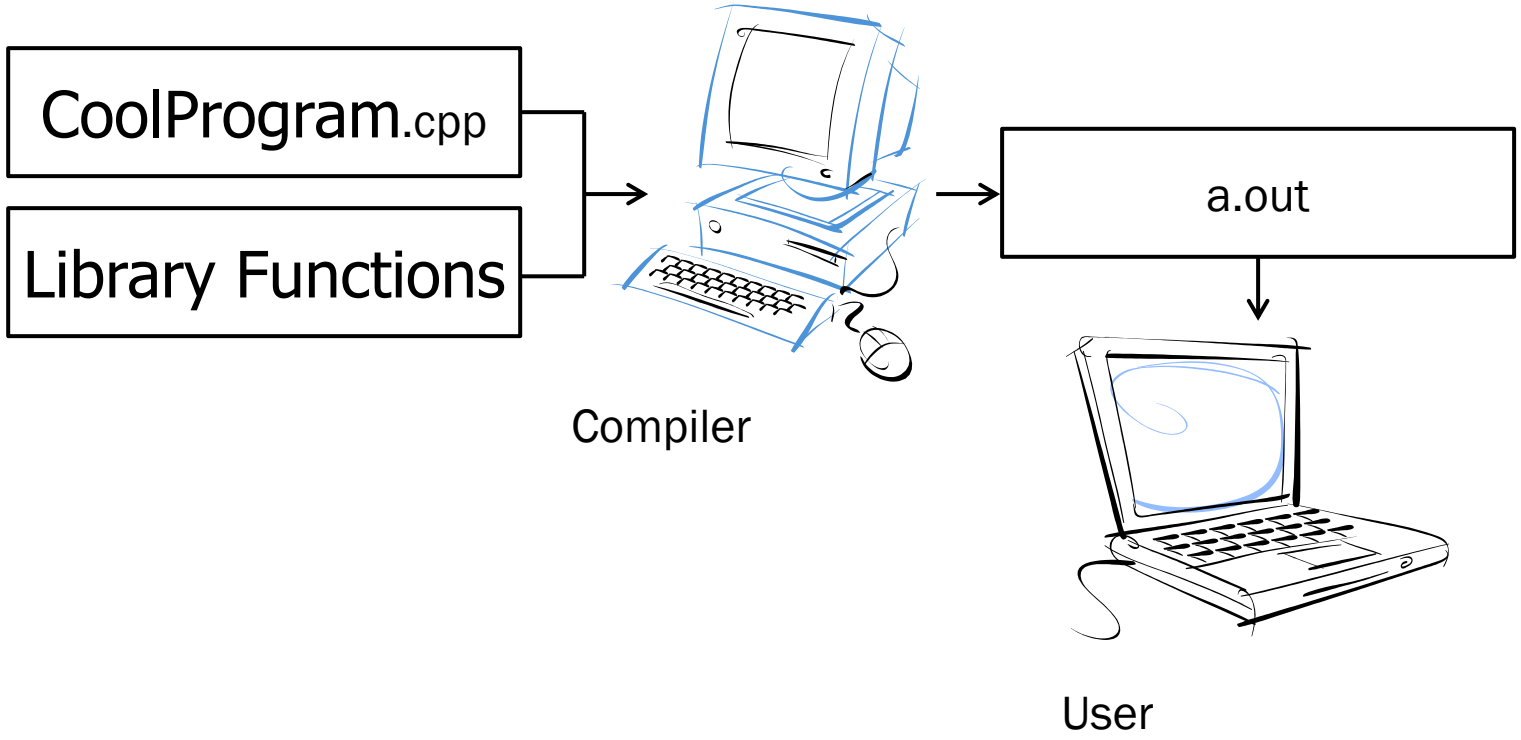
DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

BS (AI)

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.

