

---

# **COMP2014**

# **Object-Oriented Programming**

**Dr Alex Dong**

**School of Computer, Data and Mathematical Sciences**

**Western Sydney University**

---

**Part A**

**Unit Overview**

# General Information

---

- ◆ COMP2014 Object-Oriented Programming
  - Level 2 (1<sup>st</sup> year 2<sup>nd</sup> semester)
  - 10 Credit Points
- ◆ Prerequisites
  - COMP1005 Programming Fundamentals or equivalent
- ◆ Assumed Knowledge
  - Basic knowledge in programming fundamentals, including flow of control, one-dimensional array, functions, I/O operations
  - Able to write simple programs in Java, C or C++
- ◆ Contact details

**Dr Alex Dong**

Zoom: 607 668 1320 (passcode **COMP2014** if required)

Email: [A.Dong@westernsydney.edu.au](mailto:A.Dong@westernsydney.edu.au)

# Teaching team

---

- ◆ Unit Coordinator and lecturer: Dr **Alex Dong**
  - Lectures: Tuesday 2pm – 4pm
  - Zoom id: 897 4041 6917
  - Email: [A.Dong@westernsydney.edu.au](mailto:A.Dong@westernsydney.edu.au)
- ◆ Tutor 1: Dr **Alex Dong**
  - Practical classes: all classes in Parramatta South
  - Zoom id: 607 668 1320
  - Email: [A.Dong@westernsydney.edu.au](mailto:A.Dong@westernsydney.edu.au)
- ◆ Tutor 2: Mr **Vishal Patel**
  - Practical classes: all classes in Kingswood
  - Zoom id: 802 327 2508
  - Email: [V.Patel2@westernsydney.edu.au](mailto:V.Patel2@westernsydney.edu.au)

# Brief summary of the unit

---

This unit presents the concepts and principles of programming with the emphasis on object-oriented paradigm. It addresses the importance of the separation of behavior and implementation, as well as effective use of *abstraction, data hiding, encapsulation, inheritance* and *polymorphism*.

The teaching language of this unit will be C++ but Java is allowed for assignment 2.

# Presentation

- ◆ 4 Hours per week:
  - Two-hour lecture per week
  - Two-hour practical per week from Week 2
- ◆ At least **6 hours** self-study per week
- ◆ Lecture sessions: scheduled for **13 weeks** lectures
- ◆ Practical: scheduled for **12 practical sessions** (attendance is compulsory)
- ◆ Feedback giving and taking
  - Two online questionnaires (1 bonus mark each)
  - Face-to-face checking for practical and assignments
- ◆ PASS: Peer Assistant Study Sessions
  - Voluntary group work for students to help each other building up programming skills
  - Lead by Alex Dong

# Unit Schedule

Week	Topics	Readings	Tutorial and Practical
1	Unit Introduction C++ Basics	UO, LG & Ch1-2	Online survey 1
2	Function calls & parameter passing	Ch3-4	Practical 1
3	Arrays (multi-dimensional arrays)	Ch5	Practical 2
4	Classes & data abstraction	Ch6	Practical 3
5	Class design & implementation	Ch7	Practical 4 Online Tutorial 1
6	Pointers & dynamic arrays	Ch10	Practical 5
7	Strings, Streams, Static Variables & file I/O	Ch7,9&12	Assignment 1 due

# Unit Schedule

Week	Topics	Readings	Tutorials and Practicals
8	Middle break		
9	Inheritance	Ch14	Assignment 1 marking Online Tutorial 2
10	Polymorphism & virtual functions	Ch15	Practical 6
11	Operator overloading & references	Ch8	Practical 7
12	Class template & linked data structures	Ch16&17	Practical 7 (ctn) Assignment 2 due
13	Standard template library	Ch19	Online Tutorial 3
14	Unit review		Online survey 2

# Mandatory components

---

## Mandatory components:

- ◆ Attending all 12 practical sessions (face-to-face in campus or via zoom)
  - 10 sessions with attendance checking
  - 2 sessions for assignment demonstration without attendance checking
- ◆ Attempting all 3 online tutorials
  - Online quizzes are checked by vUWS system automatically
- ◆ Attempting and demonstrating 2 assignments
- ◆ Attending the final exam:

Note that all practical tasks and assignments will be individually checked during the practical sessions. No marks will be given based on email or vUWS submissions. If your tutor does not get time to check your work at the class of the due date, you need to contract the tutor via email on that date to arrange another time.

# Assessments

---

Four assessment components:

- ◆ Assignment 1 (15%): due at 5pm Friday 9 September 2022
- ◆ Assignment 2 (15%): due at 5pm Friday 28 October 2022
- ◆ Practical and tutorials (20%) : FNS if attendance is less than seven exclusive two assignment checking sessions.
  - Online tutorial performance (40-60 true/false, multiple choice or very confusing multiple answer questions in three sessions): 6%
  - Practical performance: 14%
- ◆ Final exam (50%): two-hour, open-book
- ◆ Online survey participation: 2 bonus marks (no bonus marks if your continuous assessment marks have reached 50)

# Assessments

---

- ◆ To receive a pass grade, you must achieve 50% of overall marks and **complete all mandatory components**.
- ◆ Assignment submissions and demonstration
  - **Code:** You are required to submit your code and declaration to vUWS by the deadlines.
  - **Demonstration:** Your demonstration will be at your practical session of the week immediately after the submission deadlines.

# Textbook: new version



## Absolute C++, Global Edition (6e)

Walter Savitch

University of California, San Diego

...more...



Book + Access Code

\$117.95

Add to Cart

Add to Shortlist

**Edition**

6th

**ISBN**

9781292098593

**ISBN 10**

1292098597

**Published**

25/05/2016

**Published**

Pearson Higher Ed USA

**by**

**Pages**

1008

Available once published

Questions

Absolute C++, Global Edition (6e) : 9781292098593

► I'd like to request an inspection copy

---

## Part B

# C++ Basics for non-C++ Programmers

# Topics covered by this part

---

- ◆ Structure of C++ programs
- ◆ Input and output
- ◆ Flow of control
  - Selection
  - Repetition
- ◆ Random numbers

# Compare to Java code

```
import java.util.Scanner;

public class Hello {

    public static void main(String[] args)
    {
        String name;
        System.out.println("Enter your name:");
        Scanner input = new Scanner(System.in);
        name = input.nextLine();
        System.out.println("Hello " + name+"!");
    }
}
```

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

int main()
{
    string name;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello " << name << endl;
    return 0;
}
```

Download code: Hello.java

Download code: Hello.cpp

# A Simple C++ Program

Hello.cpp

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

int main()
{
    string name;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello " << name << endl;
    return 0;
}
```

file name of source code

Include directives

Namespace declaration

Mainline function name

Variables declaration

Input statement

Output statement

# The Structure of a C++ Program

---

- ◆ A C++ program consists of:
  - Pre-processor directives (introduced by `#`, similar to `import` in Java)
  - Declarations (`class`, `function`, `variable` etc.)
  - Definitions/implementation (class members, functions, variables)
- ◆ Different from Java, a C++ program allows free functions (not a method of any class), including the `main()` , function
- ◆ There must be a file with extension `.cpp` which contains the `main()` function as the executable point.

# Basic C++ Keywords

<b>bool</b>	<b>if</b>	<b>while</b>	<b>main</b>	<b>true</b>
<b>short</b>	<b>else</b>	<b>do</b>	<b>private</b>	<b>false</b>
<b>int</b>	<b>switch</b>	<b>continue</b>	<b>protected</b>	
<b>long</b>	<b>case</b>	<b>break</b>	<b>public</b>	
<b>float</b>	<b>default</b>	<b>for</b>	<b>const</b>	
<b>double</b>			<b>static</b>	
<b>char</b>			<b>return</b>	
<b>void</b>				

String is not a built-in data type in C++  
bool is called “boolean” in Java

# Input and Output Statements

```
/* This program shows how to input and output  
data. */  
  
#include <iostream>  
#include <string>  
  
using namespace std;  
  
int main() {  
  
    string name, id;  
    int age;  
  
    cout << "Input your name:\n";  
    cin >> name;  
  
    cout << "Input your student ID:\n";  
    cin >> id;  
  
    cout << "Input your age:\n";  
    cin >> age;  
  
    cout << "\nMy name is " << name << ". " << endl;  
    cout << "My student ID is " << id << ". " << endl;  
    cout << "I am " << age << " year old." << endl;  
  
    return 0;  
}
```

comments

Include string handling library

Using standard namespace

# Arithmetic Expressions

volume.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    const double Pi = 3.14159;
    double radius, volume;

    cout << "Please enter the radius of a sphere: ";
    cin >> radius;

    volume = 4.0/3.0 * Pi *pow(radius, 3.0);
    cout << "The volume of the sphere is: \n"
        << volume;
    return 0;
}
```

Include math library,  
which support the  
basic math fuctions

A constant which  
value can't change  
in the program.

Arithmetic  
expression

# Selection Structures (if...else)

```
#include<iostream>
using namespace std;
int main() {
    int num;

    cout << "Input an integer";
    cin >> num;
    if (num > 0)
        cout << "Your input is a positive number.";
        cout << "which is " << num;
    else
        cout << "Your input is a non-positive number.";
        cout << "which is " << num;
    return 0;
}
```

Error message:  
*Misplaced else*

# Selection Structures (if...else)

```
#include<iostream>
using namespace std;
int main() {
    int num;

    cout << "Input an integer";
    cin >> num;
    if (num > 0)
    {
        cout << "Your input is a positive number.";
        cout << "which is " << num;
    }
    else
    {
        cout << "Your input is a non-positive number.";
        cout << "which is " << num;
    }
    return 0;
}
```

# Operators

## Relational Operators

<	less than
<=	less than or equal
==	equal
>	greater than
>=	greater than or equal
!=	not equal

Examples:

```
if (hour < 0 || hour > 24)  
    cout << "Incorrect Input";
```

```
if (hour >= 0 && hour <= 24)  
    cout << "Correct Input";
```

## Boolean Operators

&&  
||  
!

Meaning  
AND  
OR  
NOT

# Repetition Structures

---

## Form 1

```
while (condition)
{
    statement list;
}
```

## Form 3

```
for (initialization; test expression; update)
{
    statement list;
}
```

## Form 2

```
do
{
    statement list;
} while (condition)
```

# for loop

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

int main() {

    int num, total;
    total = 0;

    for (int counter = 1; counter <= 10; counter++)
    {
        cin >> num;
        total = total + num;
    }

    cout << total;
    return 0;
}
```

Initialization

Test expression

Update

# Random Number Generator

---

- ◆ Random numbers are widely used in simulations, games, special algorithms and so on.
- ◆ Call function `rand()` to get a random number
  - The function takes no argument
  - It returns an integer between 0 & `RAND_MAX`
  - `RAND_MAX` is a system constant, value relies on the C++ compiler you use.
- ◆ Scaling
  - Squeezes random number into smaller range, e.g.  
`rand() % 6`: *returns random value between 0 and 5*
  - `%` is modulus operator (remainder)
- ◆ Shifting
  - Shifts range between 1 & 6 (e.g., die roll)  
`rand() % 6 + 1`: *returns random value between 1 and 6*

# Random Number Seed

---

- ◆ Pseudorandom numbers
    - Calls to *rand()* producing a "sequence" of pre-produced numbers. Different time you call generates the same sequence of numbers.
  - ◆ Use function *srand()* to alter the starting point of the sequence  
`srand(seed_value);`
    - The "*seed value*" sets the start point of the random sequence, normally use `srand(time(0))` to avoid the same sequence of numbers each time you run your program
    - Note that `time()` returns system time as numeric value. You might need to include the C standard library, which contains `time()` function if it is not default
- `#include <cstdlib>`

# Random Examples

---

- ◆ Random **int** between 1 & 100:

`rand() % 100 + 1`

- ◆ Random **int** between 10 & 20:

`rand() % 11 + 10`

- ◆ Random **double** between 0.0 & 1.0:

`(RAND_MAX - rand())/(double)(RAND_MAX)`

- Type casting (**double**) is used to force double-precision division

# IDE for C++

---

- ◆ You may use any IDE that is workable at your computer
- ◆ Typical IDEs:
  - Visual Studio Community: for Windows, come with compiler
  - xCode: for Mac OS, come with compiler
  - Visual Code: for both Windows and Mac OS, you need to install a compiler firstly
- ◆ compiler can be anything workable with your IDE, typically **gcc** or **g++**

# Homework

---

- ◆ Read textbook: Chapters 1&2
- ◆ Complete the online questionnaire in vUWS. It will be turned off at the end of the week.
- ◆ Practical sessions start next week. Complete the practical tasks for week 2 at home and connect to your tutor's zoom at the time of your scheduled class.
- ◆ Warm up textbook Chapters 3-4.
- ◆ If you have not registered to a practical session, please do it as soon as possible. Contact your campus tutor or me for any questions.

# COMP 2014 Object Oriented Programming

---

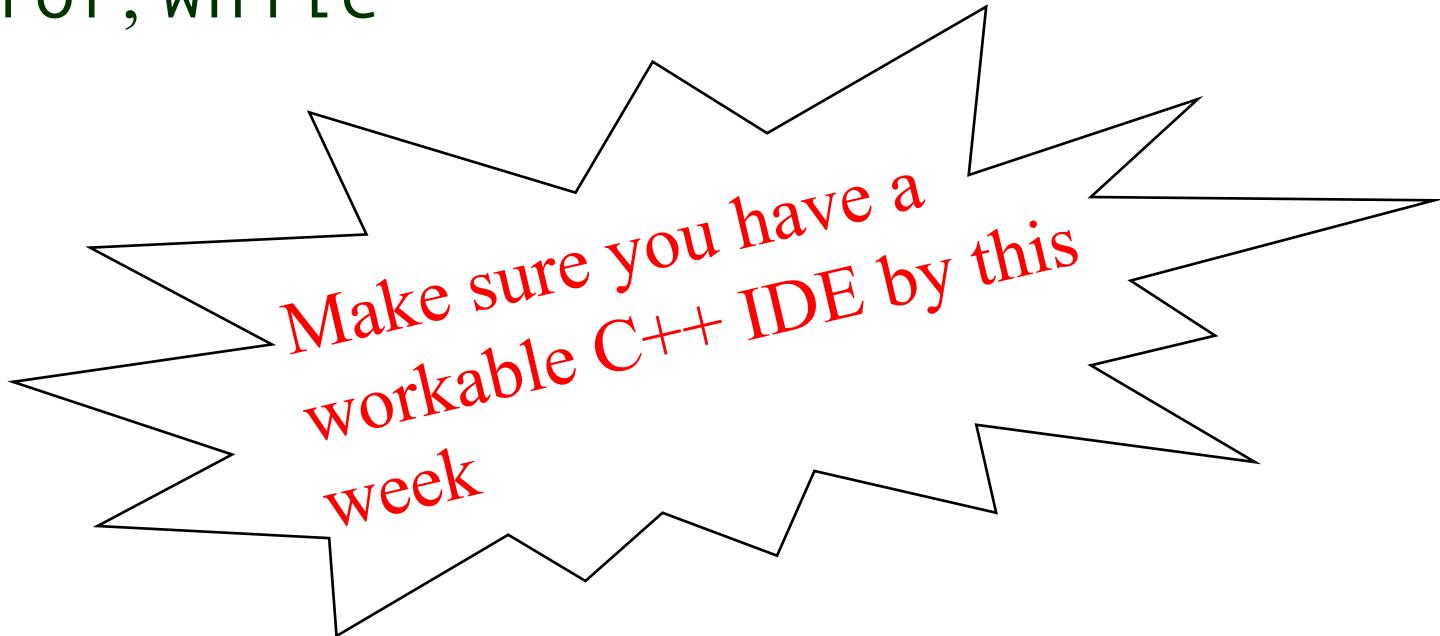
## Lecture 2

**Function calls and parameter passing**

# Topics coved in last lecture

---

- ◆ Basic syntax of C++
- ◆ Input and output
- ◆ Flow of control
  - Branching: `if-else`, `switch`
  - Loops: `for`, `while`



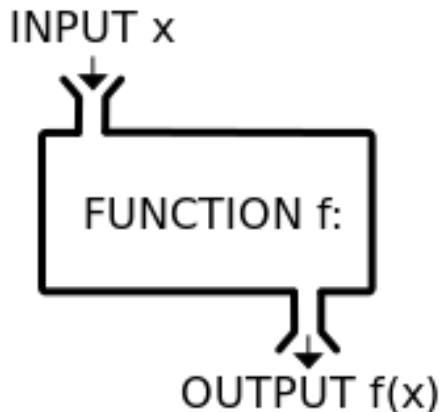
# Topics covered by the lecture

---

- ◆ Concept of function
- ◆ Write a function in C++
- ◆ Parameter passing in a function
  - Call-by-value
  - Call-by-reference
- ◆ Variable scope
- ◆ Function overloading
- ◆ Default arguments

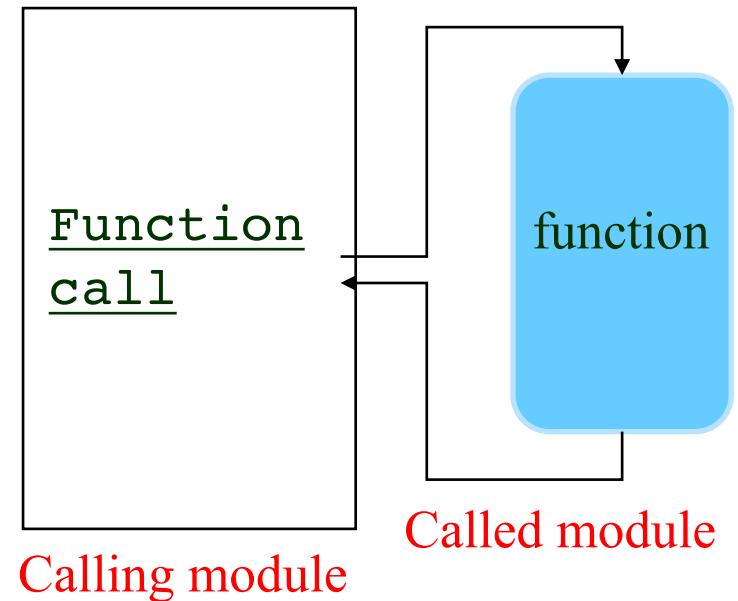
# The concept of function

**Wikipedia Definition:** A function, also known as procedure or subroutine, *takes an input and returns an output.*



$$f(x) = x^2 + 1$$

$$\begin{aligned}f(1) &= 2 \\f(2) &= 5 \\f(-2) &= 5 \\f(f(2)) &= ?\end{aligned}$$



A procedural program is composed of one or more modules, each of which consists of procedures, functions or subroutines.

The miracle of a function is that you can call it several times without extra “cost” of coding.

# Example of a function in C++

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

int f(int x) {
    int y = x*x+1;
    return y;
}

int main() {
    int x;
    cout << "Input the value of x" << endl;
    cin >> x;

    cout << "f(" << x << ")=" << f(x) << endl;
    return 0;
}
```

Function declaration and definition

$$f(x) = x^2 + 1$$

$$f(1) = 2$$

$$f(2) = 5$$

$$f(-2) = 5$$

$$f(f(2)) = ?$$

Function call

# Example of a function in C++

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

int f(int x);

int main() {
    int x;
    cout << "Input the value of x" << endl;
    cin >> x;

    cout <<"f(" << x << ")=" << f(x) << endl;
    return 0;
}

int f(int x) {
    int y = x*x+1;
    return y;
}
```

Function declaration

$$f(x) = x^2 + 1$$

$$f(1) = 2$$

$$f(2) = 5$$

$$f(-2) = 5$$

$$f(f(2)) = ?$$

Function call

Function definition

# Function declaration, definition and call

- ◆ Function declaration (function prototype): *three components in a function declaration,*

- Return type (can be void or any data type)
- Function name (better be meaningful)
- Parameter list (can be none or a few)

```
int f(int x);
```

Can either `int f(int)` or `int f(int x)`. Another example:

```
double monthlyPayment(double, int, double);
```

```
double monthlyPayment(double principal, int years, double rate);
```

Formal arguments

- ◆ Function definition: the code of the function.

- Three parts: *inputs, process & outputs*

```
int f(int x) {  
    int y = x*x+1;  
    return y;  
}
```

- ◆ Function call: use the function.

Format: `FunctionName(Actual arguments)`.

```
f(2);
```

- ◆ Any function should be **declared/defined** before it is called.

# Return Statements

- ◆ “`return`” statement transfers control back to “*calling*” module
- ◆ Return type can be any valid type, including user defined data type and “`void`”, but needs to match with the function declaration.
- ◆ Any function must have a return statement unless its type is `void`.

```
int main() {  
    ...  
    double result = areaOfCircle(10);  
    cout << result;  
    ...
```

```
double areaOfCircle(int radius) {  
    double area = 3.14*radius*radius;  
    return area;  
}
```

Mind type matching

# Main function

---

- ◆ Any C++ program must have a **main** function and have only one **main** function
- ◆ **main()** normally has **int** type and **returns** 0.
- ◆ **main()** is called by Operating System and **can only** be called by Operating System.
- ◆ If there are more than one free functions, the main function stay at the very bottom unless all other functions are declared on top of it.
- ◆ With OO paradigm, the main function is normally very small, mostly acts as a class driver (as in Java or other OO-based programming languages).

# A not-that-simple example

Calculate monthly payment of a loan, given the input:

- The loan amount: *principal*
- The length of the loan: *years*
- The interest rate (in percentage): *rate*

$$\text{payment} = \text{principal} \times \left( i + \frac{i}{(1 + i)^{\text{months}} - 1} \right)$$

Where  $\text{months} = \text{years} * 12$

$$i = \text{rate}/(12*100)$$

# A not-that-simple example of functions

Return type

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

Function name

```
double monthlyPayment(double principal, int years, double rate) {
    int months = years * 12;
    double i = rate/(12.0*100.0);
    double payment = principal * (i + i/(pow(1+i, months)-1));
    return payment;
}
```

formal arguments

Return statement

```
int main() {
```

```
    double principal;
```

```
    int years;
```

```
    double rate;
```

```
    cout << "Input the principal, years and rate" << endl;
```

```
    cin >> principal >> years >> rate;
```

```
    cout << "Monthly payment is " << monthlyPayment(principal, years, rate);
```

```
    return 0;
```

Variables that can hold values temporarily

function call

actual arguments

# Alternative Function Declaration

---

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

double monthlyPayment(double, int, double); //function prototype

int main() {
    double principal;
    int years;
    double rate;
    cout << "Input the principal, years and rate" << endl;
    cin >> principal >> years >> rate;
    cout << "Monthly payment is " << monthlyPayment(principal, years, rate);
    return 0;
}

double monthlyPayment(double principal, int years, double rate) {
    int months = years * 12;
    double i = rate/(12.0*100.0);
    double payment = principal * (i + i/(pow(1+i, months)-1));
    return payment;
}
```

# Parameters/arguments

- ◆ Formal arguments:
  - placeholders/variables
  - declared in function declaration
- ◆ Actual arguments
  - data carriers
  - used in the function call
- ◆ Parameter passing
  - actual arguments carry data in calling module and pass them to the called function via formal arguments

formal argument

```
int f(int x){  
    y = x*x;  
    return y;  
}
```

```
int z=5;  
f(z)
```

actual argument

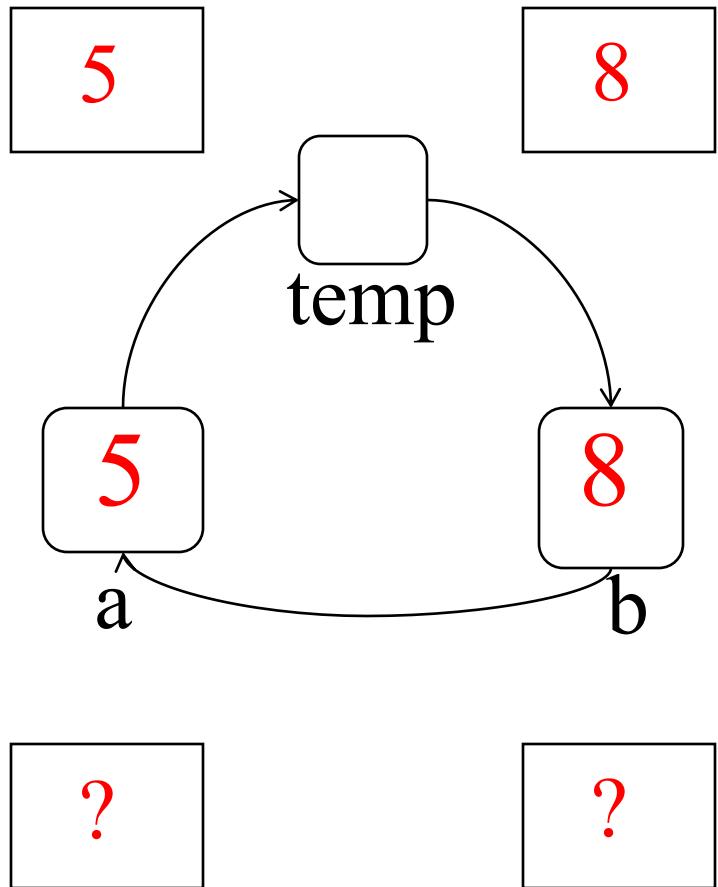
# Parameter passing

```
#include<iostream>
using namespace std;
void Swap(int a, int b) {
    int temp;
    temp=a;
    a=b;
    b=temp;
}
int main() {
    int a=5;
    int b=8;
    cout << "a=" << a << "; b=" << b << endl;
    Swap(a,b);
    cout << "a=" << a << "; b=" << b << endl;
    return 0;
}
```

**formal arguments**

**actual arguments**

Swap two numbers



Call by value

# Call-by-Value Parameters

---

- ◆ The values of the actual arguments are passed (**copied**) to the formal arguments of the called function.
- ◆ The values are **stored** in the formal arguments, which can be considered as "local variables" of the function
- ◆ If the value of the formal arguments are modified, only the "local copy" changes. The function has no access to "actual argument" from the caller.

# Another way of parameter passing

```
#include<iostream>
```

```
using namespace std;
```

```
void Swap(int& a, int& b) {
```

```
    int temp;
```

```
    temp=a;
```

```
    a=b;
```

```
    b=temp;
```

```
}
```

```
void main() {
```

```
    int a=5;
```

```
    int b=8;
```

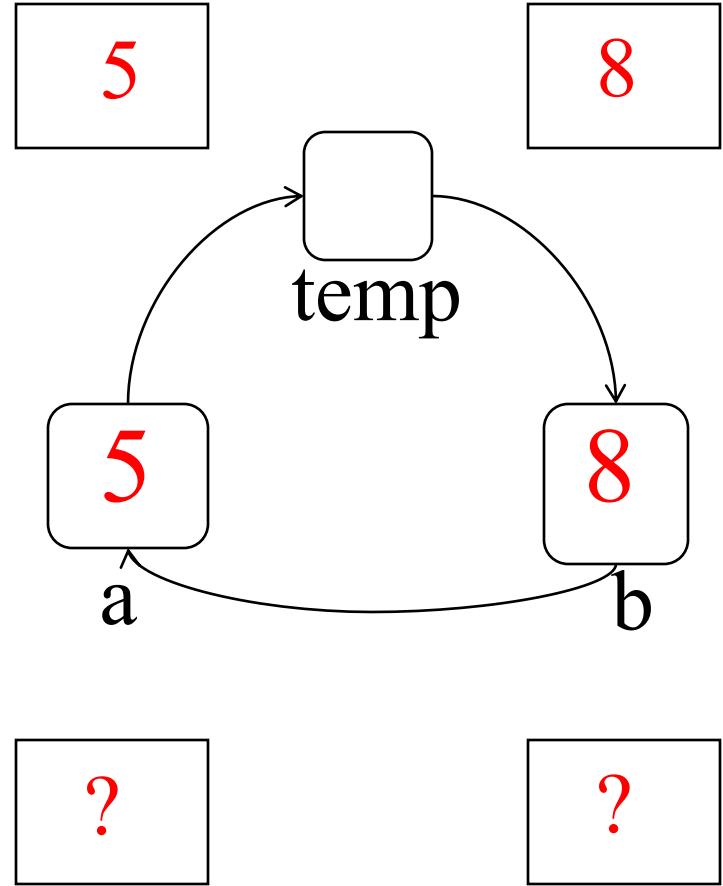
```
    cout << "a=" << a << "; b=" << b << endl;
```

```
    Swap(a,b);
```

```
    cout << "a=" << a << "; b=" << b << endl;
```

```
}
```

Address of operator



Call by reference

# Call-By-Reference Parameters

---

- ◆ A formal argument that is specified by ampersand “**&**” is called **reference argument**.
- ◆ The “addresses” of the actual arguments are passed to the reference arguments of the function when the function is called.
- ◆ The reference arguments in the function not only can get the values from those addresses but also can modify the data in the addresses.
- ◆ The calling module can then see the changed values in those addresses.
- ◆ Call-by-reference can be:
  - **Dangerous**: because caller’s data can be changed
  - **Useful**: often is desired, especially when the function has more than one return values. Avoiding **object slicing** is another motivation.
- ◆ There is no such alternation in Java (no call by reference in Java).

# Call-By-Reference vs Call-By-Value

*pass by reference*

cup = 

fillCup( )

*pass by value*

cup = 

fillCup( )

# In-class practice

---

Write a function that takes a two-digit integer and return the two digits of the number. For example,

95 => 9 and 5

14 => 1 and 4

See twodigits.cpp

# Scope rules in C++

- ◆ A program can contain number of blocks
  - A block is normally bounded by curly brackets.
  - All **function definitions** are blocks.
  - **while/if-else statements** are blocks

```
int sum = 0;  
for (int i=0; i<10; i++)  
    sum += i;
```

- ◆ A variable declared in a block is valid only in that block

```
int main() {  
    int x = 10;  
    cout << x;  
    return 0;  
}
```

```
int main() {  
    { int x = 10; }  
    cout << x;  
    return 0;  
}
```

- ◆ A variable in an outer block has access to inner blocks not the other way around.

# Local variables to a function

---

- ◆ Local variables
  - Declared inside of a function
  - Available only within that function
- ◆ Can have variables with same names declared in different functions
  - Scope is local: "that function is it's scope"
- ◆ Local variables are preferred: *maintain individual control over data*



# Global variables

- ◆ A variable is not declared in any block is called **global variable**
- ◆ A global variable has access to all blocks, especially *accessible by any functions and classes in the same file.*
- ◆ Global variables may be used for **constants**, say Pi, or for convenience (to avoid too much parameter passing).

```
#include<iostream>
int value = 10;
void change() {
    value *=10;
}
void main() {
    cout << value << endl;
    change();
    cout << value << endl;
}
```

Global variable

Give me your reasons  
whenever you use a  
global variable;  
otherwise, you lose  
marks!

# Function Overloading

C++ is a strongly typed language. Change parameter list of a function will lead to “*another*” function.

