cout << "[def, def, def] " << calcCubeVolume() << endl;
  cout << "[2, def, def] " << calcCubeVolume(2) << endl;
  cout << "[2, 2, def] " << calcCubeVolume(2, 2) << endl;
  cout << "[2, 2, 2] " << calcCubeVolume(2, 2, 2) << endl;
  return 0;
}
```

Function overloading

Which function ? → Ambiguous

ERROR!!

C++ Operator overloading

- User can overload operators for a user-defined class or types
  - Example) String s1(“yi”); String s2(“yung”); String s = s1+s2;
  - define an operator as a function to overload an existing one
  - operator followed by an operator symbol to be defined.
    - define an operator + → operator+
    - define an operator ++ → operator++
    - define an operator << → operator <<
  - To avoid confusion with built-in definition of overload operators, all operands in the basic types (int, long, float) are not allowed

Example : Operator Overloading

```
#include <iostream>
using namespace std;
enum Day { sun, mon, tue, wed, thu, fri,sat };
                                  Operator overloading
Day& operator++(Day& d)
{
  return d = (sat == d) ? sun : Day(d+1);
}
void print(Day d){
  switch(d){
   case sun : cout << "sun\n"; break;
   case mon : cout << "mon\n"; break;
   case tue : cout << "tue\n"; break;
   case wed : cout << "wed\n"; break;
   case thu : cout << "thu\n"; break;
   case fri : cout << "fri\n"; break;
   case sat : cout << "sat\n"; break;
  }
}
```

```
int main()
{
  Day d = tue;
  cout << "current : ";
  print(d);
  for(int i = 0; i < 6; i++){
      ++d;   use of overloaded operator
  }
  cout << "after 6 days : ";
  print(d);
  return 0;
}
```

result>

current : tue

after 6 days : mon

Operator Overloading

```
Passenger yung, beyonce;
…
…

if (yung == beyonce)
{
…
}
```

```
bool operator == (const Passenger &x, const Passenger &y) {
        return        x.name          == y.name
                &&    x.mealPref      == y.mealPref
                &&    x.isFreqFlyer   == y.isFreqFlyer
                &&    x.FreqFlyerNo   == y.FreqFlyerNo;
}
```


Using Overloading

```
Passenger yung, beyonce;

cout << yung;        → cout = function_<<(cout, yung)
cout << beyonce;
```

```
ostream& operator << (ostream &out, const Passenger &pass) {
        out << pass.name << “ ” << pass.mealPref;
        if (pass.isFreqFlyer) {
                out << “ ” << pass.freqFlyerNo;
        }
        return out;
}
```

Inline Functions

C (Macro functions)

```
#include <stdio.h>
#define square(i)  i*i
#define square2(i) ((i)*(i))
#define  pr(i)  printf("value
= %d\n", (i))

main( ) {
  int i = 1, j = 1, k;
  k = square(i+1); pr(k);
  k = square2(j+1); pr(k);
  k = 100/square(2); pr(k);
  k = 100/square2(2); pr(k);
}
```

i+1*i+1

100/2*2

Side effect of macro functions

result>
value = 3 // wrong answer
value = 4
value = 100 // wrong answer
value = 25

C++ (Inline functions)

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

inline int square(int i) { return i*i; }
inline void pr(int i) { cout << "value = "
<< i << endl; }

main( ) {
  int i = 1, j = 1, k;
    k = square(i+1); pr(k);
    k = 100/square(2); pr(k);
}
```

Function body is expanded at the point of function call during compile-time.

Similar to macro function

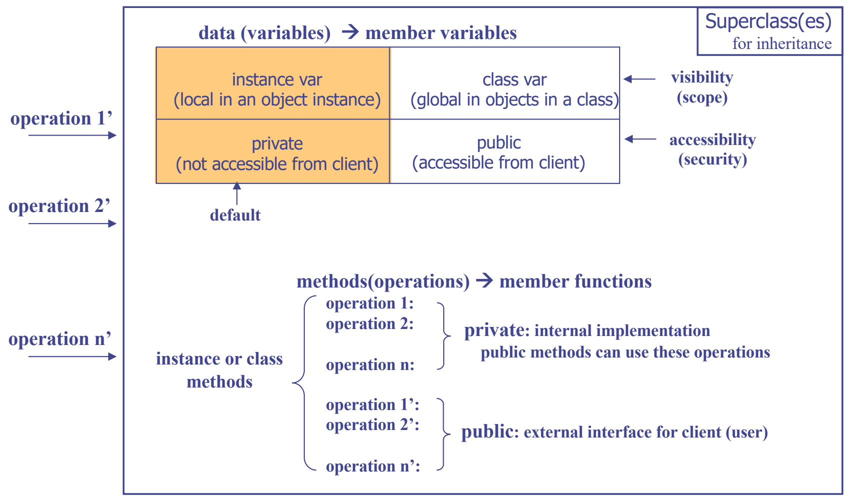
No side effect

result>
value = 4
value = 25

More on OOP and Class

Constructor and Destructor

Class Structure in General Form



Constructors

- A special, user-defined member function defined within class
  - Initializes member variables with or without arguments
- The function is invoked implicitly by the compiler whenever a class object is defined or allocated through operator *new*

```
class record {
 public:
  char name[MAX];
 private:
  int course1, course2;
  double avg;
 public:
  record ( ) {
   strcpy(name, "");
   course1 = course2 = 100;
   avg = 100;
  }
  void print(void);
};
```

same name as class

always in “public” to be used by all users for this class

must not specify a return type

Constructor

```
class record {
 public:
  char name[MAX];
 private:
  int course1, course2;
  double avg;
 public:
  record ( );
  void print(void);
};
record::record ( ) {
 strcpy(name, "");
 course1 = course2 = 100;
 avg = 100;
}
```


Default Constructor with No Argument

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
 public:
  char name[MAX];
 private:
  int course1, course2;
  double avg;
 public:
  record( );
  void print(void);
};

void record::print(void)
{ … }
```

```
record::record( ) {
 strcpy(name, "");
 course1 = course2 = 100;
 avg = 100;
}

int main( ) {
 record myRecord =
record::record();
 record hisRecord = record( );
 record herRecord;

 myRecord.print( );
 hisRecord.print( );
 herRecord.print( );
 return 0;
}
```

Same initializations

∵implicitly called without supplying an argument → Default constructor

result>

course1 = 100, course2 = 100
avg = 100

course1 = 100, course2 = 100
avg = 100

course1 = 100, course2 = 100
avg = 100

Constructors with Arguments

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
 public:
  char name[MAX];
 private:
  int course1,
course2;
  double avg;
 public:
  record();
  record(char*, int);
  record(char*, int,
int);
  void print(void);
};
```

overloading

```
record::record() {
 strcpy(name, "");
 course1 = course2 = 100;
 avg = 100;
}

record::record(char *str, int score) {
 strcpy(name, str);
 course1 = course2 = score;
 avg = score;
}

record::record(char *str, int score1, int
score2) {
 strcpy(name, str);
 course1 = score1; course2 = score2;
 avg = ((double) (course1 + course2)) /
2.0;
}
```

```
void record::print(void) { … }

int main( ) {
 record myRecord;
 record yourRecord = record("KIM", 80,
100);
 record hisRecord("LEE", 70);

 myRecord.print( );
 yourRecord.print( );
 hisRecord.print( );

 return 0;
}
```

shorthand notation

same as
record hisRecord = record(“LEE”, 70);

Destructors

- A special, user-defined class member function defined in class
- The function is invoked whenever an object of its class goes out of scope or operator *delete* is applied to a class pointer

```
class record {
 public:
  char name[MAX];
 private:
  int course1, course2;
  double avg;
 public:
  record ( ) { … }
  ~record ( ) {
   …
  }
  void print(void);
};
```

always in “public”

must not specify a return type

Destructor

the tag name of the class prefixed with a tilde (“~”)

```
int main( ) {
 record myRecord;
 …
 return 0;
}
```

record::~record( ) invoked for myRecord

Initialization, static, “this”

Initialization Style: Vars vs. Class Objects

C

```
#include<stdio.h>

int main() {
  int i = 10;
  char ch = 'a';
  printf(“%d”, i);
  printf(“%c”, ch);
  return 0;
}
```

result>
10a

C++

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

int main() {
  int i(10);
  char ch('a');
  cout << i;
  cout << ch;
  return 0;
}
```

int is a class and i is an object
char is a class and ch is an object.
Initialization at construction of objects

Initialization of Class Objects as Members

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
 public:
  int id;
  int score;
  record(int i = 0, int s = 100);
  void print(void);
};

record::record(int i, int s)
: id(i), score(s)
{
}

void record::print(void) {
  cout << id;
  cout << " : " << score << endl;
}

int main( ) {
  record myRecord(20090001, 70);
  myRecord.print( );
  return 0;
}
```

Constructor
1. Member initialization
2. Assignment

Members, id and score, are objects of class int
→ Initialization by calling constructor for class int and create objects id and score

