University of North Texas

CSCE 2110
Foundations of Data Structures

C++ and Oriented-Object Programming

Slides borrowed/adapted from Prof. Yung Li from KAIST

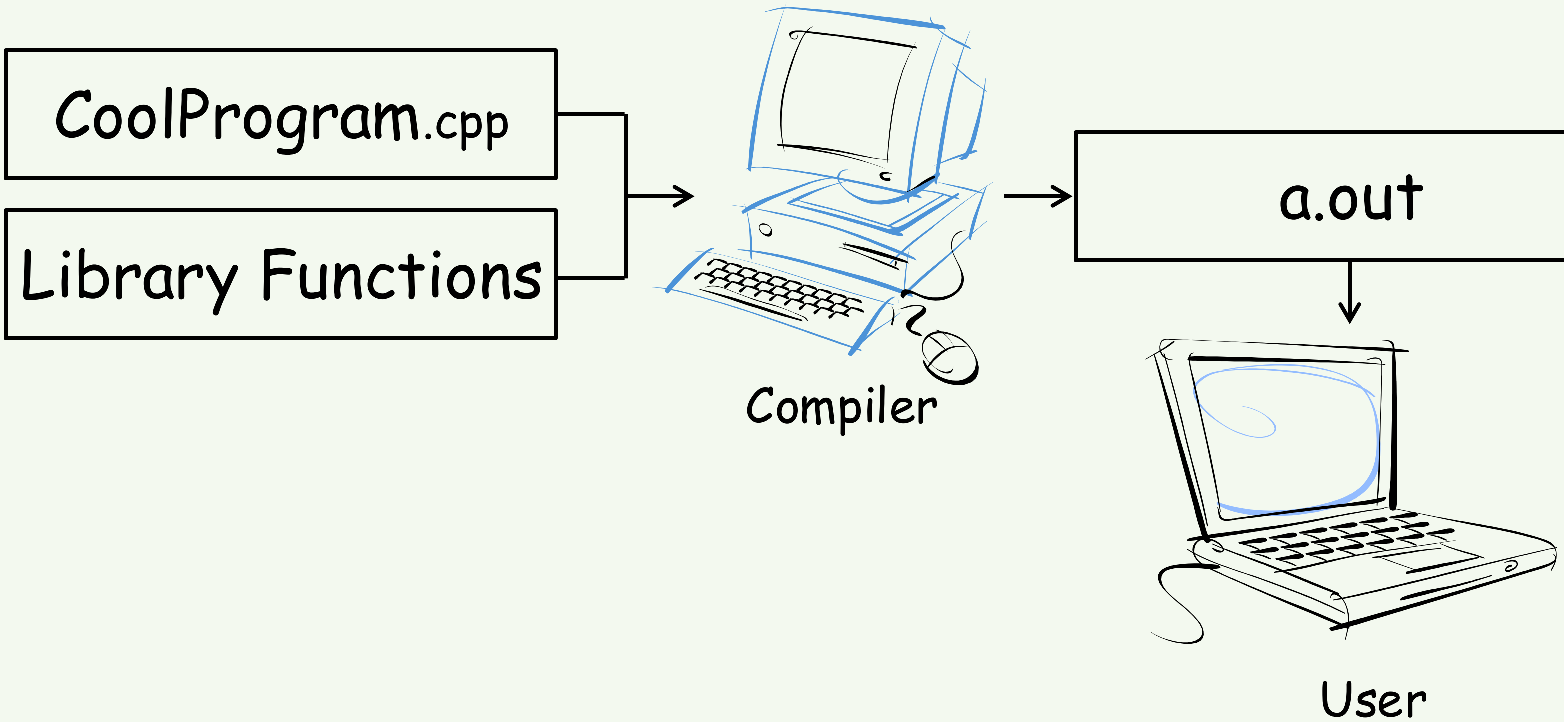
Goals of This Lecture

- 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

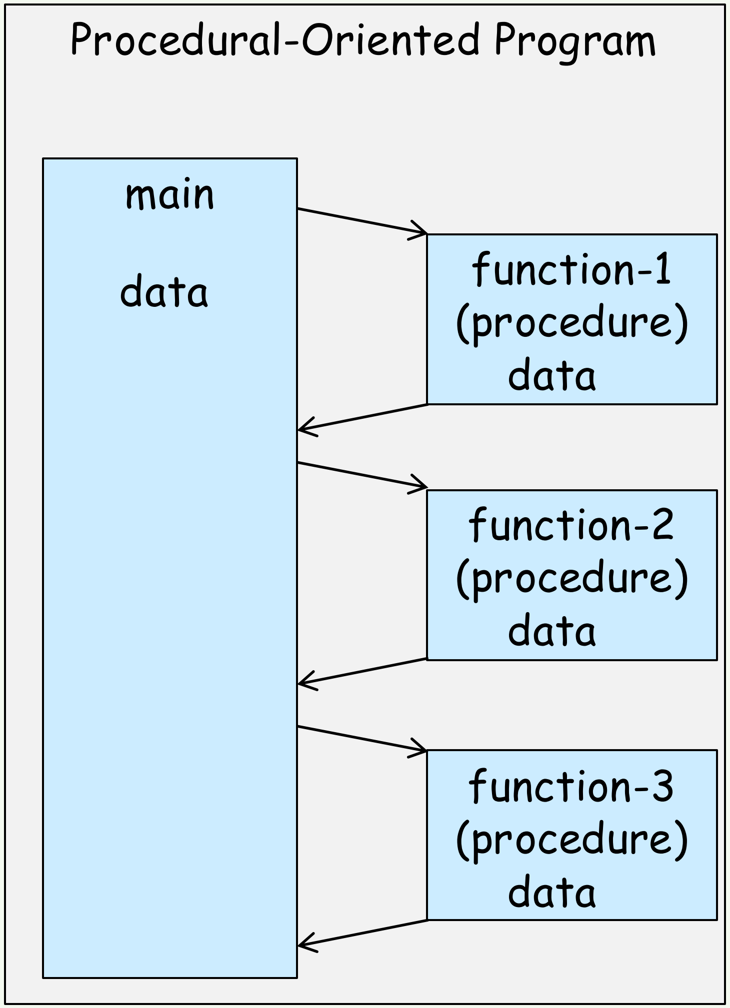
Example of ADT

- 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

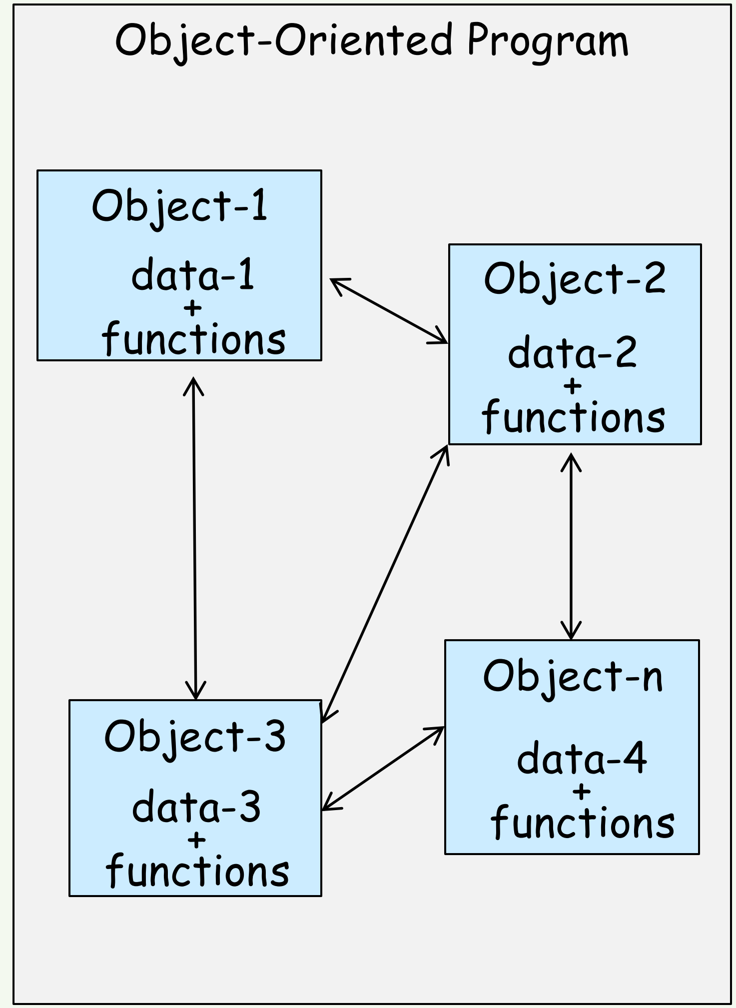
C & C++ in Abstraction View

- 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



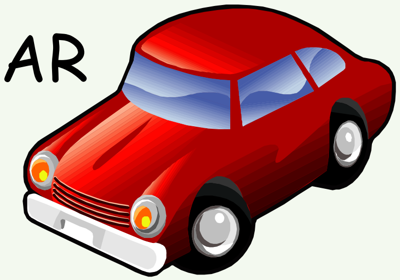
data is open to all functions.



