C++ Programming:
Introduction to C++ and OOP (Object Oriented Programming)

2019년도 2학기

Instructor: Young-guk Ha
Dept. of Computer Science & Engineering

Contents

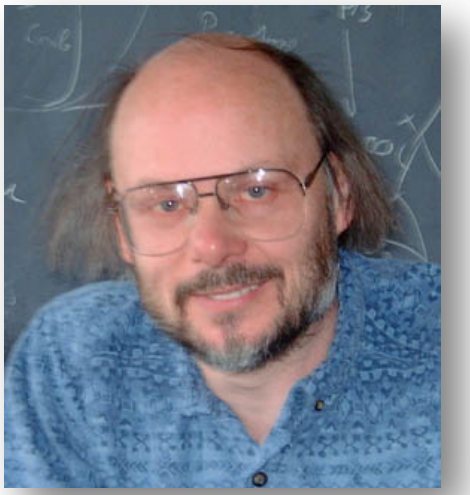
- Brief introduction to C++
- OOP vs. Procedural Programming
- Fundamental concepts of OOP
  - Information Hiding
  - Encapsulation
  - Inheritance
  - Polymorphism

Brief Introduction to C++

- C++ is originated from C language
  - C language is a ***procedural programming language***
    - Consists of functions: main / library functions / user-defined functions
    - Data types and structures: built-in / user-defined
- Prehistory of C++ language
  - 1960's: Multics written in Assembler, i.e., "***A***" (AT&T Bell Labs)
  - 1970: Ken Thompson developed language "***B***"
    - Improvement of Multics (UNIX) is written in Assembler and B
    - B is higher level than A, but has no data types and structures
  - 1973: Dennis Ritchie turned B into "***C***" language
    - Eventually, UNIX is rewritten in C
    - "***C++***" is based on C, i.e., A → B → C → C++

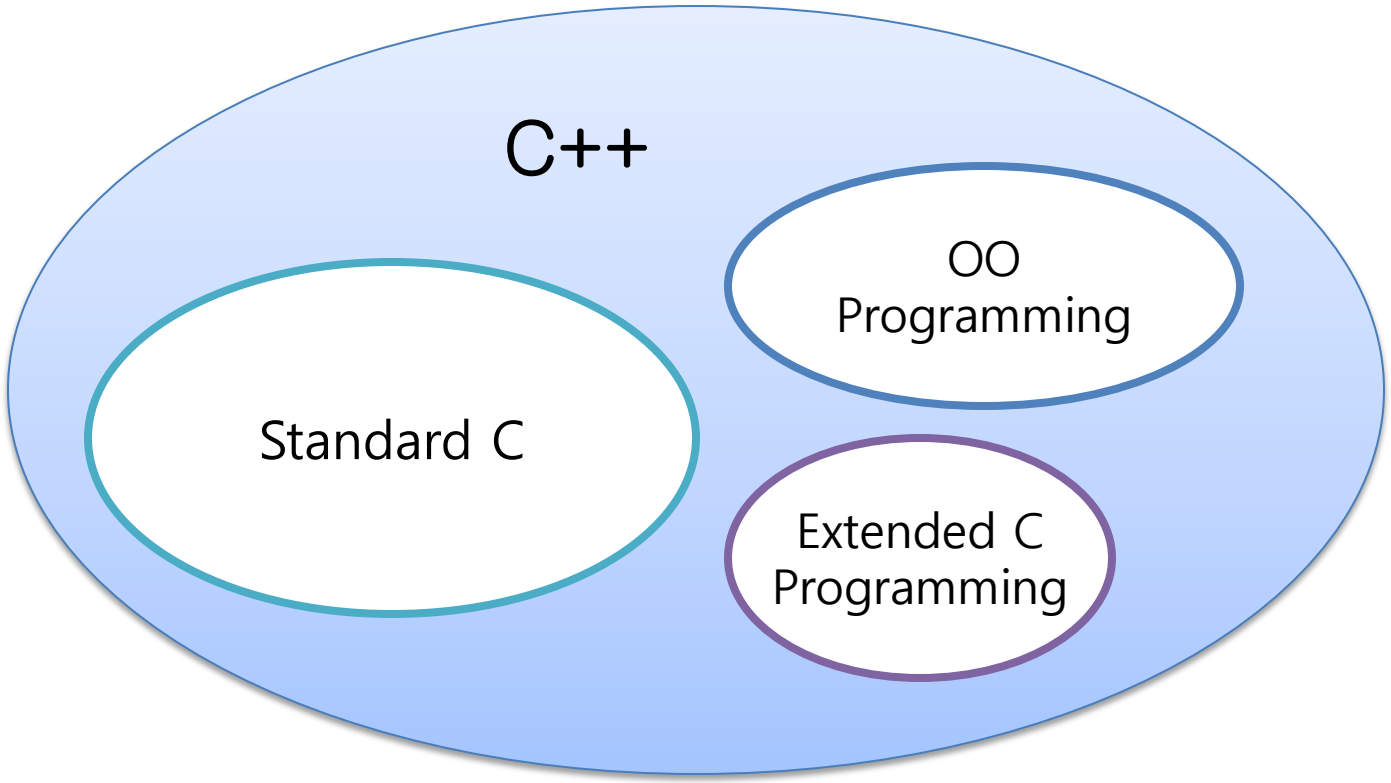
Brief Introduction to C++ (Cont.)