Assignments

result>
20090001 : 70

C.f.

```
record::record(int i, int s)
{
  id = i; score = s;
}
```

Implicit initialization of class objects by constructor for int

Global Variable

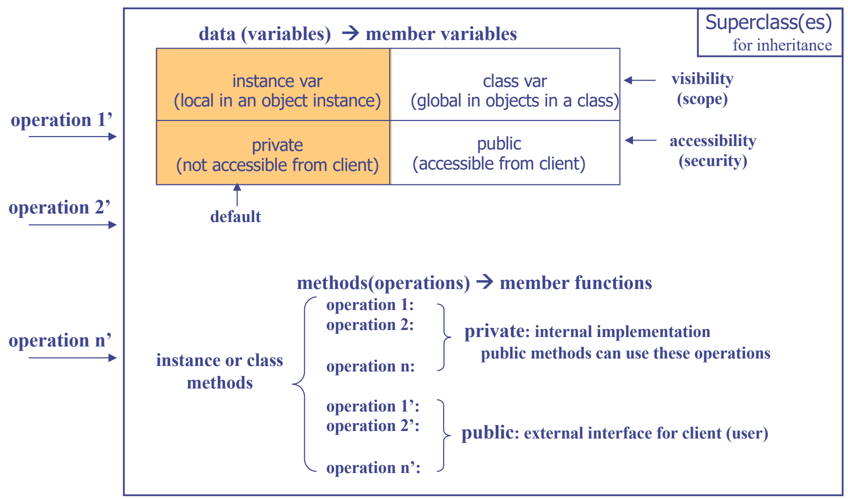
```
#include <iostream>
using namespace::std;

int count = 1;

class student{
    char name[20];
    int age;
public:
    student(char* _name, int _age){
        strcpy(name, _name);
        age = _age;
        cout << count++ << "th student" << endl;
    }
};

int main()
{
    student s1("Kim", 20);
    student s2("Seo", 28);

    return 0;
}
```

Global Variable

result>
1th student
2th student

- Global variables
  - Undesirable in Object-Oriented concept
  - All functions can access global variables
  - → Error-prone, hard to debug, etc.

Recall: Class Structure in General Form



Static: Per-class variable

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

class student{
    char name[20];
    int age;
    static int count;
public:
    student(const char* _name, int _age){
        strcpy(name, _name);
        age = _age;
        cout << count++ << "th student" << endl;
    }
};

int student::count = 1;

int main()
{
    student s1("Kim", 20);
    student s2("Seo", 28);

    return 0;
}
```

Static Member Variable = global to all objects created from the student class

Initialization at outside the class definition

result>
1th student
2th student

The Pointer *this*

- Reserved keyword
- Inside a member function, how can we access "my object itself"?
- The address of the class object through which the member function has been invoked

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

class Pointer{
public:
    Pointer* GetThis(){
        return this;
    }
};

int main()
{
    Pointer p1;
    Pointer p2;
    cout << "Object p1"       << endl;
    cout << "Address of p1: " << &p1 << endl;
    cout << "this of p1: "     << p1.GetThis() << endl;
    cout << "Object p2"       << endl;
    cout << "Address of p2: " << &p2 << endl;
    cout << "this of p2: "     << p2.GetThis() << endl;
    return 0;
}
```

result>
Object p1
Address of p1 : 0012FED7
this of p1 : 0012FED7
Object p2
Address of p2 : 0012FECB
this of p2 : 0012FECB

Example: *this* Pointer (1/2)

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

class point {
   int x, y;
  public:
   point(int a = 0, int b = 0);
   void set(int a, int b);
   void print();
};

point::point(int a, int b) {
  this->set(a, b);
}

void point::set(int a, int b) {
  this->x = a; this->y = b;
}

void point::print() {
  cout << "[" << this;
  cout << "] " << this->x;
  cout << ", " << this->y << endl;
}

int main() {
  point p(1, 1);
  p.set(2, 2);
  p.print();
  return 0;
}
```

result>
[0xbfec6f00] 2, 2

Example: *this* Pointer (2/2)

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

class point {
   int x, y;
  public:
   point(int a = 0, int b = 0);
   void set(int x, int y);
   void print();
};

point::point(int a, int b) {
  x = a; y = b;
}
```

```
void point::set(int x, int y) {
  x = x; y = y;
}
  this->x = x; this->y = y;
void point::print() {
  cout << x << "," << y << endl;
}

int main() {
  point p(1, 1);
  p.set(2, 2);
  p.print();

  return 0;
}
```

Are x and y arguments or member variables?
priority : arguments > member variables

result>
1, 1 → 2, 2

Array of Classes

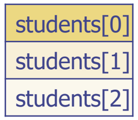
```
#include<iostream>
using namespace std;

class record {
  public:
    static int count;
    int order, id;
    int course1, course2;
    record(int i = 0, int s1 = 100, int s2 =
100);
    void print(void);
};

int record::count = 0;

record::record(int _id, int s1, int s2) {
  id = _id; course1 = s1; course2 = s2;
  order = ++count;
}
```

```
void record::print(void) {
  cout << order << "] ID : " << id
<< endl;
  cout << course1 << ", " << course2
<< endl;
}

int main( ) {
  record students[3];
  for (int i = 0; i < 3; i++)
    students[i].print( );
  return 0;
}
```

calls default constructor

result>
1] ID : 0
100, 100
2] ID : 0
100, 100
3] ID : 0
100, 100

memory



Array of Classes - Initialization

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

class record {
  public:
    static int count;
    int order, id, score;
    record(int _id = 0,
           int _score = 100);
    void print(void);
};

int record::count = 0;

record::record(int _id, int _score) {
  id = _id; score = _score;
  order = ++count;
}
```

```
void record::print(void) {
  cout << order << " [ " << id;
  cout << " ] score = " << score << endl;
}

int main( ) {
  record students[3] = { record(20090001, 99),
                         record(),
                         record(20090333) };
  for (int i = 0; i < 3; i++)
    students[i].print( );
  return 0;
}
```

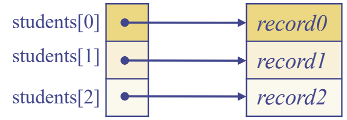
result>
1 [20090001] score = 99
2 [0] score = 100
3 [20090333] score = 100

Array of Pointers to Classes

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

class record {
  public:
    static int count;
    int order, id, score;
    record(int _id = 0,
           int _score = 100);
    void print(void);
};

int record::count = 0;

record::record(int _id, int _score) {
  id = _id; score = _score;
  order = ++count;
}
```

```
void record::print(void) { ... }

int main( ) {
  record *students[3]; // array of pointers
  for (int i = 0; i < 3; i++)
    students[i] = new record(2009000 + i, i);
  for (int i = 0; i < 3; i++) {
    students[i]->print();
    delete students[i];
  }
  return 0;
}
```

memory



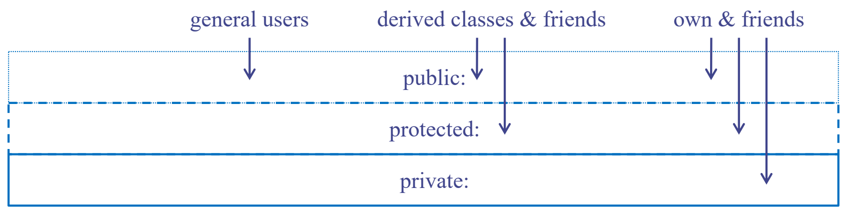
result>
1 [2009000] score = 0
2 [2009001] score = 1
3 [2009002] score = 2

Access Control, Inheritance

Access Control

```
class AccessControl {
public:                      Access specifier
        int publicData;                        public
        void publicFunc( );
protected:
        int protectedData;                     protected
        void protectedFunc( );
private:
        int privateData;                       private
        void privateFunc( );
};
```

```
int main( ) {
    AccessControl ac;
    ac.publicData = 1;        ( O )
    ac.publicFunc( );         ( O )

    ac.protectedData = 2;     ( X )
    ac.protectedFunc( );      ( X )

    ac.privateData = 3;       ( X )
    ac.privateFunc( );        ( X )
};
```



Example: Access Control

```
#include<iostream>
#define MAX 10
using namespace std;

class record{
    int course1, course2;
public:
    char name[MAX];
private:
    double avg;
public:
    void print(void) {
        cout << name << endl;
        cout << "course1 = " << course1
            << ", course2 = " << course2
<< endl;
        cout << "avg = " << avg << endl;
    }
};
```

by default, private

can be repeated

```
int main( ) {
    record myrecord;
    myrecord.name = "KIM JH";
    myrecord.course1 = 100;
    myrecord.course2 = 90;
    int sum = myrecord.course1 +
myrecord.course2;
    myrecord.avg = ((double) sum) / 2;
    myrecord.print( );
    return 0;
}
```

Access Error
→ How to modify?