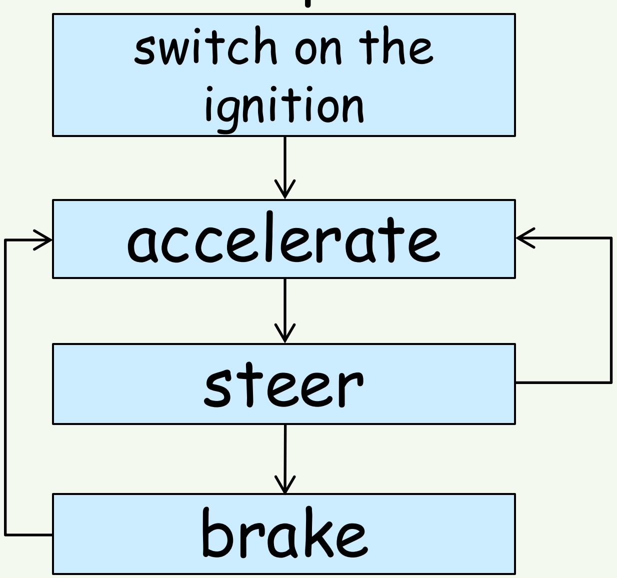
Each data is hidden and associated with an object.

Example: PO VS. OO

CAR

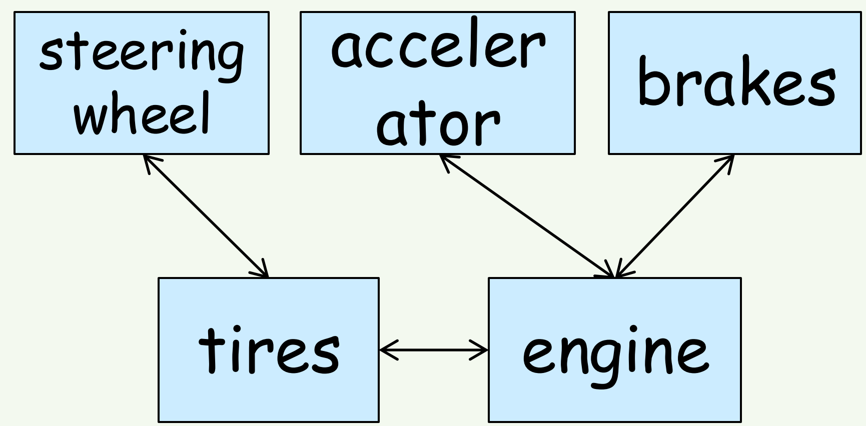


Procedure-oriented View of car operation



Car = a sequence of functions (procedures)

Object-oriented View of car operation

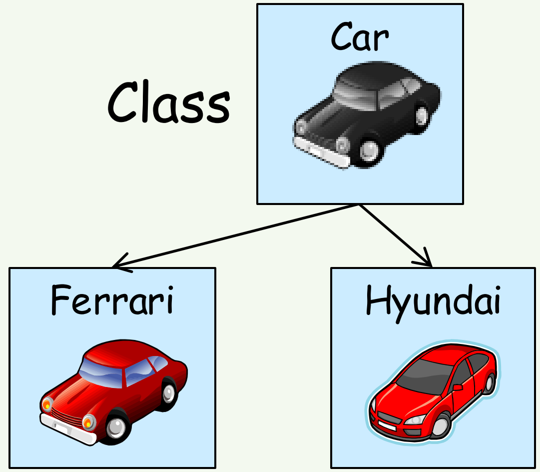


Car = interaction between components (objects)

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

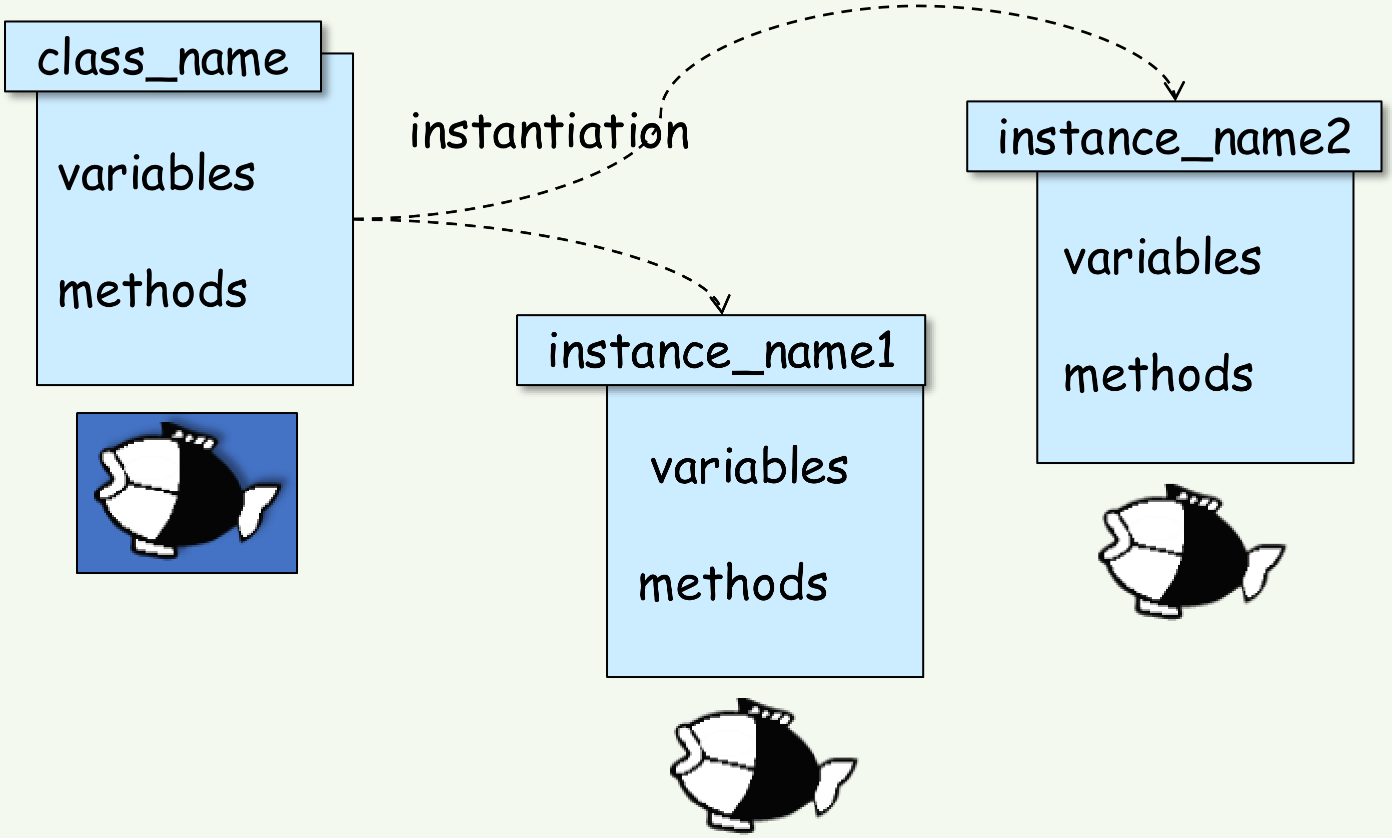
```
struct tag_name {
    type1     member1;
    type2     member2;
    …
    typeN     memberN;
} ;
```