- C++ language first developed by Dr. Bjarne Stroustrup (AT&T) in 1980's
  - Extending and improving standard C language with Classes
    - Thus, first name was "C with Classes"
  - Hybrid language
    - Supporting both procedural and object-oriented programming
  - C++ has many distinctive programming constructs that are not found in C language
    - Constructs for extended C programming
    - Constructs for object-oriented programming



Brief Introduction to C++ (Cont.)

- Basic constructs of C++



Modules

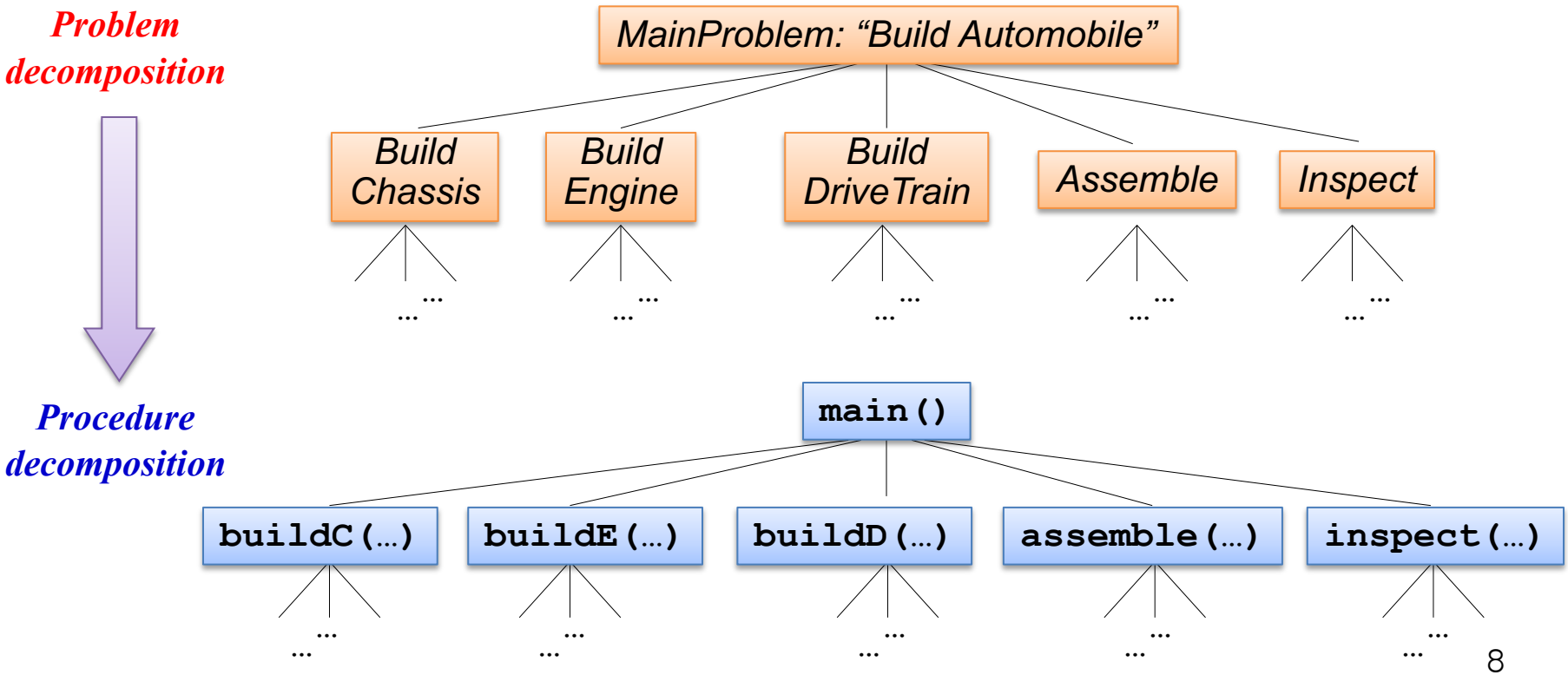
- Programs consist of ***modules***
  - Module is building block of program that can be designed, coded, and tested separately
  - Modules are assembled to form a program
    - In procedural programming
      - Modules are ***procedures***
      - E.g., Functions in C language
    - In OOP
      - Modules are ***classes***
      - E.g., Classes in C++ and Java