Example: Access Control (cont'd)

```
#include<iostream>
#define MAX 10
using namespace std;

class record{
public:
    char name[MAX];
private:
    int course1, course2;
    double avg;
public:
    void print(void); // def. is ommitted.
    void set_course1(int score) { course1 = score; }
    void set_course2(int score) { course2 = score; }
    void calculate_avg( );
};
```

provide interface to access the private vars and function

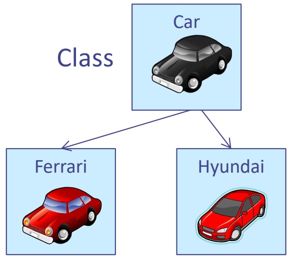
```
void record::calculate_avg( ) {
    int sum = course1 + course2;
    avg = ((double) sum) / 2;
}

int main( ) {
    record myrecord;
    myrecord.name = "KIM JH";
    myrecord.set_course1(100);
    myrecord.set_course2(90);
    myrecord.calculate_avg( );
    myrecord.print( );
    return 0;
}
```

Inheritance

Recall: What is Object ?

- Class (↔ Type in C)
  - ✓ Defines the abstract characteristics of a thing (object)
    - attributes (data) + behaviors (operations = methods)
- Object (↔ Variable in C)
  - ✓ A pattern (exemplar) of a class
- Instance
  - ✓ The actual object created at runtime
  - ✓ State: the set of values of the attributes of a particular object
- Methods
  - ✓ functions

Class



Object
(Instance at runtime)

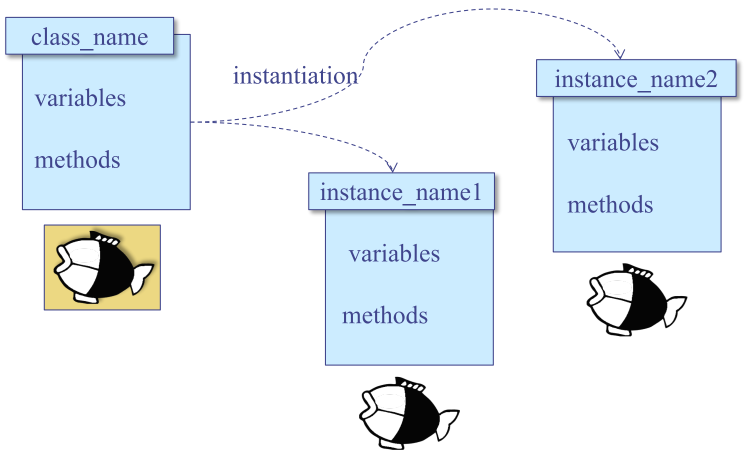
Attributes
: color, capacity, max. speed, …

Methods
: accelerate, brake, steer left, steer right, …

Recall: Class Declaration

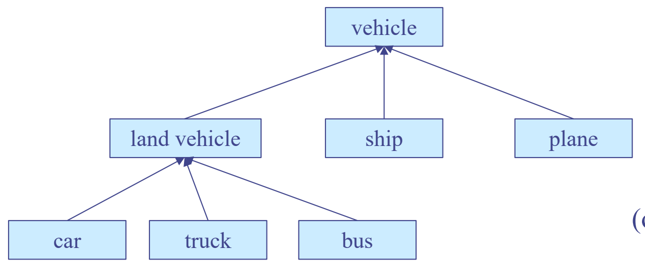
class_name *instance_name1, instance_name2;*

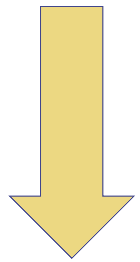
C.f. struct *tag_name struct_variable, … ;*



Inheritance (1/2)

- Subclassing: define a class based on another class
  - ✓ Another class = parent class (or superclass)
  - ✓ New class = child class (subclass)
  - ✓ Hierarchical classification in a tree form
  - ✓ Another way of "polymorphism"





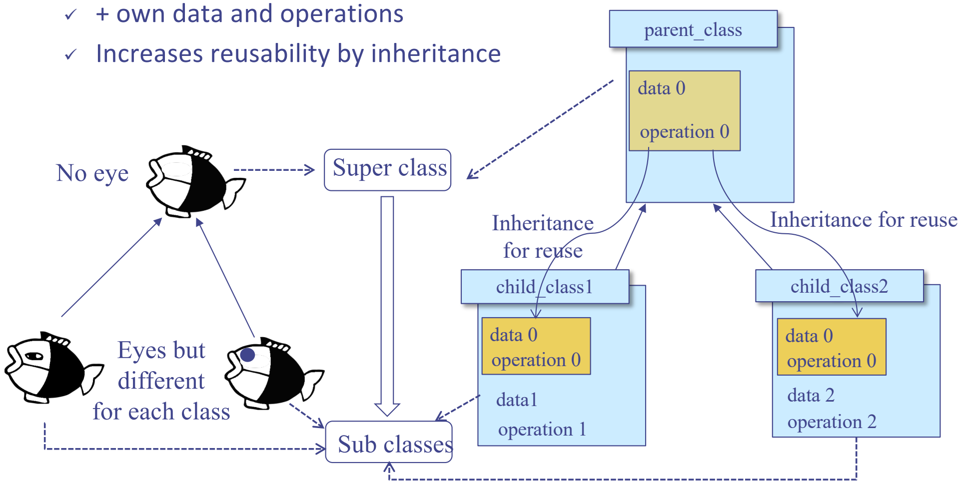
More specialized
(overridden, detailed, added)

Superclass → subclass

- overrides information in superclass
- refines information in superclass to detailed one
- adds more information to one in superclass

Inheritance (2/2)

- Inheritance
  - ✓ Inherits data (attributes) and operations (behaviors) from parent
  - ✓ + own data and operations
  - ✓ Increases reusability by inheritance



Class Example

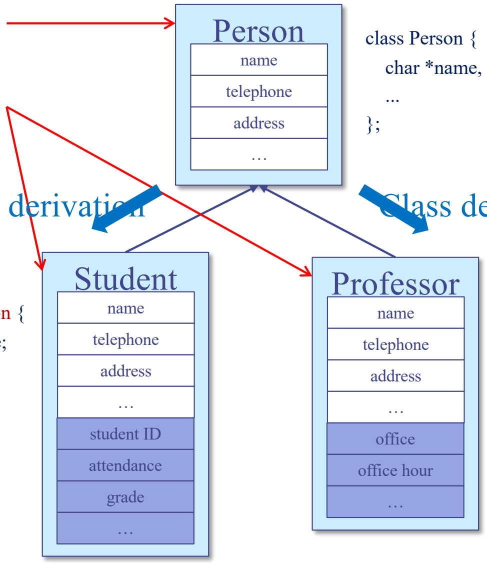
```
/* Fish Class */
class CFish {
   int color;
   char *name;
   int posx, posy;
public:
   void setcolor(int color);
   int getcolor (void);
   int setname(const char *name);
   void move(int x, int y);
};

class CJellyFish : public CFish {
   int light;
public:
   int turnlight(int on);
};

class CSquid : public CFish {
   int ink_color;
public:
   void setink_color(int color);
   int produce_ink(void);
}
```

```
CJellyFish jelly;
CSquid squid;

jelly.setname("Jelly Fish");
jelly.setcolor(WHITE_COLOR);
jelly.move(10, 10);
jelly.turnlight(LIGHT_ON);

squid.setname("Squid");
squid.setcolor(GREY_COLOR);
squid.move(40, 20);
squid.setink_color(BLACK_COLOR);
squid.produce_ink();
```


Inheritance: Mechanism for Reuse

Base class
(Parent, Superclass)

Derived class
(Child, subclass)




```
class Person {
   char *name, *telephone, *address;
   ...
};
```

```
class Student : public Person {
   int studentID, attendance;
   char grade;
   ...
};
```

```
class Professor : public Person {
   char *office, *officeHour;
   ...
};
```


Inheritance: Construct, Destruct Order

- Constructor order
  - base class ➡ derived class
- Destructor order
  - derived class ➡ base class

```
class Parent {
 public:
  Parent( ) { cout<<"Parent( )"<<endl; }
 ~Parent( ) { cout<<"~Parent( )"<<endl; }
};

class Child : public Parent {
 public:
  Child( ) { cout<<"Child( )"<<endl; }
 ~Child( ) { cout<<"~Child( )"<<endl; }
};
```

```
int main( ) {
   Child child;
   return 0;
}
```

```
result >
   Parent( )
   Child()
   ~Child()
   ~Parent()
```

Example : Constructors of Derived Class

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

class Parent {
public:
  char *_name;
  char* name() { return _name; }
  Parent(char *name = "");
  ~Parent() { delete _name; }
};

Parent::Parent(char *name) {
  _name = new
char[strlen(name)+1];
  strcpy(_name, name);
}
```

```
class Child : public Parent {
  int _age;
public:
  int age() { return _age; }
  Child(char *name = "", int age = 0);
  void print();
};

Child::Child(char *name, int age) :
Parent(name)
{
  _age = age;
}

void Child::print() {
  cout << "Name : " << _name << endl;
  cout << "age: " << _age << endl;
}
```

careful of arguments

uses Member Initialization List