a collection of heterogeneous types

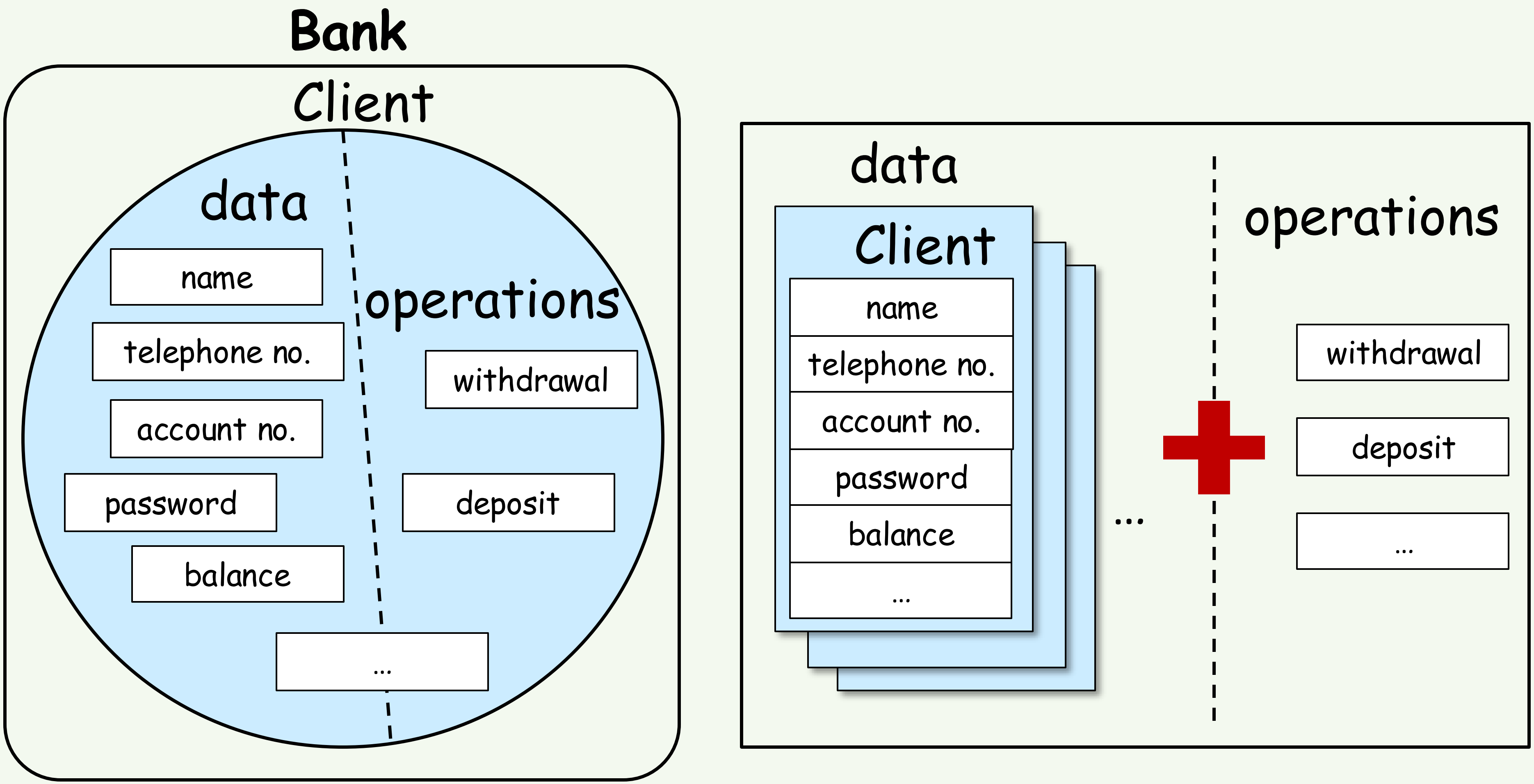
Class Declaration

class_name instance_name1, instance_name2;

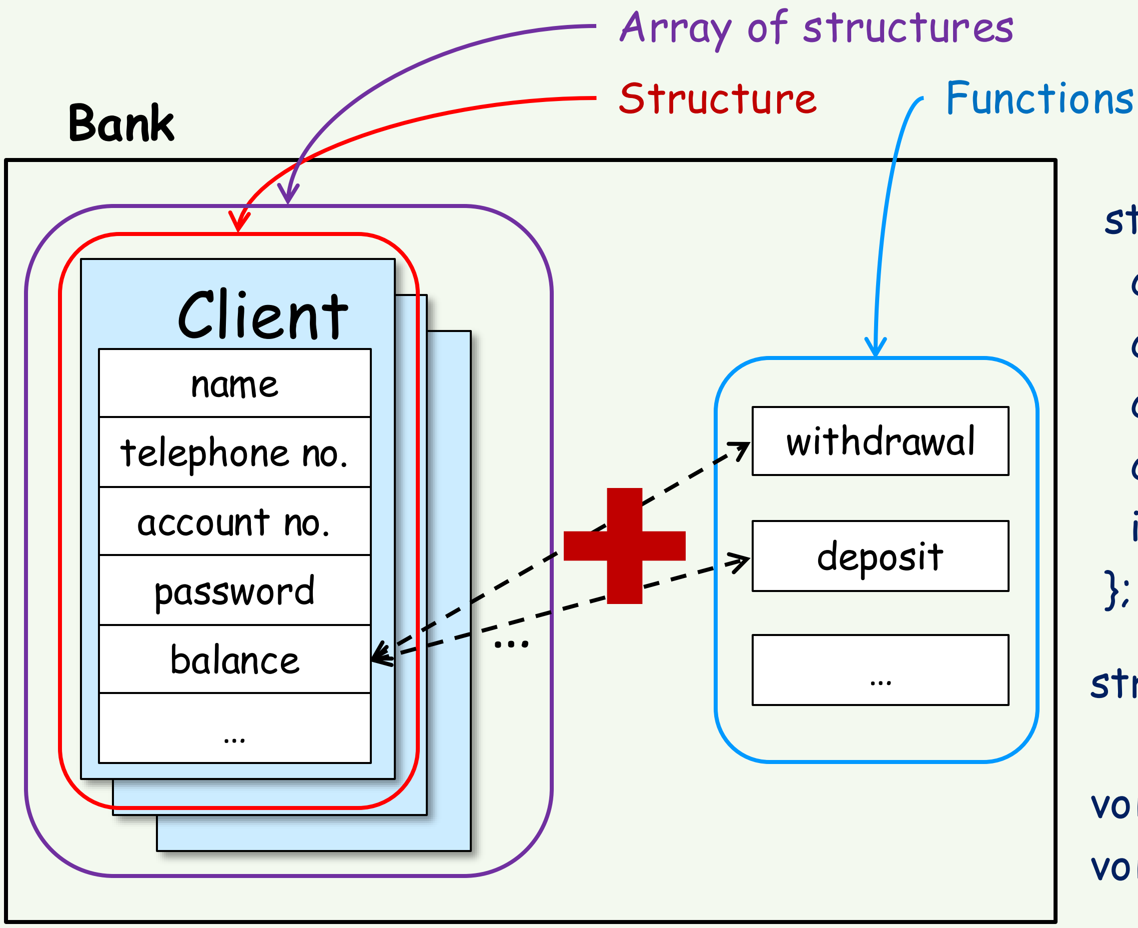
C.f. struct *tag_name struct_variable, … ;*



C Style Design (Procedural) (1/2)