Procedural Programming

- Based on “top-down design"
  - We can solve a problem with a procedure
  - When a problem that is too complex to solve straightforwardly with one procedure
    - Functionally decompose the problem (procedure) into subproblems (subprocedures)
    - And a subproblem may be further decomposed until it is straightforward enough to be solved in one procedure
  - Pros and cons
    - Intuitive way to solve many complex problems
    - Hard to maintain software
      - Cascading changes: a change in the top-level procedure (e.g., **main** function) may cascade down to its subprocedures, subprocedures, and so on

Procedural Programming (Cont.)

- Functional decomposition example
  - Main problem: to build an automobile



Object-Oriented Programming

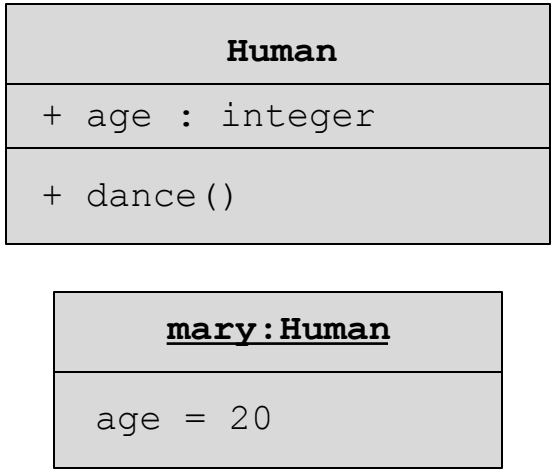
- Object-Oriented Design
  - Definition: the process of planning a system of interacting ***objects*** for the purpose of solving a software problem
    - UML (Unified Modeling Language): a standardized general-purpose modeling language in the field of object-oriented software design
  - Object-oriented design addresses major problems of the top-down design such as cascading changes
  - Focuses on classes designed to model the entities with which a problem deals, not procedures
    - A class is a blueprint for objects ⇔ an object is an instance of class (i.e., class = collection of possible objects)
    - Objects in a class share ***properties*** (= features or attributes)
    - A class has ***operations*** (= actions or processes)

Object-Oriented Programming (Cont.)

- Object-Oriented Programming (OOP)
  - A programming paradigm based on object-oriented design
    - OOP languages: Smalltalk, Eiffel, Delphi, Object-C, C#, Python, Ruby, C++, Java, …
    - C++, Java and Python are most popular object-oriented programming languages nowadays
  - Modules of OOP are classes
    - Properties are called ***member variables*** or ***data members***
    - Operations are called ***methods*** or ***function members***
  - Development of an OO program
    - To define classes and their relationships (e.g., inheritance)
    - To declare objects and make them interact with each other by invoking their methods

Object-Oriented Programming (Cont.)