```
int main() {
  Child myRecord("KIM", 21);
  myRecord.print();
  return 0;
}
```

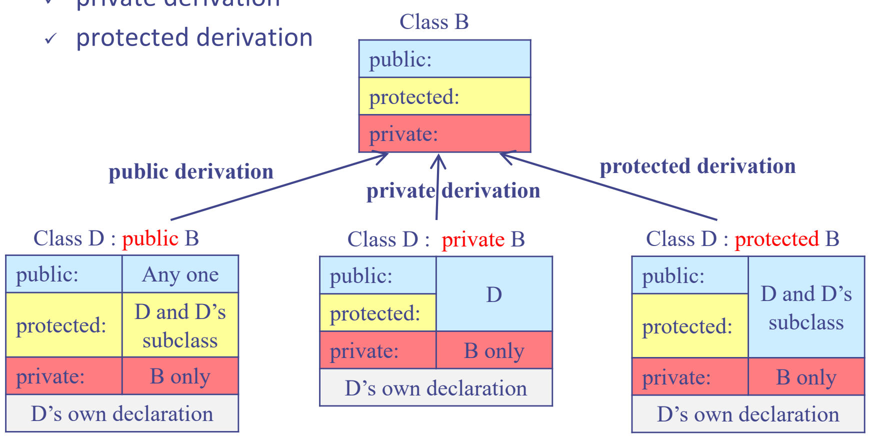
result>
Name : KIM
age: 21

Constructors of Derived Class

- If a base class has constructors, then a constructor must be invoked
  - ✓ Base class acts exactly like a member of the derived class in the constructor
    - base class' constructor is invoked in Member initialization list
  - ✓ Default constructors can be invoked implicitly
- A constructor of derived class can specify initializers for its own members and immediate bases only
  - ✓ Cannot directly initialize members of a base class

Access to Base Classes

- Access control of a base class
  - ✓ public derivation
  - ✓ private derivation
  - ✓ protected derivation

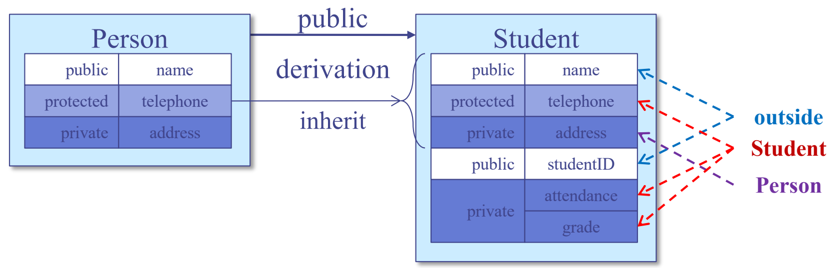


B's protected → D's protected

B's public and protected → D's private

B's public and protected → D's protected

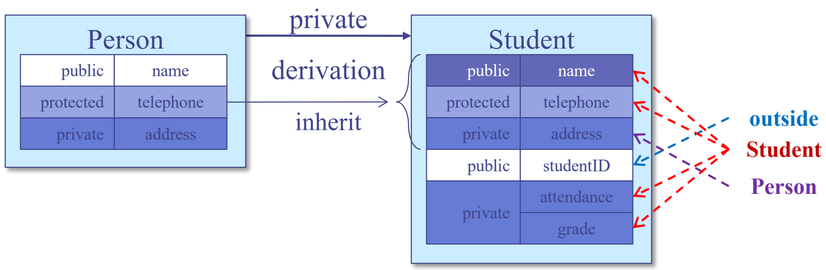
Public Derivation




```
class Person {
  public:
    char *name;
  protected:
    char *telephone;
  private:
    char *address;
};
```

```
class Student : public Person {
  public:
    int studentID;
  private:
    int attendance;
    char grade;
};
```

Private Derivation




```
class Person {
public:
    char *name;
protected:
    char *telephone;
private:
    char *address;
};
```

```
class Student : private Person {
public:
    int studentID;
private:
    int attendance;
    char grade;
};
```


Example: Public Derivation

```
#include<iostream>
using namespace std;
class Parent {
  char *_lastname;
public:
  char *_name;
  char* lastname() { return _lastname; }
  char* name() { return _name; }
  Parent(char *name = "",
          char *lastname = "");
  ~Parent() { delete _name, _lastname; }
};

Parent::Parent(char *name, char *lastname) {
  _name = new char[strlen(name)+1];
  strcpy(_name, name);
  _lastname = new
          char[strlen(lastname)+1];
  strcpy(_lastname, lastname);
}
```

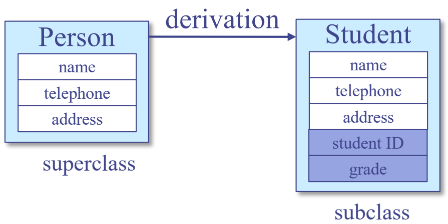
```
class Child : public Parent {
public:
  Child(char *name = "", char *lastname = "");
};

Child::Child(char *name, char *lastname) :
Parent(name, lastname)
{}

int main() {
  Child myRecord("JH", "KIM");
  cout << "Name : " << myRecord._name << endl;
  cout << "Last name : " << myRecord._lastname() << endl;

  return 0;
}
```

Name : JH

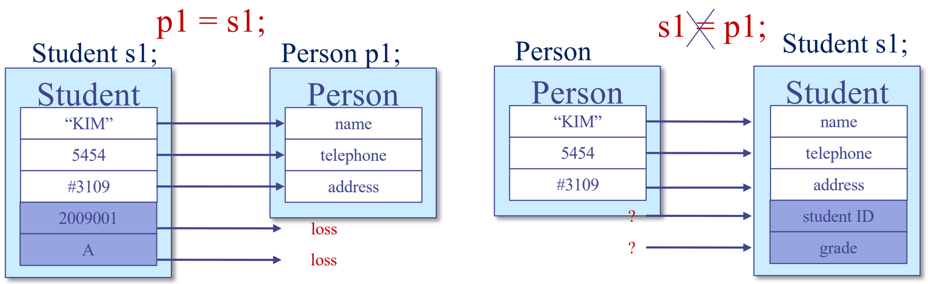
Last name : KIM

Assignment of Objects

- General Rule
  - object with less info ← object with more info
    1. Object of Superclass ← Object of Subclass
    2. Object of Subclass ~~←~~ Object of Superclass

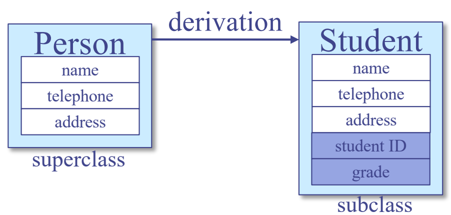


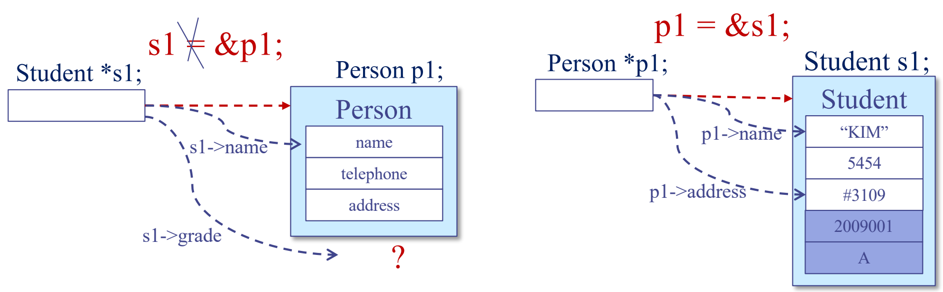
Person is a (kind of) Student? No
Student is a (kind of) Person? Yes



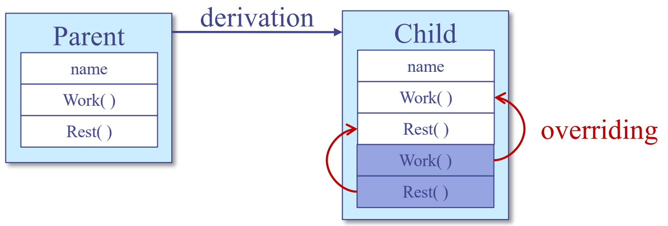
Type Conversion of Pointer & Reference

Rule
- Object of Subclass ~~← Object of Superclass~~
- Object of Superclass ← Object of Subclass





Overriding: From Subclass to Superclass




```
class Parent {
  ...
public:
  void Work ( ) { ... }
  void Rest ( ) { ... }
};
```

overriding

```
class Child : public Parent{
  ...
public:
  void Work ( ) { ... }
  void Rest ( ) { ... }
};
```


Example: Overriding (1/2)

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

class Parent {
public:
 void print( ) {
  cout << "I'm your father." << endl;
 }
};

class Child : public Parent {
public:
 void print( ) {
  cout << "I'm your son." << endl;
 }
};
```

overriding

```
int main() {
 Child child;
 child.print( );
 return 0;
}
```

result>
I'm your son.