```
double average(int n1, int n2) {  
    return ((n1 + n2) / 2.0);  
}  
  
double average(double n1, double n2) {  
    return ((n1 + n2) / 2.0);  
}
```

- ◆ **Function overloading:** the same function name with different parameters (different types or different number of parameters) represents different functions.
- ◆ **Function *signature*:** function name & parameter list
  - Must be "unique" for each function definition

# Function call: overloading resolution

---

- ◆ Given following functions:
  - 1) void f(int n, double m);
  - 2) void f(double n, int m);
  - 3) void f(double n, double m);
- ◆ These calls:  
f(5.3, 4); → Calls 2)  
f(4.3, 5.2); → Calls 3)  
f(98, 99); → Calls ??

See functioncall.cpp

# Default Arguments in a function

When making a function call, the calling module must provide all values for all formal arguments of the function. However, if part of the formal arguments are given **default** values, the calling module **does not have to** provide values for **those** arguments, in which case, the default values will be used. For example, the following code fragment is illegal:

```
double roundup(double, int); //function prototype  
void main() {  
    ...  
    cout << roundup(3.14159); //illegal, too few parameters  
}
```

However, it is ok if we declare the function as follows:

```
double roundup(double, int = 2); //function prototype
```

# Rules for Default Arguments

---

The two rules of using default parameters are:

1. Default values **can only be assigned** in the function prototype (declaration).
2. It is not necessary that all arguments of a function have to be default arguments but **the default arguments must filled from right to left**.

`double roundup(double = 0.0, int = 2); //function prototype`

`double roundup(double, int = 2); //function prototype`

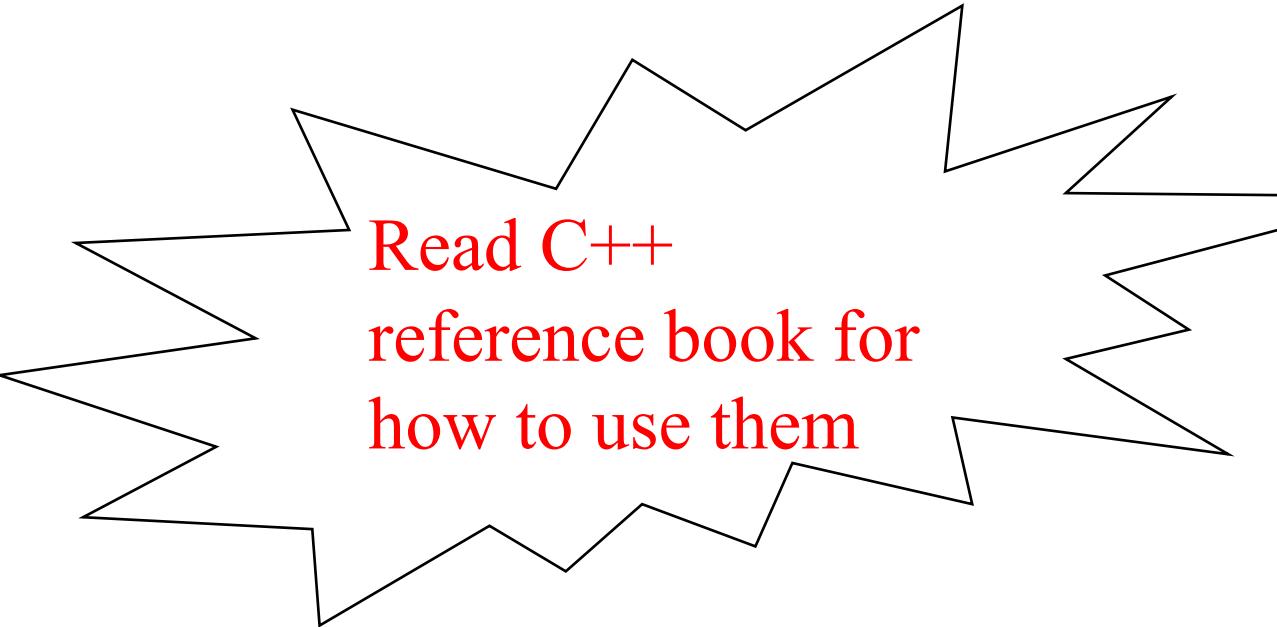
These rules allow C++ not to get '**mixed up**' when matching actual arguments with formal arguments.

`roundup(); roundup(3.14159); roundup(3.14159, 4);`

*Class constructors may take advantage of this capability to initialise data members either explicitly through actual arguments or defaulting to a set of values specified in the function prototype.*

# Pre-defined functions

- ◆ Plentiful pre-defined functions for variety of applications. They are built in library.
- ◆ They are different with different IDEs, some of them inherited from C (in `cstdlib`).
- ◆ Most useful ones are:
  - `<cmath>`
  - `<cstdlib>`
  - `<ctime>`
  - `<algorithm>`
  - `<iomanip>`



Read C++  
reference book for  
how to use them

# Some pre-defined functions

Display 3.2 Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0, 3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath

# Some pre-defined functions

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

# Summary

---

- ◆ Formal parameter is placeholder, filled in with actual argument in function calls
- ◆ Call-by-value parameters are "local copies" in receiving function body
  - Actual arguments cannot be modified
- ◆ Call-by-reference passes memory addresses of actual arguments
  - Actual arguments can be modified
  - If you don't want the function changes the value of an actual argument, use call-by-value or make the formal argument to be constant
- ◆ Same function name can define multiple: called function overloading

# Summary

---

- ◆ Default arguments: some arguments in a function can be provided with default values in the function declaration.
- ◆ Scope rules:
  - A variable is valid only in its scope
  - Local variables can only be accessed locally.
  - A variable that does not belongs to any block is called global variable, which is accessible by any function in the program
- ◆ Pre-defined functions: the functions that have been implemented and placed in C/C++ library.

# Homework

---

- ◆ Read textbook: Chapter 3-4
- ◆ Complete Practical 2 which will be due in week 3
- ◆ Warm up textbook Chapter 5.

# COMP 2014 Object Oriented Programming

---

## Lecture 3

## Arrays

# Topics covered last week

---

- ❖ Unit introduction
- ❖ Concept of functions
- ❖ Create a function
- ❖ Parameter passing in a function
  - Call-by-value
  - Call-by-reference
- ❖ Function overloading
- ❖ Default arguments

# Call-By-Reference vs Call-By-Value

*pass by reference*

cup = 

fillCup( )

*pass by value*

cup = 

fillCup( )

# Call-By-Reference vs Call-By-Value

```
#include<iostream>
using namespace std;
void Swap(int a, int b) {
    int temp;
    temp=a;
    a=b;
    b=temp;
}
int main() {
    int a=5;
    int b=8;
    cout << "a=" << a << "; b=" << b << endl;
    Swap(a,b);
    cout << "a=" << a << "; b=" << b << endl;
    return 0;
}
```

```
#include<iostream>
using namespace std;
void Swap(int& a, int& b) {
    int temp;
    temp=a;
    a=b;
    b=temp;
}
void main() {
    int a=5;
    int b=8;
    cout << "a=" << a << "; b=" << b << endl;
    Swap(a,b);
    cout << "a=" << a << "; b=" << b << endl;
}
```

# Function Overloading

C++ is a strongly typed language. Change parameter list of a function will lead to “*another*” function.

- ◆ Function *signature*: function name & parameter list
  - Must be "unique" for each function definition
- ◆ Function overloading: the same function name with different parameters (different types or different number of parameters) represents different functions.

```
double average(int n1, int n2) {  
    return ((n1 + n2) / 2.0);  
}  
  
double average(double n1, double n2) {  
    return ((n1 + n2) / 2.0);  
}
```

# Function call: overloading resolution

---

Given following functions:

- 1) void f(int n, double m);
- 2) void f(double n, int m);
- 3) void f(double n, double m);

These calls:

f(5.3, 4); → Calls 2)

f(4.3, 5.2); → Calls 3)

f(98, 99); → Calls ??

See functioncall.cpp

# Default Arguments in a function

---

When we define a function, it is possible to set some arguments of the function default values.

```
double roundup(double, int); //function prototype  
void main() {  
    cout << roundup(3.14159, 2); //set roundup to 2 digits  
}
```

```
double roundup(double, int = 2); //function prototype  
void main() {  
    cout << roundup(3.14159); //means roundup to 2 digits  
}
```

```
void main() {  
    cout << roundup(3.14159, 4); //set roundup to 4 digits  
}
```

# Rules for Default Arguments

---

The two rules of using default parameters are:

1. Default values **can only be assigned** in the function prototype (declaration).
2. It is not necessary that all arguments of a function have default values but **the default values must be filled from right to left**.

`double roundup(double = 0.0, int = 2); //function prototype`

`double roundup(double, int = 2); //function prototype`

These rules allow C++ not to get '**mixed up**' when matching actual arguments with formal arguments.

`roundup(); roundup(3.14159); roundup(3.14159, 4);`

*Class constructors may take advantage of this capability to initialise data members either explicitly through actual arguments or defaulting to a set of values specified in the function prototype.*

# Content of this lecture

- ❖ Introduction to Arrays
    - Declare arrays
    - Access arrays
  - ❖ Passing an arrays to a function
  - ❖ Array initialisation
  - ❖ Common process in arrays
    - Linear search
    - Find the largest
  - ❖ Multidimensional Arrays
- 
- Different from Java
- Essential for assignment 1

# Introductory example

**Task:** Design a program which *accepts ten integers, store them and display the total.*

```
// Bad solution 1
#include <iostream>
using namespace std;
int main() {
    int num, total;
    total = 0;
    cout << "Please input ten integers:\n";
    for ( int i=0; i<10; i++) {
        cin >> num;
        total = total + num;
    } // end for
    cout << "\n The total of the ten integers is " << total;
    return 0;
} // end main
```

An incorrect solution.  
Find the problem.

# Introductory example

```
// Bad solution 2
#include <iostream>

int main() {
    int num1, num2, num3, num4, num5,
        num6, num7, num8, num9, num10;
    int total;

    cout << "Please input ten integers:\n";
    cin >> num1 >> num2 >> num3 >> num4 >> num5;
    cin >> num6 >> num7 >> num8 >> num9 >> num10;

    total = num1+num2+num3+num4+num5
        +num6+num7+num8+num9+num10;
    cout << "\n The total of the ten integers is " << total;
    return 0;
} //end main
```

A bad solution.  
Do not follow.

Zero marks

# Introductory example

```
// Good solution
#include <iostream>

int main() {
    int num[10];
    int total=0;

    cout << "Please input ten integers:\n";
    for ( int i=0; i< 10; i++) {
        cin >> num[i];
        total = total + num[i];
    } // end for
    cout << "\n The total of the ten integers is " << total;
    return 0;
} // end main
```

A good solution:  
general and clean.

# Definition of Array

Note the difference  
between Java and C++

An array is an indexed/subscripted list of elements of the same type (a **homogeneous collection**).

4	21	5	3	16	25	11	23	2	80
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

**Declaration:**

**int a[10];**

Declare ten memory cells to store ten integers. The size must be constant.

**access:**

**a[2] = 5;  
cout << a[2];**

Put integer 5 to the third memory cell.

An array in

C++: *a collection of objects*

Java: *an object that contains a collection of objects*

**Java:**

```
int[] a = new int[10];
```

# Three components of an array

- ❖ Think of array as 3 "pieces"
  - Array **type**: What is the type of the elements in the array?
  - Array **name**: Where does it starts?
  - **Size** of the array: How many elements it can contain?
- ❖ The array name is actually a **pointer** (see lecture 6)

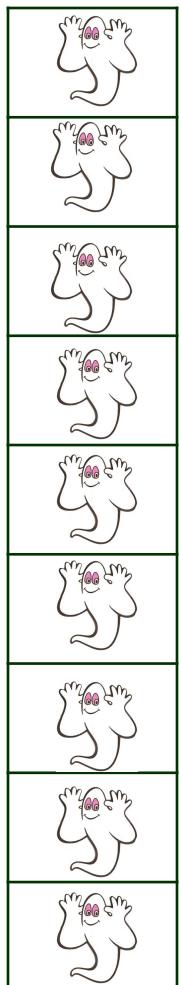
**int num[10]**

4	21	5	3	16	25	11	23	2	80
---	----	---	---	----	----	----	----	---	----

num

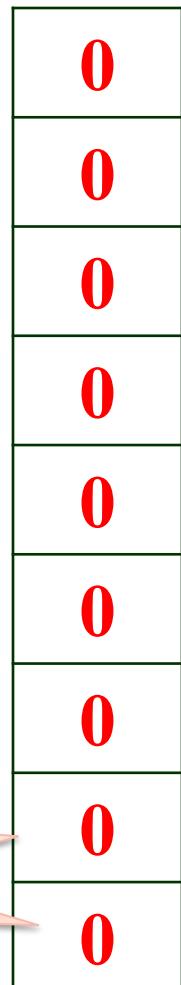
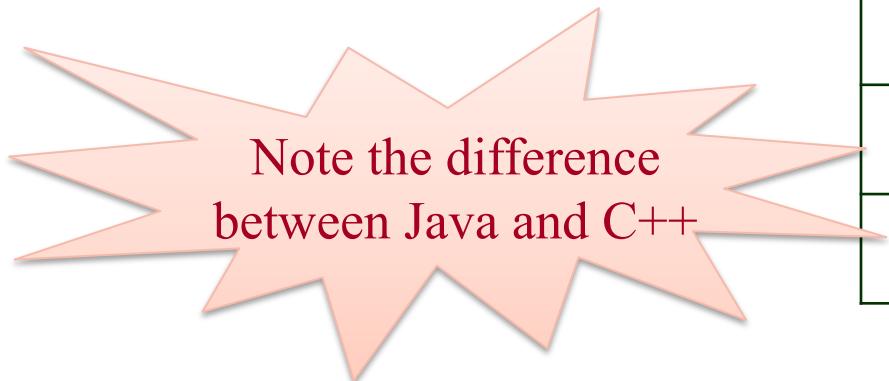
Size 10

# Initialising an Array

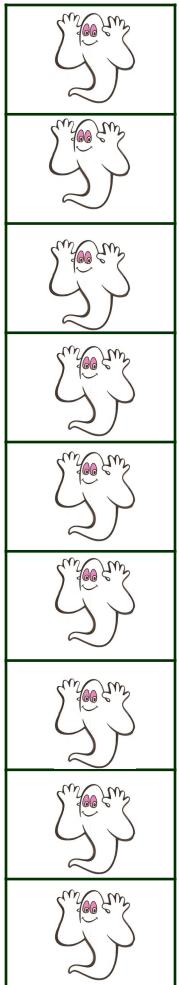


```
int scores[9];  
  
for (int i = 0; i < 9; i++) {  
    scores[i] = 0;  
}
```

What are in the memory cells before you initialise them?



# Initialising an array: Enumerating



```
int scores[ ] = {2, 5, 9, 10, 4, 2, 7, 8, 5};
```

Compiler sets the size to match the number of elements in the list

2
5
9
10
4
2
7
8
5

# Pass an array to a function

Note the difference  
between Java and C++

- ❖ When you pass an array to a function, you also need to pass the information of **type**, the **name** and the **size** of the function.
- ❖ Formal arguments in the function:
  - **Type**: the same as the array you want to pass
  - **Formal name** of the array with indicator
  - **Array size**: can be smaller than the actual array
    - e.g., `function(int list[ ], int size);`
- ❖ Actual arguments:
  - The name of the array that carries the data (it's a pointer)
  - The actual size of data you pass to the function
    - e.g., `function(num, 10);`

# Pass an array to a function

Array as a formal parameter

```
void printArray(int a[], int length) {  
    for (int i = 0; i < length; i++)  
        cout << a[i] << endl;  
}
```

```
int main() {  
    int a[] = { 1, 2, 3, 4, 5 };  
    int b[] = { 4,-1,3,20,7,9,4,11,-2};  
    printArray(a, 5);  
    printArray(b, 9);  
    return 0;  
}
```

Arrays passed to the function

See code 05-03.cpp

# Pass an array to a function

- ❖ Why do we specify the size of an array separately in C++:
  - The array name is a pointer, which points to the first element of the array
  - We can pass partial data of an array to a function
  - We can pass different arrays to the same function if the type of array is the same
  - Example:

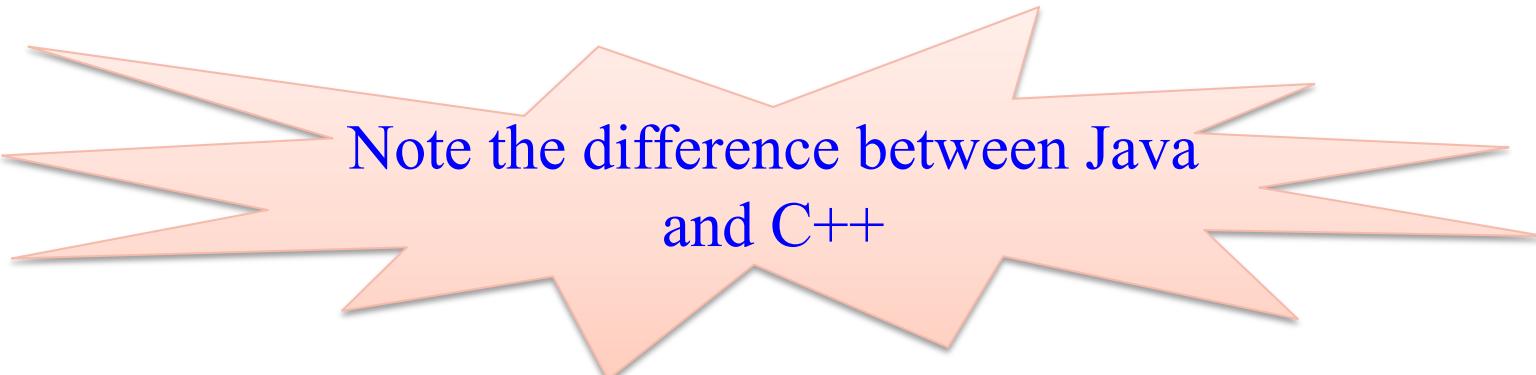
```
int score[5], time[10];  
printArray(score, 5);  
printArray(time, 10);
```

See code 05-03.cpp again

# Function that returns an array

---

- ❖ In principle, a function **cannot** return an array. There is **no** such thing as `int[ ] function( )` in C++.
- ❖ However, you can use a pointer to return a static array.
- ❖ Will talk about it Lecture 6...



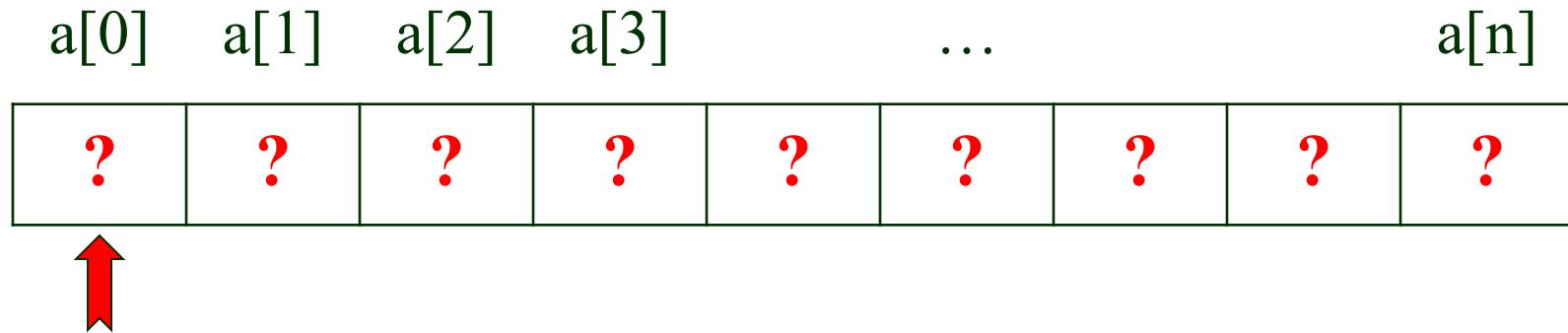
Note the difference between Java  
and C++

# Common Array Processing

---

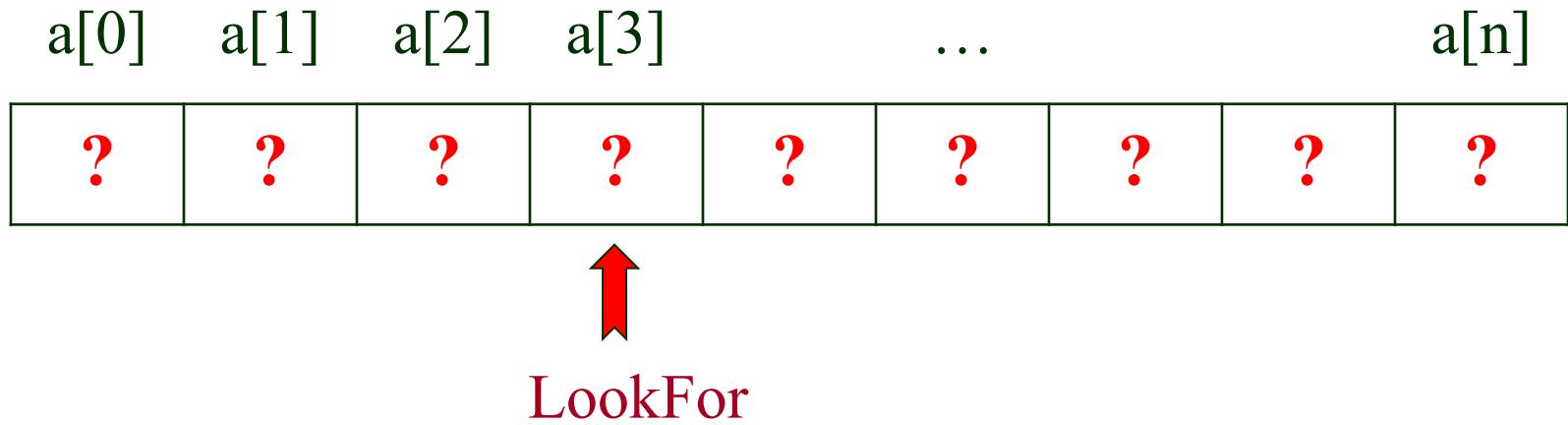
- ❖ Search: *finding a value in the array*
- ❖ Sort: *arrange the values in order*
- ❖ Calculate the average
- ❖ Calculate the minimum
- ❖ Calculate the maximum
- ❖ ...

# Linear Search



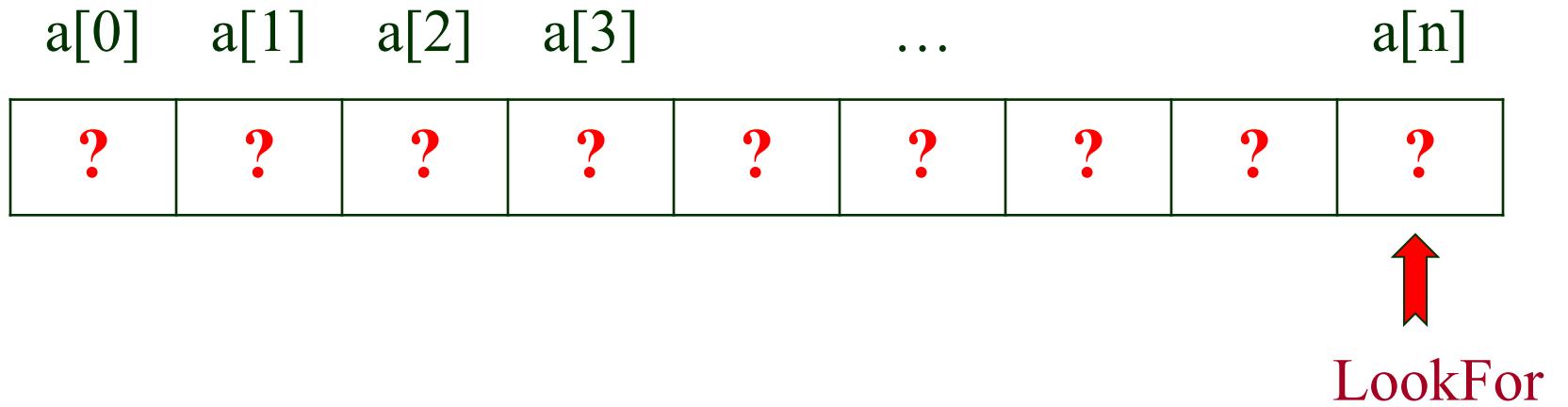
- Search: find a specific value in a set of data
- Linear search: travel through the elements one by one until find.

# Linear Search



- Search: find a specific value in a set of data
- Linear search: travel through the elements one by one until find.

# Linear Search



- Search: find a specific value in a set of data
- Linear search: travel through the elements one by one until find.

# C++ Code for Linear Search

```
int linearSearch(int data[ ], int size, int lookFor) {  
    bool found = false;  
    int index;  
    for( index = 0; index < size && !found; index++ ) {  
        if ( data[index] == lookFor )  
            found = true;  
    }  
  
    if (found)  
        return index;  
    else  
        return -1;  
} // end linearSearch
```

The index of an array start from 0!  
You get punishment if you do not remember it!

# Calculate maximum

2

5

9

10

4

2

7

8

5

```
int main( )
{
    int data[ ] = {2, 5, 9, 10, 4, 2, 7, 8, 5};

    int largest = data[0]; //or a number less than any
                           numbers in the array

    for (int i = 1; i < 9; i++) {
        if (data[i] > largest) {
            largest = data[i];
        }
    }

    cout << "The largest number is" << largest<<endl;
    return 0;
}
```

# Two-Dimensional Arrays

A two-dimensional array is made up of rows and columns (like a spread sheet or a table)

	0	...	n
0			
1			
...			
m			

In C++, a two-dimensional array is an array of arrays.

# Declaration

```
const int ROWS = 4;  
const int COLS = 3;  
int table[ROWS][COLS];
```

Must contain integer values.

OR

```
int table[ROWS][COLS] = {{2,1,3},  
                          {4,12,7},  
                          {6,2,4},  
                          {10,1,2}};
```

OR

```
int table[ROWS][COLS] = {2,1,3,4,12,7,6,2,4,10,1,2};
```

Two dimensional array is simply an array of arrays.

# Accessing array elements

```
amount = table[2][1]; //take values  
table[0][0] = 2; //set values  
newValue = 4*(table[1][0]-5);  
table[1][2]=table[3][2]+table[2][0];  
  
for(int i=0;i<4;i++)  
    for(int j=0;j<3;j++)  
        table[i][j] = i*j;
```

2	21	-3
9	4	21
11	2	-5
2	3	10

# Processing 2-D Arrays

- ◆ Getting input

```
for (int i = 0; i < ROWS; i++)  
    for (int j = 0; j < COLS; j++) {  
        cout << "Enter the next value: ";  
        cin >> table[i][j];  
    }
```

- ◆ Displaying output

```
for (int i = 0; i < ROWS; i++) {  
    for (int j = 0; j < COLS; j++)  
        cout << setw(3) << table[i][j];  
  
    cout << endl;  
}
```

See code io2Array.cpp

# Passing 2-D array to a function

```
const int ROWS=2;
const int COLS=3;
void printTab(int table[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++)
            cout << setw(3) << table[i][j];
        cout << endl;
    }
}
int main() {
    int table[ROWS][COLS];
    for (int i = 0; i < ROWS; i++)
        for (int j = 0; j < COLS; j++) {
            cout << "Enter the next value: ";
            cin >> table[i][j];
        }
    printTab(table);
    return 0;
}
```

Can be omitted

See code display2D.cpp

# In class practice

---

Write a program that calculate the total of each row in a matrix.

2	5	1
6	2	4
3	4	1
4	6	3

# Summary

---

- ❖ Array is collection of data in the same type
- ❖ Array elements stored sequentially
  - "Contiguous" portion of memory (arranged by operation system)
  - The name of the array is the point to the 1<sup>st</sup> element of the array
- ❖ The use of an array element is the same as the use of any other variable
- ❖ Programmer responsible for staying "*inbound*"
- ❖ Multidimensional arrays
  - Create "array of arrays"

# Homework

---

- ❖ Read textbook Chapter 5
- ❖ Check out the tasks of Assignment 1
- ❖ Complete practical tasks for week 3 (Practical 2).
- ❖ Warm up Chapter 6

# COMP 2014 Object Oriented Programming

---

## Lecture 4

# Objects & Data Abstraction

# Topics covered by last lecture

---

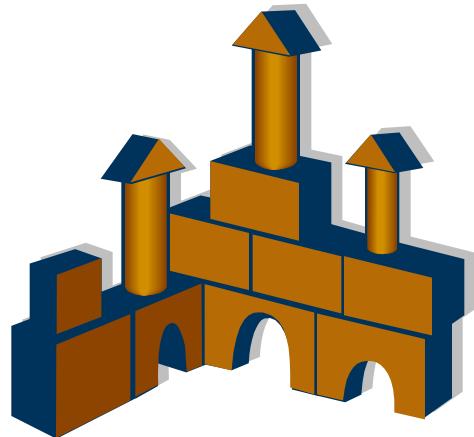
- ◆ Array declaration and initialization
- ◆ Common process in arrays:
  - linear search,
  - find the largest/smallest,
  - calculate average,
  - sorting,
  - ...
- ◆ Array as function argument
- ◆ Multi-dimensional arrays

# Topics covered by this lecture

---

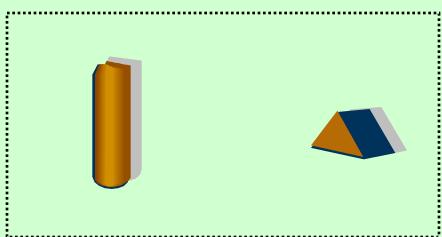
- ◆ Objects
- ◆ Data abstraction
- ◆ Classes and objects
- ◆ Class definition
- ◆ Member functions
- ◆ Applications

# Object-Oriented Analysis and Design



Class: Castle

Procedural style

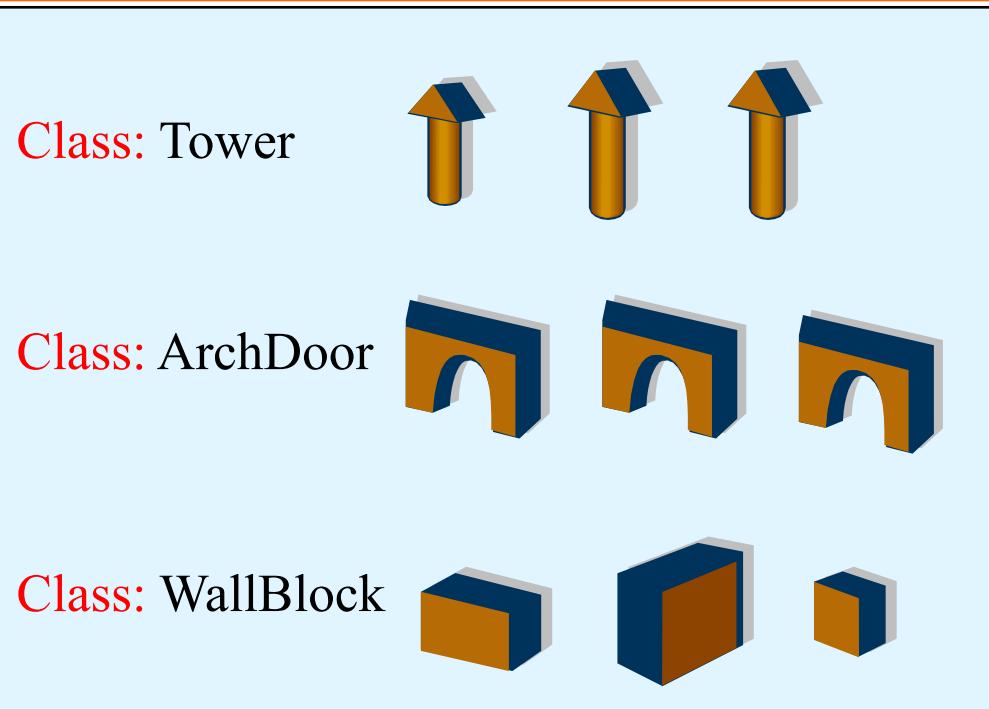


Class:  
Tower

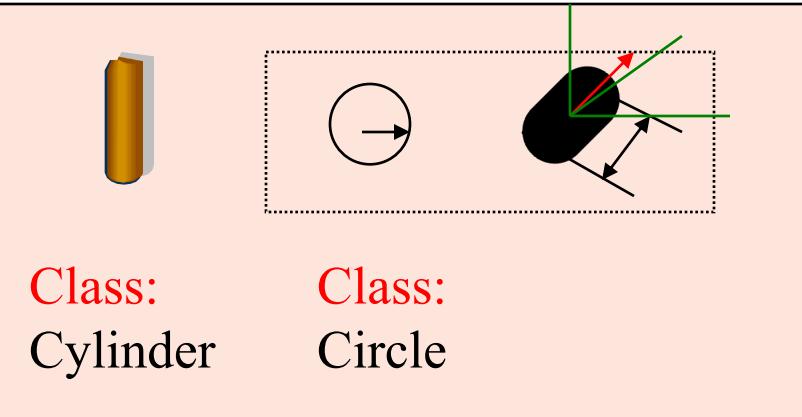
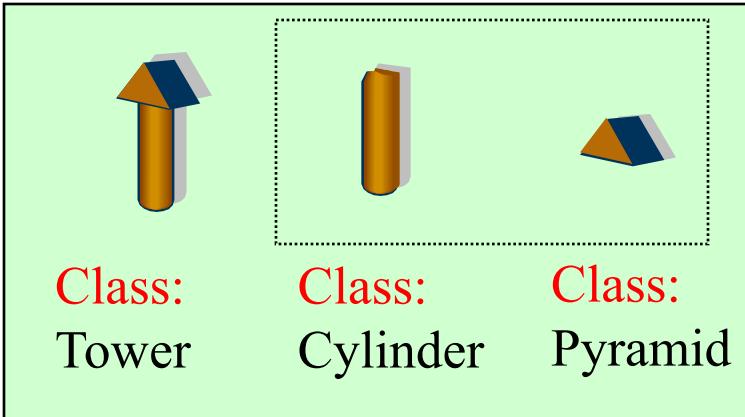
Class:  
Cylinder

Class:  
Pyramid

OO style



Class: WallBlock



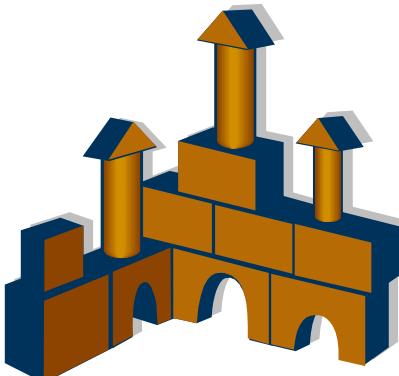
Class:  
Cylinder

Class:  
Circle

# Object-Oriented Analysis and Design

- ◆ Objects

- towers[3]
- archDoors[3]
- wallBloacks[200]
- castle



- ◆ classes

- Circle
- Cylinder
- Pyramid
- Tower
- ArchDoor
- WallBlock
- Castle



- ◆ Objects

- board, players[2], game

- ◆ Classes

- Board
- Player: Random, Smart,...
- Game

# A preview of class implementation

```
class Date {  
public:  
    int day;  
    string month;  
    int year;  
};
```



```
struct Date {  
    int day;  
    string month;  
    int year;  
};
```

date.h

```
int main() {  
    Date d;  
    d.day = 10;  
    d.month = "Aug";  
    d.year = 2021;  
  
    cout << d.day << " " << d.month << " "  
        << d.year;  
}
```

an object of Date

Alternative representation  
of date

```
int date[3];  
date[0] = 10;  
date[1] = 8;  
date[2] = 2020;
```

Class creator

dateApp.cpp

Find the differences  
between C++ and Java

Class client

# A preview of class implementation

```
class Date {  
public:  
    int day;  
    string month;  
    int year;  
    void display();  
};
```

Class declaration

Class application

```
void Date::display() {  
    cout << day << " "  
        << month << " "  
        << year;  
}
```

date1.h