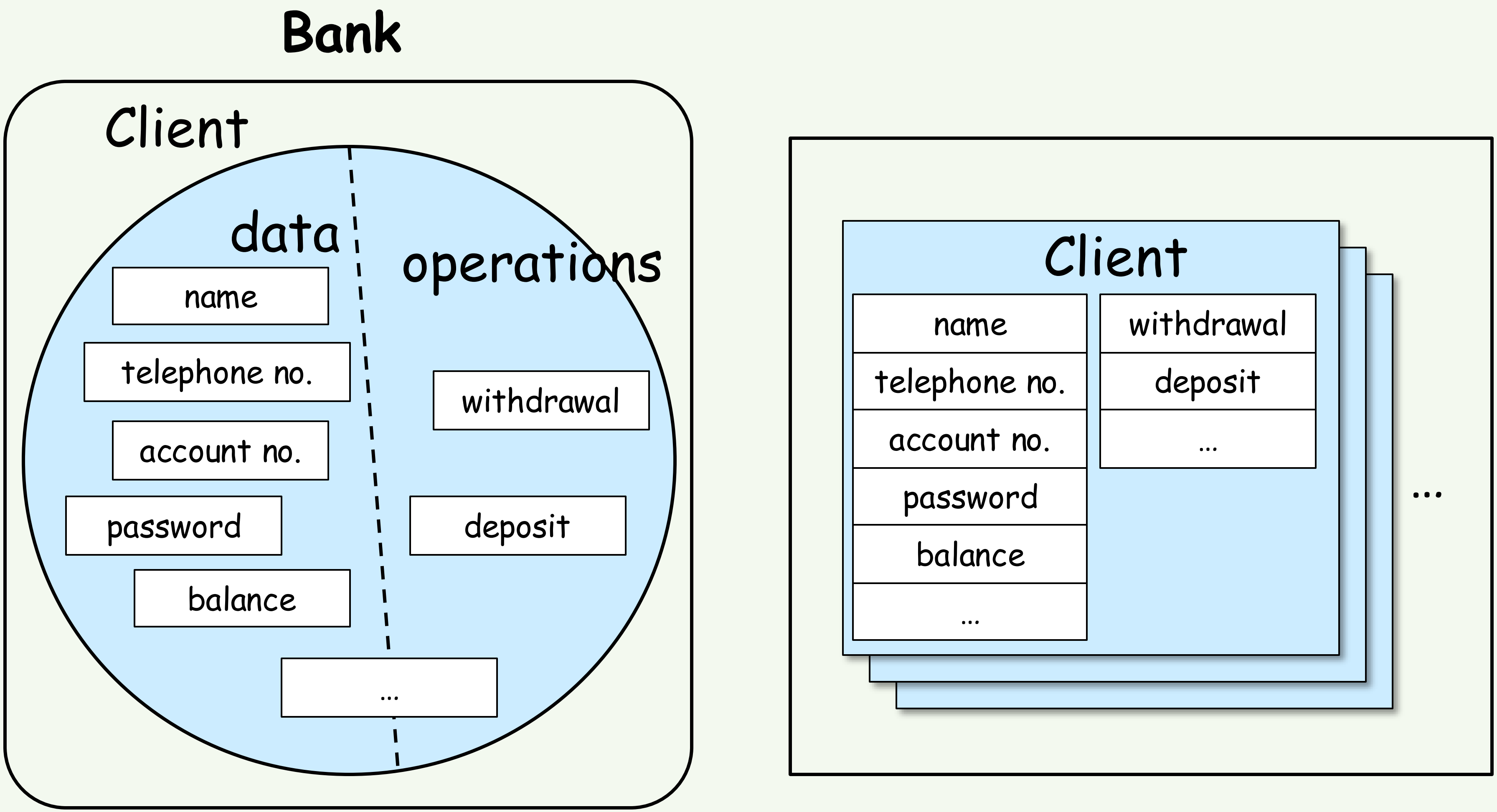
C Style Design (Procedural) (2/2)



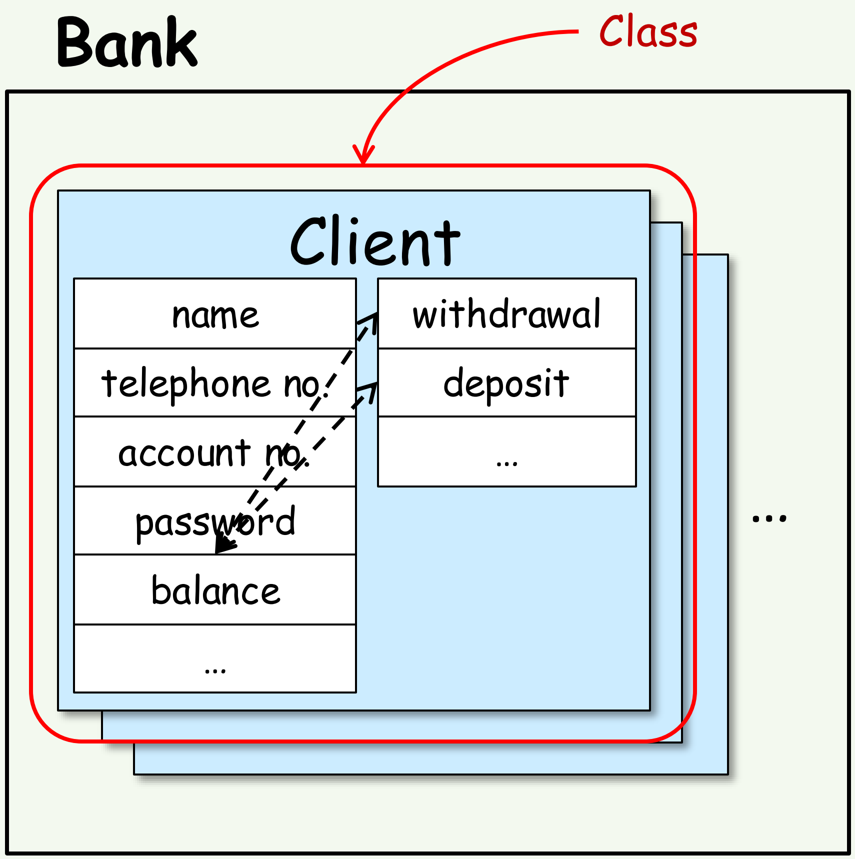

```
struct client {
  char name[MAX];
  char tel[MAX];
  char account[MAX];
  char password[MAX];
  int balance;
};

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

```
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

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

declaration

"record.h"

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;
    }
};
```

member function

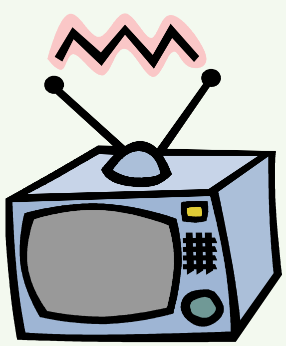
always must reference member variables with instance name

```
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

clients (users) only know interface.