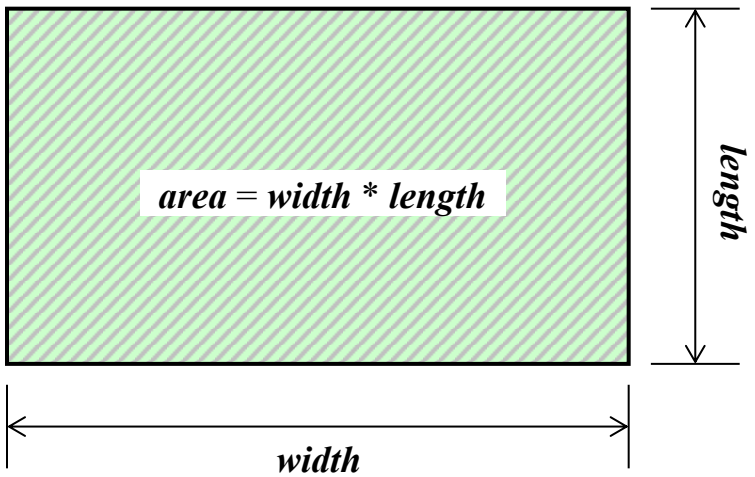
OO Design	OO Programming
class	class
object	object
property	data member (member variable)
operation	method (function member)

[UML]
 
	[C++]
	`class Human {  // class definition`
	`  public:`
	`  int age;  //  a data member`
	`  void dance() { //  a method`
	`  …`
	`  }`
	`}`
	
	`Human mary;  // object declaration`
	`mary.age = 20;`

OOP vs. Procedural Programming

- Problem: calculating rectangular area
  - Prodecdural programming to define a calculation function

```
// calcRectArea function definition
int calcRectArea(int width, int length) {
   int area;
   area = width * length;
   return area;
}
...
int area = calcRectArea(10, 20);    // function invocation
```



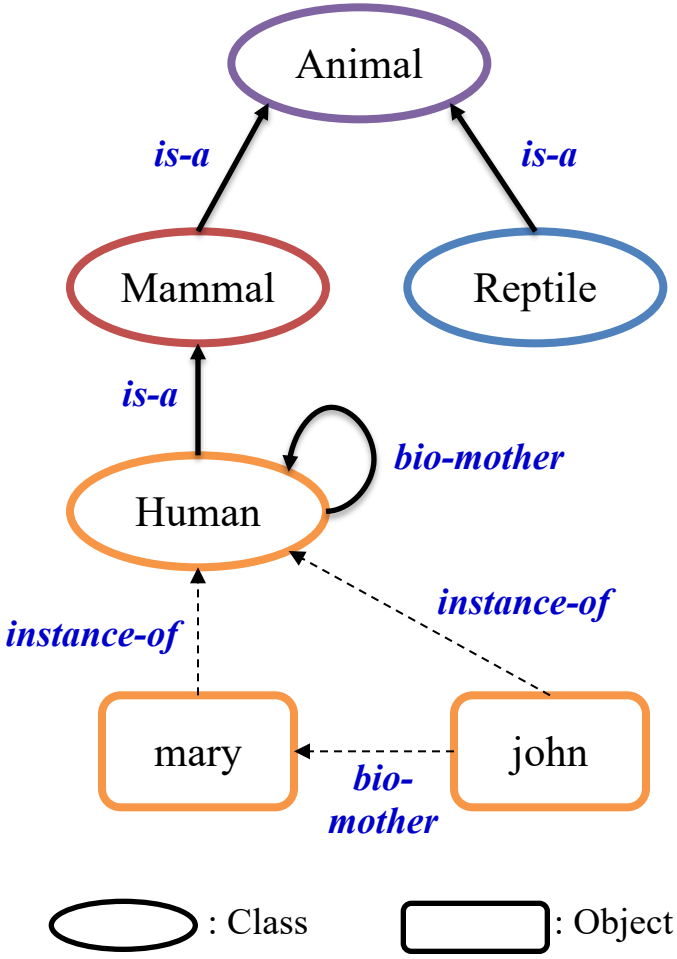
OOP vs. Procedural Programming (Cont.)