```
int main() {  
    Date d;  
    d.day = 10;  
    d.month = "Aug";  
    d.year = 2021;  
    d.display();  
}
```

dateApp1.cpp

# Data Types

	<b>Data Type</b>	<b>Definition</b>	<b>Declaration</b>	<b>Operators /methods</b>	<b>Use</b>
Built-in	<b>int</b>	built-in	<b>int</b> i, j;	+ - * / =	i=0; j=2;
	<b>float</b>	built-in	<b>float</b> a, b;	+ - * / =	a=2.1; b=a*a;
	<b>double</b>	built-in	<b>double</b> x, y;	+ - * / =	x=3.1415*y;
Structure	<b>Student</b>	<pre>struct Student { string name;   long studentID; };</pre>	<b>Student</b> a, b, c;	=	a.name = “dongmo”; b.studentID = 12345;
Class	<b>Date</b>	<pre>class Date { int day;   int month;   int year;   void display(); };</pre>	<b>Date</b> a, b, c;	= display()	a.display();

# Abstract Data Type

*Abstract Data Type* is a user-defined type that defines the types of data (data members) and the methods (member functions) that operate the data.

## Integer type

data member:

int

operations:

+

-

\*

/

## Date type

data member:

int day;

string month;

int year;

methods:

void display();

## Abstract Data Type

data members:

data item;

...

data item;

methods:

function;

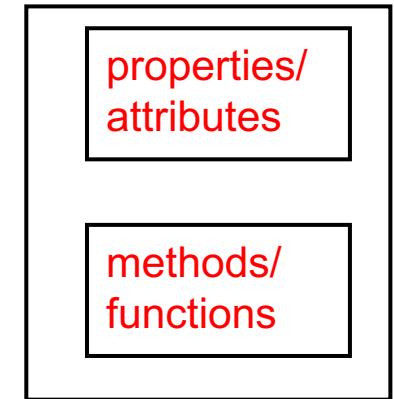
...

function;

# Abstraction, Encapsulation & Data Hiding

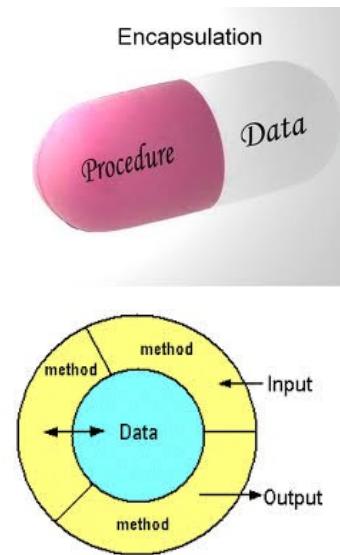
- ◆ *data abstraction*: generalisation of an object, focusing only on particular aspects of an object that are of specific interest to the problem.

e.g. Student – interested in *name*, *course*, *units* enrolled – not be interested in shoe size



- ◆ *encapsulation*: any object is encapsulated into a structure which consist of a set of *data items (attributes)* and a set of *methods* that operate the data.

- ◆ *data hiding*: ability to limit access to specific attributes and methods of an object. Avoid tightly "coupled" classes.



# Class Declaration

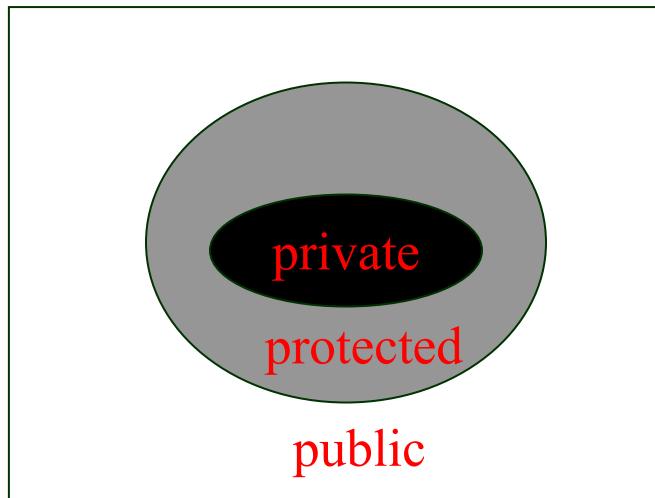
```
class className {  
    private: ←  
        data items  
        member functions  
    protected: ←  
        data items  
        member functions  
    public: ←  
        data items  
        member functions  
}; ←  
    member function definitions
```

*access specifiers:  
valid until another  
type is defined*

Be aware of the  
difference between  
C++ and Java

# Class Declaration

- ◆ access specifier specifies access privileges.
  - **private** : accessible only to class methods
  - **public** : publicly accessible
  - **protected** : accessible from derived classes



# A preview of class implementation

```
class Date {  
private:  
    int day;  
    string month;  
    int year;  
public:  
    void display();  
};
```

Cannot access  
private members  
declared in class  
'Date'

```
void Date::display() {  
    cout << day << " "  
        << month << " "  
        << year;  
}
```

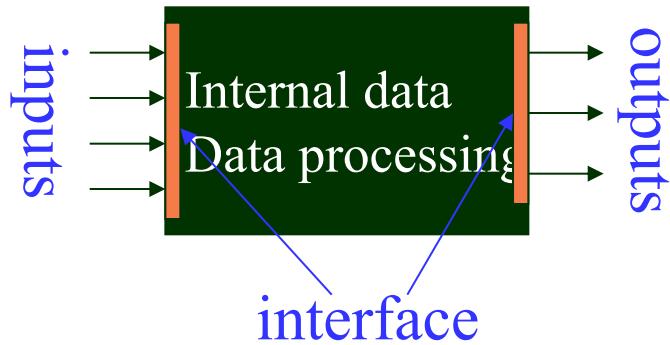
date3.h

```
int main() {  
    Date d;  
    d.day = 10;  
    d.month = "Aug";  
    d.year = 2021;  
    d.display();  
}
```

dateApp3.cpp

# Interface of a class

- ◆ The interface of a class consists of all its **public** functions



- ◆ The interface of a class is normally used for input and output data
- ◆ Two typical interface functions:
  - **Accessor:** output data (`get` functions)
  - **Mutator:** takes input as parameters (`set` function)

date3.h

dateApp3.cpp

# Data access

dateApp4.cpp

```
class Date {  
private:  
    int day;  
    string month;  
    int year;  
public:  
    void setDate(int,string,int);  
    void display();  
};
```

```
int main() {  
    Date d;  
    d.setDate(10,"Aug",2021);  
    d.display();  
}
```

```
void Date::setDate(int d, string m, int y) {  
    day = d;  
    month = m;  
    year = y;  
}
```

date4.h

# Internal data and processing

```
class Date {
private:
    int day;
    int month;
    int year;
    string mapping(int);
public:
```

Why do we set it  
to be private?

```
void setDate(int,int,int);
void display();
void showDate();
};
```

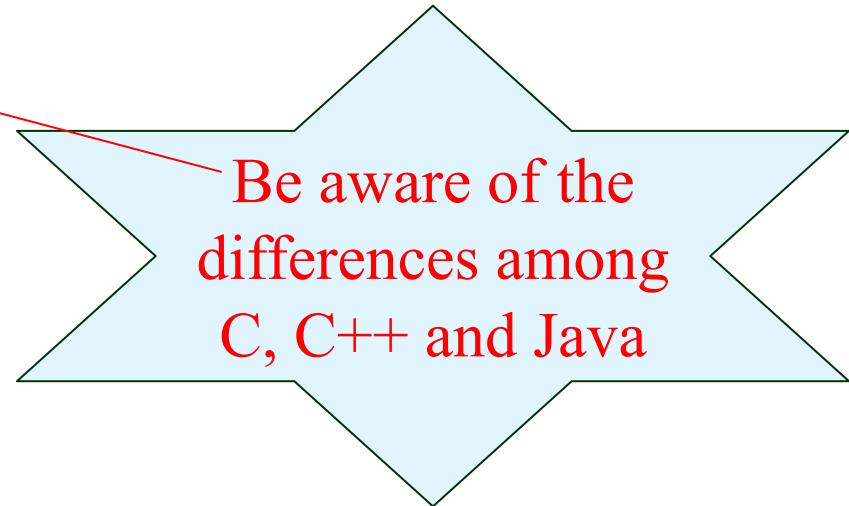
```
string Date::mapping(int month) {
    string map[12] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
                      "Sep", "Oct", "Nov", "Dec"};
    return map[month];
}
```

```
void Date::display() {
    cout << "The date is ";
    cout << day << " ";
    cout << mapping(month-1);
    cout << " " << year << endl;
}
```

```
void Date::showdate() {
    cout << "The date is ";
    cout << setfill('0')
        << setw(2) << day << '/'
        << setw(2) << month << '/'
        << setw(2) << year % 100
        << endl;
}
```

# Struct vs Class in C++

```
struct className {  
    private:  
        data items  
        member functions  
    protected:  
        data items  
        member functions  
    public:  
        data items  
        member functions  
};
```



The difference between *class* and *struct* is that if leave access specifiers out, the default access for class is *private* while the default access for struct is *public*. There is no other difference between them.

# Classes and Objects: a summary

A class contains:

- a) data items – *attributes/properties/characteristics*,
- b) methods – *member functions/behaviours/capabilities*.

An **object** is an instance of a class, which have:

- a) *specified attributes, properties, characteristics, containing data.*
- b) *abilities to use the member functions to operate the data of the object.*



# Classes and Objects: a summary

```
void main() {
```

objects

```
    Date d1, d2, d3;
```

```
    d1.setDate(20, 8 ,2021);
```

```
    d2.setDate(10,9,2021);
```

```
    d3.setDate(19,10,2021);
```

date5.h

```
    d1.showdate("Today is ");
```

```
    d2.showdate("The due date for assignment 1 is ");
```

```
    d3.showdate("The due date for assignment 2 is ");
```

```
}
```

Be aware of the  
difference between C++  
and Java

multipleClass.cpp

# Principles of Object-Oriented Programming

---

- ◆ All data types are classes, including *int*, *double*, *bool*, *array*, *char*.
- ◆ All data values are objects (variables are objects).
- ◆ Any *non-built-in* object is made up of other objects.
- ◆ A class defines the data types of its data members and their behaviour.
- ◆ Computer programs are designed by making them out of objects that interact with one another.

# From procedural to OO

```
int main() {  
    cout << "Welcome to OOP class.";  
    return 0;  
}
```

Procedural style

ProceduralStyle.cpp

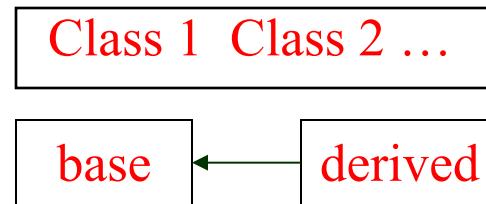
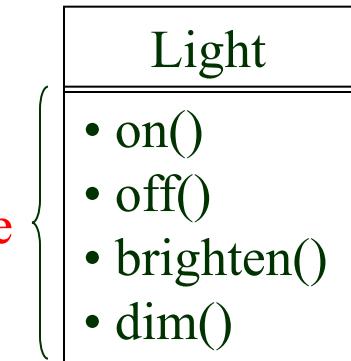
```
class Easy {  
public:  
    void run() {  
        cout << "Welcome to OOP class.";  
    }  
};  
  
int main() {  
    Easy e;  
    e.run();  
    return 0;  
}
```

OO style

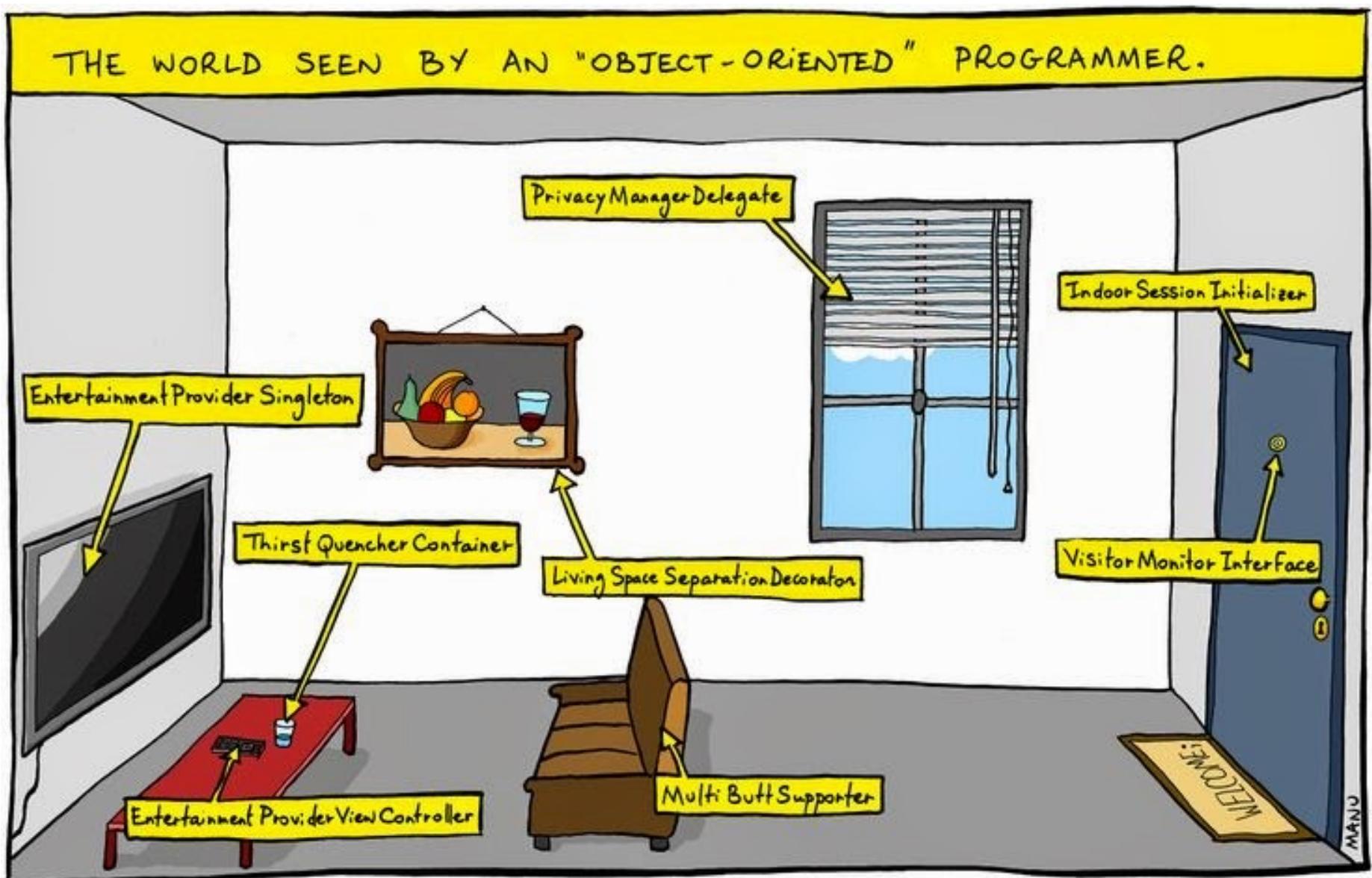
OOStyle.cpp

# Class Implementation and Reusability

- ◆ Abstraction: a class is an abstract of the same type of objects
- ◆ Interface: public members of a class -- the **interface** entries to an object.
- ◆ Hidden implementation:
  - Private
  - Protected
- ◆ Reusing the implementation:
  - Composition
  - Inheritance
- ◆ Polymorphism



# Abstraction



# Homework

---

- ◆ Read textbook: Chapter 6
- ◆ Complete and attend practical 3 this week
- ◆ Complete the **online tutorial 1** via vUWS. It will be open for one week. You can do it anywhere within the specified time period. Meanwhile practical 4 is expected to be completed within Weeks 5&6

# Attention

---

Online tutorial 1 is available now. It is due at 10 pm 04 Sept 2022. You have three attempts. The result will be based on your final attempts.

# Comp2014 Object Oriented Programming

---

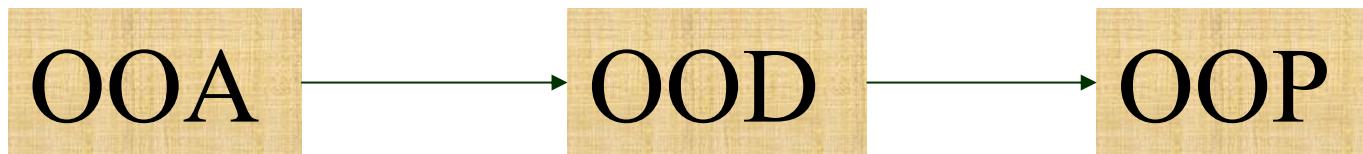
## Lecture 5

### Class Design & Implementation

# Topics covered by last lecture

---

- ◆ Objects
- ◆ Data abstraction
- ◆ Classes and objects
- ◆ Class definition
- ◆ Member functions
- ◆ Applications



# Topics covered by this lecture

---

- ◆ Class declaration, definition and application
- ◆ Constructor
- ◆ Destructor
- ◆ Object composition

# Class creation and application

## Three steps of OOP

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    void setDate();  
    void showdate();  
};
```

*class declaration*  
date.h

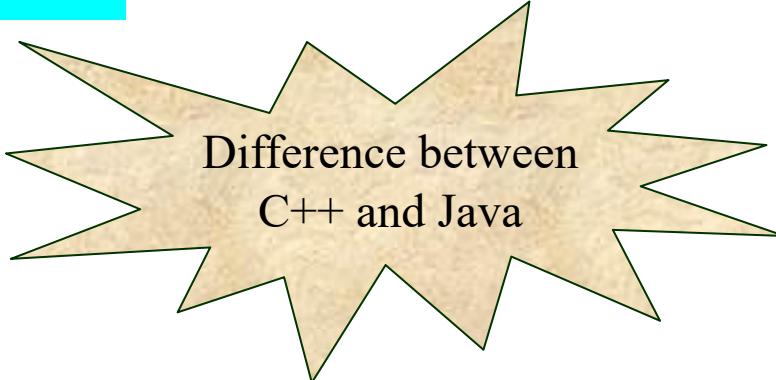
```
Void Date::setDate(  
    int d, int m, int y){  
    day = d;  
    month = m;  
    year = y;  
}  
void Date::showdate() {  
    cout<< day << " "  
        << month << " "  
        << year << endl;  
}
```

*class definition*  
date.cpp

```
int main() {  
    Date d;  
    d.setDate(27,8,2021);  
    d.showdate();  
    return 0;  
}
```

*class driver*  
dateApp.cpp

You may put all of them in one file, two or three files but you must have a .cpp file that contains the *main* function.



Difference between  
C++ and Java

# Define and use data items/member functions

## Data items

- ◆ Declaration:
  - Declare data types for each item
- ◆ Definition:
  - No need
- ◆ Uses:
  - For *internal* use, directly call their names
  - For *external* use, use . or public interface, e.g, objectVar.dataItem

## Member functions

- ◆ Declaration:
  - Declare the prototype of each member function
- ◆ Definition:
  - Implement each member function
- ◆ Uses:
  - For *internal* use, directly call their names
  - For *external* use, use . or public interface, e.g, objectVar.memberFunction(...)

# In-class Function Definitions

In C++, a member function can also be defined within the class declaration, named **inline member function**, as normally done in Java.

```
class Date {  
public:  
    int getDay() { return day; }  
};
```

Alternatively

```
class Date {  
public:  
    int day();  
};
```

Date4.h

dateApp4.cpp

```
int Date::getDay() { return day; }
```

# Class declaration

---

- ◆ You cannot declare the same class more than one time.
- ◆ To avoid declaring a class more than once, using

```
#ifndef DATE4  
#define DATE4  
    //DATE4 is the id of the file  
    //Class declared here  
#endif //DATE4
```

Your IDE can help you to generate the id of the file if you create a class by using new **Header file**

# Topics covered by the lecture

---

- ◆ Class declaration, definition and application
- ◆ Constructor
- ◆ Destructor
- ◆ Object composition

# Constructor

A constructor is a specific **member function** of each class **that is called whenever an object of the class is created**. A constructor is similar to any other member function, with three exceptions:

- 1) *Constructor must have the same name as the class.*

Date () ; //declare a constructor of Date

- 2) *Constructor has no return value.*

~~void~~ Date () ;

- 3) *Constructor is automatically called when object is declared:*

Date myBirthday; //declare an object of Date

~~myBirthday.Date () ; //incorrect call~~

Anything that can be done in a normal member function can be done in a constructor. However, constructors are mostly used for initialisation because *in many situations, initialisation cannot take place elsewhere in the class.*

# Initialisation using constructors

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date();  
    void setdate(int,int,int);  
    void showdate();  
};
```

constructor

```
Date::Date() {  
    day = 0;  
    month = 0;  
    year = 0;  
}
```

# Initialisation using constructors

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date() {  
        day = 0;  
        month = 0;  
        year = 0;  
    }  
    void setdate(int,int,int);  
    void showdate();  
};
```

```
int main() {  
    Date d;  
    d.showdate();  
}
```

Date4.h

dateApp4.cpp

# Initialisation using constructors

---

```
class TicTacToe {  
private:  
    char board[3][3];  
    int noOfMoves;  
public:  
    TicTacToe() { //Default constructor.  
        for (int row = 0; row < 3; row++)  
            for (int col = 0; col < 3; col++) {  
                board[row][col] = ' ';  
            }  
        noOfMoves = 0;  
    }  
    //More code  
};
```

# Constructor with parameters

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
  
Public:  
    Date(int, int, int);  
    void showdate();  
};
```

```
Date::Date(int d, int m, int y) {  
    day = d;  
    month = m;  
    year = y;  
}
```

Like normal functions, constructors can take parameters.

Date4\_1.h

dateApp4\_1.cpp

# Multiple Constructors

```
class Date {  
    private:  
        int day, month, year;  
    public:  
        Date ();           // take no argument  
        Date (int );      // take one argument  
        Date ( int, int ); // take two arguments  
        Date ( int, int, int ); // take three arguments  
        void showdate();  
};
```

Default constructor

Function overloading allows us to have more than one constructors. A **default constructor** is the constructor which can be called with no arguments.

# Call constructors

```
class Date {  
private:  
    int day, month, year;  
public:  
    Date ();  
    Date (int );  
    Date (int, int );  
    Date (int, int, int );  
    void showdate();  
};
```

```
int main() {  
    Date d1;  
    Date d2(17);  
    Date d3(17,8);  
    Date d4(17, 8, 2020);  
    Date d5 = Date(17,8,2020);  
    d1.showdate();  
    d2.showdate();  
    d3.showdate();  
    d4.showdate();  
}
```

The same

Date4\_2.h

dateApp4\_2.cpp

# Default constructors

If the user does not define **any** constructor, the compiler will generate a **default constructor** with empty body. However, if the user define a constructor (with or without arguments), there is no automatically generated constructor.

```
class A {  
    private: int val;  
    public: A() { val=0; }  
};  
  
class B {  
    private: int val;  
    public: B(int i) { val=i; }  
};  
  
class C {  
    private: int val;  
};
```

Which of definitions of objects gives compiling error?

```
int main() {  
    A a;  
    B b;  
    C c;  
    return 0;  
}
```

constructor.cpp

# Initializer list

Date4\_3.h

dateApp4\_3.cpp

There are four ways to initialise data members:

- ◆ Pre-set the values:

```
Date() { day=16; month=8; year=2021; }
```

- ◆ Take from the user (provide the values when create an object of the class):

```
Date(int d, int m, int y) { day=d; month=m; year=y; }
```

Date d(16,8,2021);

- ◆ Initializer list (very useful for inheritance):

```
Date(): day(26), month(8), year(2021){ }
```

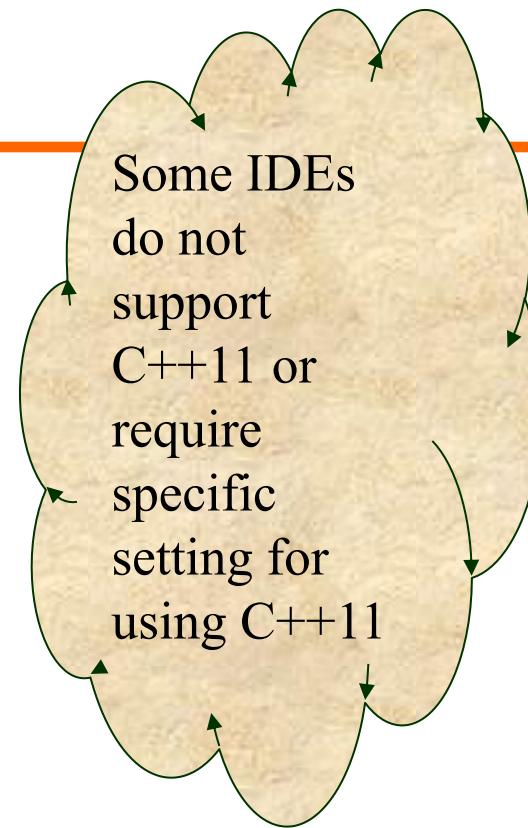
```
Date(int d, int m, int y): day(d), month(m), year(y){}
```

- ◆ Copy constructor: using an existing object to initialise the current object

InitializerList.cpp

# Initialisation

```
class Date {  
private:  
    int day = 0; // C++11  
    int month = 0; // C++11  
    int year = 0; // C++11  
public:  
    void setdate(int,int,int);  
    void showdate();  
};
```



Some IDEs  
do not  
support  
C++11 or  
require  
specific  
setting for  
using C++11

When you **define** a class, the compiler does not allocate memories to the class (except for static data members). Memories are allocated to objects of a class. Therefore, when an object is created, the object will gain memory for each data member and the “*constructors*” can then initialise these memories as specified in the constructors.

# Constructor using default values

```
#include<iostream.h>
class Time {
private:
    int hrs , mins, secs;
public:
    Time(int = 0, int = 0, int = 0);
    void display() {cout << hrs << ":"
                  << mins << ":" << secs << endl;}
};
Time::Time(int h, int m, int s) {
    hrs = h;
    mins = m;
    secs = s;
}
```

```
void main()
{
    Time t;
    Time t1(1);
    Time t2(2,20);
    Time t3(3,30,30);

    t.display();
    t1.display();
    t2.display();
    t3.display();
}
```

defaultArg.cpp

# Copy constructor: a preview

A copy constructor of a class is a special constructor for creating a new object as a copy of an existing object. The copy constructor is called whenever an object is initialized from another object of the same class. Typical declaration of a copy constructor:

```
ClassName( const ClassName& );
```

```
//Copy constructor.
TicTacToe(const TicTacToe& cboard) {
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            board[row][col] = cboard.board[row][col];

    noOfMoves = cboard.noOfMoves;
}
```

Ways of using a copy constructor:

```
TicTacToe board;
//More code on board
TicTacToe tempBoard(board); //tempBoard is a new object
TicTacToe* tempBoard = new TicTacToe(board);
```

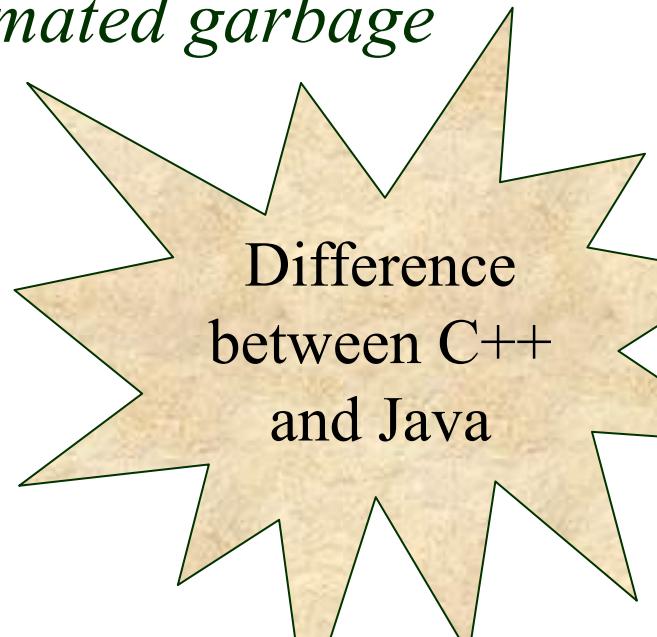
# Topics covered by the lecture

---

- ◆ Class declaration, definition and application
- ◆ Constructor
- ◆ Destructor
- ◆ Object composition

# Destructors

- ◆ A destructor is called **when an object is destroyed**. It is a function with the **same name** of the class only with a **~** (Tilda) in the beginning **without parameters**.
- ◆ One class has **only one** destructor.
- ◆ The destructor of a class is called automatically when an object of the class goes out of scope. It is typically used for clean-up and resource release (*no automated garbage collection in C++*).



# Constructors

```
class A {  
private:  
    int num ;  
public:  
    int getNumber() { return num; }  
    A(int i=0) { num=i; cout << i << "ctor" << endl;} //constructor  
    ~A() { cout << num << "dtor" << endl; } //destructor  
};
```

```
int main() {  
    A x(1);  
    {  
        { A y(2); }  
        A z(3);  
    }  
}
```

Automatic objects

destructor.cpp

What is the output of the program?

# Topics covered by the lecture

---

- ◆ Class declaration, definition and application
- ◆ Constructor
- ◆ Destructor
- ◆ Object composition

# Composition

---

- ◆ Complex objects are often built from smaller, simpler objects.
- ◆ The process of building complex objects from simpler ones is called object composition.
- ◆ Object composition models a “*has-a*” relationship between two objects.

```
class Room {  
    Desk console;  
    Chair chairs[50];  
    Doors doors[2];  
};
```

Any object of room contains space to store one object of console, 50 objects of chairs and 2 objects of doors.

# Class communication in composition

- ◆ Objects in a composed class can communicate each other via their interface (public functions).

```
class Board {  
    char grid[BOARDSIZE][BOARDSIZE];  
public:  
    bool addMove(int x1,int y1, int x2,int y2);  
    bool checkWin();  
    bool validInput(int x, int y);  
    void printBoard();  
};
```

```
class Player {  
protected:  
    int playerType;  
public:  
    void getMove(Board b,int& x,int& y);  
    int getType();  
};
```

```
class Game {  
    Board board;  
    Player player[2];  
public:  
    void play();  
};
```

```
void Game::play() {  
    while(!board.checkWin()) {  
        int x1, y1, x2, y2;  
        player[0].getMove(board, x1, y1);  
        player[1].getMove(board, x2, y2);  
        board.addMove(x1,y1,x2,y2);  
    }  
}
```

A game consists of a game board and two players.

# Homework

---

- ◆ Read textbook Chapter 6 & 7.
- ◆ Complete online tutorial 1 if you haven't.
- ◆ Show your tutor the solution of practical 4 if you haven't
- ◆ Start to work on assignment 1 if you haven't. Feel free to ask us any questions related to the assignment.

# Comp2014 Object Oriented Programming

---

## Lecture 6

# Pointers and Dynamic Memory Allocation

# Topics covered by last two lectures

---

- ◆ Objects and classes
- ◆ Abstraction, encapsulation and data hiding
- ◆ Class declaration, definition and application
- ◆ Constructor
- ◆ Destructor
- ◆ class composition

```

#include <iostream>
using namespace std;
const int SIZE = 10;

void fillArray(int a[], int size){
    for (int i = 0; i < size; i++)
        a[i] = rand() % 100;
}

void printArray(int a[], int size){
    for (int i = 0; i < size; i++)
        cout << a[i] << " ";
}

int main() {
    int arr[SIZE];
    srand(time(0));

    fillArray(arr, SIZE);
    printArray(arr, SIZE);

    return 0;
}

```

Procedural style

```

#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
public:
    void fillArray(int a[], int size){
        for (int i = 0; i < size; i++)
            a[i] = rand() % 100;
    }

    void printArray(int a[], int size){
        for (int i = 0; i < size; i++)
            cout << a[i] << " ";
    }
};

int main() {
    int arr[SIZE];
    srand(time(0));

    ooArray oo;
    oo.fillArray(arr, SIZE);
    oo.printArray(arr, SIZE);

    return 0;
}

```

Fake OO style

```

#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
public:
    void fillArray(int a[], int size){
        for (int i = 0; i < size; i++)
            a[i] = rand() % 100;
    }

    void printArray(int a[], int size){
        for (int i = 0; i < size; i++)
            cout << a[i] << " ";
    }
};

int main() {
    int arr[SIZE];
    srand(time(0));

    ooArray oo;
    oo.fillArray(arr, SIZE);
    oo.printArray(arr, SIZE);

    return 0;
}

```

Fake OO

```

#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
    int a[SIZE];
    int size;
public:
    ooArray() {size = SIZE;}
    void fillArray() {
        for (int i = 0; i < size; i++)
            a[i] = rand() % 100;
    }

    void printArray() {
        for (int i = 0; i < size; i++)
            cout << a[i] << " ";
    }
};

int main() {
    srand(time(0));

    ooArray oo;
    oo.fillArray();
    oo.printArray();

    return 0;
}

```

Real OO

# Object Oriented Programming Style

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class ooArray {
    int a[SIZE];
    int size;
public:
    ooArray() {size = SIZE;}
    void fillArray();
    void printArray();
};

void ooArray::fillArray() {
    for (int i = 0; i < size; i++)
        a[i] = rand() % 100;
}

void ooArray::printArray() {
    for (int i = 0; i < size; i++)
        cout << a[i] << " ";
}
```

```
int main() {
    srand(time(0));

    ooArray oo;
    oo.fillArray();
    oo.printArray();

    return 0;
}
```

MyooApp.cpp

Well organised OO style

Myoo.h

# Class communication in composition

```
class Board {  
    char grid[BOARDSIZE][BOARDSIZE];  
public:  
    bool addMove(int x1,int y1, int x2,int y2);  
    bool checkWin();  
    bool validInput(int x, int y);  
    void printBoard();  
};
```

```
class Player {  
protected:  
    int playerType;  
public:  
    void getMove(Board b,int& x,int& y);  
    int getType();  
};
```

```
class Game {  
    Board board;  
    Player player[2];  
public:  
    void play();  
};
```

A game consists of a game board and two players.



```
void Game::play() {  
    while(!board.checkWin()) {  
        int x1, y1, x2, y2;  
        player[0].getMove(board, x1, y1);  
        player[1].getMove(board, x2, y2);  
        board.addMove(x1,y1,x2,y2);  
    }  
}
```

# Topics of this lecture

---

- ◆ Pointers
- ◆ Pointers and arrays
- ◆ Pointers and functions
- ◆ `new` operator
- ◆ Dynamic memory allocation
- ◆ Copy constructor

# Pointers

- ◆ Pointer is a variable that stores the **address** of a memory cell in certain data type.

```
int a =10;
```

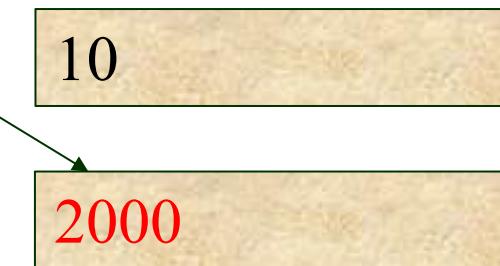


*2000*

```
int *pa;
```



*3000*



```
pa = &a;
```

# Declaration of pointers

- ◆ Declare pointers of built-in data types:

```
int *ptr1;           int* ptr1; //either way  
double *ptr2;  
string *ptr3;
```

- ◆ Declare pointers of user-defined data types:

```
Office *o;  
LectureTheater *lt;  
Kitchen *k;  
Board *board;
```



01001100	10111001	01101010
----------	----------	----------

Note that a pointer only points to the first memory cell of the data. Therefore we need to know the type of data.