Example: Overriding (2/2)

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

class Parent {
public:
 void print( ) {
  cout << "I'm your father." << endl;
 }
};

class Child : public Parent {
public:
 void print(int i = 1) {
  for (int j = 0; j < i; j++)
   cout << "I'm your son." << endl;
 }
};
```

overriding

```
int main() {
 Child child;
 child.print( );
 child.print(3);
 return 0;
}
```

result>
I'm your son.
I'm your son.
I'm your son.
I'm your son.

Call Overridden Functions

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

class Parent {
public:
 void print( ) {
  cout << "I'm your father." << endl;
 }
};

class Child : public Parent {
public:
 void print( ) {
  cout << "I'm your son." << endl;
 }
};
```

overriding

```
int main() {
 Child child;
 child.print( );
 child.Parent::print( );
 return 0;
}
```

result>
I'm your son.
I'm your father.

Static Binding

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

class Parent {
public:
  void print( ) {
    cout << "I'm your father." << endl;
  }
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm your son." << endl;
  }
};
```

overriding

```
int main() {
  Child *child = new Child( );
  child->print( );

  Parent *father = child;
  father->print( );

  delete child;

  return 0;
}
```

Static binding (compile-time binding)

result>
I'm your son.
I'm your father.

How does father do as child ?
→ Dynamic binding

Dynamic Binding: Virtual Functions

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

class Parent {
public:
  virtual void print( ) {
    cout << "I'm your father." << endl;
  }
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm your son." << endl;
  }
};
```

virtual function

overriding

```
int main() {
  Child *child = new Child( );
  child->print( );

  Parent *father = child;
  father->print( );

  delete child;

  return 0;
}
```

Dynamic binding (run-time binding)

result>
I'm your son.
I'm your son.

- Polymorphism → Ability to have many forms
  - Objects with different internal structures can share the same external interface
  - virtual function and class derivation are means to realize polymorphism

Virtual and Non-Virtual Functions

```
class Parent {
public:
  virtual void vpr( ) { cout << "vpr: parent" << endl; }
  void nvpr ( ) { cout << "nvpr: parent" << endl; }
};

class Child : public Parent {
public:
  void vpr( )   { cout << "vpr: child" << endl; }
  void nvpr( ) { cout << "nvpr: child" << endl; }
  }
};
```

```
Parent father;
Child  son;

Parent *par_pt = &son
```

father.vpr() → vpr: parent
father.nvpr() → nvpr: parent
son.vpr() → vpr: child
son.nvpr() → nvpr: child
par_pt -> vpr() → vpr: child
par_pt -> nvpr() → nvpr: parent

Virtual Destructor (1/2)

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

class Parent {
  char* familyName;
public:
Parent(char* _familyName) {
 familyName  = new
   char[strlen(_familyName)+1];
 strcpy(familyName, _familyName);
 }

 ~Parent( ){
   cout << "~Parent()" << endl;
   delete familyName;
 }
 virtual void PrintName() {
   cout << familyName << ',';
 }
};
```

```
class Child : public Parent {
  char* name;
public:
  Child(char* _familyName, char*
_name)
     : Parent(_familyName) {
    name = new char[strlen(_name)+1];
   strcpy(name, _name);
  }
  ~Child( ){
   cout << "~Child()" << endl;
   delete name;
  }
  virtual void PrintName() {
   Parent::PrintName();
   cout << name << endl;
  }
};
```

```
int main() {
Parent *parent = new Child("KIM", "JH");
Child *child = new Child("KIM", "HS");
parent->PrintName();
child->PrintName();
cout << endl;
delete child;
cout << endl;
delete parent;

return 0;
}
```

How to delete child's name?

result>
KIM,JH
KIM,HS

~Child()
~Parent()

~Parent()

Virtual Destructor (2/2)

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

class Parent {
  char* familyName;
public:
  Parent(char* _familyName) {
   familyName  = new
    char[strlen(_familyName)+1];
   strcpy(familyName, _familyName);
  }
  virtual ~Parent( ){
   cout << "~Parent()" << endl;
   delete familyName;
  }
  virtual void PrintName() {
   cout << familyName << ',';
  }
};
```

```
class Child : public Parent {
  char* name;
public:
  Child(char* _familyName, char*
_name) : Parent(_familyName) {
   name = new
char[strlen(_name)+1];
   strcpy(name, _name);
  }
  ~Child( ){
   cout << "~Child()" << endl;
   delete name;
  }
  virtual void PrintName() {
   Parent::PrintName();
   cout << name << endl;
  }
};
```

```
int main() {
  Parent *parent = new Child("KIM", "JH");
  Child *child = new Child("KIM", "HS");
  parent->PrintName();
  child->PrintName();
  cout << endl;
  delete child;
  cout << endl;
  delete parent;

  return 0;
}
```

```
result>
KIM,JH
KIM,HS

~Child()
~Parent()

~Child()
~Parent()
```


Template: Function and Class

Function Template (1)

```
int integerMin(int a, int b)          // returns the minimum of a and b
  { return (a < b ? a : b); }
```

- Useful, but what about min of two doubles?
  - ✓ C-style answer: double doubleMin(double a, double b)

- Function template is a mechanism that enables this
  - ✓ Produces a generic function for an arbitrary type T.

```
template <typename T>
T genericMin(T a, T b) {              // returns the minimum of a and b
  return (a < b ? a : b);
}
```


Function Template (2)

```
template <typename T>
T genericMin(T a, T b) {              // returns the minimum of a and b
  return (a < b ? a : b);
}
```

```
cout << genericMin(3, 4) << ' '       // = genericMin<int>(3,4)
  << genericMin(1.1, 3.1) << ' '      // = genericMin<double>(1.1, 3.1)
  << genericMin('t', 'g') << endl;    // = genericMin<char>('t','g')
```

Function Overloading vs. Function Template

- Function overloading
  - ✓ Same function name, but different function prototypes
  - ✓ These functions do not have to have the same code
  - ✓ Does not help in code reuse, but helps in having a consistent name
- Function template
  - ✓ Same code piece, which applies to only different types

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

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```


Class Template (1)

- In addition to function, we can define a generic template class
- Example: BasicVector
  - ✓ Stores a vector of elements
  - ✓ Can access i-th element using [] just like an array

```
template <typename T>
class BasicVector {                    // a simple vector class
public:
  BasicVector(int capac = 10);         // constructor
  T& operator[](int i)                 // access element at index i
    { return a[i]; }
  // ... other public members omitted
private:
  T* a;                                // array storing the elements
  int capacity;                        // length of array a
};
```


Class Template (2)

- BasicVector
  - ✓ Constructor code?

```
template <typename T>                  // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];                 // allocate array storage
}
```

- How to use?

```
BasicVector<int>    iv(5);     iv[3] = 8;
BasicVector<double> dv(20);    dv[14] = 2.5;
BasicVector<string> sv(10);    sv[7] = "hello";
```


Class Template (3)

- The actual argument in the instantiation of a class template can itself be a templated type
- Example: Twodimensional array of int

```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

- BasicVector consisting of 5 elements, each of which is a BasicVector consisting of 10 integers
  - ✓ In other words, 5 by 10 matrix

Exceptions

Exceptions: Intro

- Exception
  - ✓ Unexpected event, e.g., divide by zero
  - ✓ Can be user-defined, e.g., input of studentID > 1000
  - ✓ In C++, exception is said to be “thrown”
  - ✓ A thrown exception is said to be “caught” by other code (exception handler)
  - ✓ In C, we often check the value of a variable or the return value of a function, and if… else… handles exceptions
    - Dirty, inconvenient, hard to read

Exception: Also a class

```
class MathException {                     // generic math exception
public:
  MathException(const string& err)        // constructor
    : errMsg(err) { }
  string getError() { return errMsg; }    // access error message
private:
  string errMsg;                          // error message
};
```

```
class ZeroDivide : public MathException {
public:
  ZeroDivide(const string& err)
  : MathException(err) { }
};
```

```
class NegativeRoot : public MathException {
public:
  NegativeRoot(const string& err)
  : MathException(err) { }
};
```


Exception: Throwing and Catching

```
try {
  // . . . application computations
  if (divisor == 0)                        // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

ZeroDivide “is a” MathException? Yes

Exception Example (1)

```
#include <iostream>
using namespace std;
double division(int a, int b){
     if( b == 0 ) {
         throw "Division by zero condition!";
     }
     return (a/b);
}

int main () {
     int x = 50;   int y = 0;   double z = 0;
     try {
           z = division(x, y);
           cout << z << endl;
     } catch (const char* msg) {
           cerr << msg << endl;
     }
     return 0;
}
```


Exception Specification

- In declaring a function, we should also specify the exceptions it might throw
  - ✓ Lets users know what to expect