encapsulation

Encapsulation in C++

Class in C++

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

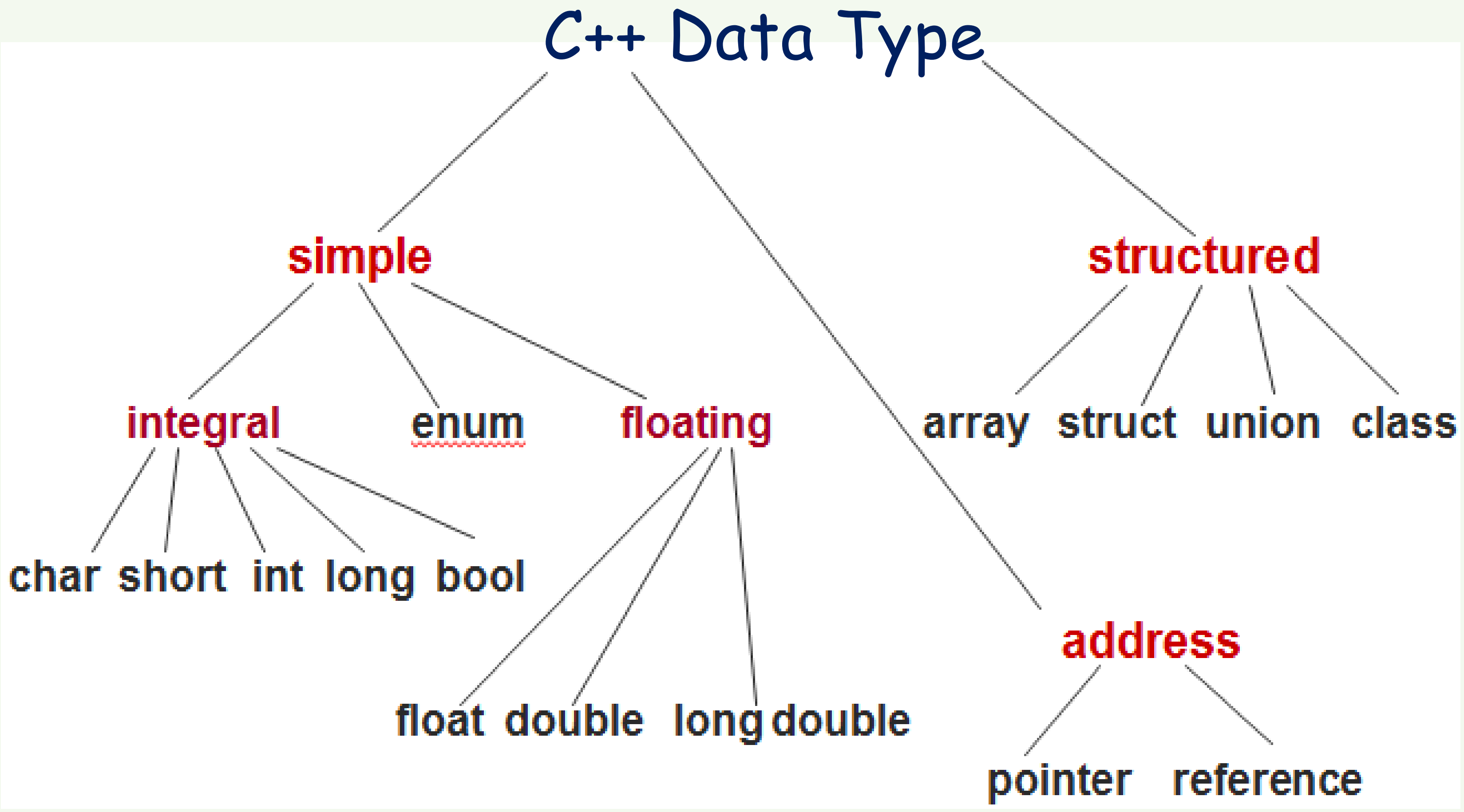
Access specifier

Interfaces : open outside

Encapsulation: hide inside

Basic Features (Mostly same as C)

C++ Data Types


C++ Data Type
simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
address
float
double
long double
pointer
reference

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 = "KAISTT";
  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, Control Flow

Local and Global Variables

- Block
  - Enclosed statements in {…} define a block
  - Can be nested within other blocks
- 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;          // local x hides global x
      x = 1;
      ::x = 2;         // assign to global x

      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 "FunGame"
- 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
```

Control Flow: If Statement

if (<boolean_exp>)

<true_statement>

[else if (<boolean_exp>)

<else_if_statement>]

[else

<else_statement>]

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

int main() {
    int number;

    cout << "Enter a number: ";
    cin >> number;

    if (number > 0) {
        cout << "The number is positive." << endl;
    } else if (number < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }

    return 0;
}
```

Control Flow: Switch Statement

```
char command;
cin >> command;
switch (command) {
    case 'I' :
        editInsert();
        break;
    case 'D' :
        editDelete();
        break;
default :
        cout << "Error\n";
        break;
}
```

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

int main() {
    int n;
    cin >> n;
    switch (n) {
        case 1: cout << "One"; break;
        case 2: cout << "Two"; break;
        default: cout << "Other";
    }
}
```

Control Flow: While & DO-While

while (<boolean_exp>)

<loop_body_statement>

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

int main() {
    int i = 1;
    while (i <= 3) {
        cout << i << endl;
        i++;
    }
}
```

do

<loop_body_statement>

while (<boolean_exp>)

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

int main() {
    int i = 1;
    do {
        cout << i << endl;
        i++;
    } while (i <= 3);
}
```

Control Flow: For Loop

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

<body_statement>

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

int main() {
    for (int i = 1; i <= 5; i++) {
        cout << i << endl;
    }
}
```

Constructor and Destructor

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;
}
```

without supplying an argument
→ Default constructor

Same initializations

∵implicitly called

```
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;
}
```

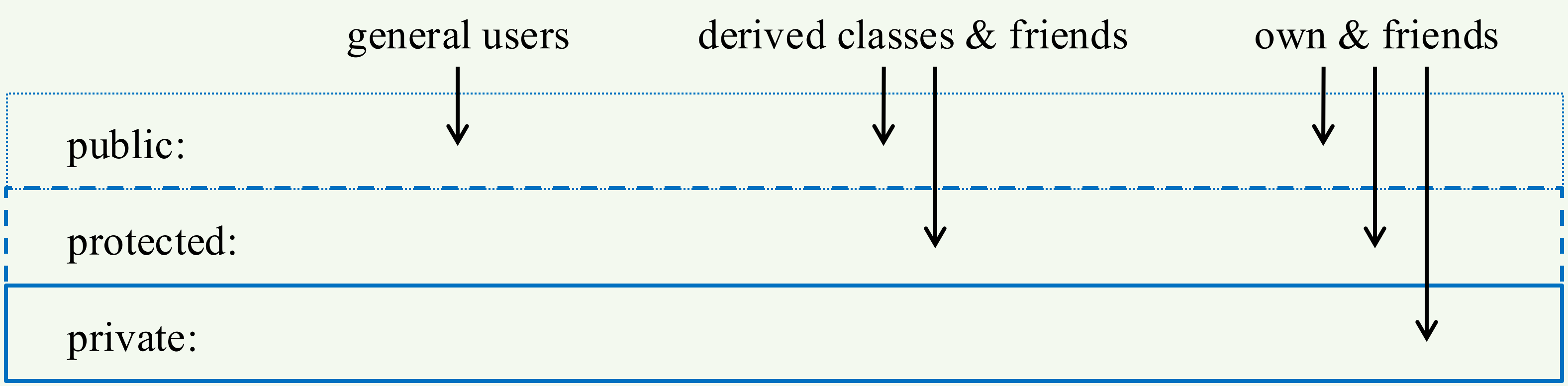
record::~record( ) invoked for myRecord

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

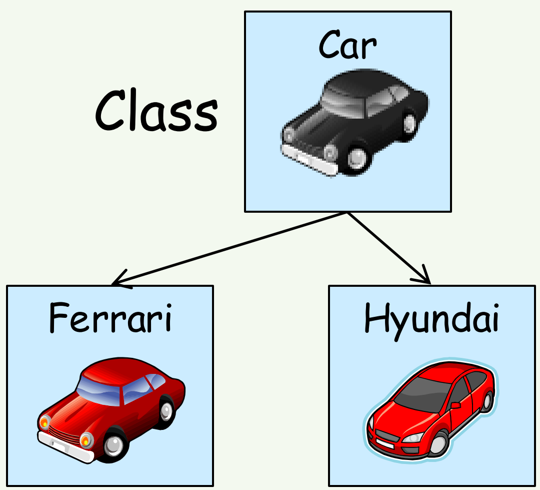
```
void record::calculate_avg( ) {
    int sum = course1 + course2;
    avg = ((double) sum) / 2;
}

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)