# Address operator &

Ooh, now I understand call-by-reference

- The “*address of*” operator, **&**, returns the address of a variable

```
int value = 10;  
int *ptr;  
ptr = &value;
```

```
TicTacToe board;  
TicTacToe *boardPtr;  
boardPtr = &board;
```

- Assign right address to right pointer: *match types*.

```
int value = 10;  
int *ptr1;  
double *ptr2;  
ptr1 = &value; //legal  
ptr2 = &value; //illegal
```

# Dereference operator \*

pointer.cpp

- The dereference operator \* obtains the data item (can also be an *l-value*) the pointer points to.
  - \*p means “the variable that p points to”

```
int value;  
int *ptr;  
ptr = &value;  
  
value = 100;  
cout << *ptr << endl;  
  
*ptr = 200; /  
cout << value << endl;
```

Use the pointer to get the value

ptr

2000



Assign directly

value 2000

100

Assign to the variable ptr point to

ptr

2000



value 2000

200

& and \* are dual operators

# Pointer Assignments

- ◆ Pointer variables can be "assigned":

```
int *p1, *p2;  
p2 = p1;
```

assign address

- Assigns one pointer to another
- "Make p2 point to where p1 points"



# Pointer Assignments

- ◆ Pointer variables can be "assigned":

```
int *p1, *p2;  
*p2 = *p1;
```

assign value

- Get the value of the variable p1 point to
- Assign its value to the variable p2 point to



# Initialise pointers and pointer offset

---

- A pointer can be initialized by **NULL**, which indicate that the pointer points to nothing.

```
int *ptr = NULL;
```

- Pointer can be offset to another address:

```
cout << *(ptr+1) << endl;  
ptr = ptr +1;  
ptr++;
```

# Pointers and Functions

- ◆ An address can be **passed** to a function via a pointer
- ◆ A function can also **return** an address via a pointer
- ◆ Example:

return an address via a pointer

pass an address via a pointer

```
int* functionPointer(int* ptr);
```

- A pointer as a formal argument. Note that the actual argument must be either a pointer variable of the same type or an address of the type.
- If a function returns a pointer, you need to create a pointer of the same type in the calling module to catch the value of the function.

functionPointer.cpp

# Array and Pointers

- ◆ An array name is a pointer, pointing to the first memory cell of the array:

```
int a[ ] = {1,2,3,4,5};  
cout << *a << endl;
```

arrayPointer.cpp

- ◆ Can perform arithmetic on pointers
  - "Address" arithmetic

```
double d[ 10 ];
```

- d contains the address of d[0]: `*d` is equivalent to `d[0]`.
- `d+1` contains the address of `d[1]`: `*(d+1)` is equivalent to `d[1]`.
- `d+2` contains the address of `d[2]`: `*(d+2)` is equivalent to `d[2]`.
  - » *Address oriented vs value oriented*

# Return an array by returning a pointer

- ◆ Array type is NOT allowed as return-type of a function.

Example:

```
int[ ] someFunction(); // ILLEGAL!
```

- ◆ Instead return a pointer of the array:

```
int* someFunction(); // LEGAL!
```

arrayFunction.cpp

# The new Operator

newOperator.cpp

- ◆ What's wrong with the following program?

```
int main() {  
    int *ptr;  
    *ptr = 20;  
    cout << *ptr << endl;  
}
```

You have memory to store an address of an integer

You do not have memory to store an integer value

- ◆ Operator *new* creates memories (dynamic memory allocation)

```
int *ptr;  
ptr = new int;  
*ptr = 20;
```

**ptr** is the only indicator for the  
memory cell

- The **new** operator creates a new memory cell of integer and returns the address of the memory cell
- You can access the memory cell using a pointer

# Memory Management

---

- ◆ Dynamically-allocated memories are from the “*freestore*”, also called “**heap**”
- ◆ You use **new** operator to request a memory cell from the freestore.
- ◆ If your request is succeeded, you will receive an address with the allocated memory.
- ◆ The heap is limited. Once it is full, future **new** operations will fail. In this case, **new** will return a NULL.
- ◆ You release memory using the “**delete**” operator.
- ◆ The technology is called *dynamic memory allocation*.

# Dynamic Object Creation

---

- ◆ Create an object with *new* operator
  - *MyType \*fp = new MyType;*
  - *MyType \*fp = new MyType(1, 'a');*
- ◆ Free an object with *delete* operator
  - *delete fp;*
- ◆ Allocate an array of objects with *new[]* operator
  - *MyType \*fparr = new MyType[100];*
- ◆ Free a list of objects with *delete[]* operator
  - *delete[] fparr;*

# Dynamic Object Creation

---

- ◆ Two ways to create variables/objects

```
int value; //get memory to store an integer  
int *ptr; //get memory to store an integer pointer  
ptr = new int; //get memory to store an integer  
delete ptr; // return memory
```

```
Board board; //get memory to store a Board object
```

```
Board *boardPtr; //get memory to store a Board pointer  
boardPtr = new Board; //get memory to store a Board object  
  
delete boardPtr; // return memory
```

# The **->** operator and pointer of pointers

---

Call a data member or member function from a pointer:

```
Date *p;  
p = new Date;  
( *P ) . showdate();
```

Equivalently, we write:

```
p->showdate();
```

```
Board board;
```

```
board.play(); //Call a method from an object
```

```
Board *boardPtr;
```

```
boardPtr = new Board;
```

```
boardPtr->play(); //Call a method from an object pointer
```

# Dynamic array

```
class DynamicArray {  
private:  
    int* darray;  
    int size;  
public:  
    DynamicArray() {  
        cout << "Input the size of the array:" << endl;  
        cin >> size;  
        darray = new int[size];  
    }  
  
    void input() {  
        for (int i = 0; i < size; i++) cin >> darray[i];  
    }  
  
    void output() {  
        for (int i = 0; i < size; i++) cout << darray[i] <<  
        endl;  
    }  
  
    ~DynamicArray() { delete[ ] darray; }  
};
```

You may try to use vector, which is a dynamic array template.

You do not have to know the size of the array at the time you program.



dynamicArray.cpp  
arrayFunction.cpp  
10-09.cpp

two-dimensional  
dynamic array

```
int* grid[10]; // array that stores 10 integer pointers  
int** grid; // grid is an array of arrays
```

# Return an array

creatArray.cpp

```
int* createArray(int size) {  
    int* temp = new int[size];  
    for (int i = 0; i < size; i++)  
        temp[i] = 0;  
    return temp;  
}
```

```
int main( )  
{  
    int* a = createArray(1000000);  
    for (int i = 0; i < 1000000; i++)  
        cout << a[i] << " ";  
    delete[] a;  
    return 0;  
}
```

Borrow memory

TIMELY RETURN OF A LOAN  
MAKES IT EASIER  
TO BORROW  
A SECOND TIME

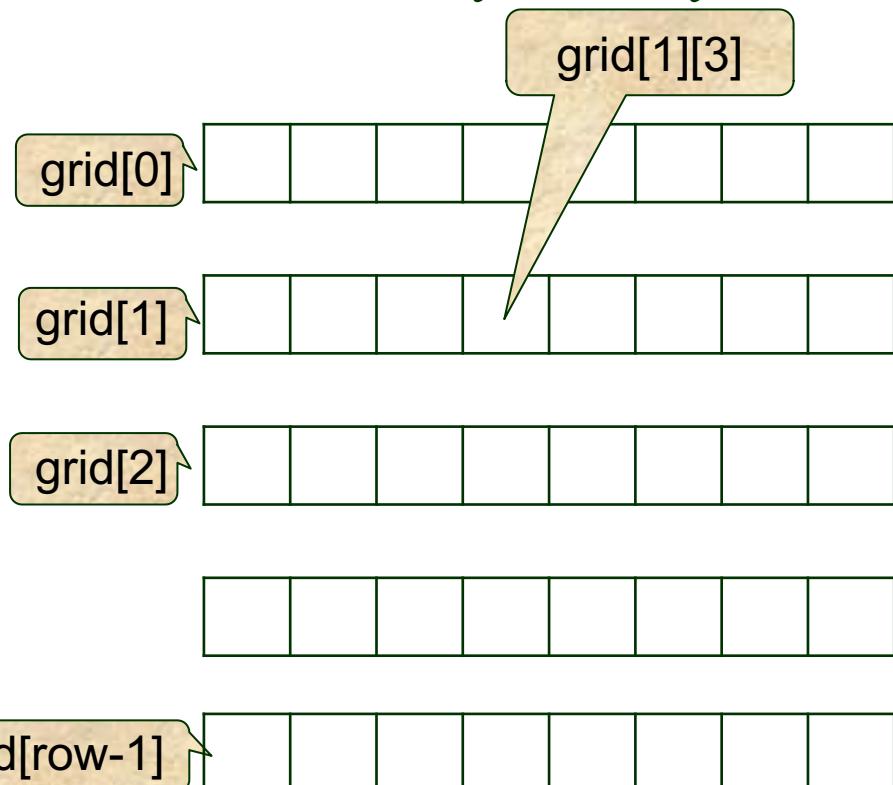
QUOTESOFAHEAVEN.BLOGSPOT.COM

Return memory

# Two-dimensional array

- ◆ A two-dimensional array is an array of arrays thus the name of a two-dimensional array is **a pointer of pointers**
- ◆ To create a 2D array, you must create **an array of pointers** to store the addresses of rows (a set of one-dimensional arrays).
- ◆ To delete a 2D array, you also need to delete an array of arrays

```
int **grid = new int*[row];
for (int i = 0; i < row; i++)
    grid[i] = new int[col];
```



```
for (int i = 0; i < row; i++)
    delete[] grid[i];
delete[] grid;
```

# When do we need a pointer?

---

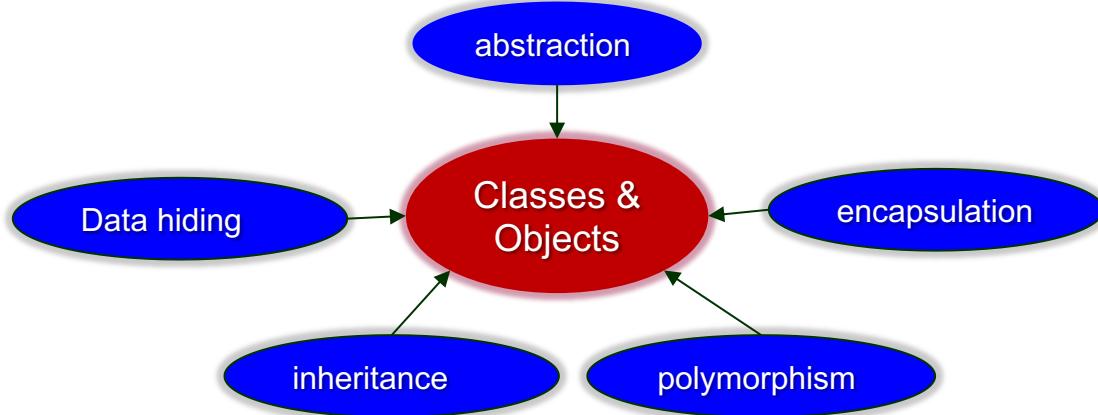
- Dynamic memory allocation requires a pointer to access the memory
- Sometime pass the address of an object is more **efficient** and **safer** than pass the object itself.
- You can return an array from a function if the array is created using dynamic memory allocation.
- You can even pass a function to another function because the function name is a pointer points to the code of the function.
- Sometimes, inheritance requires to use pointers to avoid **object slicing** (Lecture 10).
- Other chances to use pointers. The more you use pointers, the more you love them.

# Homework

---

- ◆ Read textbook Chapter 10.
- ◆ Complete practical 4 if you haven't. The practical contains more training for assignment 1.
- ◆ Work on practical 5

# Object Oriented Style

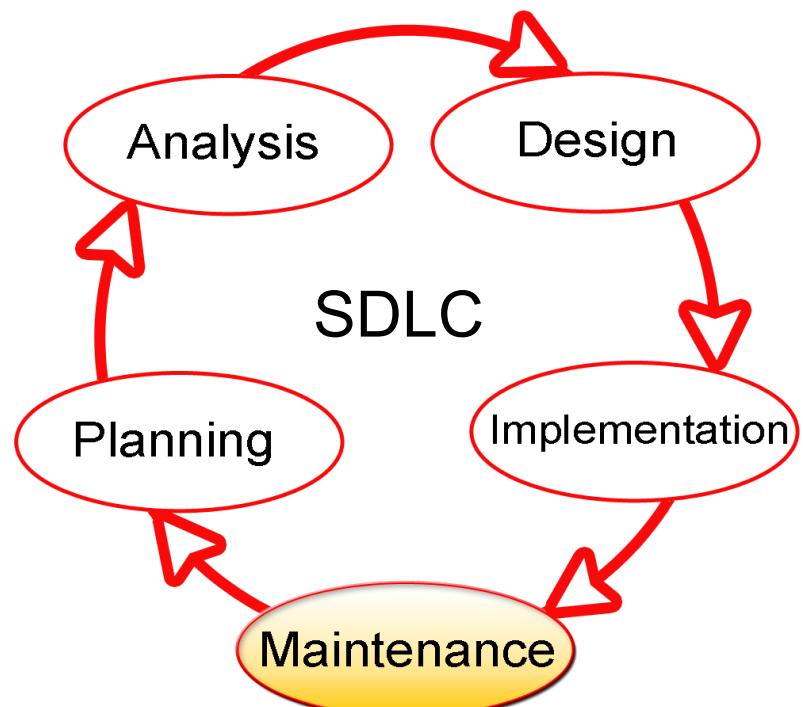


Object Oriented Programming

Software Development Life Cycle  
Iteration and Evolution

Focus of assignment 1:

- Abstraction
- Encapsulation
- Data hiding



# OO Style for Assignment 1 – levels

1. Procedural code for Tic Tac Toe game
2. OO code for Tic Tac Toe (starting point for assignment 1)
3. Class Board – single class solution
4. Add Class Game - two class solution
5. Add Class Player and more – multiple class solution

```
class Board {  
    int grid[boardSize][boardSize];  
public:  
    char addMove(int x1, int y1, int x2, int y2);  
    void winningStatus();  
    bool isValid(int x, int y);  
    bool isFull();  
    void displayBoard();  
    ...  
    // char play() // single class solution  
};  
class Game {  
    Board board;  
    //Player* players[2]; //for three+ class solution  
public:  
    //Game(Player*, Player*); //for three+ class solution  
    char play(); //two or multiple class solution  
    ...  
};
```

```
class Player {  
protected:  
    int player;  
public:  
    virtual void getMove(Board*,  
        int&, int&) = 0;  
    ...  
};
```

```
int main() {  
    Player* players[2];  
    players[0] = new HumanPlayer(1);  
    players[1] = new MindfulPlayer(-1);  
    Game game(players[0], players[1]);  
    game.play();  
    // Board board;  
    // board.displayBoard();  
}
```

# Comp2014 Object Oriented Programming

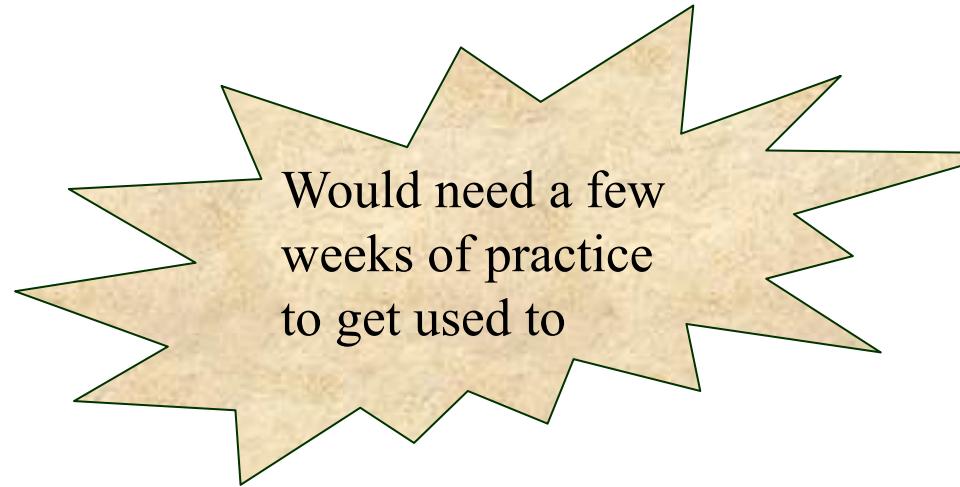
---

## Lecture 7

**Strings, Stream, Static  
Constant and File I/O**

# Topics covered by last lecture

- ◆ Pointers
- ◆ Pointers and arrays
- ◆ Pointers and functions
- ◆ **new** operator
- ◆ Dynamic memory allocation



Buy a house

```
House myhouse;  
//you live there until you die
```

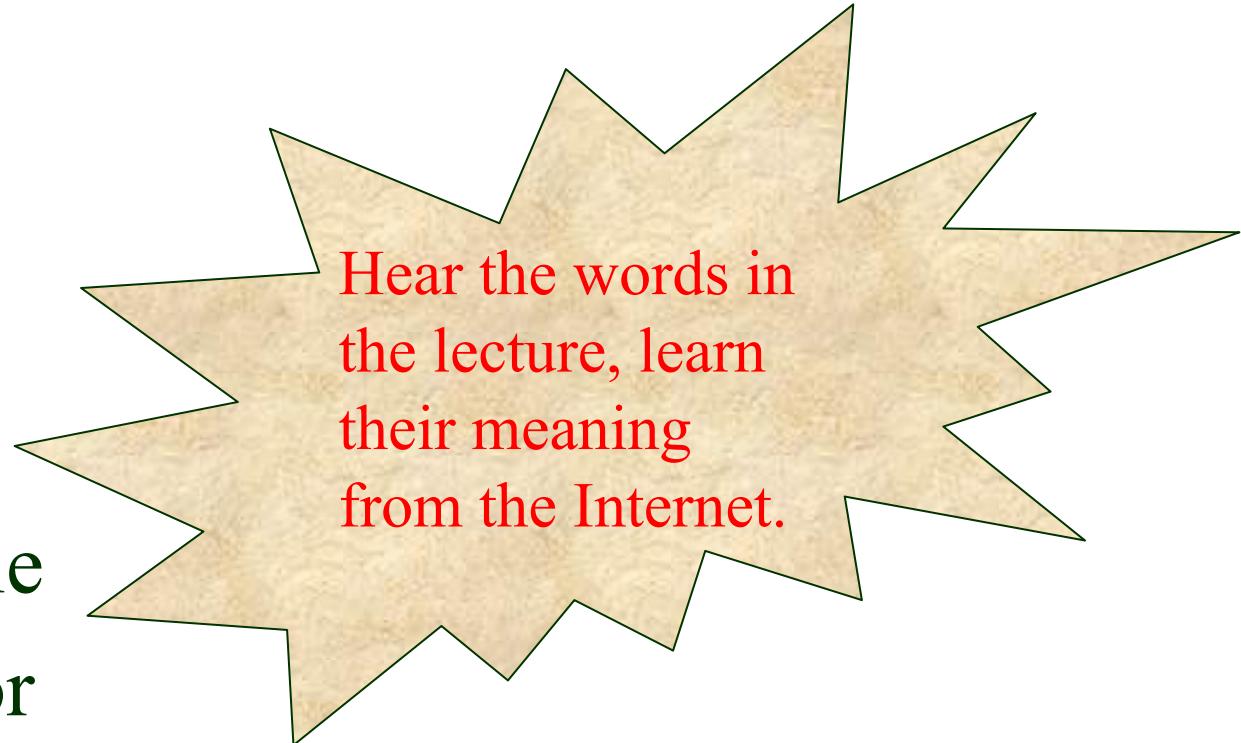
Rent a house

```
House *housekey = new House;  
//you live there as long as you pay rent  
delete housekey;
```

# Topics covered by this lecture

---

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor



# Two string types

string1.cpp

- ◆ C-string: inherited from C
  - Define a string as an array of `char`, say  
`char s[10];`
  - End of string will be marked with null, ‘`\0`’, automatically
  - Operate the string as an array with dozens of extra functions, a bit complicated and hard to remember though.

```
#include <cstdlib>
#include <cstring>
```

- ◆ Standard class: `string`
  - Header file: `#include <string>`
  - Define a string as an object of Class `string`, say  
`string s;`
  - Also dozens of member functions for string operations

Highly recommended

# I/O with Class string

- ◆ Use `cin` and `cout` for input and output

stringIO.cpp

```
string s;  
cin >> s;  
cout << s;
```

- ◆ It stops at a whitespace:

Input: “Hello dongmo!”

Output: “Hello”

- ◆ For complete lines, use `getline()` function:

```
string line;  
cout << "Enter a line of input: ";  
getline(cin, line);  
cout << line << endl;
```

# getline(): more options

getline.cpp

- ◆ Can stop reading by specifying "delimiter" character:

```
string line;  
cout << "Enter input: ";  
getline(cin, line, '?');
```

- Receives input until "?" encountered

- ◆ Be careful mixing `cin >> var` and `getline()`

```
int n;  
string line;  
cin >> n;  
getline(cin, line);
```

Note that `cin>>n` stops at “`\n`”, leaving it on the stream for `getline()`!

# Typical C-style functions

---

- **atof**: Convert string to double
- **atoi**: convert string to integer
- **strcpy**: copy string
- **strcat**: concatenate strings
- **strcmp**: Compare two strings
- **strchr**: Locate first occurrence of character in string
- **strstr**: Locate substring



# Some Member Functions of Class `string`

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
<b>Constructors</b>	
<code>string str;</code>	Default constructor; creates empty <code>string</code> object <code>str</code> .
<code>string str("string");</code>	Creates a <code>string</code> object with data "string".
<code>string str(aString);</code>	Creates a <code>string</code> object <code>str</code> that is a copy of <code>aString</code> . <code>aString</code> is an object of the class <code>string</code> .
<b>Element access</b>	
<code>str[i]</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> .
<code>str.at(i)</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> .
<code>str.substr(position, length)</code>	Returns the substring of the calling object starting at <code>position</code> and having <code>length</code> characters.
<b>Assignment/Modifiers</b>	
<code>str1 = str2;</code>	Allocates space and initializes it to <code>str2</code> 's data, releases memory allocated for <code>str1</code> , and sets <code>str1</code> 's size to that of <code>str2</code> .
<code>str1 += str2;</code>	Character data of <code>str2</code> is concatenated to the end of <code>str1</code> ; the size is set appropriately.
<code>str.empty( )</code>	Returns <code>true</code> if <code>str</code> is an empty <code>string</code> ; returns <code>false</code> otherwise.

(continued)

# Typical Member Functions of Class `string`

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
<code>str1 + str2</code>	Returns a <code>string</code> that has <code>str2</code> 's data concatenated to the end of <code>str1</code> 's data. The size is set appropriately.
<code>str.insert(pos, str2)</code>	Inserts <code>str2</code> into <code>str</code> beginning at position <code>pos</code> .
<code>str.remove(pos, length)</code>	Removes substring of size <code>length</code> , starting at position <code>pos</code> .
<b>Comparisons</b>	
<code>str1 == str2</code> <code>str1 != str2</code>	Compare for equality or inequality; returns a Boolean value.
<code>str1 &lt; str2</code> <code>str1 &gt; str2</code>	Four comparisons. All are lexicographical comparisons.
<code>str1 &lt;= str2</code> <code>str1 &gt;= str2</code>	
<code>str.find(str1)</code>	Returns index of the first occurrence of <code>str1</code> in <code>str</code> .
<code>str.find(str1, pos)</code>	Returns index of the first occurrence of string <code>str1</code> in <code>str</code> ; the search starts at position <code>pos</code> .
<code>str.find_first_of(str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character in <code>str1</code> , starting the search at position <code>pos</code> .
<code>str.find_first_not_of(str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character <i>not</i> in <code>str1</code> , starting search at position <code>pos</code> .



# C-string and string: object conversions

- ◆ Automatic type conversions

- From c-string to string object:

```
char aCString[ ] = "My C-string";  
string stringObj = aCString; //copy constructor
```

- » Perfectly legal and appropriate!

- From string object to c-sting

```
aCString = stringObj; //Illegal!
```

- » Cannot automatically convert a string object to a c-string

- Must use explicit conversion:

```
strcpy(aCString, stringObj.c_str());
```



Remember the way of conversion,  
get benefit from both

# Topics covered by this lecture

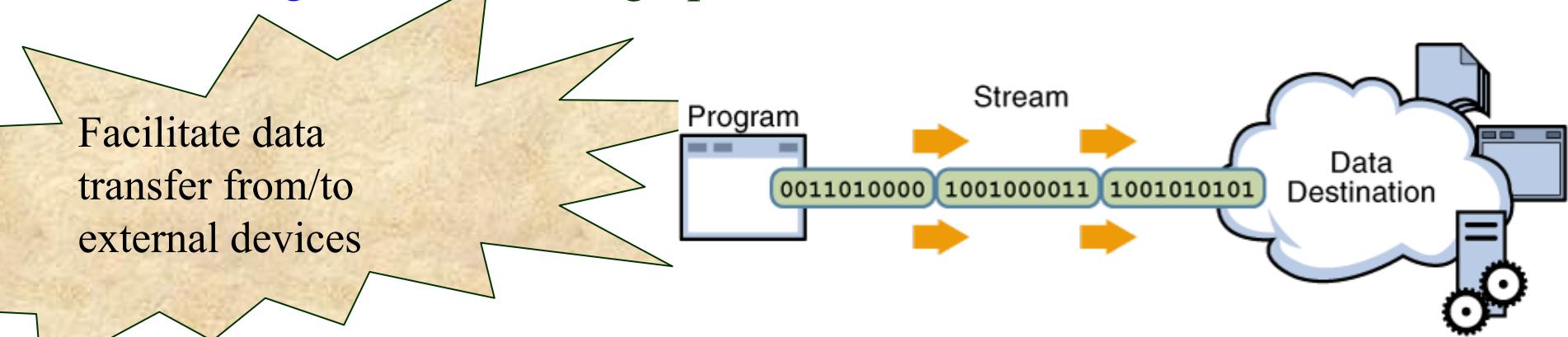
---

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor

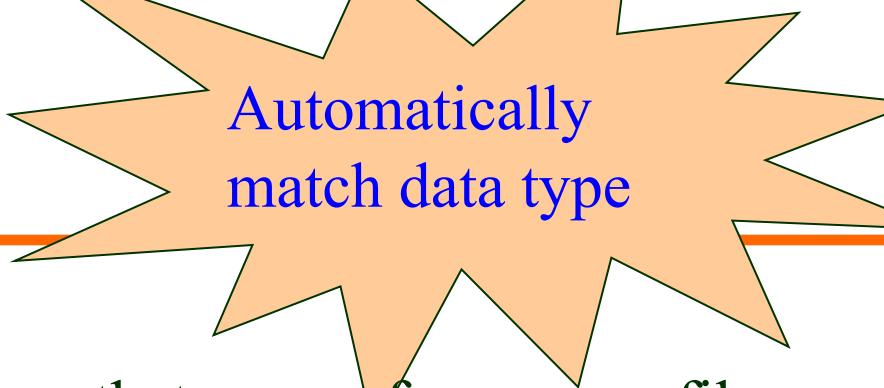


# Streams

- ◆ A stream is a flow of characters.
  - If the flow is into your program, the stream is called an **input stream** (istream).
  - If the flow is out of your program, the stream is called an **output stream** (ostream).
- ◆ Most useful stream classes
  - **<iostream>**: input and output via standard devices
  - **<fstream>**: input and output via files
  - **<stringstream>**: string operation via buffer



# Streams



Automatically  
match data type

- ◆ Use operators: >> and <<:

- Assume that `inStream` is a stream that comes from some file:

```
int value;
```

```
inStream >> value;
```

- » Reads value from the stream, assigned it to *value* as an integer

```
double num;
```

```
inStream >> num;
```

- » Reads value from the stream, assigned it to *num* as a double.

- Assume that `outStream` is a stream that goes to some file

```
outStream << value;
```

- » Writes value to stream



It's called buffer in Java.

# I/O stream character operations

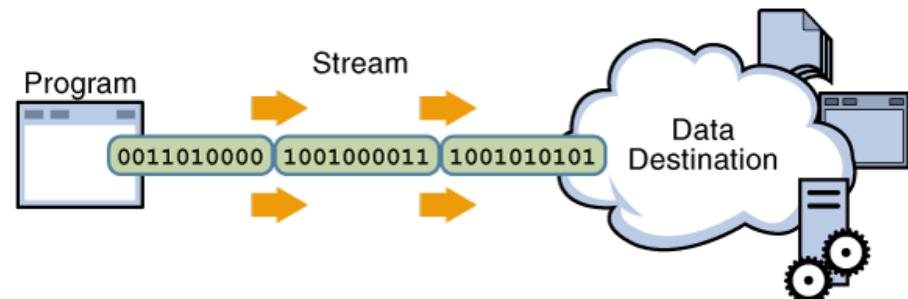
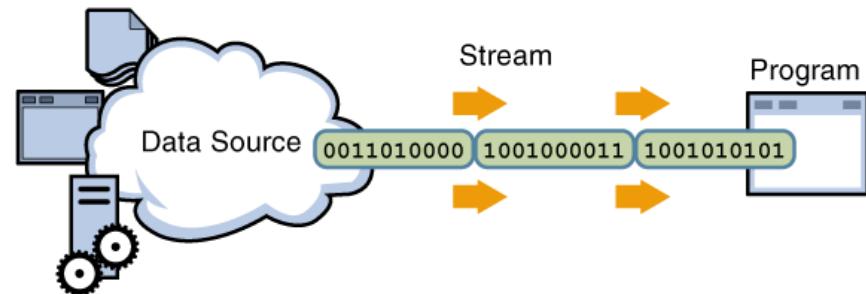
- ◆ More I/O stream member functions:

- `get(char)`
- `getline(string)`
- `put(char)`
- `putback(char)` // used in istream
- `peek()` // reads the next character from the input stream without extracting it
- `ignore()`



# Topics covered by this lecture

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor



# File Input: a template

```
#include <iostream>
#include <fstream>
use namespace std;
int main( )
{
    ifstream fin;
    fin.open("data_in.txt");
    string word;

    while (!fin.eof()) {
        fin >> word;
        cout << word << endl;
    }
    fin.close();

    return 0;
}
```

File I/O stream header file

File input stream object

Open file

Read from the stream word by word

Close file

fileInput.cpp

# File output: a template

```
#include <iostream>
#include <fstream>
use namespace std;
int main( ) {
    ofstream fout;
    fout.open("data_out.txt");
    int x = 2;
    int y = 3;
    fout << "x + y = "
        << (x+y);
    fout.close();
    return 0;
}
```

File I/O stream header file

File input stream object

Open file

Output to the stream

Close file

fileOutput.cpp

# Appending to a File

---

- ◆ Standard open operation begins with empty file
  - Even if file exists, contents lost
- ◆ Open for appending:

```
ofstream fout;  
fout.open("data_out.txt", ios::app);
```

- If the file doesn't exist, create one
- If the file exists, append the new input to the end of the file

# Checking End of File

---

- ◆ Use loop to process file until end
- ◆ Two methods to test *end of file*
  - Use member function `eof()`

```
ifstream fin;  
fin.open("data_in.txt");  
char next;  
while (!fin.eof()) {  
    fin.get(next);  
    cout << next;  
}
```

- Reads each character until file ends
- `eof()` member function returns bool

# End of File Check with Read

---

- ◆ Use `>>` operator

```
ifstream fin;
fin.open("data_in.txt");
double next, sum = 0;
while (fin >> next) {
    sum += next;
}
cout << "the sum is " << sum << endl;
```

- ◆ Read operation returns `bool` value!

`(fin >> next)`

- » Expression returns `true` if read successful

- » Returns `false` if attempt to read beyond end of file

- ◆ Copy-and-paste would introduce invisible characters into a file, which might cause bugs.

# Tools: File Names as Input

---

## ◆ Stream open operations

```
char fileName[16];
ifstream fin;
cout << "Enter file name: ";
cin >> fileName;
fin.open(fileName);
```

- Allows the user to provide file name in real time
- Include a full path to the file in filename unless it is located in the default folder

# Formatting Output with Stream Functions

- ◆ Format decimal number output:

```
fout.setf(ios::fixed);  
fout.setf(ios::showpoint);  
fout.precision(2);
```

- ◆ They are in library <iomanip>, called manipulator.