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body . . .
}
```

The function calculator (and any other functions it calls) can throw two exceptions or exceptions derived from these types

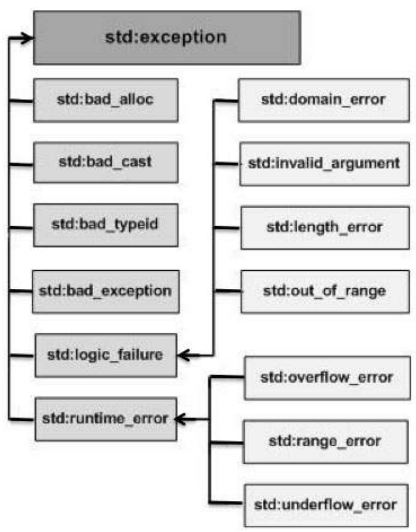
- Exceptions can be “passed through”

```
void getReadyForClass() throw(ShoppingListTooSmallException,
                              OutOfMoneyException) {
    goShopping();  // I don't have to try or catch the exceptions
                   // which goShopping() might throw because
                   // getReadyForClass() will just pass these along.
    makeCookiesForTA();
}
```


Exception: Any Exception and No Exception

```
void func1();              // can throw any exception
void func2() throw();      // can throw no exceptions
```


C++ Standard Exceptions



Exception Example (2)

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

class MyException : public exception {
    const char * what () const throw (){
        return "C++ Exception";
    }
};

int main()
{
    try {
       throw MyException();
    }catch(MyException& e){
       std::cout << "MyException caught" << std::endl;
       std::cout << e.what() << std::endl;
    } catch(std::exception& e){
       //Other errors
    }
}
```

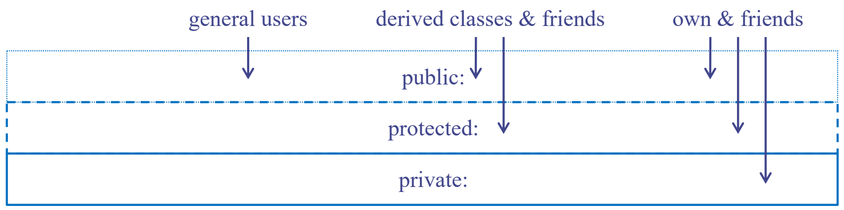

Friend

Recall: Access Control

```
class AccessControl {
public:
        int publicData;
        void publicFunc( );
protected:
        int protectedData;
        void protectedFunc( );
private:
        int privateData;
        void privateFunc( );
};
```

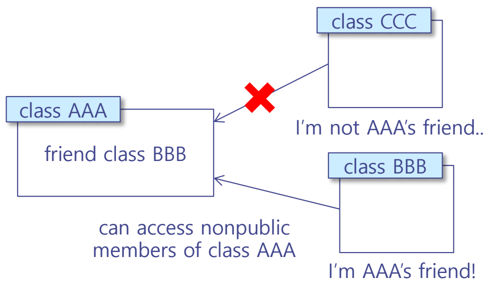
public / protected / private

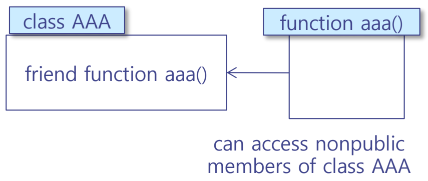
```
int main( ) {
  AccessControl ac;
  ac.publicData = 1;        ( O )
  ac.publicFunc( );         ( O )

  ac.protectedData = 2;     ( X )
  ac.protectedFunc( );      ( X )

  ac.privateData = 3;       ( X )
  ac.privateFunc( );        ( X )
};
```



Friends to a Class

- In some cases, information-hiding is too prohibitive.
  - ✓ Only public members of a class are accessible by non-members of the class
- “friend” keyword
  - ✓ To give nonmembers of a class access to the nonpublic members of the class
- Friend
  - ✓ Functions
  - ✓ Classes





Example: Friend Functions

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

class point {
   int x, y;
  public:
   point(int a = 0, int b = 0);
   void print();

   friend void set(point &pt, int a, int b);
};

point::point(int a, int b) {
  x = a; y = b;
}
```

not member function,
but friend function

```
void point::print() {
  cout << x << ", " << y << endl;
}

void set(point &pt, int a, int b) {
  pt.x = a; pt.y = b;
}

int main() {
  point p(1, 1);
  p.print();
  set(p, 2, 2);
  p.print();

  return 0;
}
```

call-by-reference

not "p.set( );"

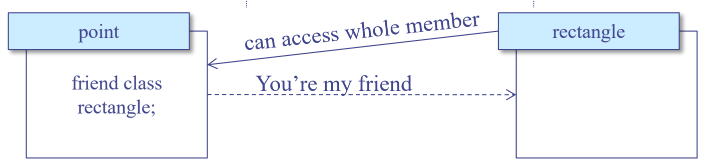
result>
1, 1
2, 2

Friend Class

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

class point {
   int x, y;
   friend class rectangle;
  public:
   void set(int a, int b);
};

void point::set(int a, int b) {
  x = a; y = b;
}
```

```
class rectangle {
   point leftTop, rightBottom;
  public:
   void setLT(point pt);
   void setRB(point pt);
   void print();
};

void rectangle::setLT(point pt) {
  leftTop.set(pt.x, pt.y);
}

void rectangle::setRB(point pt) {
  rightBottom.set(pt.x, pt.y);
}
```

```
void rectangle::print() {
  cout << "LT:" << leftTop.x;
  cout << ","    << leftTop.y << endl;
  cout << "RB:" << rightBottom.x;
  cout << ","    << rightBottom.y << endl;
}

int main() {
  rectangle sq;
  point lt, rb;
  lt.set(1, 1); sq.setLT(lt);
  rb.set(9, 9); sq.setRB(rb);
  sq.print();
  return 0;
}
```



result>
LT:1, 1
RB:9, 9

Wrap Up

- You may not have a big problem in reading the codes in the book
- You may not have a big problem in doing the homework assignments
- However,
  - ✓ Be ready to debug your program
  - ✓ Be ready to search more things in Google
  - ✓ Be ready to meet "compilation errors"

Supplementary Materials

Example : Constructors

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
  public:
    char name[MAX];
  private:
    int course1, course2;
    double avg;
  public:
    record( );
    void print(void);
};

void record::print(void) { … }
```

```
record::record( ) {
  strcpy(name, "");
  course1 = course2 = 100;
  avg = 100;
}

int main( ) {
  record yourRecord = { "HONG GD", 100, 100 };
  yourRecord.print();

  record myRecord = record::record( );
  myRecord.print( );

  return 0;
}
```

Error

∵member variables in "private"

Example: Constructors & Destructors

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

class record {
  public:
    char *name;
  private:
    int course1, course2;
    double avg;
  public:
    record(char *str = "", int s1 = 100, int s2 =
100);
    ~record();
    void print(void);
};

record::~record() {
  delete []name;
}
```

```
record::record(char *str, int s1, int s2) {
  name = new char[strlen(str)+1];
  strcpy(name, str);
  course1 = s1; course2 = s2;
  avg = ((double) (s1 + s2)) / 2;
}

void record::print(void) { … }

int main( ) {
  record *myRecord = new record( );
  record *yourRecord = new record("KIM", 90,
100);

  myRecord->print( );
  yourRecord->print( );

  delete myRecord, yourRecord;

  return 0;
}
```


Constructors with Arg. and Default Values

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
  public:
    char name[MAX];
  private:
    int course1, course2;
    double avg;
  public:
    record(char *str = "", int s = 100);
    void print(void);
};

record::record(char *str, int score) {
  strcpy(name, str);
  course1 = course2 = score;
  avg = score;
}
```

```
void record::print(void) { … }

int main( ) {
  record myRecord;
  record yourRecord = record("KIM",
90);
  record hisRecord = "LEE";

  myRecord.print( );
  yourRecord.print( );
  hisRecord.print( );

  return 0;
}
```

implicitly call with default values
(default constructor)

result>

course1 = 100, course2 =
100
avg = 100
KIM
course1 = 90, course2 =
90
avg = 90
LEE
course1 = 100, course2 =
100
avg = 100

shorthand notation

same as
record hisRecord = record("LEE");

A Special Constructor : Copy Constructor

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

class point {
  public:
    int x, y;
    point(int _x, int _y) {
      x =_x; y = _y;
    }
    point(const point &pt) {
      x = pt.x; y = pt.y;
    }
    void set(int _x, int _y) {
      x = _x; y = _y;
    }
    void print();
};
```

```
void point::print() {
  cout << x << "," << y << endl;
}

int main() {
  point p1(1, 1);
  point p2(p1);
  p1.set(2, 2);
  cout << "P1 : ";
  p1.print();
  cout << "P2 : ";
  p2.print();

  return 0;
}
```

result>
P1 :
2,2
P2 :
1,1

copy constructor

Syntax : X(const X& X1)

Default Copy Constructor

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

class point {
  public:
    int x, y;
    point(int _x, int _y) {
       x =_x; y = _y;
    }
    void set(int _x, int _y) {
       x = _x; y = _y;
    }
    void print();
};
```