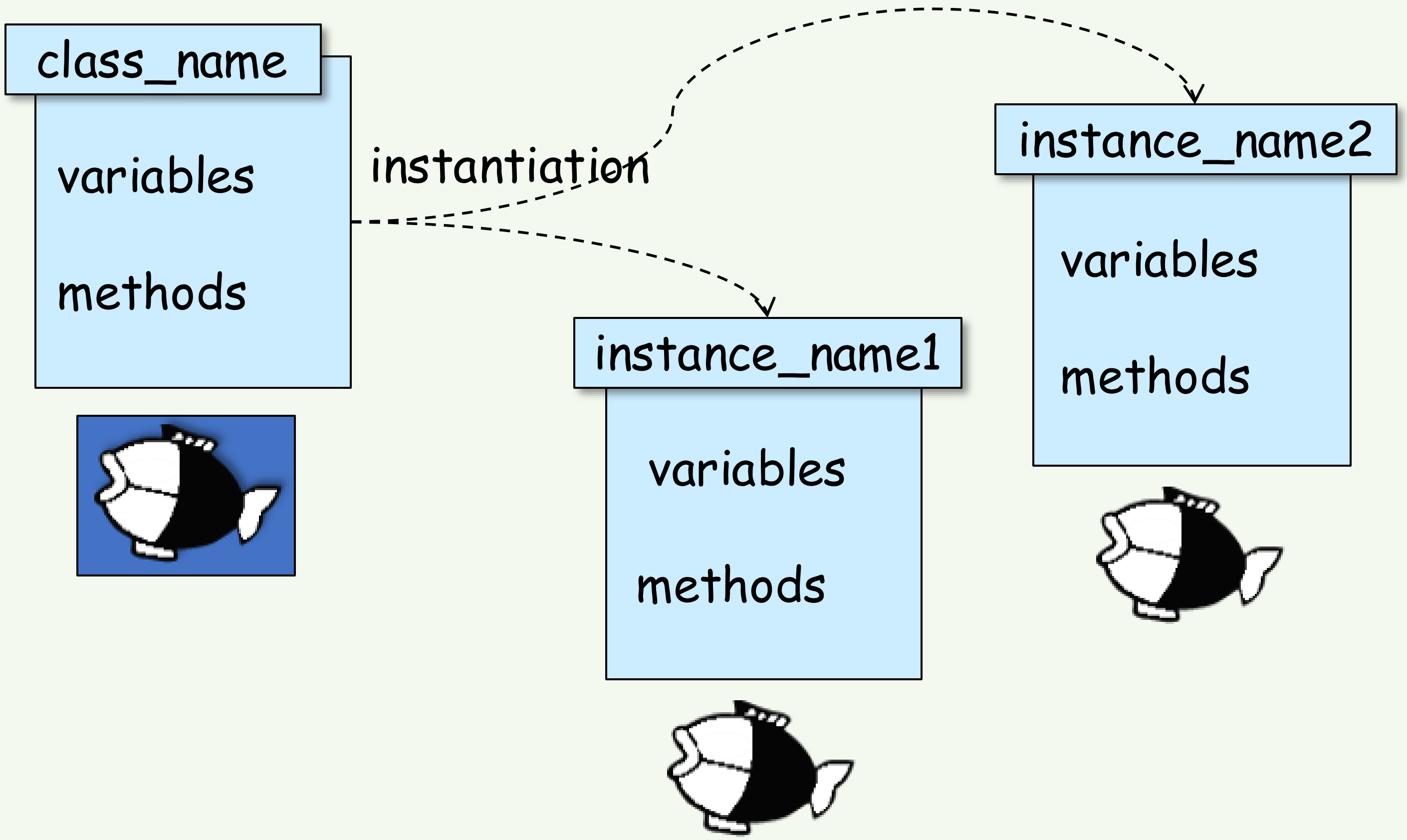
Attributes: color, capacity, max. speed, …

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

Recall: Class Declaration

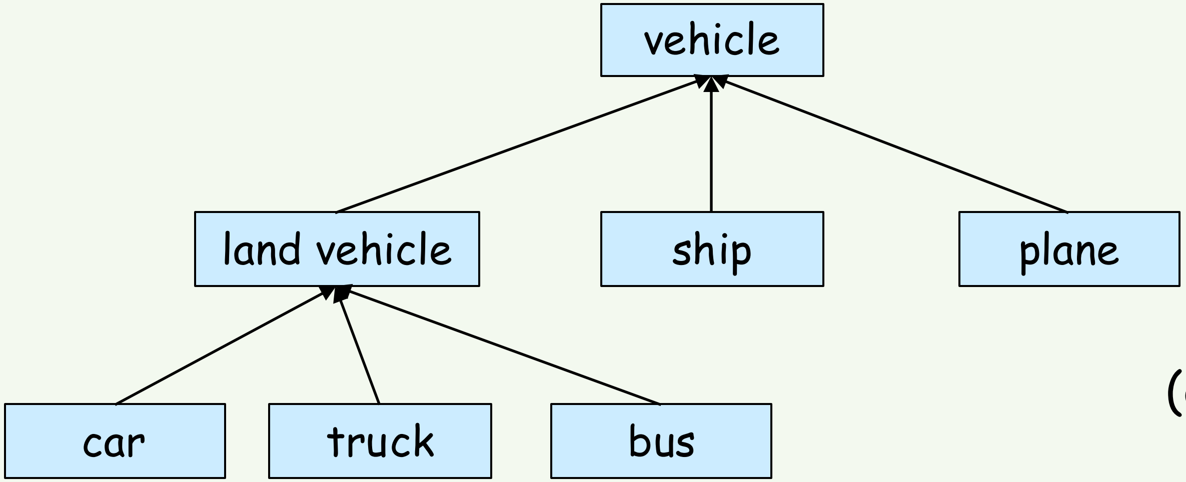
class_name instance_name1, instance_name2;

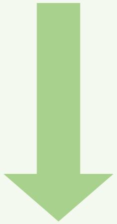
C.f. struct tag_name struct_variable, … ;



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"