- ◆ Member function **setf()**

- Allows multitude of output flags to be set

- ◆ Member function **precision(x)**

- Decimals written with "x" digits after decimal

- ◆ Member function **width(x)**

- Sets width to "x" for outputted value
  - Only affects "next" value outputted

# Topics covered by this lecture

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor



# Static variables

Think about the difference between the following two versions of function `func()`:

```
int func() {  
    int i = 0;  
    i++;  
    return i;  
}
```

```
int func() {  
    static int i = 0;  
    i++;  
    return i;  
}
```

```
int main() {  
    cout << func() << endl;  
    cout << func() << endl;  
    cout << func() << endl;  
    return 0;  
}
```

staticinFunction.cpp

# Static variables

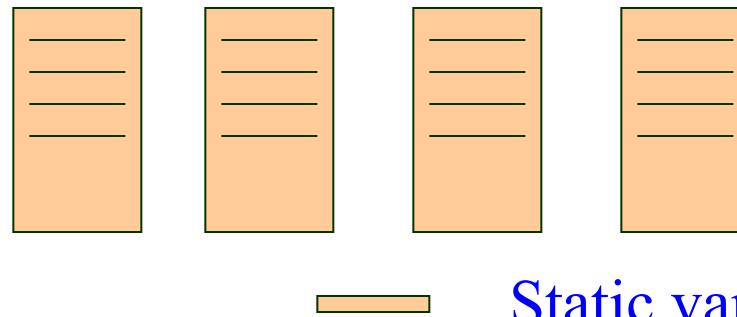
---

- ◆ In C/C++, the keyword **static** has two basic meaning:
  - Allocated once at a fixed address: *static storage*.
  - Only initialize once (it's done when the program is compiled).
- ◆ Static variables in a function: *remember values between function calls ( initialized when the first time being called and remain the value for further call without re-initialized ).*
- ◆ Static variables in a class: *belongs to the class but not to an object. All the objects of the class share the same static variables.*

# Static data members of a class

Properties of a static data member

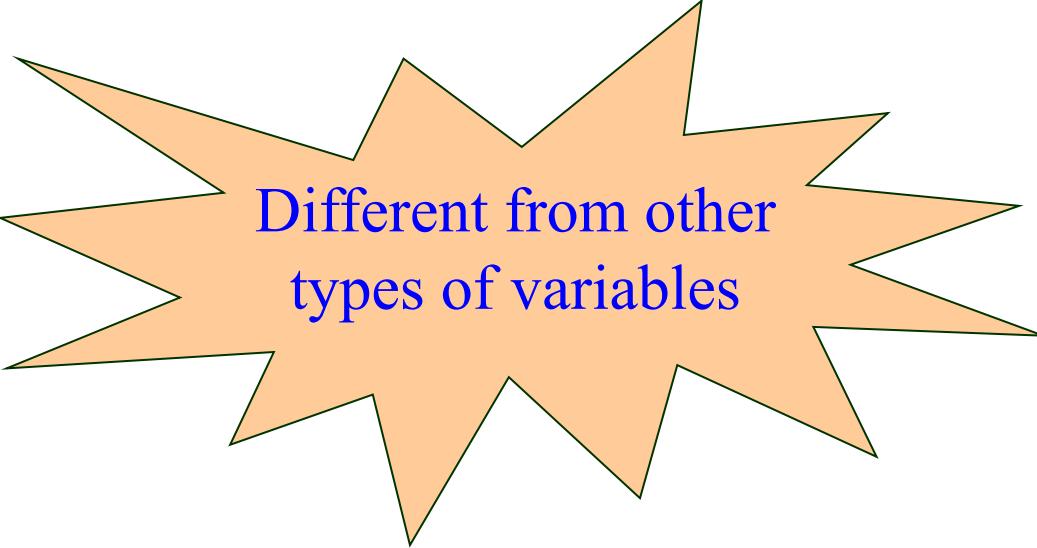
- ◆ All objects of the class "share" the same storage
- ◆ Useful for "tracking" objects of the same class
  - How often a member function is called?
  - How many objects of a class have been created?
  - Identify different object, say object ids.



# Static data members of a class

- ◆ A static member in a class should be initialized independently to individual objects.
- ◆ The way to initialize a static member of a class:
  - `int WithStatic::x = 10;`

StaticInClass.cpp

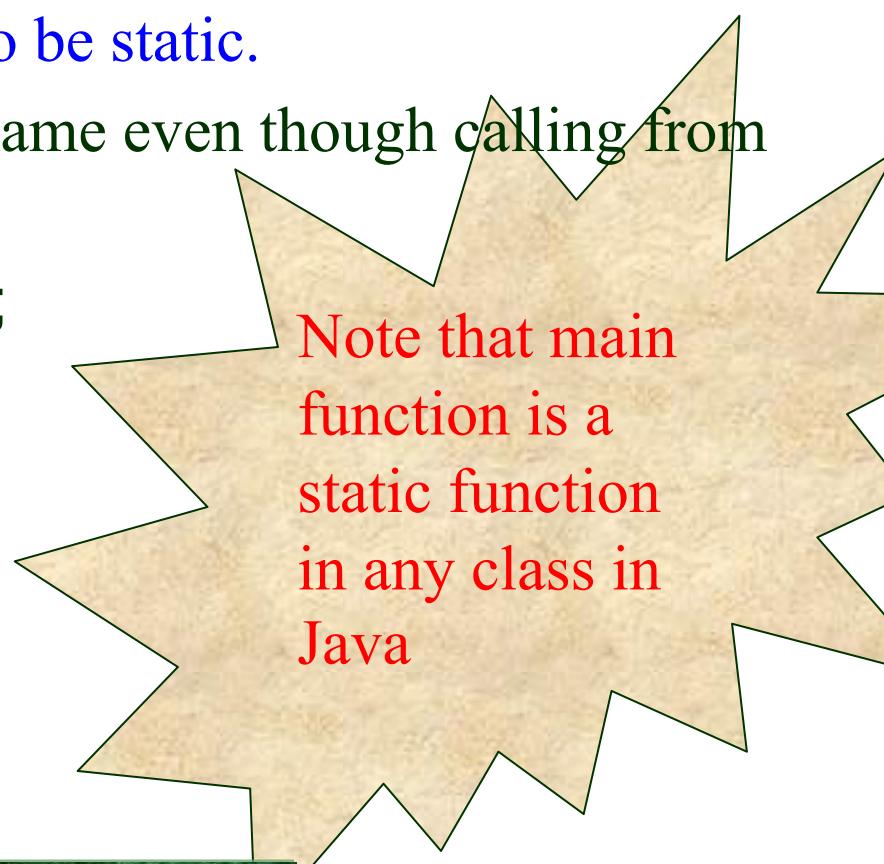


Different from other  
types of variables

# Static member functions of a class

A member function in a class can also be static:

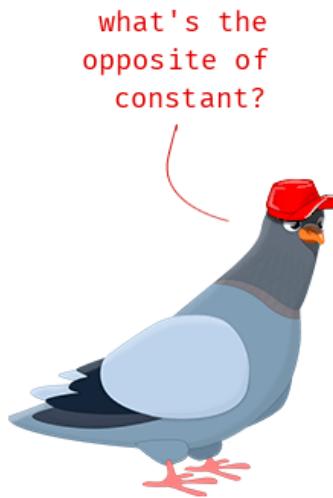
- ❖ It is independent to the objects of the class, which requires **all variables in a static function needs to be static.**
- ❖ Normally called by using the class name even though calling from an object is ok.
  - `ClassName::StaticFunction();`



Note that main function is a static function in any class in Java

# Topics covered by this lecture

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor



inconstant, fickle, unstable,  
irregular, unsteady,  
intermittent, changeable,  
disloyal, untrue, fluctuating



# The concept of constants

- ◆ The concept of constant (`const` in C++) is created to allow the programmer to draw a line between what changes and what doesn't. It's a **technique** that a programmer takes use of compiler to help him check his programs.
- ◆ `const` is used for variables which values never change once they are initialized. It can be also used for functions.
- ◆ A constant variable must be initialized while its declaration. You will get an error message if you accidentally try to change the value of a constant.
- ◆ Pre-processor vs constants:
  - C style: `#define BUFSIZE 100`
  - C++ style: `const int bufsize = 100;`

constant.cpp

# Constant objects and member functions

---

- ◆ Constants can be any user defined data type.
  - `const Date newYear(1,1);`
  - `const Date ChristmasOfTheYear(25,12,2017);`
- ◆ A constant object can only call a constant member function.
- ◆ A member function that is specifically declared `const` is treated as one that will not modify data members in the class or will not call a non-`const` member function.
  - `int getDay() const;`
  - `void printDate() const;`

# Constant member functions

- ◆ *const* modifier after function declaration indicates that the function does not modify the state.

```
class Date {  
    int d, m, y;  
public:  
    int getDay() const { return d ; }  
    int getNextYear() const ;  
    // ...  
};
```

Time.h

timeApp.cpp

```
int Date::getNextYear() const { return y++; }      // Error  
int Date::getNextYear() { return y; }                //wrong  
int Date::getNextYear() const { return y; }          // correct
```

# Topics covered by this lecture

---

- ◆ String
- ◆ Stream
- ◆ file I/O
- ◆ Static variable
- ◆ Constant variable
- ◆ Copy constructor

# Copy constructor: a review

A copy constructor of a class is a special constructor for creating a new object as a copy of an existing object. The copy constructor is called whenever an object is initialized from another object of the same class. Typical declaration of a copy constructor:

```
ClassName( const ClassName& );
```

```
Board(const Board& cboard) {  
    boardSize = cboard.getBoardSize();  
  
    grid = new int*[boardSize];  
    for(int i = 0; i < boardSize; i++)  
        grid[i] = new int[boardSize];  
  
    for(int i = 0; i < boardSize; i++)  
        for(int j = 0; j < boardSize; j++) {  
            grid[i][j] = cboard.grid[i][j];  
        }  
}
```

Ways of using copy constructor:

```
Board tempBoard = board;
```

```
Board tempBoard(board);
```

```
Board* tempBoard  
= new Board(board);
```

# Homework

---

- ◆ Read relevant content in Chapters 7, 9 & 12.
- ◆ Please work on your assignment 1. You have all knowledge now for all the tasks. However, if you still have difficulties to complete practical tasks up to practical 5, please come to PASS or try to get help from your tutor.
- ◆ If you have finished your assignment 1, your tutor can mark it in the practical class and you can always improve your code if you do not like your marks.

**You are not allowed to fail this  
assignment!**

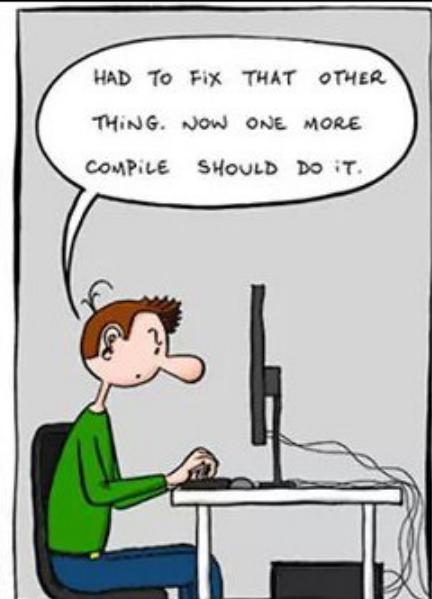


COMPILES  
AND WORKS FIRST TIME.  
WHAT DID I DO  
WRONG?



ALL RIGHT, ONE LAST  
COMPILATION AND THIS  
SHOULD WORK.

THEN I'LL GO  
GET LUNCH.



HAD TO FIX THAT OTHER  
THING. NOW ONE MORE  
COMPILE SHOULD DO IT.



AH, THAT WAS THE  
ACTUAL PROBLEM. NOW  
IT SHOULD WORK AFTER  
THIS FINAL COMPILATION.



I THINK I NAILED IT.  
JUST THIS ONE LAST  
COMPILE...

THEN WHICH.



# Assignment 1 is over!

# Comp2014 Object Oriented Programming

---

## Lecture 8

## Inheritance

# Topics covered by last lecture

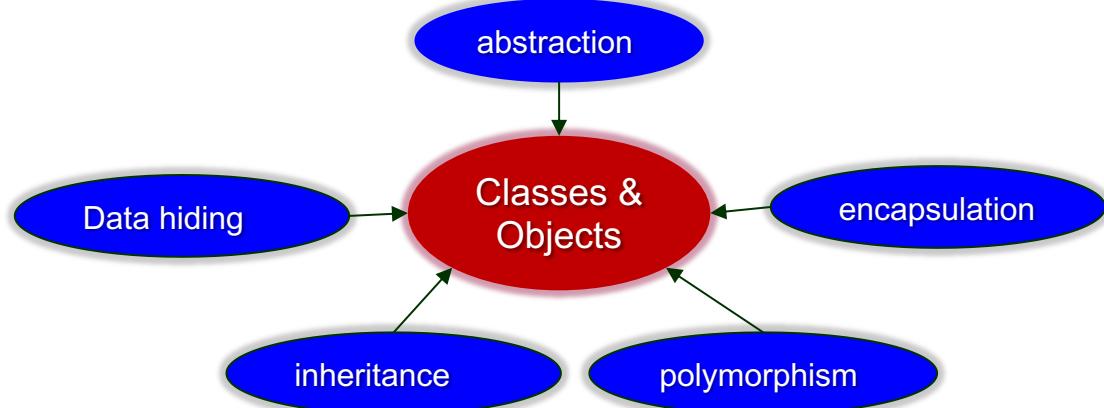
---

- ◆ String as an array of characters
- ◆ String as an object of string class
- ◆ Stream and file I/O
- ◆ Static variables
- ◆ Constant variables
- ◆ Copy constructor



# Topics covered by this lecture

- ◆ Composition
- ◆ Inheritance
  - Class declaration
  - Inheritance type
  - Inherited access
  - Constructor and destructor
- ◆ Type conversion
- ◆ Class hierarchies



Object Oriented Programming

# Composition

---

There are two ways to define a new data type from existing data type:

- ◆ **Composition**: declare a new data type by using existing data types
- ◆ **Inheritance**: derive a new data type from existing data types.
- ◆ Use which for what?
  - **Composition**: *A is part of B*. e.g. engine is part of car.
  - **Inheritance**: *A is a B*. e.g. a SmartPlayer is a Player.  
A “ComputerScienceBook” is a Book.

# Composition vs Inheritance

- ◆ Objects in a composited class can communicate each other via their interface (public functions).

```
class Board {  
    TicTacToe grid[boardSize][boardSize];  
    BoardCoordinate focus;  
public:  
    bool addMove(int,int,int,int);  
    char checkWin();  
    void printBoard();  
    . . .  
};
```

```
class Game {  
    Board* board;  
    Player *players[2];  
public:  
    Game(Board*,Player*,Player*);  
    void play();  
};
```

A game consists of a game board and two players - Composition

```
class Player {  
protected:  
    char playerSymbol;  
public:  
    virtual void getMove(Board*,int&,int&)=0;  
    char getSymbol();  
    . . .  
};
```

```
void Game::play() {  
    while(!board.checkWin()) {  
        int x1, y1, x2, y2;  
        player[0]->getMove(board,x1,y1);  
        player[1]->getMove(board,x2,y2);  
        board.addMove(x1,y1,x2,y2);  
        board.printBoard();  
    }  
}
```

# Composition vs Inheritance

```
class Player {  
protected:  
    char playerSymbol;  
public:  
    virtual void  
        getMove(Board*,int&,int&)=0;  
    char getSymbol();  
    ...  
};
```

All these classes are almost the same except the implementation of *getMove* function

```
class HumanPlayer : public Player {  
public:  
    void getMove(Board*,int& x,int& y) {  
        cin >> x >> y;  
        x--;y--;  
    };
```

```
class RandomPlayer : public Player {  
public:  
    void getMove(Board* bPtr,int& x,int& y) {  
        do {  
            x = index / bPtr->getSize();  
            y = index % bPtr->getSize();  
        } while(!bPtr->isValid(x,y))  
        return;  
    };
```

```
class MonteCarloPlayer : public Player {  
    double simulation(Board b);  
    double expansion(Board b);  
public:  
    void getMove(Board*,int&,int&);  
};
```

# Inheritance

## A Manager is an Employee

```
class Employee {  
public:  
    string    firstName, lastName;  
    int      employeeId;  
    Date     hiringDate;  
};
```

```
class Manager: public Employee {  
public:  
    int      level ;  
    string   officeNumber;  
};
```



- ◆ Manager is derived from Employee. (Derivation), or Employee is a base class for Manager.
- ◆ Manager has all the members of Employee in addition to its own members.

Manager m1;

m1.firstName = "John"; m1.lastName = "Smith";

# Inheritance

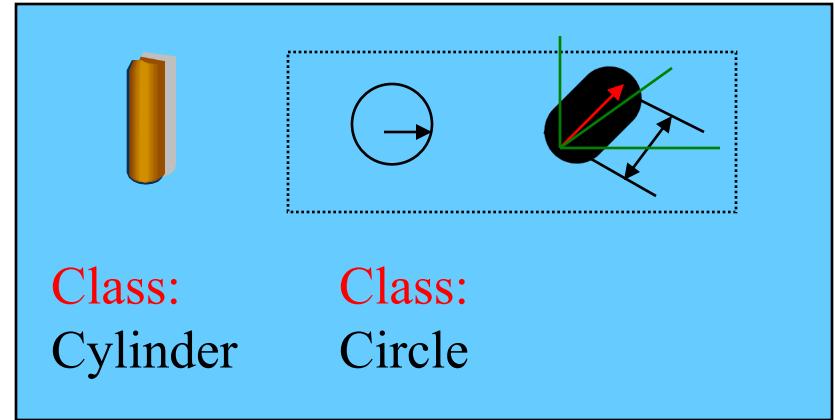
---

- ◆ Inheritance allows one class to be derived from existing classes by acquiring, or **inheriting**, their *data items* and *member functions*. It can then be altered by **adding new** data members or member functions, or by **modifying** (**overriding**) **existing** member functions and their access privileges.

# Inheritance Declaration

```
class Circle {  
    private:  
        double radius;  
    public:  
        Circle(double r=1.0) { radius = r; }  
        double calVal();  
};
```

```
double Circle::calVal(void) // this calculate the area of the circle  
{  
    return (PI * radius * radius);  
}
```

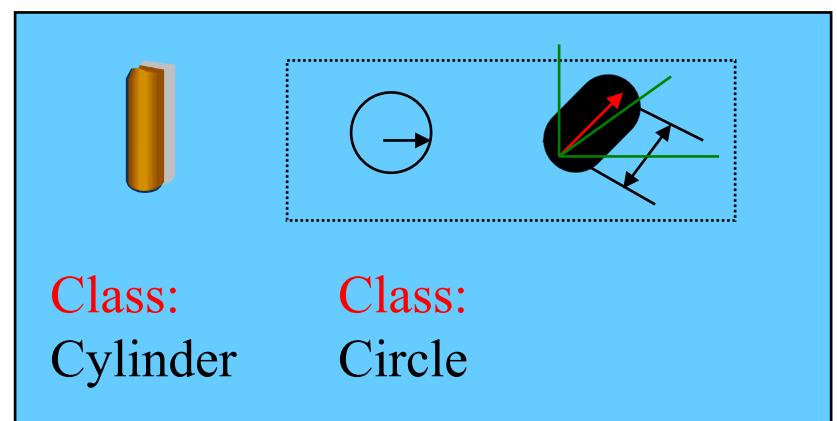


It is important to have a constructor to initialise data members!

# Inheritance Declaration

```
class Cylinder : public Circle {  
private:  
    double length;  
public:  
    Cylinder(double r, double l): Circle(r),length(l) {}  
    double calVal();  
};  
  
double Cylinder::calVal(void)  
{  
    return (length*Circle::calVal());  
}
```

Initialise the data members of the base class.



# Inheritance Declaration

---

```
int main() {  
    Circle circle1, circle2(2); // create two Circle obj  
    Cylinder cylinder1(3,4); // create one Cylinder obj  
  
    cout << "The area of circle_1 is " << circle1.calVal() << endl;  
    cout << "The area of circle_2 is " << circle2.calVal() << endl;  
    cout << "The volume of cylinder1 is " << cylinder1.calVal()  
        << endl;  
  
    circle2 = cylinder1; // assign a cyl to a Circle  
    cout << "\nThe area of circle_1 is now " << circle2.calVal() << endl;  
}
```

Circle.cpp

# Member ownership and access

- ◆ A member of a derived class can use any **public** or **protected** members of its base class.

```
class Employee {  
private:  
    string firstName, lastName;  
    char middleInitial;  
protected:  
    string fullName() {return firstName+' '+middleInitial+' '+lastName; }  
public:  
    // constructor is needed here.  
    void print() {cout << "Employee " << fullName() << endl;}  
};  
class Manager: public Employee {  
    int level;  
public:  
    // constructor is needed here.  
    void printManager() { cout << "Manager " << fullName() << " at level ="  
        level << endl; }  
};
```

# Function overriding

- ◆ A member of a derived class can override a **public** or **protected** members of its base class.

```
class Employee {  
private:  
    string firstName, lastName;  
    char middleInitial;  
protected:  
    string fullName() {return firstName+' '+middleInitial+' '+lastName; }  
public:  
    // constructor is needed here.  
    void print() {cout << "Employee " << fullName() << endl;}  
};  
class Manager: public Employee {  
    int level;  
public:  
    // constructor is needed here.  
    void print() {cout << "Manager " << fullName() << " at level ="  
        level << endl; }  
};
```

Function overriding

# Use member functions of base class

- ◆ You can also call directly the *print()* member function of the base class in the derived class by using scope operator :

```
void Manager::print() const {  
    Employee::print() ;          // print Employee information  
    // Print extra information for managers  
    cout << "at level:" << level << endl ;  
}
```

manager.cpp

# Override existing code

---

Suppose you received a task to work on a project with existing code. You can keep the existing code untouched by overriding the functions you do not like (you might need to delete useless variables and change the accessibility of some data members or member functions from private to protected).

Overriding.cpp

# Accessibility

Base case: `private`, `protected`, `public`

```
class B {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```

```
int main() {  
    B b;  
    cout<<b.x; //wrong  
    cout<<b.y; //wrong  
    cout<<b.z; //correct  
};
```

# Accessibility

## Accessibility.cpp

Base case: **private**, **protected**, **public**

Derived: **private**, **protected**, **public**

Outside: accessible or not accessible

```
class B {  
private:  
    int x;  
protected:  
    int y;  
public:  
    int z;  
};
```

```
class D: public B {  
private:  
    int r;  
public:  
    void sum() {  
        r = x+y+z; //wrong  
        r = y+z; //correct  
    }  
};
```

```
int main() {  
    B b;  
    cout<<b.x; //wrong  
    cout<<b.y; //wrong  
    cout<<b.z; //correct  
    D d;  
    cout<<d.x; //wrong  
    cout<<d.y; //wrong  
    cout<<d.z; //correct  
    cout<<d.r; //wrong  
    d.sum(); //correct  
};
```

# Inheritance type

How to make data access correct?

```
class B {  
private:  
    int x;  
protected:  
    int y;  
    int getX() {  
        return x;  
    }  
public:  
    int z;  
};
```

```
class D: public B {  
protected:  
    int r;  
public:  
    void sum() {  
        r = getX() + y + z;  
    }  
};  
//all correct
```

```
class E: public D {  
public:  
    void print() {  
        cout << getX() + y + z + r;  
    }  
};  
//all correct
```

```
int main() {  
    E e;  
    e.z = 0;  
    e.sum()  
    e.print();  
};  
//all correct
```

Accessibility2.cpp

# Inheritance type

## How do they affect data access?

Three ways to extend a class:

```
class DerivedClassName: public BaseClassName  
class DerivedClassName: private BaseClassName  
class DerivedClassName: protected BaseClassName
```

```
class B {  
private:  
    int x;  
protected:  
    int y;  
    int getX() {  
        return x;  
    }  
public:  
    int z;  
};
```

```
class D: private B {  
protected:  
    int r;  
public:  
    void sum() {  
        r = getX() + y + z;  
    }  
};
```

```
class E: protected D {  
public:  
    void print() {  
        cout << getX() + y + z + r;  
    }  
};
```

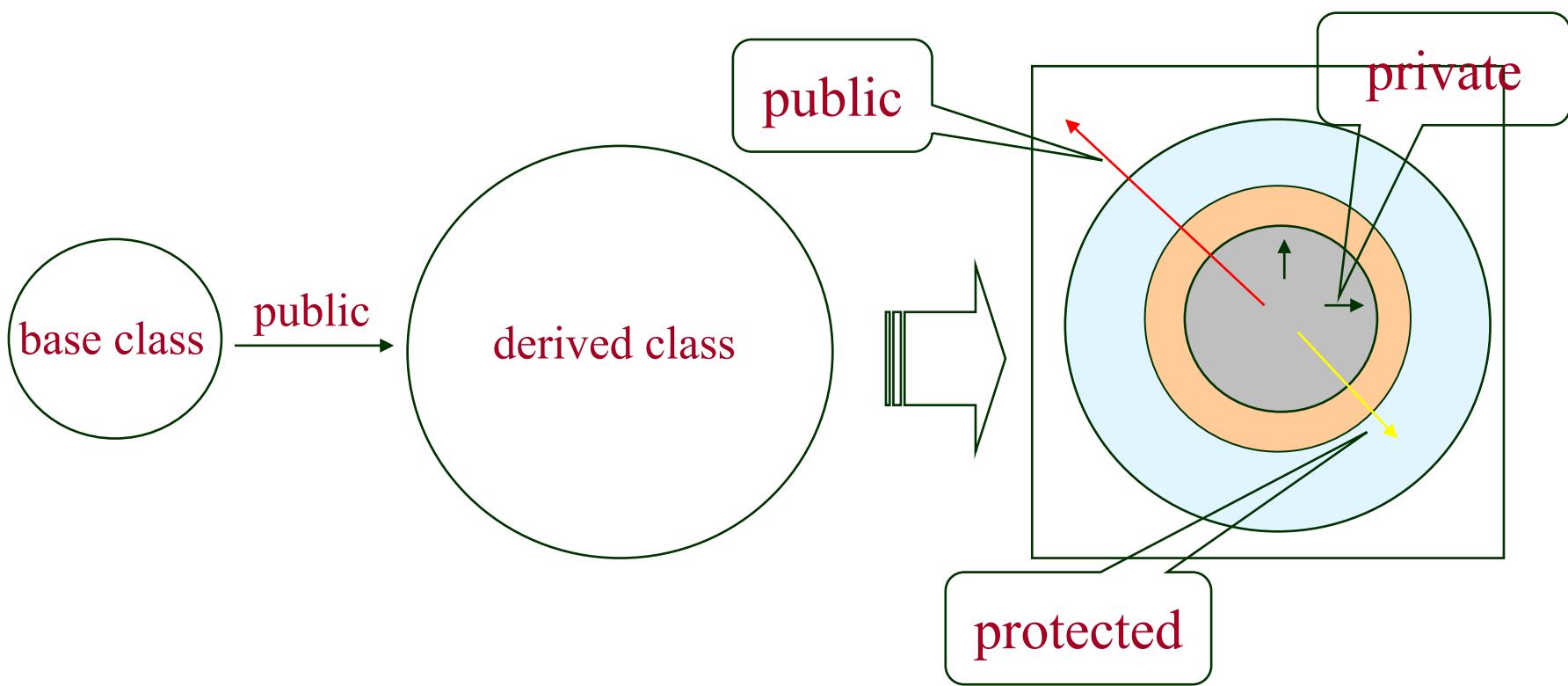
```
int main() {  
    E e;  
    e.z = 0;  
    e.sum();  
    e.print();  
};
```

Incorrect

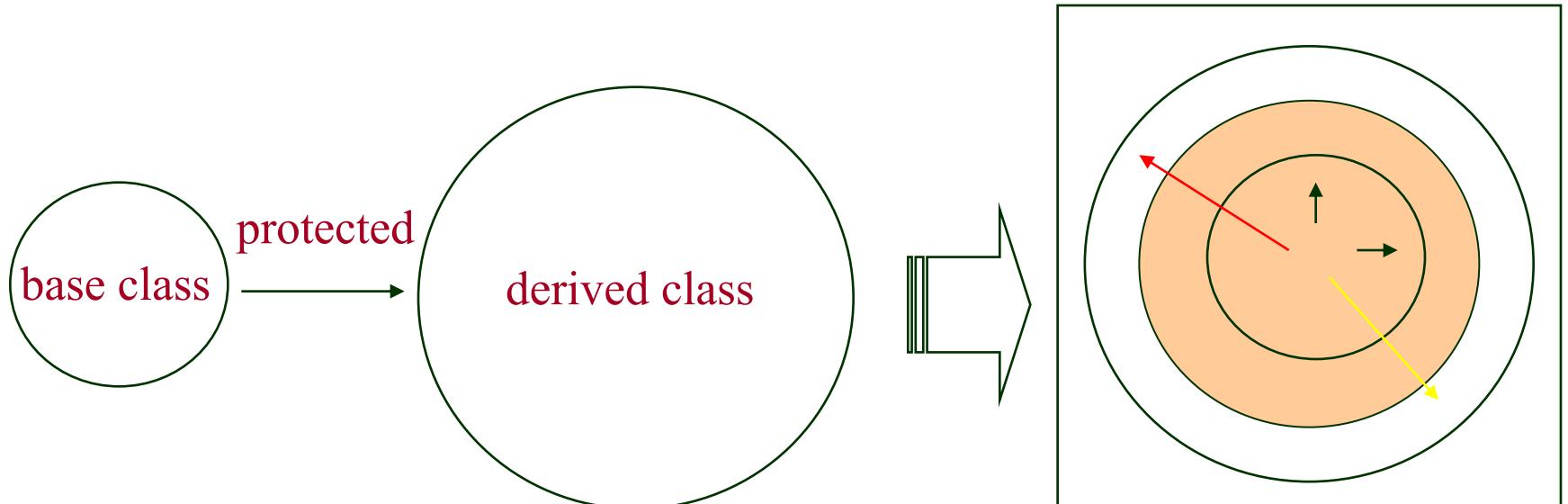
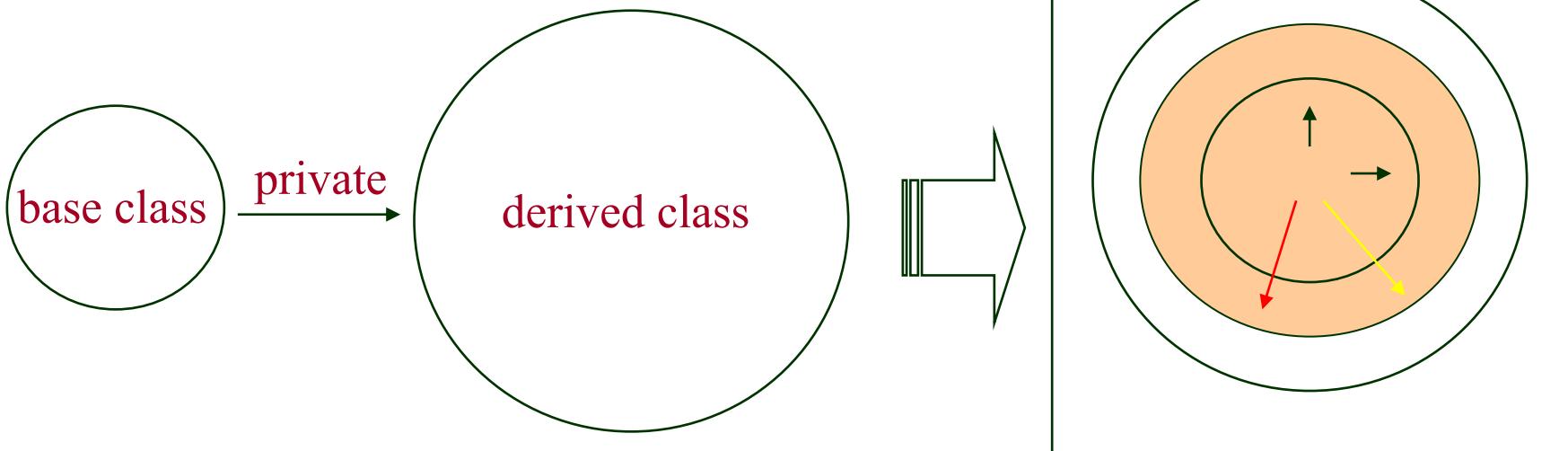
Incorrect

# Class Access

The manner in which a derived class is declared provides a mean of **further** limiting member access to inherited classes. In the example of Cylinder class, the class access was declared as **public**. This provides any derived class of Cylinder class the ability to access the base class (Circle)'s members with the same privileges as the base class.



# Class Access



# Class Access

class Cylinder : public Circle

Type of inheritance Data member in base class	: public	: private	: protected
private	Inaccessible from derived class inaccessible outside	Inaccessible from derived class Inaccessible outside	Inaccessible from derived class inaccessible outside
protected	protected in derived class Inaccessible outside	private in derived class Inaccessible outside	protected in derived class Inaccessible outside
public	public in derived class Accessible outside	private in derived class Inaccessible outside	protected in derived class Inaccessible outside

```
class Circle {  
private: double radius;  
public: double calVal();  
};
```

# The constructor initializer list

- ◆ It is important that **every** member item of an object is initialized.
- ◆ When a member item of a base class is **private**, the derived class can only initialize the variables through **initializer list via constructors**.

```
class Circle {  
    private:  
        double radius;  
    public:  
        Circle(double r) {  
            radius = r;  
        }  
        double calVal();  
};
```

```
class Cylinder : public Circle {  
    private:  
        double length;  
    public:  
        Cylinder(double r, double l):  
            Circle(r), length(l) {}  
        double calVal();  
};
```

# Constructors in inheritance

There must be suitable **constructors** in base class and derived classes.

```
class A {  
    private:  
        int val;  
    public:  
        A(int x) { val=x; }  
};  
class B: public A {  
    public:  
        B(int x): A(x) {}  
        B(int x) {}      // INCORRECT  
        B() : A(5) {}  
        B() {}          // INCORRECT  
};  
  
class C: public A { // INCORRECT no constructor for class C  
};
```

ab.cpp

# Constructor and destructor execution

- ◆ First execute the constructor of base class and then of derived class when an object of derived class is created.
- ◆ First execute the destructor of derived class and then of base class when an object of derived class is destroyed.

```
class A {  
    public: A() { cout << "ctor:A()" << endl ; }  
            ~A() { cout << "dtor:~A()" << endl ; }  
};  
class B : public A {  
    public: B() { cout << "ctor:B()" << endl ; }  
            ~B() { cout << "dtor:~B()" << endl ; }  
};  
int main() {  
    B b ;  
}
```

call.cpp

managerOrder.cpp

# Type conversion and casting

- ◆ **Upcasting:** If a class “Derived” has a **public** base class “Base” then a “Derived” object can be assigned to a variable of type “Base” without explicit casting.