```
void point::print() {
  cout << x << "," << y << endl;
}

int main() {
  point p1(1, 1);
  point p2(p1);
  p1.set(2, 2);
  cout << "P1 : ";
  p1.print();
  cout << "P2 : ";
  p2.print();
}
```

```
result>
P1 :
2,2
P2 :
1,1
```

same result

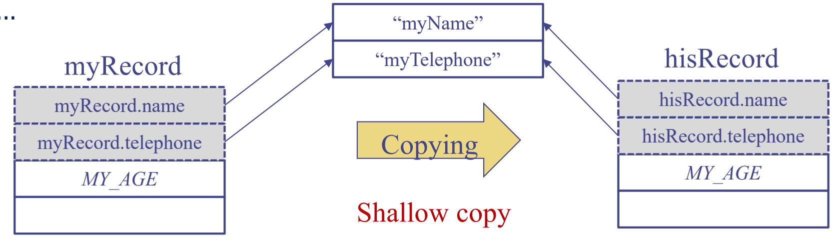
default copy constructor
- simply copies all members implicitly
- can be used without definition

Limitation of Default Copy Constructor

```
class record{
  public:
    char *name;
    char *telephone;
    int age;
    ...
};
```

```
int main() {
  record myRecord;
  record hisRecord = myRecord;
  ...
}
```

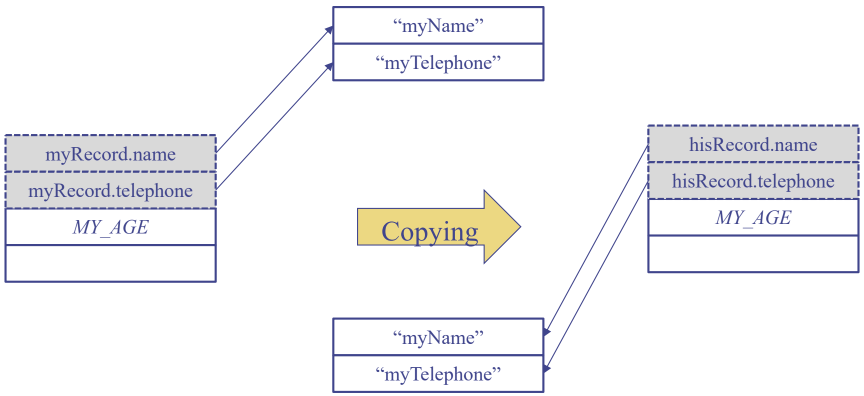
calls default copy constructor
= record hisRecord(myRecord);



Shallow copy

Member variables of an object are two pointers (name and telephone) and one integer
→ Copied two pointer variables and one integer variable
→ Two points variables point to the same locations as ones in original objects.
One integer variable copies its own.

Deep Copy Constructor



Deep copy

Deep copy of two member variables of type pointers
→ Copied pointer variables points to different locations from ones in original ones.

Example: Deep Copy Constructor

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

class record {
  public:
    char *name;
    char *tel;
    record(char *, char *);
    record(const record &);
    ~record();
    void modifyTel(char *_tel);
    void print(void);
};

record::record(char *_n, char
*_tel) {
  name = new char[strlen(_n)+1];
  strcpy(name, _n);
  tel = new char[strlen(_tel)+1];
  strcpy(tel, _tel);
}
```

```
record::record(const record &_record) {
  name = new
char[strlen(_record.name)+1];
  strcpy(name, _record.name);
  tel = new char[strlen(_record.tel)+1];
  strcpy(tel, _record.tel);
}

record::~record() {
  delete name, tel;
}

void record::modifyTel(char *_tel) {
  delete tel;
  tel = new char[strlen(_tel)+1];
  strcpy(tel, _tel);
}
```

deep copy

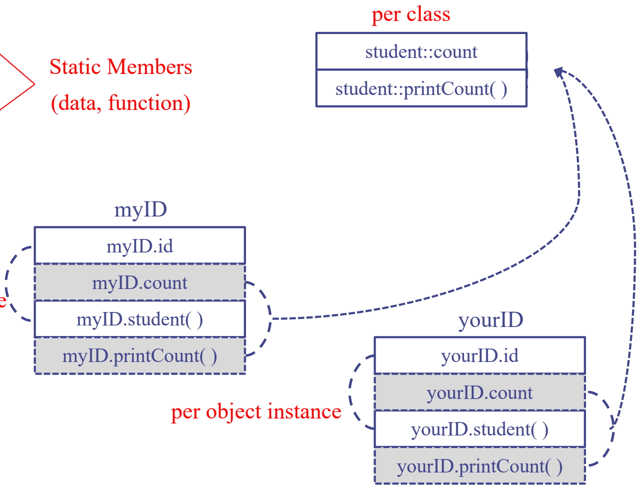
```
void record::print(void) {
  cout << name;
  cout << " : " << tel << endl;
}

int main( ) {
  record myRecord("KIM",
"6565");
  record hisRecord(myRecord);
  myRecord.modifyTel("5454");
  myRecord.print( );
  hisRecord.print( );
  return 0;
}
```

```
result>
KIM : 5454
KIM : 6565
```

Static Members

```
class student {
  public:
    int id;
    static int count;

    student(int i = 0);
    static void printCount( )
};

student myID, yourID;
```

Static Members
(data, function)



Example: Static Members (1/3)

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

class student {
  public:
    int id;
    student(int i = 0);
    static void printCount();
  private:
    static int count;
};

int student::count = 0;

student::student(int i) {
  id = i;
  count++;
}
```

```
void student::printCount() {
  cout << "count = " << count
<< endl;
}

int main() {
  student myID = 20090001;
  myID.printCount();
  student yourID;
  myID.printCount();
  student hisID, herID;
  student::printCount();
}
```

result>
count = 1
count = 2
count = 4

Static member function can be accessed directly with class name

A static data member must be initialized outside the class definition in the same manner as a non-member variable
∵only one copy of static member

Access of a static member is syntactically identical

Example: Static Members (2/3)

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

class student {
  public:
    int id;
    int order;
    student(int  i= count);
    static void printCount();
  private:
    static int count;
};

int student::count = 0;

student::student(int i) {
  order = i;
  count++;
}
```

A static data member can appear as a default argument to a member function of the class.

(A non-static member cannot.)

Example: Static Members (3/3)

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

class math {

private:
    static int sum;
    static int facto;
    static int permu;
public:
    math(){
        sum = 0;
        facto = 0;
        permu = 0;
    }
    static int summation(int a);
    static int factorial(int a);
    static int permutation(int a, int b);
};

int math::sum = 0;
int math::facto = 1;
int math::permu = 1;

int math::summation(int a){
    sum = 0;
    for(int i=0; i<=a; i++)
        sum += i;
    return sum;                }
```

Static Member Variables Initialization

```
int math::factorial(int a){
    facto = 1;
    while(a != 0){
        facto  *=  a;
        a -= 1;
    }
    return facto;
}

int math::permutation(int a, int b){
    permu = 1;
    permu = math::factorial(a) / math::factorial(a-
b);

    return permu;
}

int main() {
    int result1, result2, result3;

    result1 = math::summation(5);
    result2 = math::factorial(4);
    result3 = math::permutation(6,2);

    cout << " sum: " << result1 << endl;
    cout << " factorial: " << result2 << endl;
    cout << " permutation: " << result3 << endl;

    return 0;
}
```

Calls of static Member functions

result>
sum: 15
factorial : 24
permutation : 30

Const Keyword

- Declare constant variables, pointers, member functions
- Once initialized, the value of the const variables cannot be overriden.

```
int n1 = 10; int n2 = 20

const int* p1 = &n1;     /* p1 is a pointer to a constant  integer*/
           p1 = &n2 ;             /* ok! */
           *p1 = 20;         Compile Error!

int* const p2 = &n1;    /* p2 is a constant pointer  to an integer*/
           *p2 = 20;              /* ok! */
           p2 = &n2;         Compile Error!

const int* const p3 = &n1;    /* p3 is a constant pointer to a constant  integer */
           *p3 = 20;         Compile Error!
           p3 = &n2;         Compile Error!
```


Const Member Variables

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

class record {
  public:
    const int id;      constant
    int course1, course2;
    record(int i = 0, int s1 = 100, int s2 = 100);
    void print(void);
};

record::record(int i, int s1, int s2) {     assignment
  id = i;                                   (not initialization)
  course1 = s1; course2 = s2;
}

void record::print(void) {
  cout << "ID : " << id << endl;
  cout << "course1 = " << course1;
  cout << ", course2 = " << course2 << endl;
}

int main( ) {
  record myRecord(20090001, 90, 100);
  myRecord.print( );
  return 0;
}
```

Error

```
record::record(int i, int s1, int s2)
        : id(i)
{
    course1 = s1; course2 = s2;
}
```