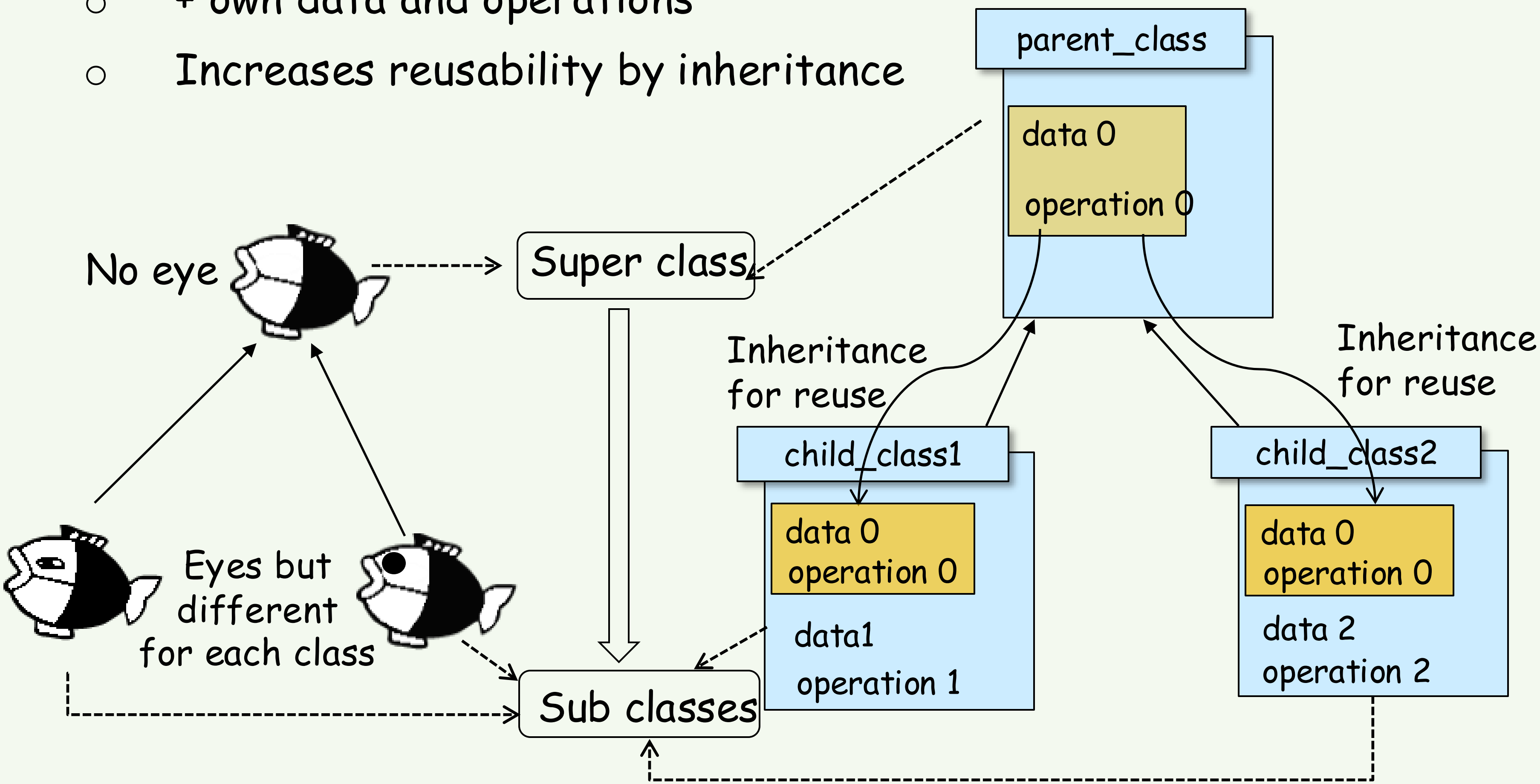


More specialized (overridden, detailed, added)

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();
```

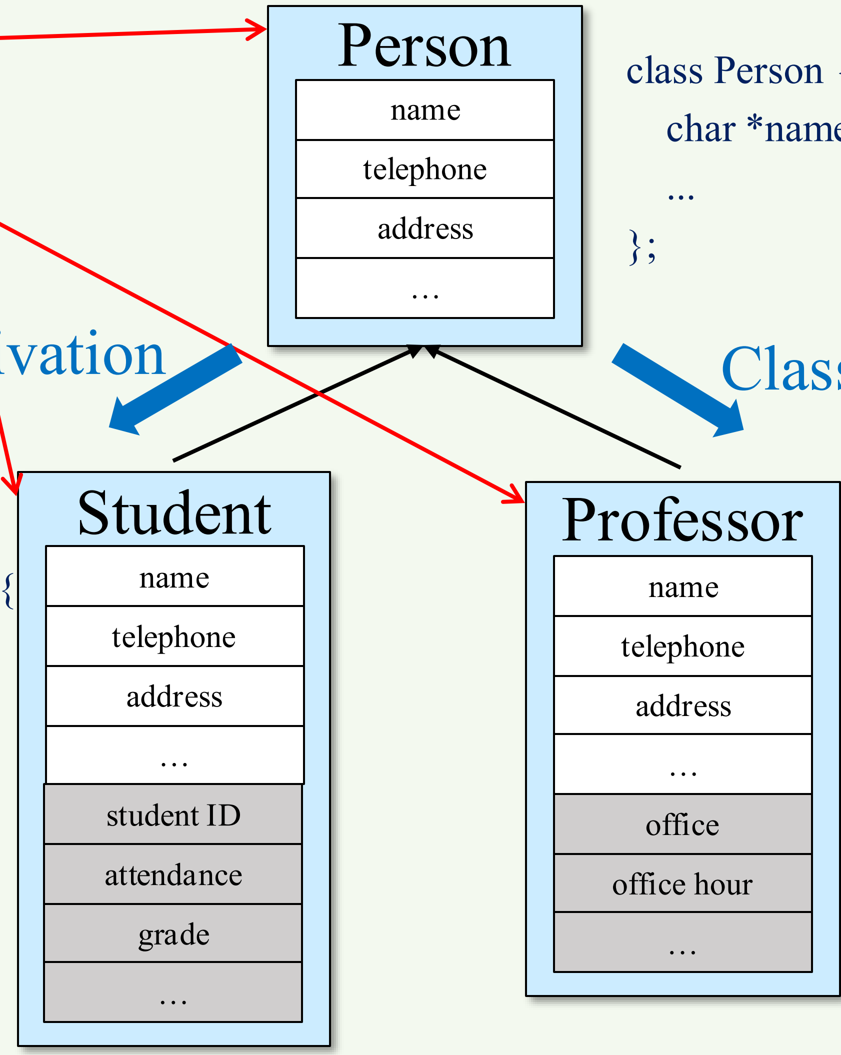
Inheritance: Mechanism for Reuse

Base class
(Parent, Superclass)

Derived class
(Child, subclass)

Class derivation

Class derivation




```
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

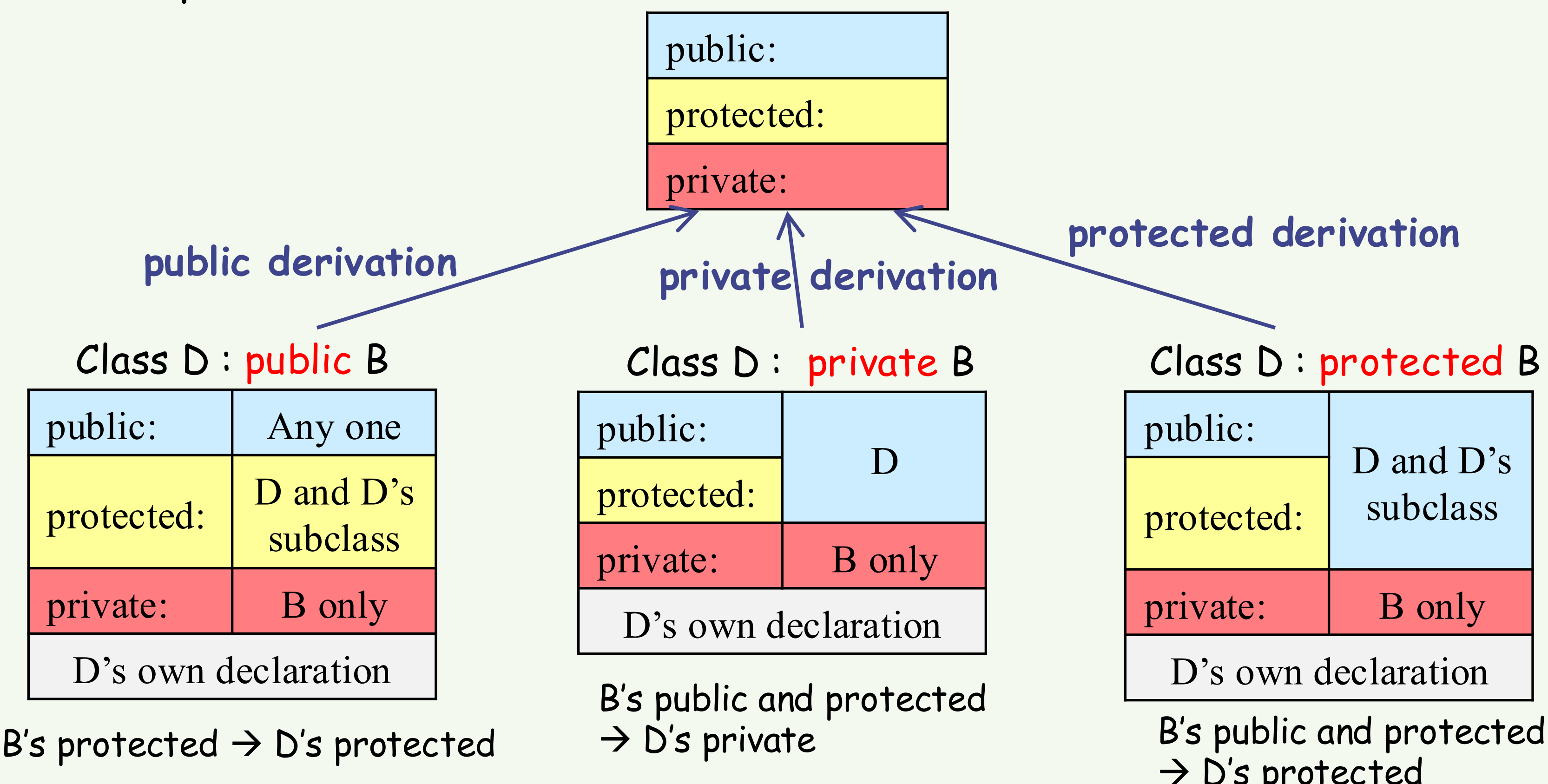
- If a base class has constructors, then a constructor must be invoked
  - Base class acts exactly like a member of the derived class in the constructor
    - base class' constructor is invoked in Member initialization list
  - Default constructors can be invoked implicitly