```
Employee staff1("John", "Smith", 'D');  
Manager staff2("Peter", "Wang",' ');  
staff2.setLevel(2);  
staff1 = staff2; //object slicing  
staff1.print();
```

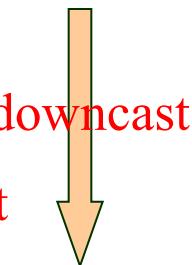
managerCopy.cpp

Note:

- ◆ Only the members of Employee class are copied to staff1.
- ◆ Inheritance type should be **public**.
- ◆ A better way to upcast an object is to use pointers.

Base class

upcast



ObjectSlicing.cpp

Derived class

# Type conversion and casting

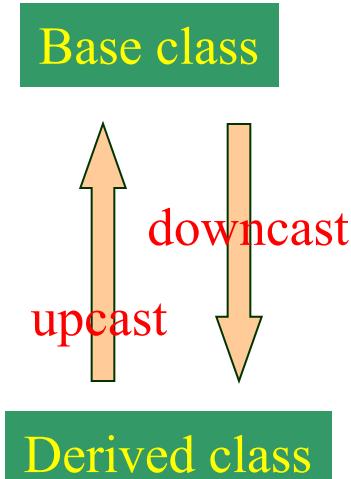
- ◆ **Downcasting:** *A base class object cannot be automatically converted to the derived class.*

```
staff2 = staff1; //invalid
```

- ◆ You may use **static\_cast<>** to force a downcasting in your risk

```
void g ( Manager mm, Employee ee) {  
    Employee* pe = &mm;           // ok:  
    Manager* pm = &ee;           // INCORRECT  
    pm->level = 2;              // INCORRECT  
    pm = static_cast<Manager*>(pe); // works.  
    pm->level = 2;              // fine.  
    // ...  
}
```

managerCopy.cpp



# Automatic type conversion and casting

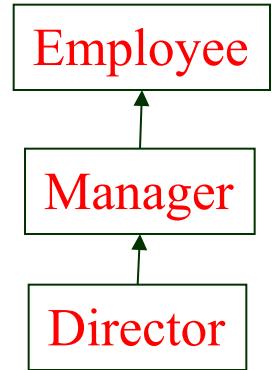
---

- ◆ Safety Issues of type conversions:
  - Upcasting is always possible and safe especially use pointers.
  - Down-casting needs to be performed with great care.
  - To allow safe down-casting, C++ introduced the concept of **dynamic casting**. The technique is available for polymorphic classes (will be introduced in Lecture 10).

# Class Hierarchies

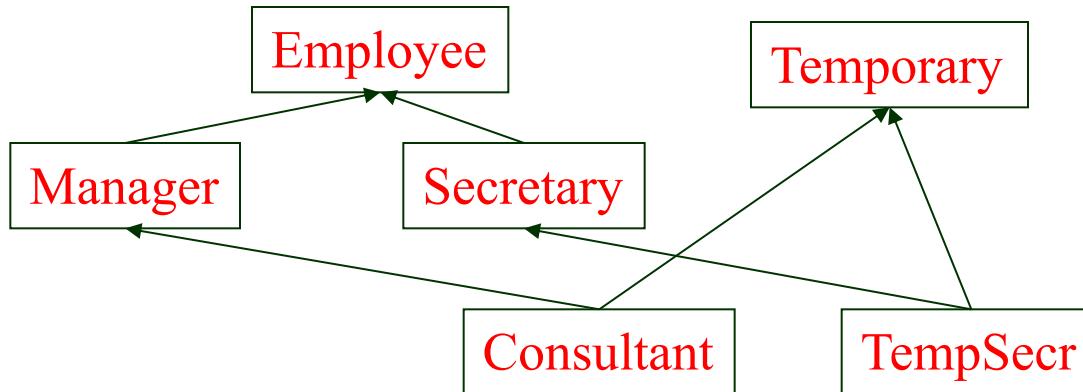
- ◆ A derived class can itself be a base class.

```
class Employee { /*... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```



- ◆ Multiple Inheritance

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
class TempSecr : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```



# Homework

---

- ◆ Read textbook: Chapter 14
- ◆ Assignment 1 demonstration is in this week. Each student will take 10 to 15 minutes. You would experience a significant delay for the demonstration depending on the size of your class.
- ◆ Assignment 2 is available now. Please check it out and let me know if you have any questions.

# Comp2014 Object Oriented Programming

---

## Lecture 9

### Polymorphism & Virtual Functions

# Topics covered in last lecture

---

- ◆ Composition
- ◆ Inheritance
  - Class declaration
  - Inheritance type
  - Inherited access
  - Constructor and destructor
- ◆ Type conversion
- ◆ Class hierarchies

# Inheritance

An object of Player represents only one player!

```
class Player {  
private:  
    char playerSymbol;  
public:  
    Player(char s) { playerSymbol = s; }  
    virtual char getMove(Board b, int& x, int& y) = 0;  
    char getPlayer() {return playerSymbol;}  
};
```

```
class HumanPlayer : public Player {  
public:  
    HumanPlayer(char s): Player(s) {}  
    char getMove(Board b, int& x, int& y);  
};
```

```
class RandomPlayer : public Player {  
public:  
    RandomPlayer(char s): Player(s) {}  
    char getMove(Board b, int& x, int& y);  
};
```

```
class SmartPlayer : public Player {  
public:  
    SmartPlayer(char s): Player(s) {}  
    char getMove(Board b, int& x, int& y);  
};
```

Any specific player *is a* player. All these classes are almost the same except the implementation of *getMove* function

# Polymorphism

```
class Board {  
    char grid[BOARDSIZE][BOARDSIZE];  
public:  
    bool addMove(int x1,int y1,int x2,int y2);  
    bool checkWin();  
    bool validInput(int, int);  
    void printBoard();  
};
```

```
class Game {  
    Board *board;  
    Player *player1, *player2;  
public:  
    Game(Board* b, Player* p1, Player* p2);  
    void play();  
};
```

```
int main() {  
    Board* board = new Board(10);  
    Player* p1 = new HumanPlayer('C');  
    //Player* p1 = new RandomPlayer('C');  
    Player* p2 = new SmartPlayer('B');  
public:  
    Game game(p1, p2);  
    game.play();  
};
```

```
class Game {  
    Board *board;  
    RandomPlayer *player1; //hard coded  
    SmartPlayer *player2; //hard coded  
public:  
    Game(Board* b, Player* p1, Player* p2);  
    void play();  
};
```

Bad solution

Right solution

How are different players' moves passed to Game class?

# Topics covered in the lecture

- ◆ Function calls and binding
- ◆ Static binding
- ◆ Function overriding and dynamic binding
- ◆ Polymorphism
- ◆ Virtual functions
- ◆ Abstract classes

Difficulty factor:  
\*\*\*\*

composition

abstraction

encapsulation

data hiding

inheritance

polymorphism

Object oriented analysis, design and programming

# Function calls and binding

```
class Binding {  
public:  
    void print(int value) { cout << value << endl; }  
};
```

binding.cpp

```
Void print(int value) {  
    cout << value << endl;  
}
```

```
int main()  
{  
    Binding b;  
    b.print(10);  
    print(10);  
    return 0;  
}
```

binding

binding

Find the code that  
implement the  
function.

# Static binding

- ◆ Connecting a function call to a function body (the code) is called **binding**.
- ◆ **Static binding**: binding is performed at compiling before the program is run.

```
void testOverloading( int numerator, int denominator) {  
    int fraction = numerator / denominator;  
    cout << "Fraction1 = " << fraction << endl;  
}
```

```
void testOverloading(double numerator, double denominator)
```

```
{
```

```
    double fraction = numerator / denominator;  
    cout << "Fraction2 = " << fraction << endl;
```

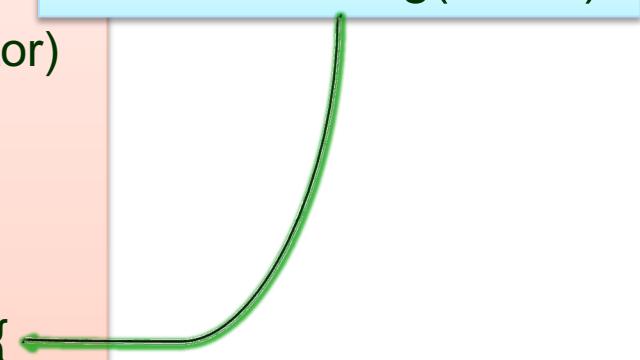
```
}
```

```
void testOverloading( int numerator, double denominator) {
```

```
    double fraction = numerator / denominator;  
    cout << "Fraction3 = " << fraction << endl;
```

```
}
```

testOverloading(3, 7.0);



# Function overriding and binding

- ◆ Function **overriding** (method overriding) allows a derived class to provide a new implementation of a method that is already implemented in its base class.
- ◆ With static binding, a function call from an object of a class is bound to the implementation of the function in that class.
- ◆ Do not mix up function overloading and function overriding.

staticBinding.cpp

## Function overloading vs Function overriding

The diagram consists of two overlapping orange ovals. The left oval represents function overloading, and the right oval represents function overriding. The intersection of the two ovals is shaded green, while the non-overlapping parts are white.

Same function name but  
different (or different  
number of) parameters  
`f1(double) & f1(int)`

Same function name and parameters  
but defined in different classes in a  
hierarchy  
`classOne::f1(int) & classTwo:: f1(int)`

# Static binding for overriding functions

- ◆ Static binding: *Call own implementation if any; otherwise call base implementation.*

```
class One {  
public:  
    double f1(double);  
    double f2(double);  
};  
  
double One::f1(double num)  
{  
    return num+1;  
}  
double One::f2(double num)  
{  
    return f1(num) * f1(num);  
}
```

```
class Two: public One {  
public:  
    double f1(double);  
};  
double Two::f1(double num)  
{  
    return num+2;  
}
```

Class One:

$$f1(n) = n+1;$$

$$f2(n) = f1(n)^2;$$

Class Two:

$$f1(n) = n+2;$$

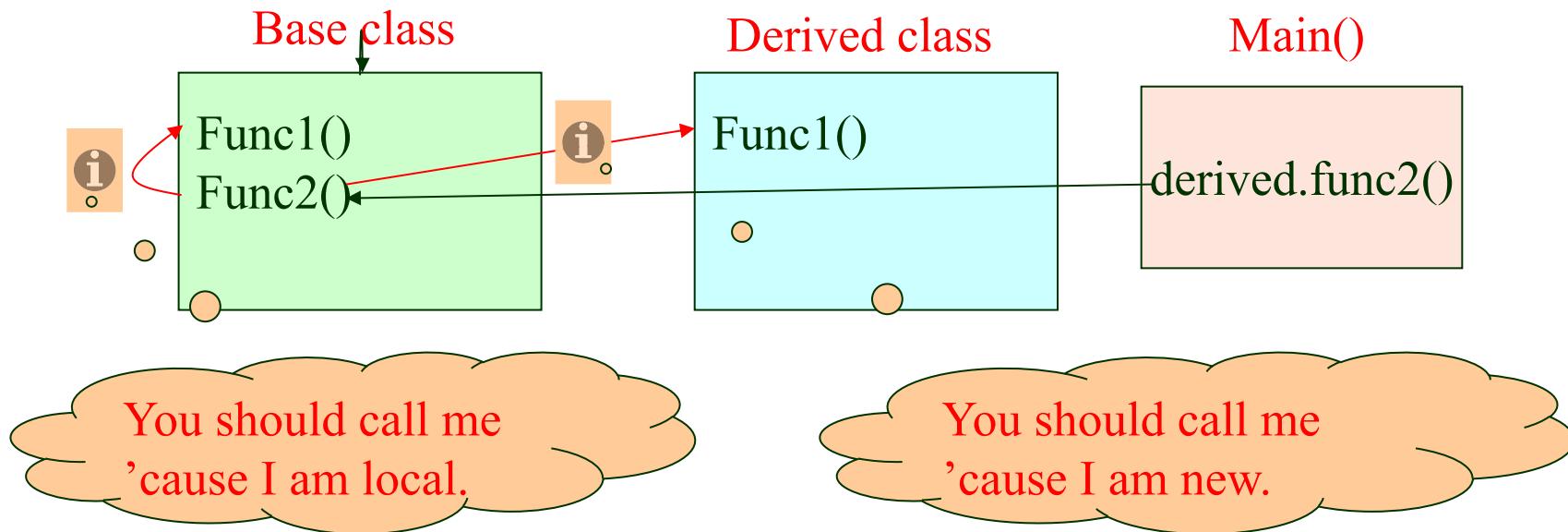
$$f2(n) = f1(n)^2;$$

Question: *If an object in class Two call f2, which f1 will be called?*

See example access1.cpp

# Polymorphism

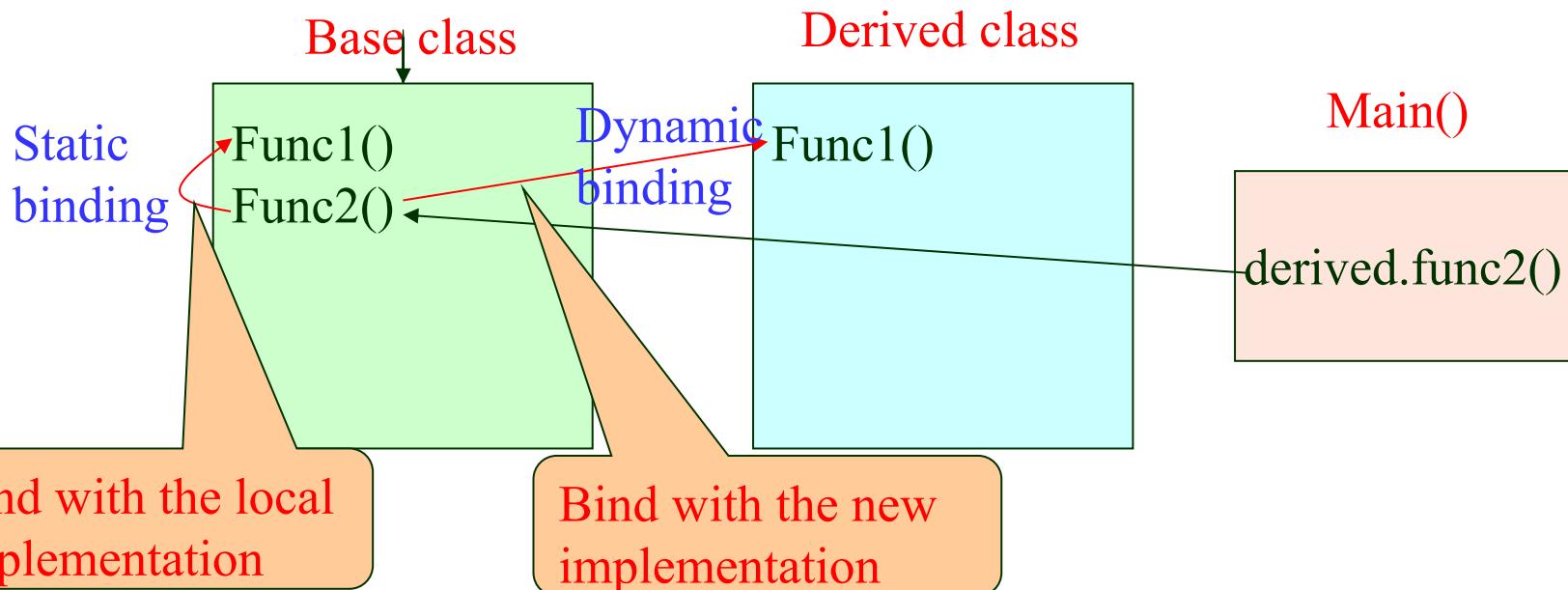
- Unlike function overloading where different parameters determine which version of the function to execute, for function overriding, the name and its signature of a function are exactly the same in both base class and derived class, which one should be executed when it is invoked?



# Polymorphism

Why should we allow polymorphism?

- ◆ *Polymorphism* permits the same function name to invoke one response in objects of a base class and another response in objects of a derived class.
- ◆ The way that C++ determines which function to call is through two types of binding, *static* and *dynamic*.



# Polymorphism

- ◆ If we want a binding method that is capable of determining which function should be invoked at **run time**. This type of binding is referred to as **dynamic binding**. To allow this happening, we use the following keyword in the **base class**:

```
virtual double f1(double);
```

- ◆ A polymorphic function that is dynamically bound is called a **virtual function**.

See example access2.cpp

# Polymorphism

- The use of virtual functions allows dynamic binding. This means that depending upon the object which calls an overridden function, the appropriate function will be used. A virtual function effectively creates a pointer to that function, but does not assign an actual value to the pointer until **run-time**.

```
class Game {  
    Board* board;  
    Player* player[2];  
public:  
    Game(Board* b, Player* p[]);  
};
```

```
class Player {  
protected:  
    char playerSymbol;  
public:  
    virtual void getMove(Board, int&, int&)=0;  
};
```

Why can't determine it  
during compiling?

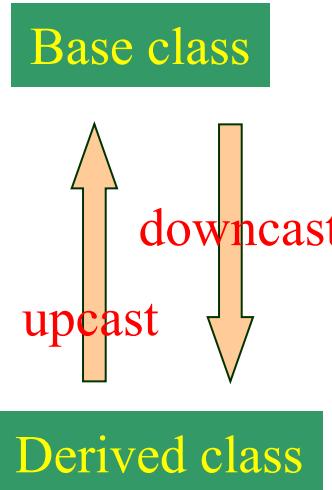
```
class HumanPlayer: public Player {  
public:  
    HumanPlayer(char ps)  
        {playerSymbol = ps;}  
    void getMove(Board, int&, int&);  
};
```

```
class SmartPlayer: public Player {  
public:  
    SmartPlayer(char ps)  
        {playerSymbol = ps;}  
    void getMove(Board, int&, int&);  
};
```

# Dynamic binding and up-casting

- ◆ Note that results of dynamic binding depend on the ways of casting.
  - Casting by pointer or reference: **works better**

```
Game(Board* b, Player* p[])
{
    board = b;
    player[0] = p[0]; //deep copy
    player[1] = p[1];
}
//in main function
Board* b = new Board(10);
Player* p[2];
P[0] = new HumanPlayer('C'); // upcasting
P[1] = new RandomPlayer('B'); //upcasting
Game game(b, p);
```



```
class Player {
protected:
    char playerSymbol;
public:
    virtual void getMove(Board, int&, int&);
};
```

# Dynamic binding and upcasting

- ◆ However, when casting the whole object or pass parameter by value, the object would be **sliced** to the object of its base class

Check the differences from the following two functions:

```
void describe(Pet p) { // Slice the object
    cout << p.description() << endl;
}
```

```
void describe(Pet *ptr) { // no object slicing
    cout << ptr->description() << endl;
}
```

See example ObjectSlicing.cpp

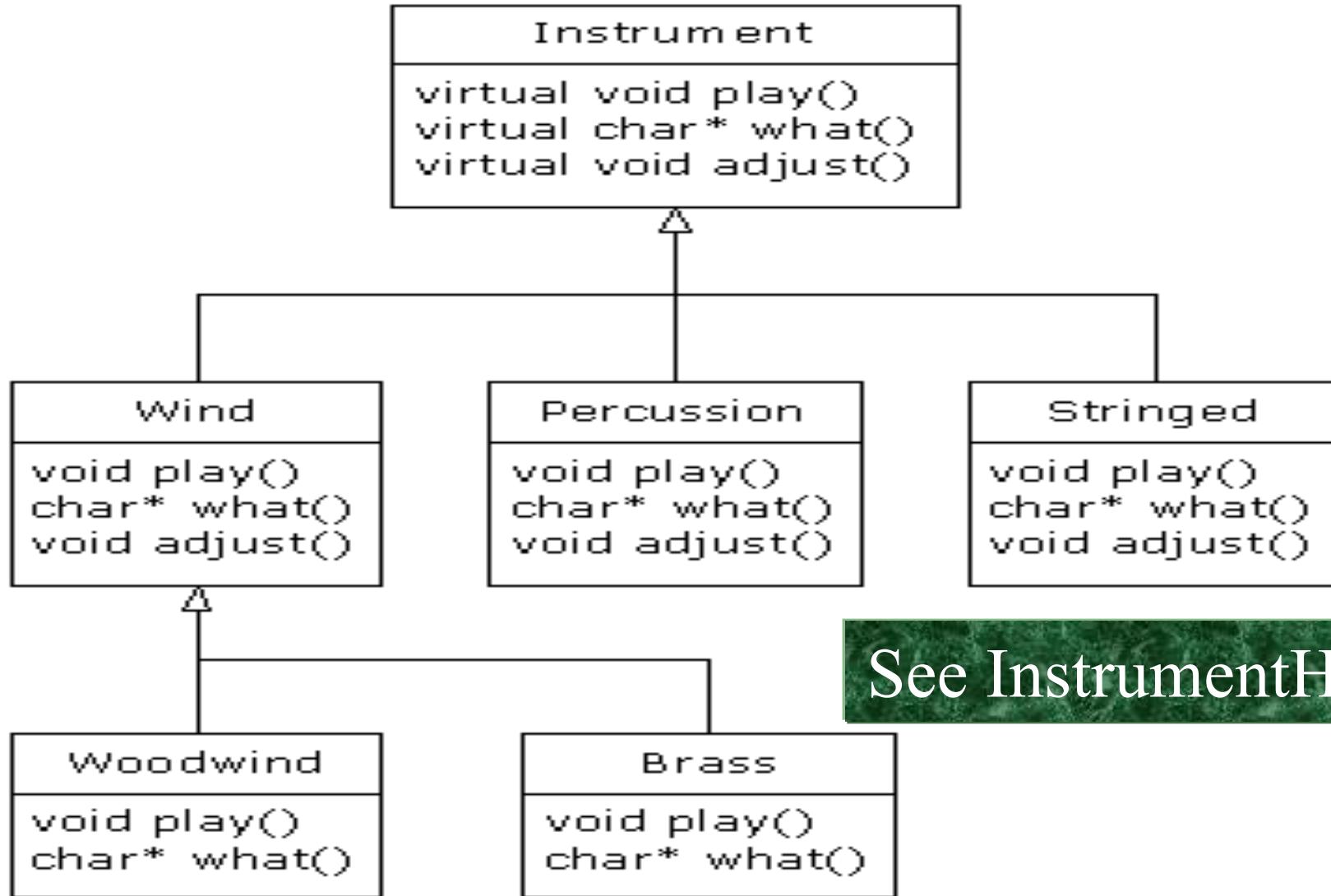
# Pure Virtual Function and Abstract Classes

- ◆ Pure Virtual Functions: A virtual function is made pure by the initializer of `=0`.
- ◆ No definition is needed for pure virtual functions.
- ◆ Abstract Class: A class with at least one virtual function is called an abstract class. No object of an abstract class can be created but a pointer of an abstract class can be defined.

```
class Player {  
protected:  
    Char playerSymbol;  
public:  
    virtual void getMove(Board*, int&, int&) = 0;  
};
```

```
Player p; //illegal  
Player* p; //legal
```

# Hierarchy of Instruments



See InstrumentH.cpp

# Mixture of overloading & overriding

---

- ◆ Overriding a virtual function can't change the return type. See ChangeReturn.cpp
- ◆ If overriding one of the overloaded member functions in the base class, the other overloaded versions become hidden in the derived class.

See NameHidding.cpp

# Virtual destructor

See destructors.cpp

- ◆ You may have experienced warning message “no virtual destructor” even though you don’t think you need a destructor.

```
class Base {  
    virtual void function();  
    // but no destructor here
```

```
};  
  
class Derived : public Base {  
    void function() { int* p = new int[1000]; }  
    ~Derived() {  
        delete [] p;  
    }
```

```
//in another part of the program  
Base *b = new Derived();  
delete b; // Here's the problem!
```

Borrowed memories, which are used by the overridden methods

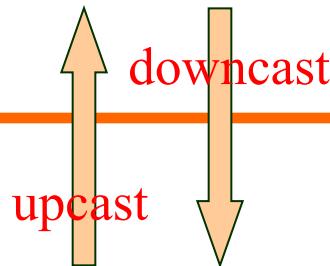
The base class’s destructor can’t do the clean-up for the derived class because b is a pointer of the base class. You might incur some memory leaking.

# Downcasting

- ◆ Two ways to downcast an object:
  - `static_cast`: **unsafe**
  - `dynamic_cast`: **safe but no guaranty of success.**

*When you use `dynamic_cast` to try to cast down to a particular type, the return value will be a pointer to the desired type only if the cast is proper and successful; otherwise, it will return zero to indicate that this was not the correct type.*

Derived class



See DynamicCast.cpp

```
Pet* d = new Dog;  
Dog* d1 = dynamic_cast<Dog*>(d);  
Cat* c = dynamic_cast<Cat*>(d);
```

# Homework

---

- ◆ Read Textbook chapter 15.
- ◆ Complete online tutorial 2.
- ◆ Attend this week's PASS session, which provides necessary training for assignment 2.

# Comp2014 Object Oriented Programming

---

## Lecture 10

# Operator Overloading, References and Friends

# Topics covered in last lecture

- ◆ Function calls and binding
- ◆ Static binding
- ◆ Function overriding and dynamic binding
- ◆ Polymorphism
- ◆ Virtual functions
- ◆ Abstract classes

Difficulty factor:  
\*\*\*\*

//function prototype in a class

Type **className::functionName(Type1 parameter1, Type2 parameter2)**

//function call

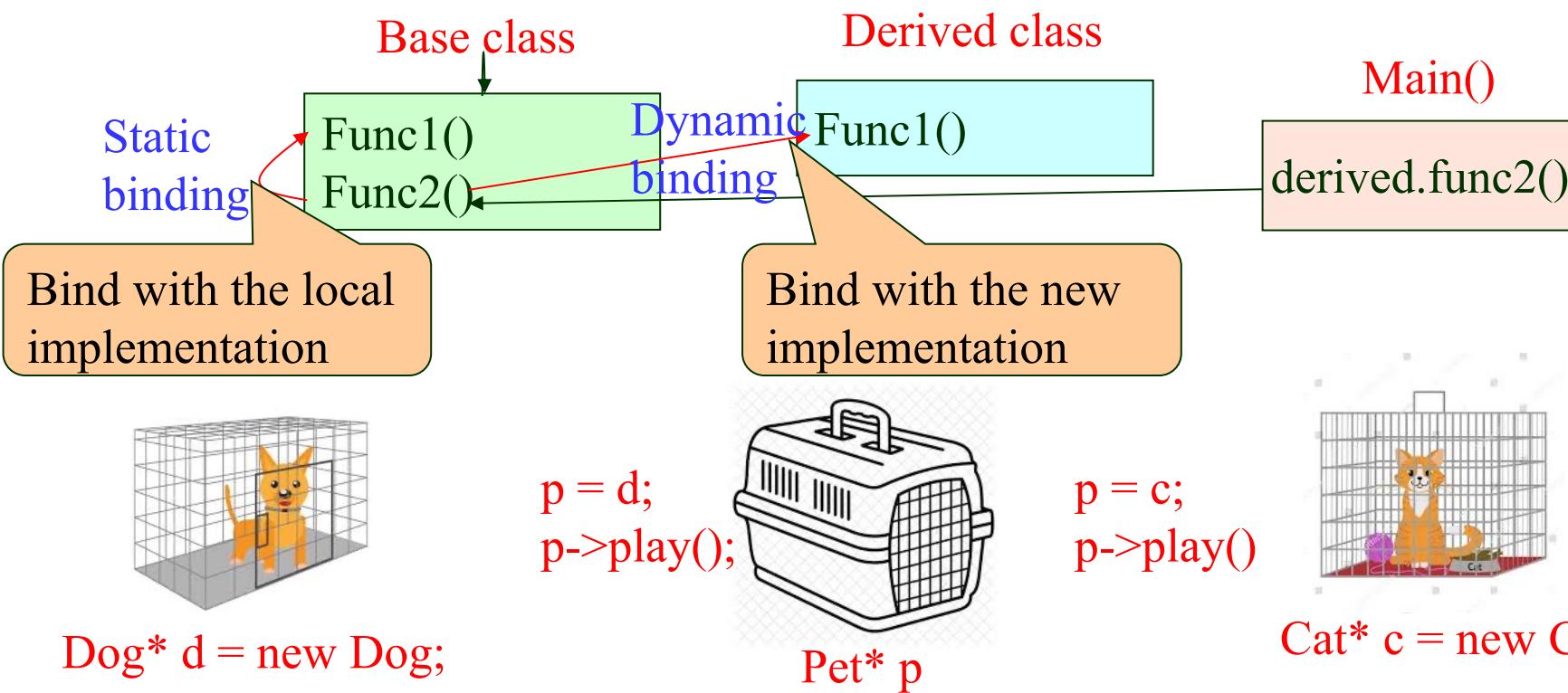
**className object;**

binding

Type o = object.functionName(object1, object2);

# Polymorphism

- ◆ The way that C++ determines which function to call is through two types of binding, *static* and *dynamic*.
- ◆ *Polymorphism* permits the same function name to invoke one response in objects of a base class and another response in objects of a derived class.



# Topics covered by the lecture

---

- ◆ Function and function overloading (review)
- ◆ Operator overloading
- ◆ References
- ◆ Friends
- ◆ Review of assignment 1 and preview of assignment 2



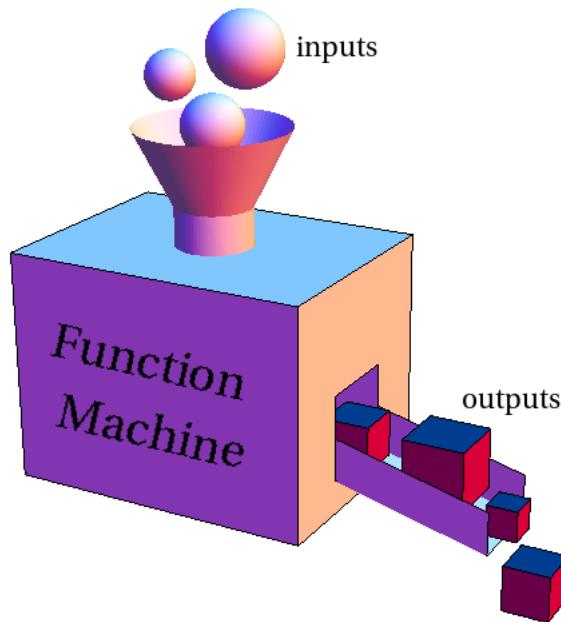
**Difficulty factor:**

\*\*\*\*

# Functions and operators

- ◆ A function is a process that takes inputs and generates outputs.
- ◆ A function can be overloaded, e.g.,

```
int addition(int x, int y)  
double addition(double x, double y)
```



```
int addition(int x, int y) {  
    int z = x + y;  
    return z;  
}  
  
int main() {  
    cout<< addition(20,30) <<endl;  
    return 0;  
}
```

# Built-in operators

---

- ◆ Beside the functions we created, many mathematical operators are also functions and defined for built-in data types, such as
  - `+, -, *, /, %`: arithmetic operations (for *int, float, double, long*)
  - `<=, <, >=, >, ==, !=`: relational operations (for *expressions*)
  - `&&, ||`: logical operations (for *Boolean variables*)

These *built-in* functions are called *operators*. For instance,

```
int value, x, y, z;  
  
z = x + y;  
value = (2*x + y) / z;  
  
x <= y;  
  
bool u, v, w;  
  
u && v;  
  
w == u || v;
```

# Built-in operators

---

- ◆ Other built-in operators you may be less aware of:
  - Incremental operators: `++`, `--`
  - Accumulation operators: `+=`, `-=`, `*=`, `/=`, `^=`
  - Stream operators (free functions): `<<`, `>>`
  - Array operators: `[]`

```
int value, x, y, z;  
ifstream fin;  
ofstream fout;  
  
value += x; //means value = value + x;  
y = ++x; //means x=x+1; and y=x;  
fin >> x;  fout << y;  
array[x] // the xth value of array
```

# Functions and operators

- ◆ An operator can be either *prefix*, *infix* or *postfix*:

*Prefix*:  $--x$ ,  $++x$ ,  $\&value$

*Postfix*:  $x++$ ,  $y--$

*Infix*:  $x + y$ ,  $x * y$ ,  $x < y$ ,  $x == y$ ,  $u \&& v$

Unless prefix, infix and postfix operators are most for human users.  
The implementation of all operators can be represented as function in prefix format:

$--x$	$x.operator--()$	$x--$	$operator--(x)$
$x + y$	$x.operator+(y)$		
$x * y$	$x.operator*(y)$		
$x < y$	$x.operator<(y)$		
$u \&& v$	$u.operator\&\&(v)$		

# Built-in operators (i.e. functions)

- ◆ Are these built-in operators applicable to the user-defined data-types, including the classes you created? **NO**

```
class Date {  
public:  
    int day;  
    int month;  
    int year;  
};
```

```
int main() {  
    Date d(30,9,2021), d1;  
    d1 = d + 100;  
    while (d < d1)  
        d++;
```

Next day

- ◆ If we want the operators of +, < or ++ to be used for Date data type, we need them to be **overloaded** to that data type, i.e,

**Date operator+( int ), Date operator<( Date ),  
Date operator++()**

# Operator Overloading

---

Built-in operators can be overloaded so that when they are used with class objects, the operators have meaning appropriate to the new types. In fact, to use an operator on non built-in data type, the operator *must* be overloaded in the class, with two exceptions:

- 1). *The assignment operator, =, will allow copying of contents of corresponding data members called member-wise assignment. This operation, though, is dangerous if any of the data members are pointers (shallow copy)*
- 2). *The other exception is the address-of operator, &. It returns the address of the object in memory for any data type.*

# Operators that can be overloaded

- ◆ Almost all operator can be overloaded but common ones are:

- Arithmetic operators: + - \* / %
- Incremental operators: ++ -- Commonly for search and comparison
- Comparison operators: != == > < >= <=
- Boolean operators: && ||
- Accumulation operators: += -= \*= /= ^=
- Stream operators (free functions): << >>
- Array operators: [] Commonly for easy i/o operations
- Assignment and address of: = Commonly for deep copy

Unary operators: **--x, ++x, ...**

Binary operators: **x + y, x || y, cout >> x, ...**

*Overload an operator only if it is necessary or actually helpful!*

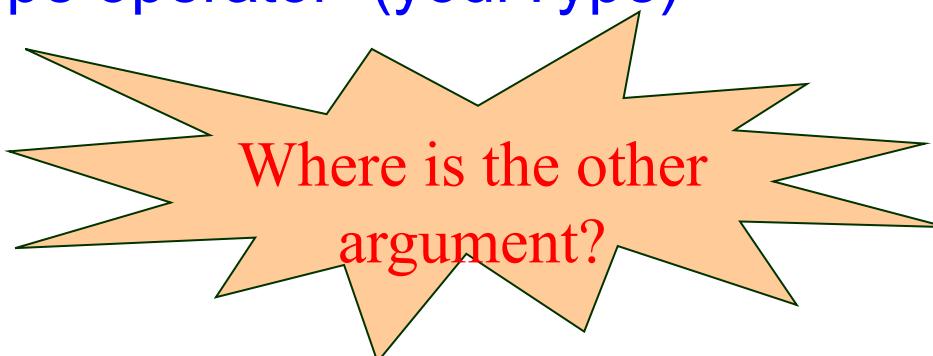
# Operator Overloading - Syntax

- ◆ To overload an operator “**i**” in class **className**,
  - If “**i**” is a unary operator, define a public function:  
**className operator*i*();**
  - If “**i**” is a binary operators, define a public unary function:  
**className operator*i*(className x);**
  - Note that this rule may vary for special cases!