Example of ADT

Example: ADT modeling a simple stock trading system

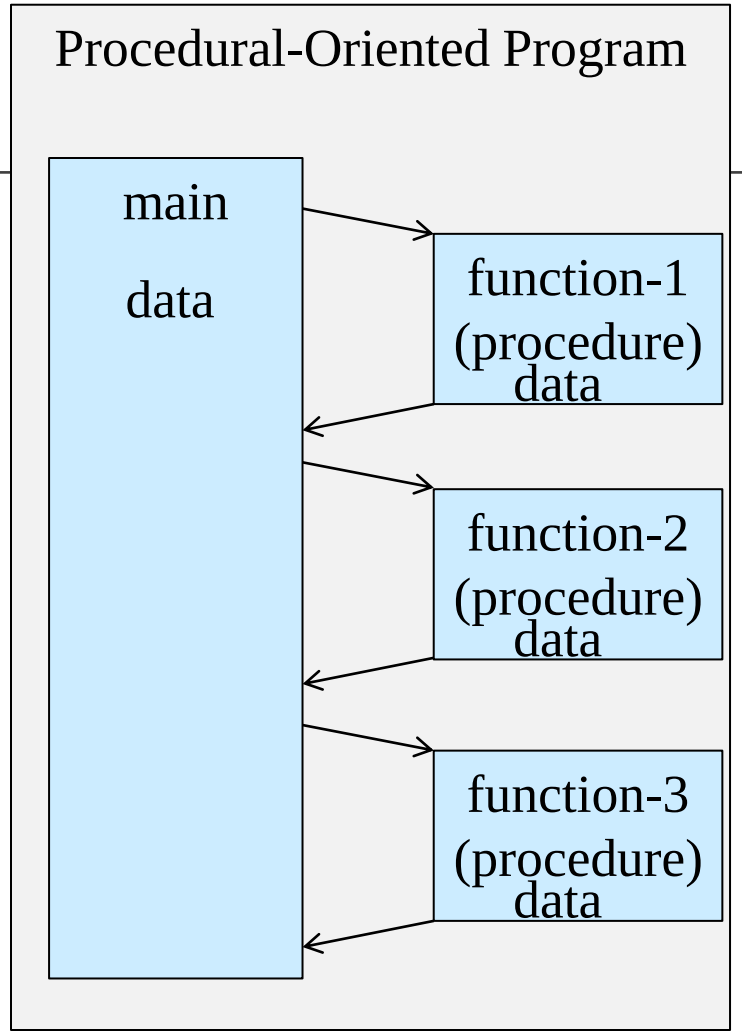
- The data stored are buy/sell orders
- The operations supported are
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

C++ in Abstraction View

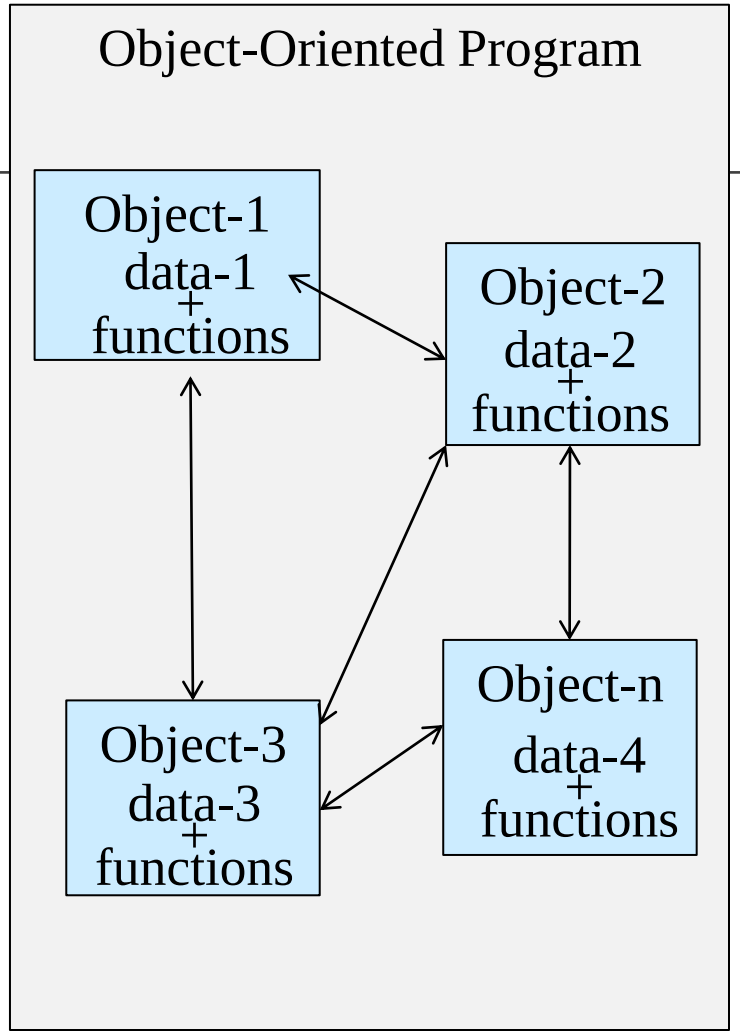
C++ supports Object-Oriented programming

- 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.