Const Member Functions

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

class point {
    int x, y;
  public:
    point(int = 0, int = 0);
    void set(int, int) const;
    void print( ) const;
};

point::point(int a, int b) {
  x = a; y = b;
}

void point::set(int a, int
const {                       ERROR
  x = a; y = b;               ∵x, y are non-constant
}

void point::print( ) const {
  cout << x << "," << y
<< endl;
}
```

const Member Function: only applied to const data, not to non-const. data

```
int main( ) {
  point p(1, 1);
  p.print();

  const point p2(2, 2);       ERROR
  p2.set(3, 3);               ∵A const class object cannot
  p2.print();                 invoke non-const member functions.
  return 0;
}
```


Reference Member Variables (1/2)

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

class record {
  public:
    int &id;       reference
    int course1, course2;
    record(int i = 0, int s1 = 100, int s2 = 100);
    void print(void);
};

record::record(int i, int s1, int s2)     initialization
: id(i)
{
  course1 = s1; course2 = s2;
}

void record::print(void) {
  cout << "ID : " << id << endl;
  cout << "course1 = " << course1;
  cout << ", course2 = " << course2 << endl;
}

int main( ) {
  record myRecord(20090001, 90, 100);
  myRecord.print( );
  return 0;
}
```

result>

ID : garbage

course1 = 90, course2 = 100

Reference Member Variables (2/2)

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

class record {
  public:
    int &id;        reference
    int course1, course2;
    record(int &i, int s1 = 100, int s2 = 
100);
    void print(void);
};

record::record(int& i, int s1, int s2)   initialization
: id(i)
{
  course1 = s1; course2 = s2;
}
```

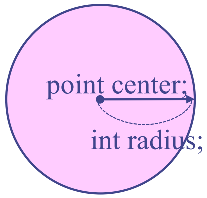
```
void record::print(void) {
  cout << "ID : " << id << endl;
  cout << "course1 = " << course1;
  cout << ", course2 = " << course2
<< endl;
}

int main( ) {
  int common = 20090001;
  record Record1(common, 90, 100);
  record Record2(common, 70, 80);
  common = 20090002;
  Record1.print( );
  Record2.print();
  return 0;
}
```

result>

```
ID : 20090002
course1 = 90, course2 = 100
ID : 20090002
course1 = 70, course2 = 80
```


Inheritance VS. Nested Class

Nested	Has-a relation	A circle has a point.
Inheritance	Is-a relation	A student is a person.



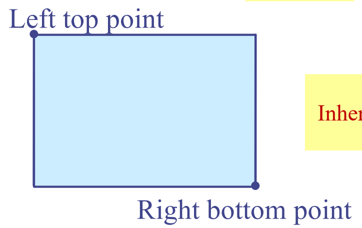
Nested

```
class circle {
  point center;
  int radius;
  ...
};
```

Inheritance

```
class circle : public point {
  int radius;
};
```

Unnatural !



Inheritance

How about with 2 points ?

Example: Private Derivation

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

class Parent {
  char *_lastname;
public:
  char *_name;
  char* lastname() { return _lastname; }
  char* name() { return _name; }
  Parent(char *name = "", char *lastname = "");
  ~Parent() { delete _name, _lastname; }
};

Parent::Parent(char *name, char *lastname) {
  _name = new char[strlen(name)+1];
  strcpy(_name, name);
  _lastname = new char[strlen(lastname)+1];
  strcpy(_lastname, lastname);
}
```

```
class Child : private Parent {
public:
  Child(char *name = "", char *lastname = "");
};

Child::Child(char *name, char *lastname) : Parent(name, lastname) {
}

int main() {
  Child myRecord("JH", "KIM");
      cout << "Name : " << myRecord.name() << endl;
    cout << "Last name : " << myRecord.lastname() << endl;

  return 0;
}
```

```
Name : JH
Last name : KIM
```


Example: Type Conversion of Pointer (1)

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

class Person {
  public:
    void Sleep() { cout<<"Sleep"<<endl; }
};

class Student : public Person {
  public:
    void Study() { cout<<"Study"<<endl; }
};

class Undergraduate : public Student {
  public:
    void Research()
    { cout<<"Research"<<endl; }
};
```

```
int main() {
  Person *p1 = new Person;               ( O )
  Student *p2 = new Person;              ( X )
  Undergraduate *p3 = new Person;        ( X )

  p1->Sleep();
  p2->Sleep();
  p3->Sleep();
  return 0;
}
```

Example: Type Conversion of Pointer (2)

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

class Person {
  public:
    void Sleep() { cout<<"Sleep"<<endl; }
};

class Student : public Person {
  public:
    void Study() { cout<<"Study"<<endl; }
};

class Undergraduate  : public Student {
  public:
    void Research()
  { cout<<"Research"<<endl; }
};
```

```
int main() {
  Person *p1 = new Student;          ( O )
  Person *p2 = new Undergraduate;    ( O )
  Student *p3 = new Undergraduate;   ( O )

  p1->Sleep();
  p2->Sleep();
  p3->Sleep();
  return 0;
}
```


Example: Type Conversion of Pointer (3)

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

class Person {
  public:
    void Sleep() { cout<<"Sleep"<<endl; }
};

class Student : public Person {
  public:
    void Study() { cout<<"Study"<<endl; }
};

class Undergraduate : public Student {
  public:
    void Research() { cout<<"Research"<<endl; }
};
```

```
int main() {
  Person *p1 = new Person;           ( O )
  Person *p2 = new Student;          ( O )
  Person *p3 = new Undergraduate;    ( O )

  p1->Sleep();
  p2->Sleep();
  p3->Sleep();
  return 0;
}
```


Overriding and Overloading

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

class Parent {
public:
  void print( ) {
    cout << "I'm your father." << endl;
  }

  void print(int i) {
    for (int j = 0; j < i; j++)
      cout << "I'm your father." << endl;
  }
};
```

overriding

Overloading (within class)

```
class Child : public Parent {
public:
  void print( ) {
    cout << "I'm your son." << endl;
  }
};

int main() {
  Child child;
  child.print( );        ERROR
  child.print(3);
  return 0;
}
```


Multiple Inheritance

- In C++, a class can have more than one immediate bass class
  - ✓ Not supported in JAVA
- Multiple Inheritance
  - ✓ The use of more than one immediate base class
    - Inheritance tree → Inheritance graph with no cycle
  - ✓ Usage
    - *class* child : *public* parent1, *public* parent2 { … }
  - ✓ Combined two unrelated classes together as a part of an implementation of a third class
  - ✓ Conflict of names: Two base classes have a member function with the same name
    - To resolve ambiguity, use following expression
      - parent class name : : function()
    - Ex. when two parents have the same function A()

```
ch->A();              // error  → Ambiguity for inheritance
ch->parent1::A();     // ok
ch->parent2::A();     // ok
```

Example : Multiple Inheritance

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

class Output
{
public:
  Output(){}
  void Print() { cout << contents
<< endl;}
protected:
  char contents[20];
};
class IntInput
{
public:
   IntInput(){}
   void In() { cin >> number; }
protected:
   int number;
};
```

```
class IO : public Output, public IntInput
{
public:
  IO(){}
  void Delivery(){
   sprintf(contents, "%d", number);
  }
};

int main()
{
   IO *a = new IO();
   cout << "Input : ";
   a->In();              // from IntInput class
   a->Delivery();        // from IO class
   cout << "Output : ";
   a->Print();           // from Output class
  return 0;
}
```

Result>

Input : 10

Output : 10

Heterogeneous List

- Homogenous List
  - List of objects in the same class (type) → Implementation in array
- Heterogeneous List
  - List of objects in different classes
  - Use points to objects in base class and derived classes → array of pts
  - Uniform interface for objects in different classes

```
class Parent {
public:
  virtual void vpr( ) { cout << "vpr: parent" << endl; }
};

class Child : public Parent {
public:
  void vpr() { cout << "vpr: child" << endl; }
};
```

```
Parent par1, par2;
Child  son1, son2;

Parent *list[4 ];
list[0] = &par1;
list[1] = &son1;
list[2] = &son2;
list[3] = &par2;
```

Heterogeneous List in uniform interface

```
for  (int i = 0; i < 4; i++)
   list[i] -> vpr( );
```

→

vpr( ): parent
vpr( ): child
vpr( ): child
vpr( ): parent

Pure Virtual Functions and Abstract Class

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

class Parent {
public:
  virtual void print( ) = 0;
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm your son." << endl;
  }
};
```

Pure virtual function

1. A virtual function is made "pure" by the initialzer = 0.
2. A virtual function cannot be called within an abstract class.

Abstract class

1. A class with one or more pure virtual functions
2. No object from class is created.
3. Means to provide an interface without exposing any implementation details

Example: Pure Virtual Functions

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

class Parent {
public:
  virtual void print( ) = 0;
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm your son." << endl;
  }
};
```

```
int main() {
  Parent parent;
  parent.print( );
  Child child;
  child.print( );
  child.Parent::print( );
  return 0;
}
```

ERROR

ERROR

∵ Cannot invoke a virtual function

∵ No objects of an abstract class can be created.