- ◆ Examples:

**x++**      Overloading: **yourType operator++()**

**x = y ;**    Overloading: **yourType operator=(yourType)**



# Example of operator overloading

## ◆ Overload < for class Date

```
class Date {  
private:  
    int day;  
    int month;  
    int year;  
public:  
    Date(int,string,int);  
    void display();  
    bool operator<(Date d);  
};
```

```
bool operator<(Date d) {  
    if(year < d.year )  
        return true;  
  
    if(year == d.year && month < d.month)  
        return true;  
  
    if(year == d.year && month == d.month  
&& day < d.day)  
        return true;  
  
    return false;  
}
```

# Overloading assignment operator =

Overload operator= for class DynamicArray

- Shallow copy:

```
class DArray {  
private:  
    int* arr;  
    int size;  
public:  
    DArray(int s):size(s) {  
        arr = new int[s];  
    }  
  
    ~DArray() {  
        delete[] arr;  
    }  
};
```

```
void operator=(DArray c) {  
    this->size = c.size;  
    arr = c.arr;  
}
```

- Deep copy:

```
void operator=(DArray c) {  
    this->size = c.size;  
    arr = new int[size];  
    for(int i=0;i<size;i++)  
        arr[i] = c.arr[i];  
}
```

# Topics covered by the lecture

---

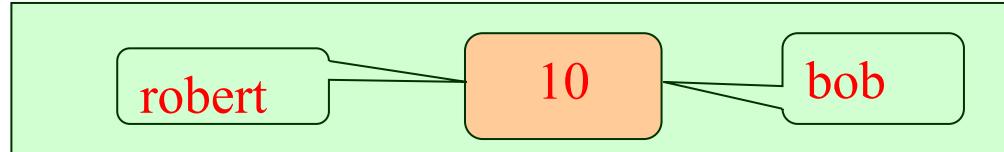
- ◆ Function and function overloading (review)
- ◆ Operator overloading
- ◆ References
- ◆ Friends
- ◆ Review of assignment 1 and preview of assignment 2

# References

Why does a variable  
need another name?

- ◆ **Reference:** a reference is an *alias of an object variable*, that is, another name for an already existing variable.

```
int robert;  
int& bob = robert;
```



```
robert = 10; //bob is 10 now
```

```
bob = 20; // robert equals 20 as well
```

- *bob* is a reference to the storage location of *robert*
- Changes made to *bob* will apply to *robert*, so does *bob*.

- ◆ **Benefit:** Two variables share the same memory cell.
- ◆ **Call-by-reference** is an example of using references

# Reference



Mind deference  
with Java

Things you should always remember about references.

- ◆ A reference must always refer to something. NULL is not allowed.
- ◆ A reference must be initialized when it is created. An unassigned reference can not exist.
- ◆ Once initialized, it cannot be changed to another variable.

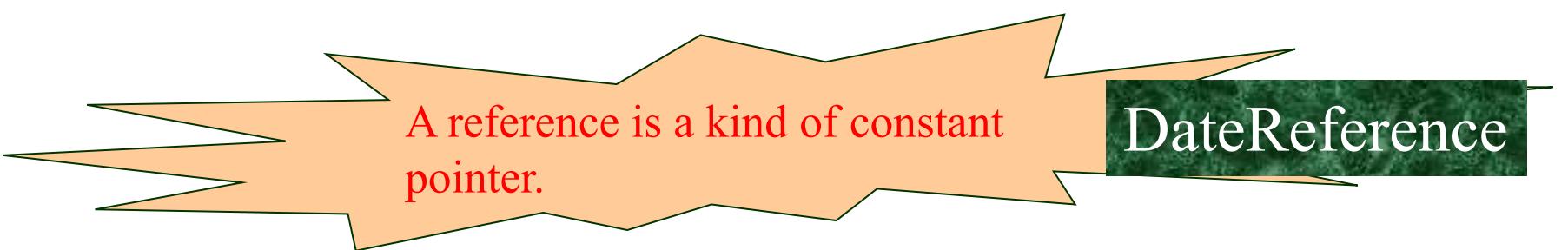
```
int main() {  
    int a=9;  
    int& aref = a;  
    a++;  
    cout << "The value of a is " << aref;  
    return 0;  
}
```



Date.Java

# Reference vs pointer

- ◆ References are often confused with pointers but there are three major differences between them
  - You cannot have a NULL reference, but you can assign NULL to a pointer.
  - Once a reference is initialized to a variable, it cannot be referred to another variable. Pointers can point to any objects at any time if type matches.
  - A reference must be initialized when it is created. Pointers can be initialized at any time.



A reference is a kind of constant pointer.

DateReference

# Friends

---

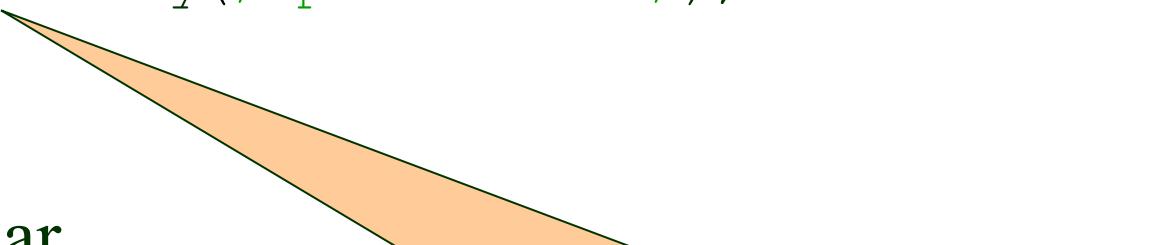
- ◆ Encapsulation is an OOP concept that binds together the data and functions, and that keeps both safe from outside interference and misuse.
- ◆ Normally a class restricts outside accesses to its data and hence these data are not accessible to data members or member functions of other classes and free functions.
- ◆ There may be some circumstances where the programmer wishes to allow such accesses. << and >> operators are typical examples.
- ◆ This can be done via the use of **friends**.
- ◆ Friends can be either functions or classes

# Friend Functions of Classes

- ◆ Friend function (of a class): a non-member function of the class that has access to all the members of the class, including private members
- ◆ Declare a **friend** function in a class

```
class classIllusFriend {  
    friend void homestay /*parameters*/;  
    ...  
};
```

- ◆ Friend class is similar



Function homestay() is not a member function of classIllusFriend but can access all the data members and member functions of this class

Friend.cpp

date.h

# Topics covered by the lecture

---

- ◆ Function and function overloading (review)
- ◆ Operator overloading
- ◆ References
- ◆ Friends
- ◆ Review of assignment 1 and preview of assignment 2

# Homework

---

- ◆ Read textbook Chapter 8.
- ◆ Online tutorial 3 will be open next week.
- ◆ Practical 7 will be checked from this week.
- ◆ Assignment 2 will be due next Friday 5 pm 14 Oct 2022.

# Comp2014 Object Oriented Programming

---

## Lecture 11

### Class Templates and Linked Data Structures

# Topics covered by the lecture

---

- ◆ Function and function overloading (review)
- ◆ Operator overloading
- ◆ References (mind the difference between C++ and Java)
- ◆ Friends (function or class)

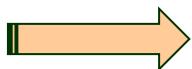
**Difficulty factor: \*\*\*\***

# Topic covered in the lecture

---

- ◆ Template
  - Function template
  - Class template
- ◆ General concept of data structures
- ◆ Linked list

Programming learner



Professional programmer

functionalities

efficiency

# Function Templates

- Suppose that we have a set of arrays, each in different data type. We need to write a function to print the members of each array.

```
void printArray(int *array, int count) {  
    for( int i = 0; i < count; i++)  
        cout << array[ i ] << " ";  
    cout << endl;  
}
```

```
void printArray(double *array, int count) {  
    for( int i = 0; i < count; i++)  
        cout << array[ i ] << " ";  
    cout << endl;  
}
```

```
void printArray(char *array, int count) {  
    for( int i = 0; i < count; i++)  
        cout << array[ i ] << " ";  
    cout << endl;  
}
```

Say no to  
copy-and-paste

Can we merge them into a single function?

Yes, we can!

# Function Templates

Templates are used in cases where we want to use a range of similar functions or develop a set of similar classes. In these situations we write a base version which is then adapted to different **data types**.

```
template <class Datatype>
void printArray( Datatype *array, int count) {
    for( int i = 0; i < count; i++)
        cout << array[ i ] << " ";
    cout << endl;
}
```

For the above piece of code, when the compiler finds a reference to the **printArray** function in the code it substitutes the type of the first parameter throughout the function.

See arraypft.cpp

# Class Templates

They are called *containers*.

Many data structures, such as *array*, *linked lists*, *stacks*, *queues* etc., can be thought of independently from the type of objects within them. Operationally they are the same irrespective of whether they contain integers, doubles, characters or any user defined data type. Thus if we define a class for a data structure it would be useful if it could contain data in any data type. To do this we must use a **CLASS TEMPLATE**.



# Class Templates: declaration

```
template<class T>
class dArrayT {
private:
    T* arr;
    int size;      // The number of filled numbers
    int capacity; // The capacity of the array
    void resize();
public:
    dArrayT(int c);
    ~dArrayT();
    bool insert(int pos, T val);
    bool remove(int pos);
    int length() const {return size;}
    T& operator[] (int i); //operator overloading (as a l-value)
    T operator[] (int i) const; //operator overloading
};
```

Demonstrate an implementation of vector

There is a data type variable **T** here.

# Class Templates: definition

```
template<class T>
bool dArrayT<T>::insert(int pos, T val) {
    if (pos < 0 || pos > size)
        return false;
    if(size + 1 > capacity)
        resize();
    if(pos == size) {
        arr[size++] = val;
    } else {
        // move right
        for(int i=size-1;i>=pos;i--)
            arr[i+1] = arr[i];
        arr[pos] = val;
        size++;
    }
    return true;
}
```

Each method of a template class is a template function

Remember to use the *template* keyword before **each member function** outside the class declaration and the sharp angle bracket **<T>** in the class scope.

More methods ...

# Class Templates: applications

```
int main() {  
    dArrayT<int> a1;      // array of integers  
    dArrayT<double> a2; // array of doubles  
    dArrayT<char> a3;    // array of characters  
    dArrayT<FlightTicket> a4;  
    dArrayT<HotelVoucher> a5;  
    dArrayT<EventTicket> a6;  
    dArrayT<Package> a7;  
  
    //...  
  
    return 0;  
}
```

See dArrayTUsage.cpp

# Vector: a dynamic array template

**vector** is a dynamic array in the **Standard Template Library (STL)**.  
Many methods are implemented.

Header file:

```
#include <vector>  
  
vector<Order> OrderBundle;
```

Simple operations:

- `OrderBundle.push_back(order)`: *appending a new element value at the end.*
- `OrderBundle.pop_back()`: *removing an element from the end.*
- `OrderBundle[index]` : *random access with index*

See vectorApp.cpp

# Array vs vector

- ◆ Array is relatively more efficient than vector. If you know the size of the data, use array.
- ◆ Vector is more flexible, especially if you do not know the size of data.
  - Be careful of the difference:

```
extPersonType list[500]; //500 objects
vector<extPersonType> list; //0 objects
```
- ◆ A set of built-in functions in the Standard Template Library can be used with vector, such as sort, max, min,

...

See Practical Task 7.5

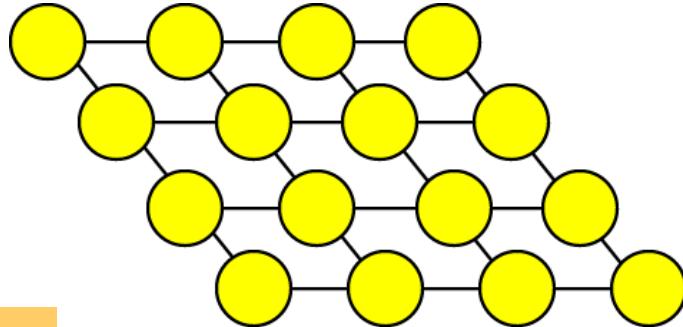
# Data structure: general concept

---

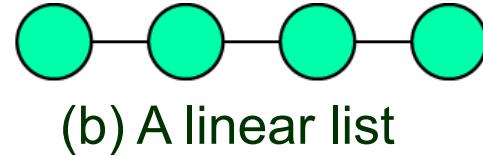
- ◆ A dynamic array can be more useful and convenient than the normal array but
  - Changing the size of the array requires creating a new array and then copying all data from the array with the old size to the other array with the new size
  - The data in the array are next to each other sequentially in memory, which means that inserting an item inside the array requires shifting some other data in the array.
- ◆ We do not have to store data in an array!

# Data structure: general concept

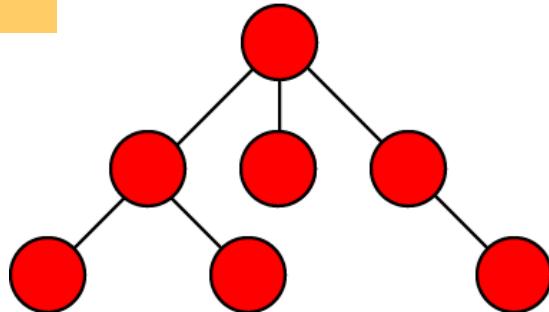
Typical data structures



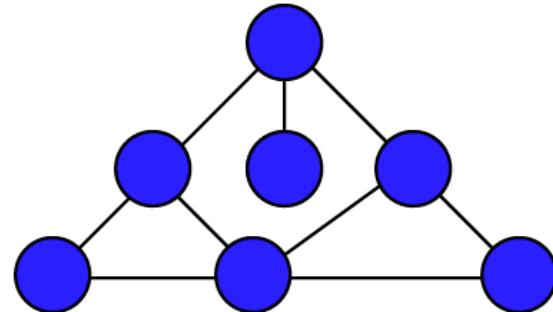
(a) A matrix



(b) A linear list



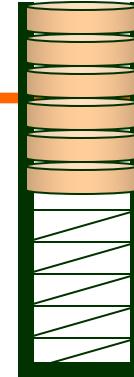
(c) A tree



(d) A network

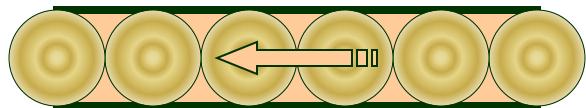
# Typical linked data structures

- ◆ Linked list:



- ◆ Stack: first in last out (FILO)

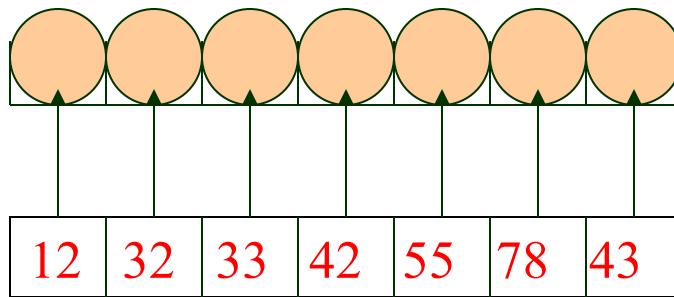
- ◆ Queue: first in first out (FIFO)



- ◆ Hash table:

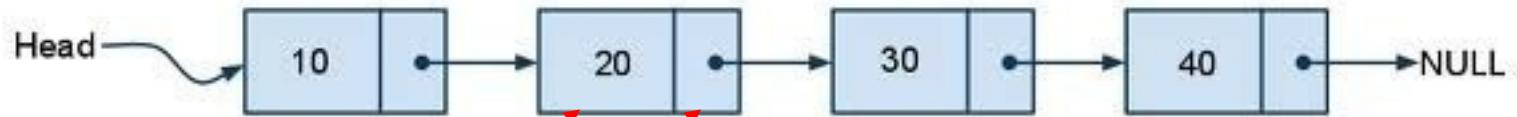


- ◆ Map:



What are inside? **objects!!!**

# Linked List Concept

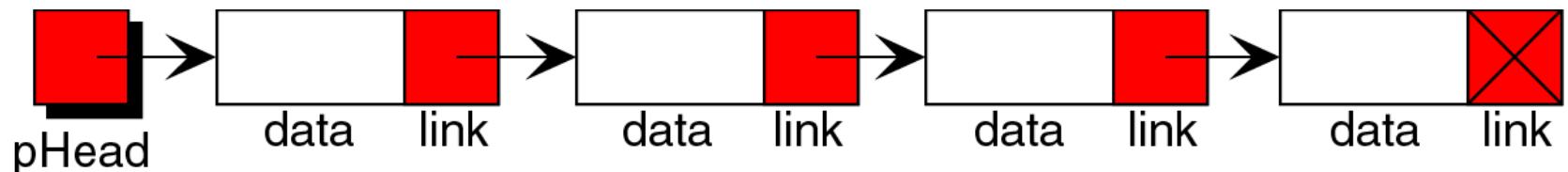


```
Template<class Type>
class Node {
public:
    Type data;
    Node *link;
};
```

Linked list: an ordered collection of data in which each element (node) contains the location of the next element

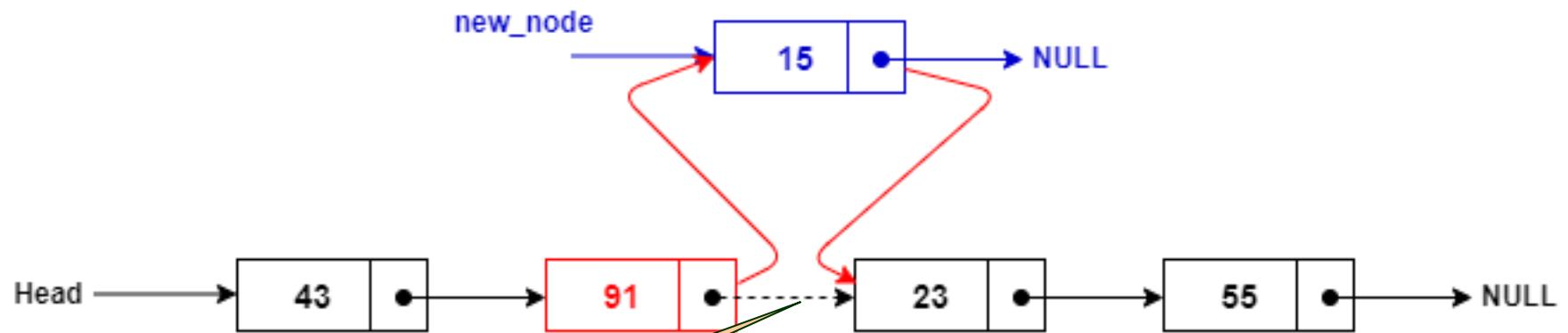
See example SimpleLinkedList.cpp

# Typical operations



- ◆ Insert a node
- ◆ Delete a node
- ◆ Modify a node
- ◆ Search for a data item in a node
- ◆ Traversal over a linked list

# Insert a node into a linked list



Insert this node

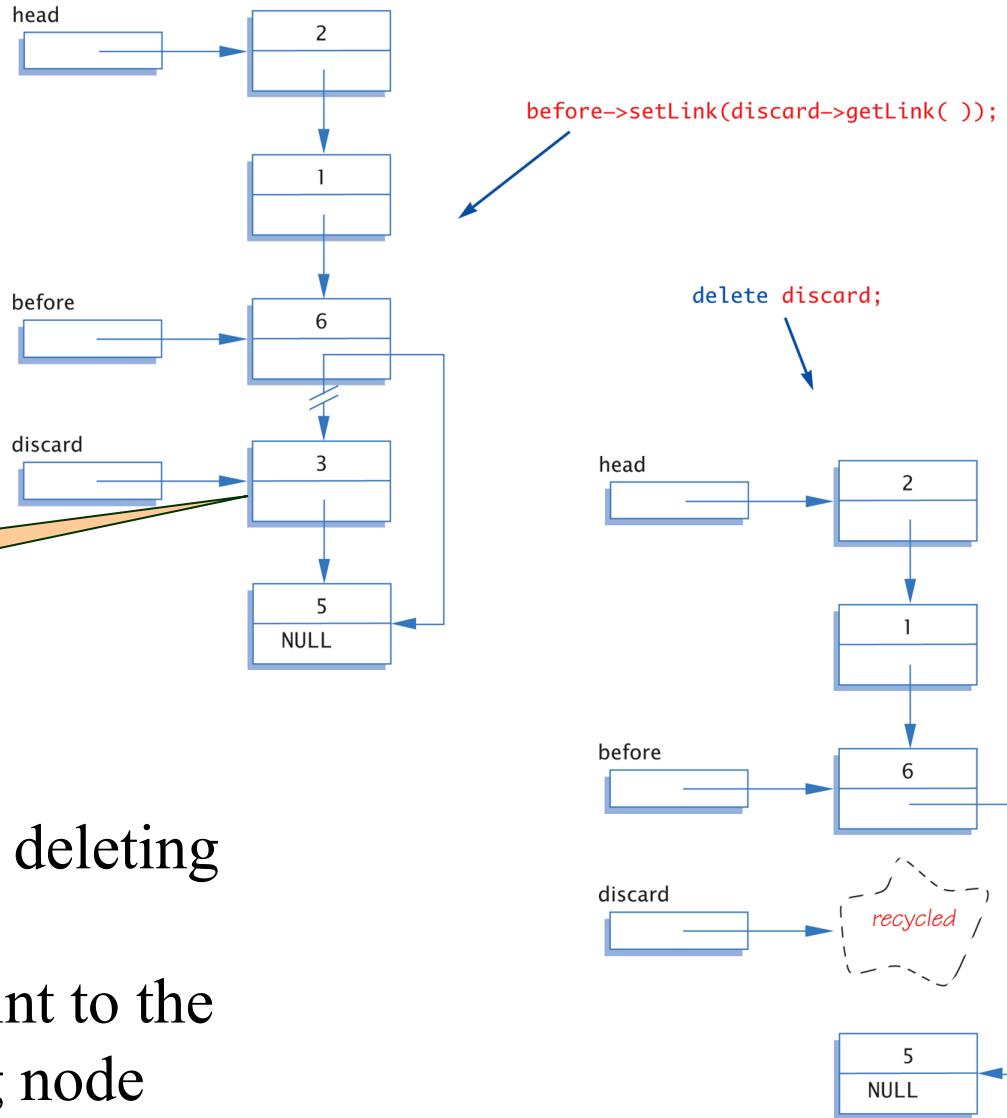
- Find the position to insert
- Create a node
- Put in the data
- Put in the pointer of next node
- Change the pointer of the node before

# Deleting a Node

Delete this node

- Find the node before the deleting node
- Change its pointer to point to the node next to the deleting node
- Recycle the memory.

Display 17.7 Removing a Node

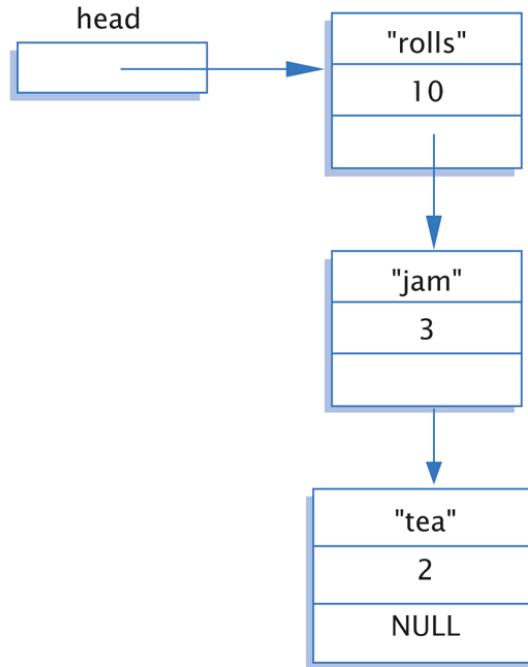


# Modify a node

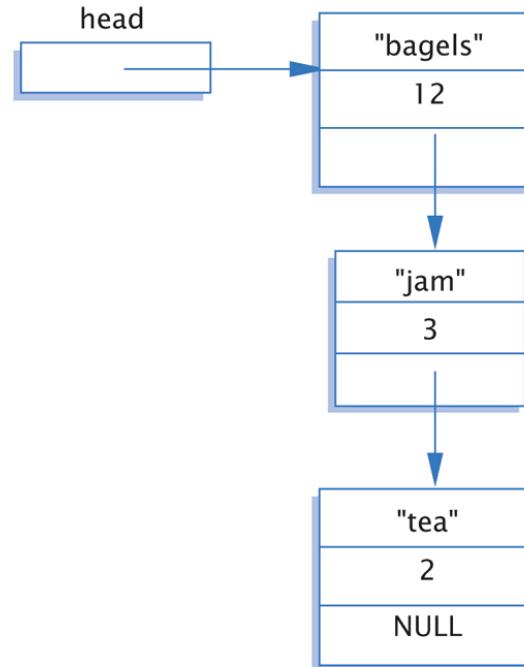
---

```
head->count = 12;  
head->item = "bagels";
```

*Before*



*After*



# Homework

---

- ◆ Read textbook Chapters 16 & 17.
- ◆ Work on your assignment 2 if you have not complete. The deadline for assignment 2 is 5pm Friday 14 Oct 2022.
- ◆ Demonstration of assignment 2 will be in your practical class next week.

# Comp2014 Object Oriented Programming

---

## Lecture 12

### Standard Template Library

# Topic covered in last lecture

---

- ◆ Function template
- ◆ Class template
- ◆ General concept of data structures
- ◆ Linked list

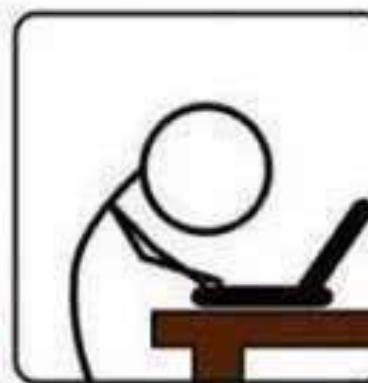
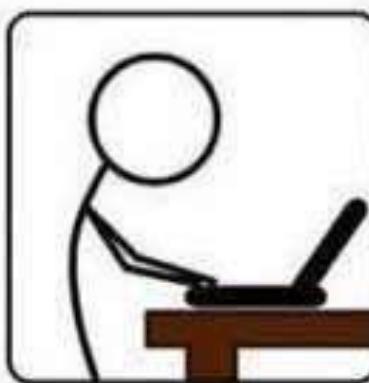
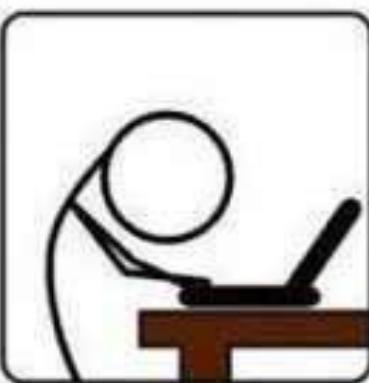
# The Programmers Life

WORK

HOME

PLAY

SLEEP



# The Zen of Python, by Tim Peters

---

*Beautiful is better than ugly.*

*Explicit is better than implicit.*

*Simple is better than complex.*

*Complex is better than complicated.*

*Flat is better than nested.*

*Sparse is better than dense.*

*Readability counts.*

*Special cases aren't special enough to break the rules,*

*Although practicality beats purity.*

*Errors should never pass silently,*

*Unless explicitly silenced.*



# The Zen of Python, by Tim Peters

---

*In the face of ambiguity, refuse the temptation to guess.*

*There should be one - and preferably only one - obvious way to do it  
Although that way may not be obvious at first unless you're Dutch.*

*Now is better than never,*

*Although never is often better than \*right\* now.*

*If the implementation is hard to explain, it's a bad idea.*

*If the implementation is easy to explain, it may be a good idea.*

*Namespaces are one honking great idea - let's do more of those!*



# Complex is better than complicated

---

```
#include "Constants.h"
#include "Ticket.h"
#include "EventTicket.h"
#include "FlightTicket.h"
#include "HotelVoucher.h"
#include "ClientRequest.h"
#include "RequestGenerator.h"
#include "Package.h"
#include "TravelAgent.h"
#include "SmartTravelAgent.h"
```

# Complex is better than complicated

---

```
cout <<"Choose a class to test: " << endl;
cout <<"1. Test ClientRequest class" << endl;
cout <<"2. Test FlightTicket class" << endl;
cout <<"3. Test HotelVoucher class" << endl;
cout <<"4. Test EventTicket class" << endl;
cout <<"5. Test RequestGenerator class" << endl;
cout <<"6. Test Package class" << endl;
cout <<"7. Run Travel Agent" << endl;
cout <<"8. Run Smart Travel Agent" << endl;
cout <<"9. Quit" << endl;
```

# Complex is better than complicated

---

```
void runTravelAgent() {  
    TravelAgent agent;  
    agent.readClientRequests();  
    agent.generatePackages();  
    agent.printSuccessfulPackages();  
}
```

# Complex is better than complicated

---

```
void runSmartTravelAgent() {  
    SmartTravelAgent smartagent;  
    smartagent.readClientRequests();  
    smartagent.generatePackages();  
    smartagent.printSuccessfulPackages();  
    smartagent.vacancy();  
}
```

# Complex is better than complicated

---

```
void TravelAgent::generatePackages() {  
    for (int i = 0; i < counter; i++) {  
        Package* p = generatePackage(requests[i]);  
        if (p != NULL && p->validayPackage()) {  
            successfulPackages.push_back(p);  
        } else {  
            cout << "Client " << requests[i].cId  
            << ": No sufficient budget or resources." << endl;  
        }  
    }  
}
```

# Complex is better than complicated

---

```
Package* TravelAgent::generatePackage(ClientRequest cr) {  
    Package* p = new Package;  
    p->setRequest(cr);  
    int flyinDay = cr.earliestEventDay();  
    int flyoutDay = cr.latestEventDay();  
    p->addFlightTicket(0, flyinDay);  
    p->addFlightTicket(1, flyoutDay);  
    for (int i = 0; i < NUMBEROFEVENTS; i++) {  
        if (cr.events[i])  
            p->addEventTicket(i);  
    }  
    for (int i = flyinDay; i < flyoutDay; i++)  
        p->addHotelVoucher(cr.hotelType, i, 0.0);  
    return p;  
}
```

# Topic covered in the lecture

---

- ◆ Standard Template Library (STL)
  - Sequential containers
  - Associative containers
  - Container adaptors
  - Iterators
  - Algorithms

# Standard Template Library

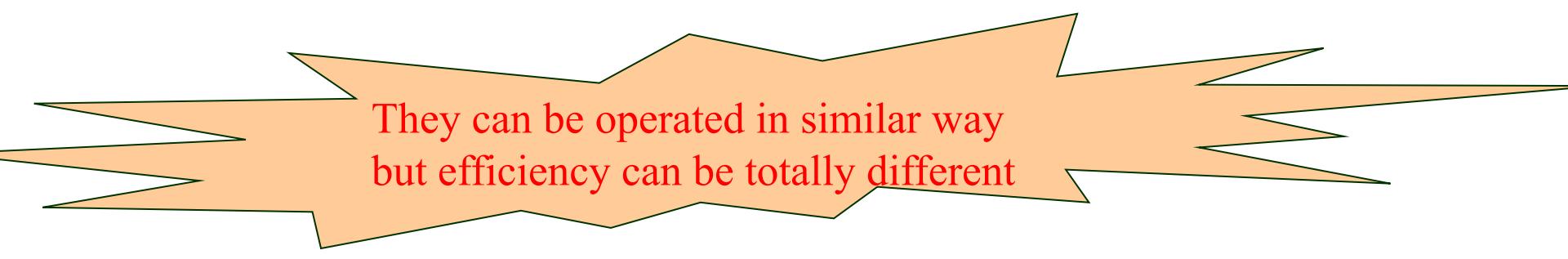
---

- ◆ To make programming simpler and easier, most C++ compilers provide class library for users to use. ANSI/ISO recommended a certain list of such classes as Standard Library. Most of these classes are in template form so that they are **independent** to data types. These class templates become the basis of the **Standard Template Library** (STL).
- ◆ STL includes:
  - **Containers**: are *template classes from which individual data structures can be constructed.*
  - **Iterators**: are pointers *keeping track of positions in a data structure and establishing the boundaries of sequences of elements.*
  - **Algorithms**: are *template functions that provide useful search, sort, location, and other numeric functions.*

# Container types in STL

**Container classes** are class templates which provide a possibility to use higher abstraction level in the programming. There are three types of containers:

- ◆ **Sequential container:** *Each element has a certain position that is determined when the element is inserted.* Typical sequential containers are:
  - **vector:** *similar to an array but the size of vector is variable.*
  - **deque:** *double ended queue, elements added at front or back while allow random access via an iterator.*
  - **list:** *doubly linked list, insertions & deletions anywhere in list.*



They can be operated in similar way  
but efficiency can be totally different

# Container types in STL

---

- ◆ Associate container: *an associative container contains a list of elements with variable size that supports efficient retrieval of elements based on keys.* STL has four associative containers:
  - set: *a collection of elements with no duplicates (sorted)*
  - multiset: *a collection of elements with duplicates (sorted)*
  - map: *store and retrieve data with unique keys (hashtable or balanced binary tree).*
  - multimap: *store and retrieve data with possible multiple keys (hashtable or balanced binary tree).*

# Container types in STL

---

- ◆ Container adapters *are class templates which provide member functions push and pop that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure.* STL has three container adapters:
  - Stack Adapter: LIFO
  - Queue Adapter: FIFO
  - Priority\_queue Adapter: *highest priority element first out*

# Vector and iterator



**Vector:** *an array-like container that stores a varying number of elements and provides random access to them.*

Header file:

```
#include <vector>
```

Typical operations which are effective:

- [index] : *random access with index*
- push\_back(): *appending a new element value at the end.*
- pop\_back(): *removing an existing value from the end.*

Operations that are not effective:

- insert(iterator position, const T& x): inserting a new element into the vector.  

Iterator will be explained later
- erase(iterator position): removing an element from the vector.