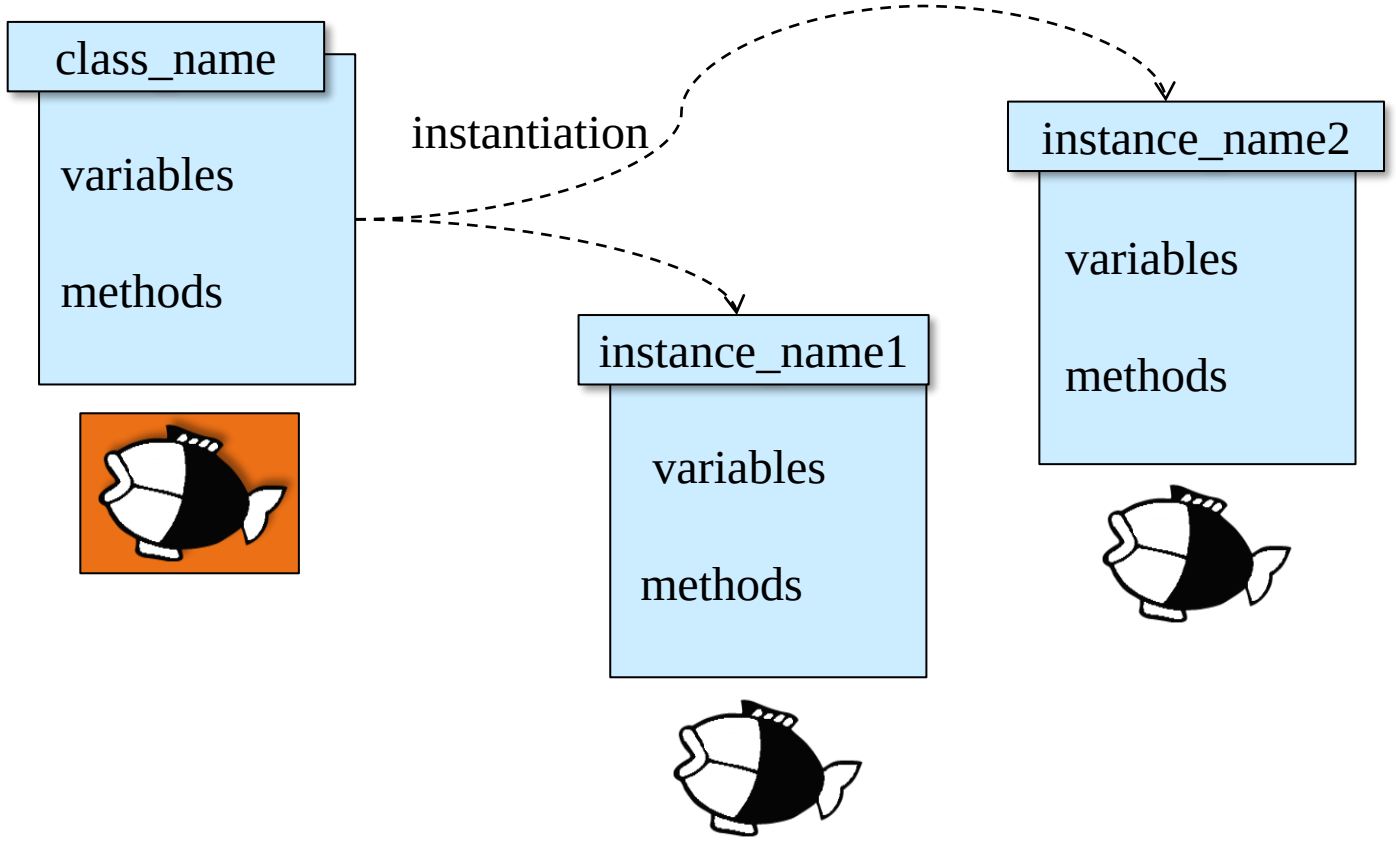
C++ Classes

Similar to structure in C

Class in C++	Structure in C
class class_name { *public:* *// member variables* *int a, b, c;* *…* *// member methods (functions)* *void print(void);* *…* *} ;*	struct *tag_name* { *type1 member1;* *type2 member2;* *…* *typeN memberN;* } ;
a collection of types and associated functions	a collection of heterogeneous types

Class Declaration

class_name *instance_name1, instance_name2;*



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

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

declaration
definition : always after declaration

"record.h"

"record.cpp"

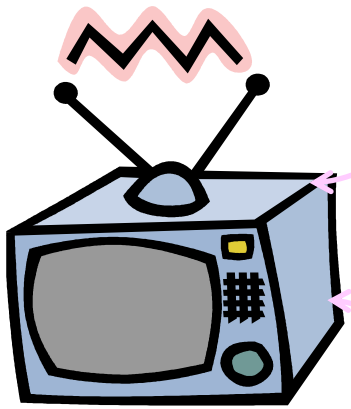
```
void record::print(void) {
    cout << name << endl;
    cout << "course1 = " << course1
        << ", course2 = " << course2 << endl;
    cout << "avg = " << avg << endl;
}
```

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

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



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

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
```

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

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

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;
}
```

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);
};
```

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

record::~record( ) invoked for myRecord

always in "public"

must not specify a return type

Destructor

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

Credits and Acknowledgements

These slides are largely borrowed from Prof. Yung Yi, KAIST, South Korea.