- A constructor of derived class can specify initializers for its own members and immediate bases only
  - Cannot directly initialize members of a base class

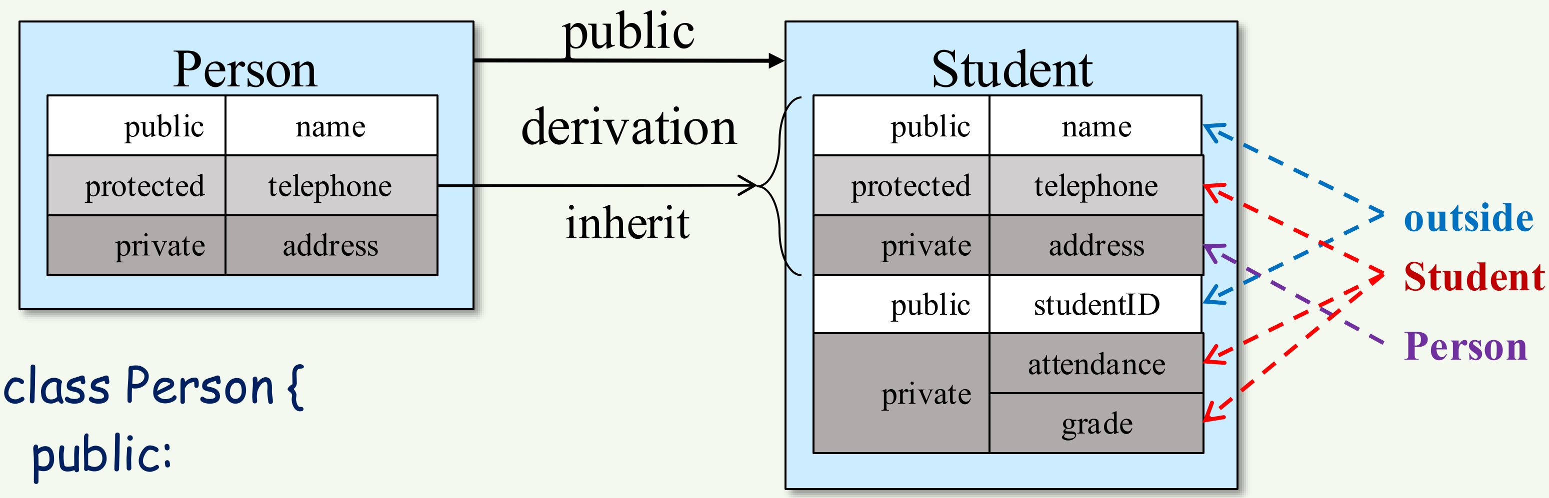
Access to Base Classes

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

Class B



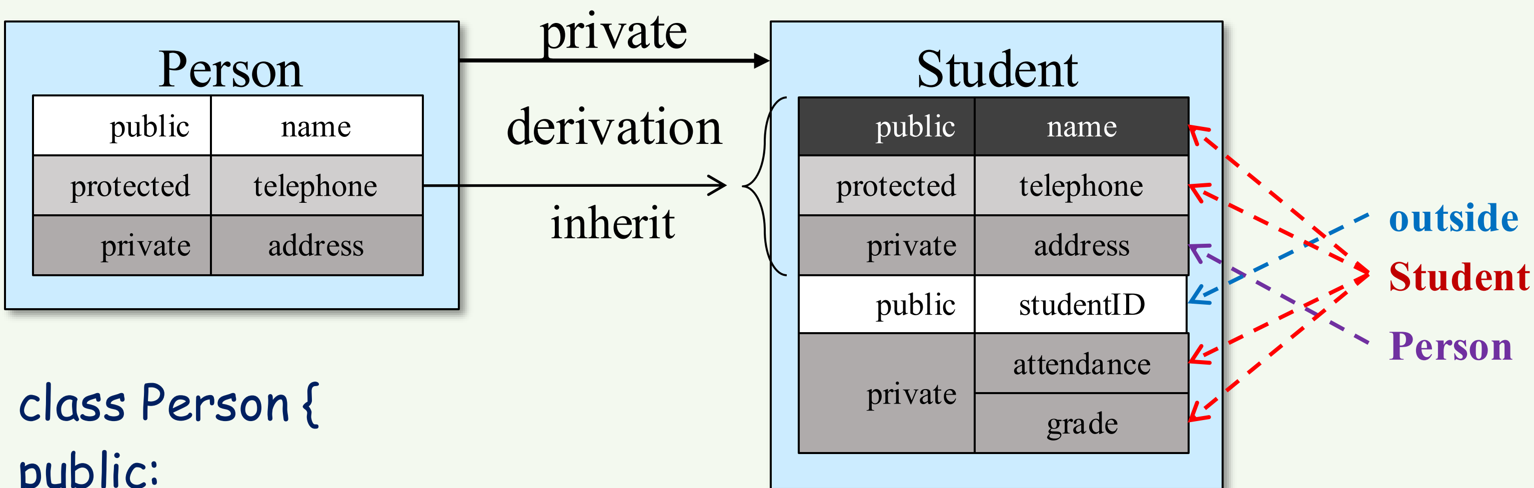
Public Derivation




```
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

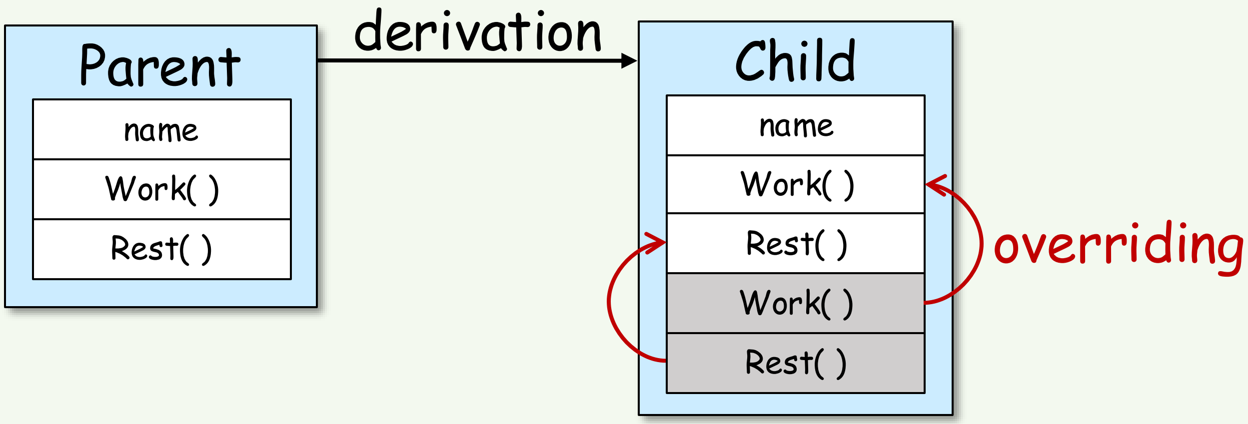
```
#include<iostream>
using namespace std;
class Parent {
  char *_lastname;
public:
  char *_name;
  char* lastname() { return _lastname; }
  char* name() { return _name; }
  Parent(char *name = "",
           char *lastname = "");
  ~Parent() { delete _name, _lastname; }
};

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

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.
```

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", "J
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()
```