# Iterator

An iterator is like a pointer. It points to an element in a container.  
Mostly we use an iterator as a loop variable.

Given a vector: `vector<ClientRequest> requests;`  
we can declare an iterator as follows:

```
vector< ClientRequest>::iterator itor;
```

Then to use it, we can have

```
for (itor = requests.begin(); itor != requests.end(); itor++)  
    cout << itor->budget;
```

Alternative ways:

```
for (int i=0; i< requests.size(); i++)  
    cout << requests[i].budget;
```

or

```
for (ClientRequest cr : requests) // C++11  
    cout << cr.budget;
```

See iterator.cpp

# deque

---

*Deque (double ended queue): a double-ended queue which elements can be added to or removed from the front or the back.*

Header file:

```
#include<deque>
```

Typical operations:

- `push_back()`: appending a new element value at the end.
- `push_front()`: appending a new element value at the beginning.
- `pop_back()`: removing an existing value form the end.
- `pop_front()`: removing an existing value form the beginning.
- `insert(iterator, value)`: insert a value into the deque at the position  
(not efficient)

Check out other methods of deque via

<http://www.cplusplus.com/reference/deque/deque/>

# list

**List:** an implementation of doubly linked list template. Each node of a linked list contains two pointers, pointing to the previous and the next nodes, respectively.

Header file:

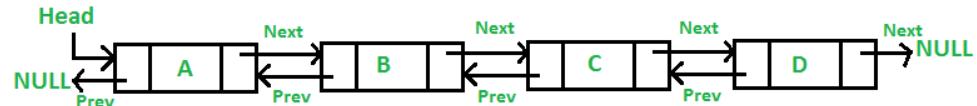
```
#include<list>
```

Typical operations:

- `insert(iterator position, const T& x)`: inserting a new element into the list.
- `erase(iterator position)`: removing an element from the list.

Remark 1. The operations `push_back`, `pop_back`, `push_front` and `pop_front` are defined also for list.

Remark 2. If you want to iterate over all elements you need **iterator** to do it. You can't use index.



# Typical methods for sequential containers

---

- ◆ Member function `size()` indicates the current number of elements stored in the container (it has nothing to do with the size of allocated space).
- ◆ Member function `capacity()` indicates how many elements can be stored to the container without the need to allocate more memory space.
- ◆ Function `push_back()` and `push_front()` allocates automatically more memory space when needed.
- ◆ Function `reserve()` can be used to allocate a certain amount of memory. This is often useful to prevent unnecessary memory allocations in `push_xxx` operations.
- ◆ Function `resize()` can also be used to resize the underlying array.

# set

---

**Set** is used for efficient management of object collections with **sortable keys** so that insertion, deletion, and search operations can be performed with logarithmic runtimes. Sets are implemented internally with **binary search trees**. Set will automatically sort elements and eliminate duplicated elements.

Operator “`<`” for **elements** determines the sorting order by default (users have to overload the `operator<` for user defined data type).

The new elements are put in a set using the function `insert()` (`push_back()` and `push_front()` are not available).

The best way to access the elements is to use **iterators**.

Header file `<set>` for set.

See `set.cpp`

# map

**Unordered\_map** and **Map** store pairs of sorted keys and objects using **hashing technology** and **binary search trees**, respectively. The key of each object is used to identify the object but is stored separately from the object. The comparison criterion is applied to the keys.

**Example:**

```
map<string, double> stringDoubleMap;  
stringDoubleMap[ "e" ] = 2.71828;  
stringDoubleMap[ "pi" ] = 3.14159;  
stringDoubleMap.insert(pair<string, double>( "key" , 10.0 ));  
  
map<int, ClientRequest> requests;  
map<int, Record> studentRecords;
```

See example map.cpp

# Algorithms

---

STL provides a great number of algorithms which can be used across the various containers. Examples include *searching*, *sorting*, *inserting* and *deleting*. There are approximately 70 algorithms included.

**Eg1.** *copying all elements to a stream object:*

```
vector<int> v;  
copy( v.begin(), v.end(), output);
```

this copies all elements of a vector v from the beginning to the end to *output*.

**Eg2.** *insert an element into a vector:*

```
v.insert(v.begin()+1, 22);
```

this inserts 22 at the second position.

**Eg3.** *count elements:*

```
int result = count( v.begin(), v.end(), 8);
```

this will count the number of occurrences of 8 between the beginning and the end of the vector.

# Mathematical functions

---

## Eg4. *find minimum element*

```
cout << "The minimum element is " << *(min_element(v.begin(), v.end()));  
this uses the function min_element to locate the smallest element in  
the vector and returns a pointer to that smallest element.
```

## Eg5. *calculate total*

```
cout << "The total of vector v is " << accumulate(v.begin(), v.end(), 0);  
this uses the function accumulate (in the <numeric> header file) to  
total all elements in vector v.
```

# Example

```
#include <vector> #include <iterator> #include <algorithm> #include <iostream>
using namespace std;
void main() {
    float c_array[5] = {3.0, 4.0, 5.0, 2.0, 1.0};
    vector<float> stl_array(c_array, c_array + 5);
    sort(stl_array. begin(), stl_array.end()); //sort is STL algorithm
    vector<float>::iterator pos;
    pos = find(stl_array.begin(), stl_array.end(), 2.0);
    if (pos == stl_array.end())
        cout << "2.0 was not found " << endl;
    else
        cout << "2.0 was found in the position " << pos - stl_array.begin() << endl;
}
```

See example algorithm.cpp

# Homework

---

- ◆ Read textbook Chapter 19.
- ◆ Demonstrate your code of assignment 2 during your practical session. **No demonstration, no marks.**
- ◆ There would be a significant delay for the assignment checking. If you have any urgency, please let your tutor know. You may be granted to show your work in week 14.
- ◆ Online tutorial 3 should have been available.
- ◆ The lecture in next week will be a review of this unit. I will show you the format of the final examination and some example questions. No previous examination papers are available in the library.

# Comp2014 Object Oriented Programming

---

## Lecture 13

## Unit Review

**and Sample exam questions**

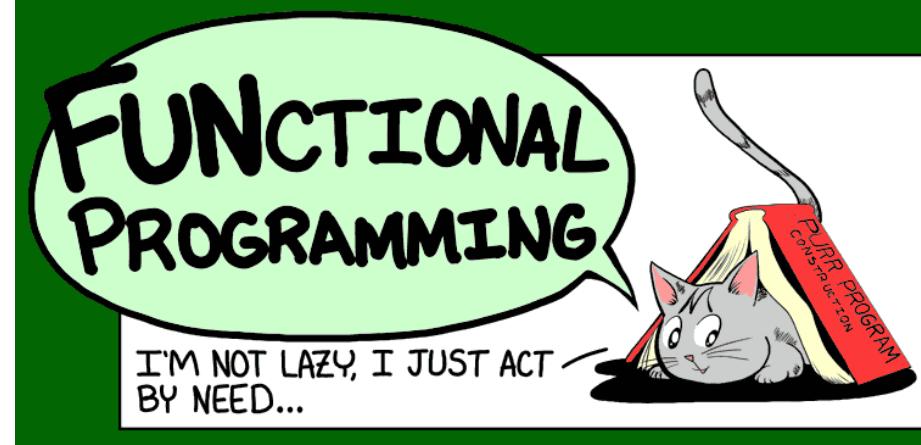
# What are covered today

---

- ◆ Review of topics that were covered in the unit
- ◆ Structure of final exam
- ◆ Sample exam questions
- ◆ Review of Assignment 2
- ◆ Announcements

# Topics covered by the unit

- ◆ Review of fundamental concepts of programming
  - Variable declaration and **variable scopes**
  - Input and output
  - Control logic:
    - » branching and looping



- ◆ Functions
  - Function declaration and definition
  - Function calls and returns
  - Parameter passing: *call-by-value* and *call-by-reference*  
*e.g., void getMove(Board b, int& x, int& y)*
  - Function overloading: *f(int)*, *f(double)*, *f(int, int)*
  - Default arguments: *f(int i=0, double j=2.3)*

call with a pointer

# Topics covered by the unit

---

## ◆ Arrays

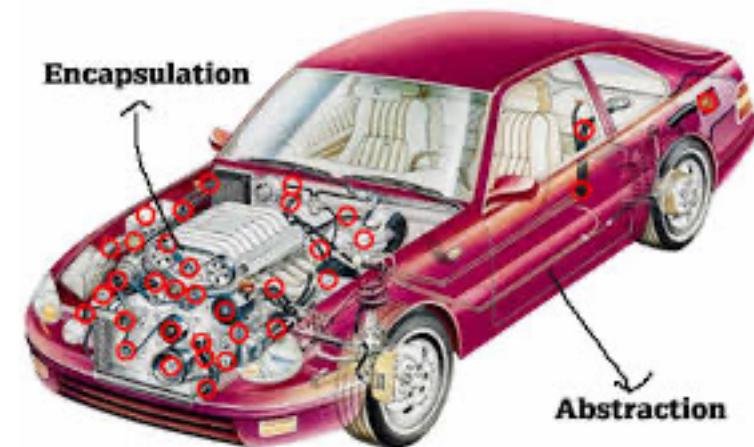
- Declaration: `DataType a[ ]` in C++ and `DataType[ ] a` in Java;
- Array name is a **pointer**.
- Data access using index **or** pointer.
- Common process in an array: *find maximum and minimum, average, linear search, binary search, sorting and etc.*
- Pass an array into a function: *its **name** and **size***
- Return an array from a function: *static array can be returned as a pointer*
- Multi-dimensional array: *array of arrays*
- Dynamic array: *memories are from heaps and size is flexible*
  - » *vector in STL is a typical dynamic array*
  - `vector<ClientRequest> requests;`
  - `vector<Package> packages;`

# Topics covered by the unit

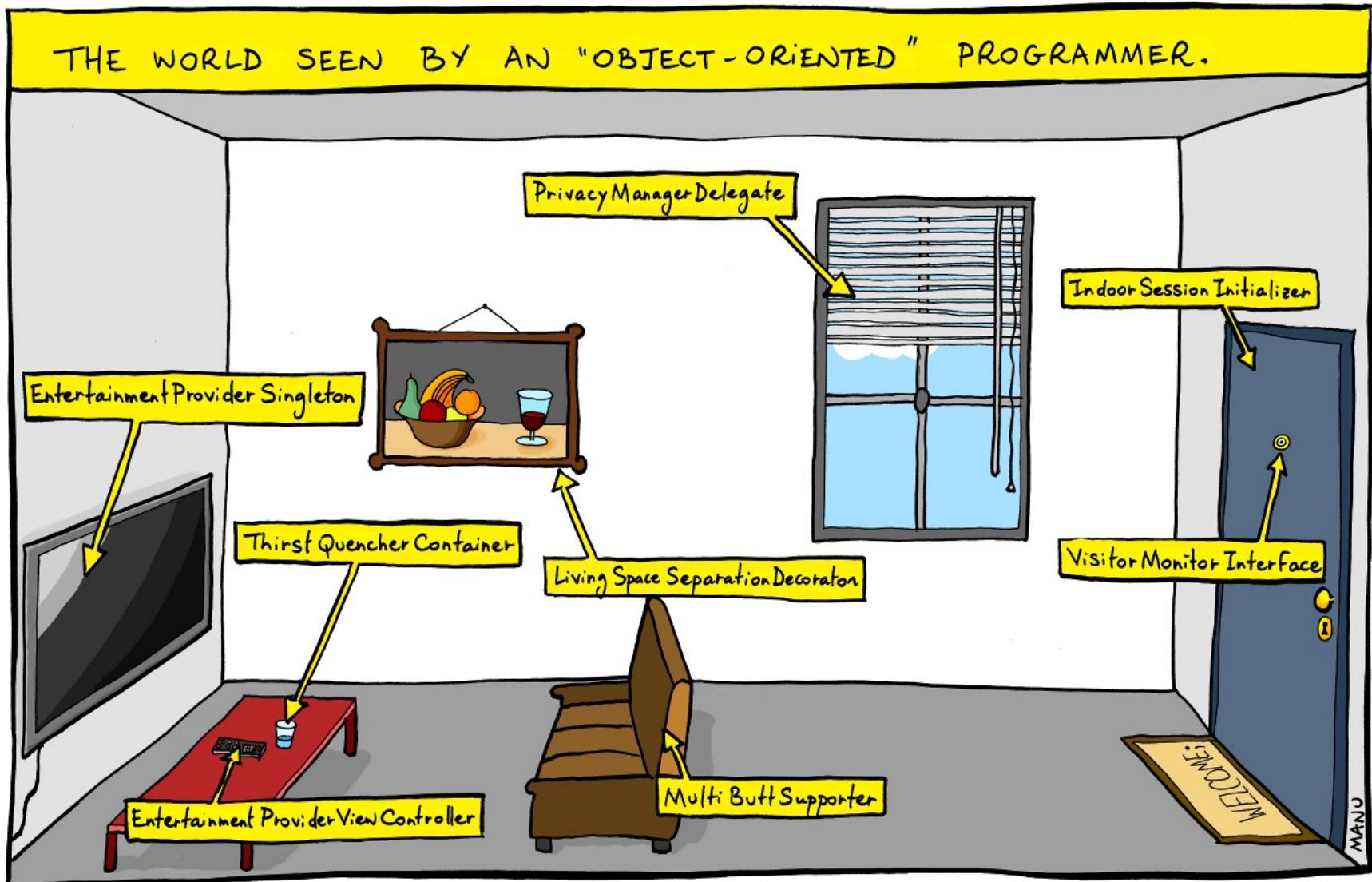
## ◆ Objects and classes

- Data abstraction: abstract data type (ADT)
- Data encapsulation: data items and methods
- Data hiding: private, protected and public
- Build a class: declaration and definition
- Class applications: create objects (variables are objects).
- Constructor and destructor

Abstraction  
Encapsulation  
Data hiding

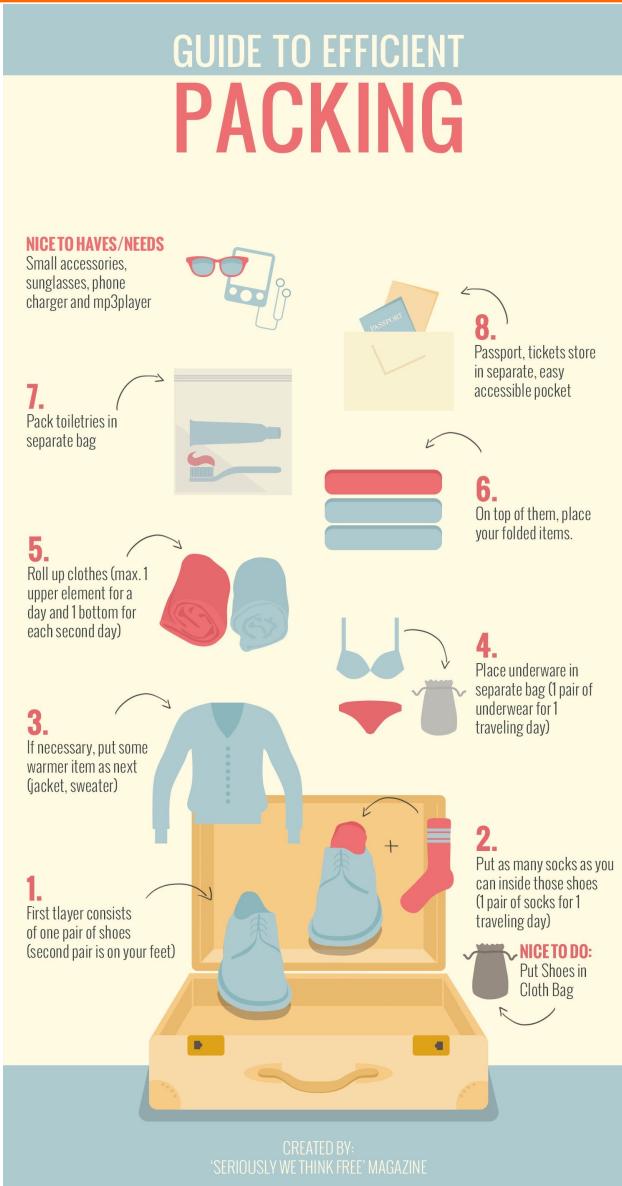


# Thinking in OO: abstraction



Picture was acquired from the Internet

# Thinking in OO: encapsulation



Methods  
(less memory!!!)

Data members  
(occupy  
memory!!!)

Picture was acquired from the Internet

# Thinking in OO: data hiding



Picture was acquired from the Internet

# Topics covered by the unit

- ◆ **Static variables:** in a function or in a class
  - Unique to a function (independent to function calls, initialise once)
  - Unique to a class (share by all the objects of the class)
- ◆ **Reference:** an alias of a variable, not a copy of object but referred to the same object it is assigned to.
- ◆ **File I/O:** input/output to/from iostream
  - Make use of redirection operators << and >>, which automatically recognize built-in data types.

```
ClientRequest request; char c;
Fin >> request.budget >> c >> request.hotelType >> c;
while (c != ']') {
    int eid;
    fin >> eid;
    request.events[eid] = true;
    fin >> c;
}
```

9020, 4 [14, 2, 7]

# Topics covered by the unit

## ◆ Strings

- C-style strings
- String class
- Conversion

## ◆ Operator overloading

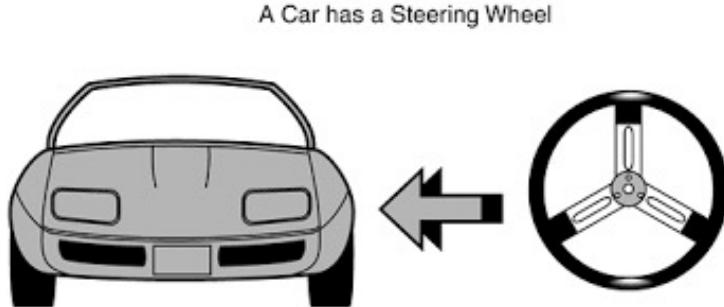
- Make the operators that have been used for built-in data type also available for the data types you define.
- Typical ones that may need to be overloaded: `<`, `==`, `<<`, `++`,  
`+=`, `[ ]`, ...
- Overload an operator only if you feel it is helpful.

```
bool operator<(const ClientRequest& cr) const {
    if (budget > cr.budget) return true;
    return false;
}
```

# Topics covered by the unit

inheritance

- ◆ **Composition:** *part-of* or *has-a* relation
- ◆ **Inheritance:** *is-a* relation
  - Class declaration: use : for *extends*
  - Inheritance type: *private*, *protected* and *public*
  - Access control: *how to use base class's members*
  - Constructors and destructors
  - Type conversion: *upcasting (always possible)* and *downcasting*
  - Class hierarchy: a tree of “is-a” relations (base classes on top)



# Thinking in OOP

- ◆ Derived classes of Ticket:

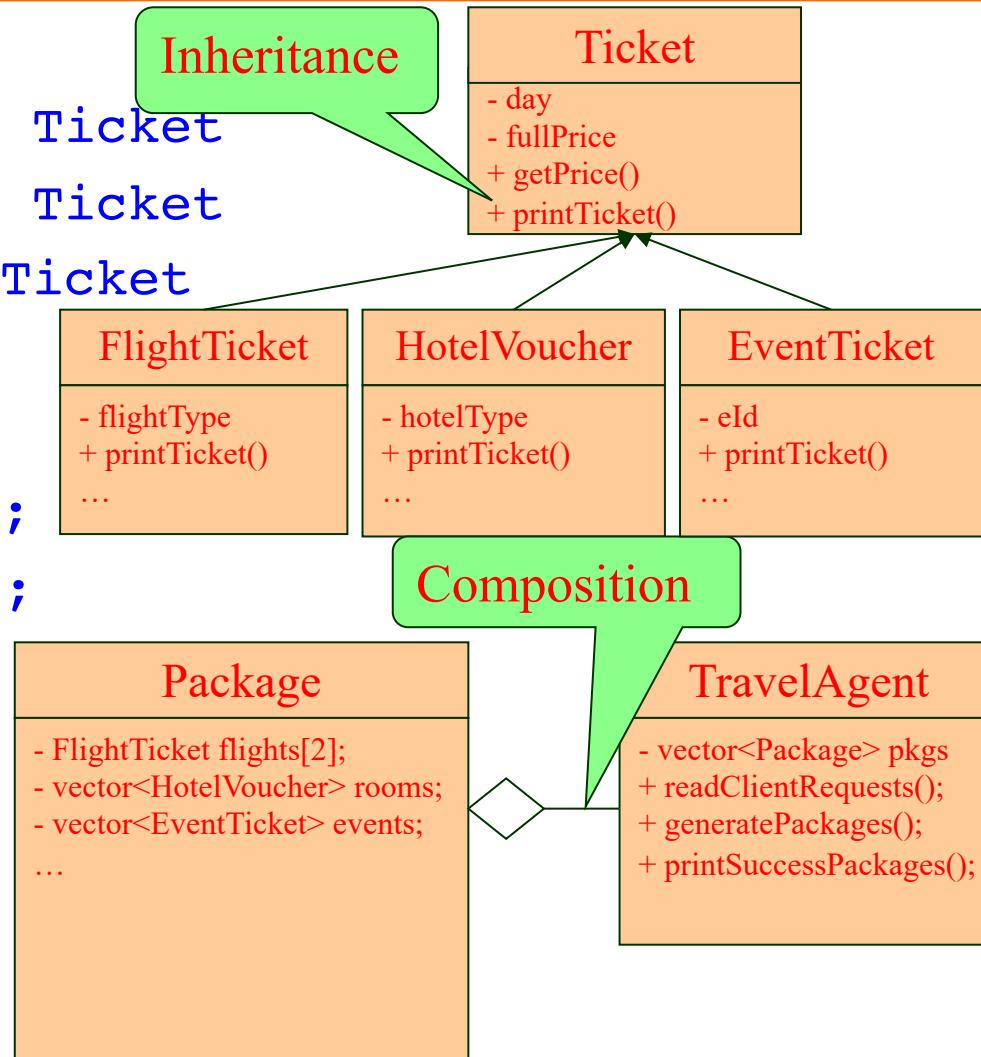
```
class FlightTicket: public Ticket  
class HotelVoucher: public Ticket  
class EventTicket: public Ticket
```

- ◆ A travel package contains:

```
FlightTicket flights[2];  
vector<HotelVoucher> rooms;  
vector<EventTicket> events;
```

- ◆ A travel agent does:

```
readClientRequests();  
generatePackages();  
printSuccessfulPackages();
```



Unified Modeling Language: UML

# Topics covered by the unit

## ◆ Polymorphism

- Static binding
- Dynamic binding
- Virtual functions
- Purely virtual functions and abstract classes



polymorphism

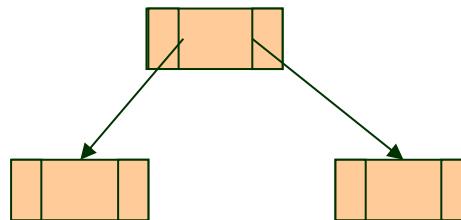
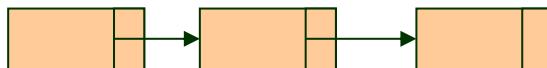
virtual

<b>Player</b>
- playerType - Board b + getMove(Board, int&, int&)
<b>SmartPlayer</b>
+ getMove(Board, int&, int&) ...
<b>Game</b>
- Board b - Player players*[2]; + Game(Board b, Player* p1, Player* p2) ...

# Topics covered by the unit

## ◆ Linked list

- The simplest data structure other than array
- More efficient for insertion and deletion
- Fundamental for understanding tree structures
- Typical operations in a linear structure
- Efficiency of operations with different implementations



# Topics covered by the unit

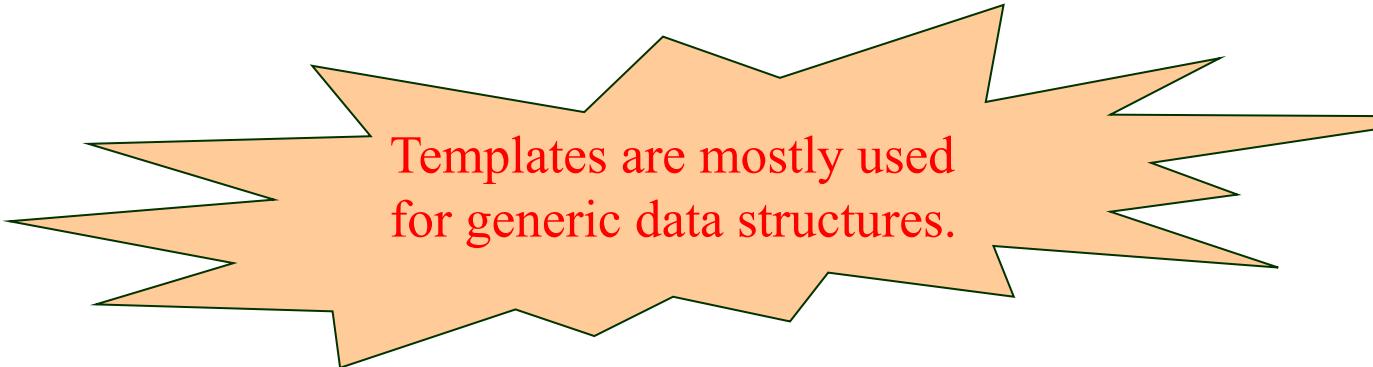
---

## ◆ Templates

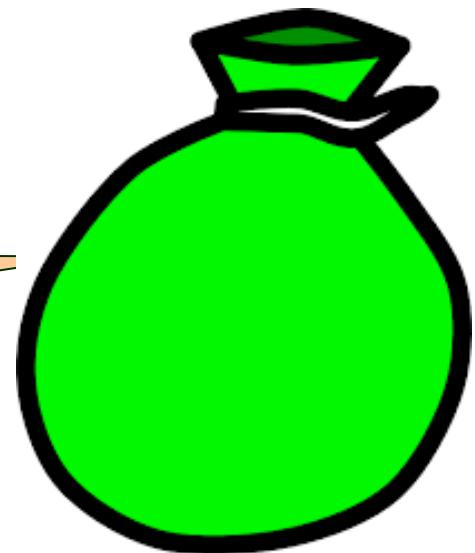
- Function template
- Class template

## ◆ Standard Template Library

- Containers: *sequential containers, associative containers and container adaptors*
- Iterators
- Algorithms



Templates are mostly used for generic data structures.



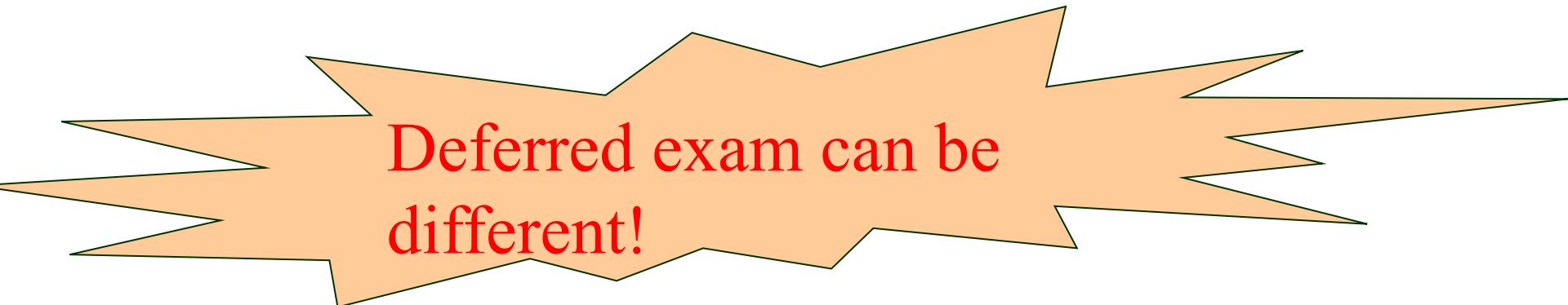
# Format of final exam

- ◆ Due to the COVID-19 Pandemic, the final examination in this semester will be run online again via zoom and vUWS.
- ◆ All students who have enrolled this unit are required to take the final examination during the university scheduled time (unless a deferred examination is approved).
- ◆ The students are requested to connect via zoom to the unit coordinator's zoom id 30 minutes before the examination starts. The connection should remain with video showing students face until the end of examination.
- ◆ The students must check out with the teaching team to make sure all submissions have been recorded by the system.
- ◆ The whole examination will be recorded for future reference.

9:30-11:30am 7 Nov 2022

# Format of Final Exam

- ◆ **Part A:** multiple choice questions (30 questions 1 mark each)
- ◆ **Part B:** Programming (5 tasks for 20 marks)



Deferred exam can be  
different!

A practice site of the final  
examination will be set up and  
available on vUWS soon!

# Format of final exam

---

- ◆ **Multiple choice questions (30 marks):**

- 30 questions, one mark each.
  - Designed for being completed within 30-45 minutes.
  - Completed on vUWS at the first hour, automatically closed at the end of the first hour. No access after that.
  - Questions are taken from a big pool (120 questions) and presented one question each time in random order. You would expect some typos with the multiple choice questions. Screenshot any possible errors and email me **after the examination**. I will investigate it and adjust marks accordingly.

# Format of final exam

---

## ◆ Programming tasks (20 marks):

- The programming part consists of five **tasks** for 20 marks in total.
- Designed to be completed within 75 minutes.
- You can only use C++ to complete the tasks.
- The code for the programming tasks must be submitted on vUWS by the end of the examination (will be given an additional 10 minutes for submission).
- Only source code is needed for submission. It can be based on any IDE. If we can't run your code, we will contact you but we will mainly focus on your implementation logic.

# Format of final exam

---

- You will be required to submit two identical versions: one **for Turnitin checking** and the other one **for documentation and marking**.
- The submission to the Turnitin site is for similarity verification. If the similarity of two submissions is higher than a threshed, the related students will be requested to demonstrate their work to a tutor during a scheduled time after the examination.

# Format of deferred exam

---

- ◆ The deferred examination will be in the same format as the final examination except that every student is required to demonstrate their code for the programming tasks.
- ◆ As long as the campuses are accessible to students, the deferred exam will be held at the campus.
- ◆ Students with AIP will take examination at the same time but will be possibly given longer time for completion based on AIP specification.

# Source of the questions

---

- ◆ **Lecture notes:** demonstrated code and basic concepts.
- ◆ **Practical questions:** practice for both multiple choice questions and programming
- ◆ **Online tutorials:** practice for multiple choice questions
- ◆ **Assignment tasks:** practice for programming
- ◆ A few multi-choice questions might be out of teaching scope.

# Sample questions: Part A

---

## Part A Multiple-choice questions

Which of the following statements is INCORRECT?

- A. A constructor is a special kind of member function. It is automatically called when an object of that class is declared.
- B. A constructor has the name of its class name.
- C. A constructor can return any type of values.
- D. A constructor can have multiple parameters.

# Sample questions: Part A

---

## Part A Multiple-choice questions

Which of the following statements is INCORRECT?

- A. A constructor is a special kind of member function. It is automatically called when an object of that class is declared.
- B. A constructor has the name of its class name.
- C. A constructor can return any type of values.
- D. A constructor can have multiple parameters.

# Sample question: Part A

---

If a variable is declared as \_\_\_\_\_ data member in a class, then it is common to all the objects of the class.

- A. static
- B. const
- C. public
- D. private

# Sample question: Part A

---

If a variable is declared as \_\_\_\_\_ data member in a class, then it is common to all the objects of the class.

- A. static
- B. const
- C. public
- D. private

# Sample questions: Part A

Given the following program, which of the class member accesses in the `main()` function are **LEGAL**?

```
class DayOfYear {  
public:  
    void input();  
    string output();  
    // other public members  
private:  
    int month;  
    int day;  
    // other private members  
};
```

```
int main() {  
    DayOfYear birthDay;  
    birthDay.input();           // a)  
    birthDay.day = 25;          // b)  
    cout << birthDay.output();  // c)  
    if(birthDay.month == 1)      // d)  
        cout << "January\n";  
}
```

- A. b) and d)
- B. a) and c)
- C. a), b) and c)
- D. None of the above

# Sample questions: Part A

Given the following program, which of the class member accesses in the `main()` function are **LEGAL**?

```
class DayOfYear {  
public:  
    void input();  
    string output();  
    // other public members  
private:  
    int month;  
    int day;  
    // other private members  
};
```

```
int main() {  
    DayOfYear birthDay;  
    birthDay.input();           // a)  
    birthDay.day = 25;          // b)  
    cout << birthDay.output();  // c)  
    if(birthDay.month == 1)      // d)  
        cout << "January\n";  
}
```

- A. b) and d)
- B. a) and c)
- C. a), b) and c)
- D. None of the above

# Part B

---

Download the program `BaseProgram.cpp` from. Convert it into OO-Style by making the following changes on the original code:

**Task 1 (5%):** Change the definition of `CDAccount` from `struct` to `class`. Make all the data members of the class to be private. Move the free function `inputDate(Date& theDate)` and `getCDDData(CDAccount& the Account)` in the class as public member functions of the class (remove the parameters of function `getCDDData`)

**Task 2 (5%):** Change the data member `double balanceAtMaturity` of class `CDAccount` into a public member function `double balanceAtMaturity()` to calculate the balance at maturity (simply move the related code from the main function of the original program to implement the function).

# Part B: Programming tasks

---

- ◆ **Task 3 (5%):** Add a new public member function, *printAccount()*, to class **CDAccount** to print the information of a CD account ( you may also simply move the related code from the main function of the original code).
- ◆ **Task 4 (5%):** Make appropriate changes on the whole program to guarantee the new program does the same as the original code.

# Look ahead

---

- OOP is just the beginning towards professional programmer
- Other units for further programming training:
  - **300103 Data Structures and Algorithms**
  - 300167 Systems Programming 1
  - 300130 Internet Programming
  - 300698 Operating Systems Programming
  - 300115 Distributed Systems and Programming
  - 300960 Mobile Applications Development
  - 300165 Systems Administration Programming
  - 300582 Technologies for Web Applications
- AI Major: M3110 Major in Artificial Intelligence
- Units related to Artificial Intelligence:
  - 301174 Artificial Intelligence
  - 300093 Computer Graphics
  - 300569 Computer Security
  - 301205 Robotic Programming