- Problem: calculating rectangular area
  - OOP to define Rectangle class

```
class Rectangle {                       // Rectangle class definition
 private:
  int width;                            // data member 1
  int length;                           // data member 2
 public:
  setRect(int w, int l) {               // method 1
     width = w; length = l;
  }
  int calcArea() {                      // method 2
     return width * length;
  }
}
...
Rectangle rec;                          // Rectangle object declaration
rec.setRect(10, 20);                    // method 1 invocation
int area = rec.calcArea();              // method 2 invocation
```

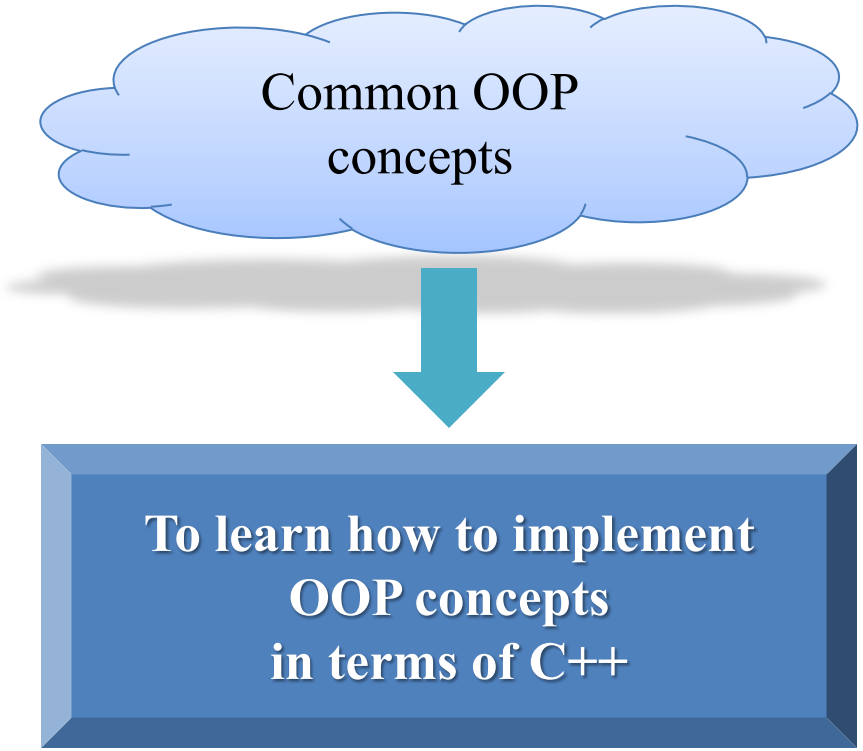
Classes, Objects and Their Relationships

- Relationships among classes
  - ***is-a*** (***subclass-of***) relationship
    - Eg., Human ***is-a*** Mammal,
      Mammal ***is-an*** Animal
    - ***is-a*** relationships create an ***inheritance*** hierarchy (will be discussed in detail later)
  - Relationship by a property
    - Eg., Human's ***biological-mother*** is Human
- Relationships for objects
  - ***instance-of*** (***object-of***) relationship
    - Eg., mary is an ***instance-of*** Human
  - Relationship among objects
    - Eg., john's ***biological-mother*** is mary



Fundamental Concepts of OOP

- ✓ *Information hiding and Encapsulation*
- ✓ *Inheritance*
- ✓ *Polymorphism*



Information Hiding

- Programmers using a class method need **NOT** know the details on the implementation
  - You only need to know "what the method does"

- Information hiding
  - To design a method so programmers can use the method without knowing the details on "how it works"
    - To separate "what" from "how" when designing methods
    - Also referred to as "***abstraction***"
  - E.g., **`public`** and **`private`** modifiers of C++
    - Used to control access to a class's data members and methods from the outside of the class
      - **`public`**: expose to the outside world
      - **`private`**: hide from the outside world

Encapsulation

- An OO design technique to achieve the principle of information hiding
  - Divides a class into
    - Exposed part ("what it does"): ***interface***
    - Hidden part ("how it does"): ***implementation***
  - Real world example: driving a car
    - Interface: we see and use "brake", "accelerator", and "steering wheel"
    - Implementation: we do not know "mechanical details"

- Example in C++
  - Interface
    - Public data members and headings of public methods
    - Documentations and manuals
  - Implementation
    - Bodies of public methods
    - Private data members and methods

Encapsulation Example (1)

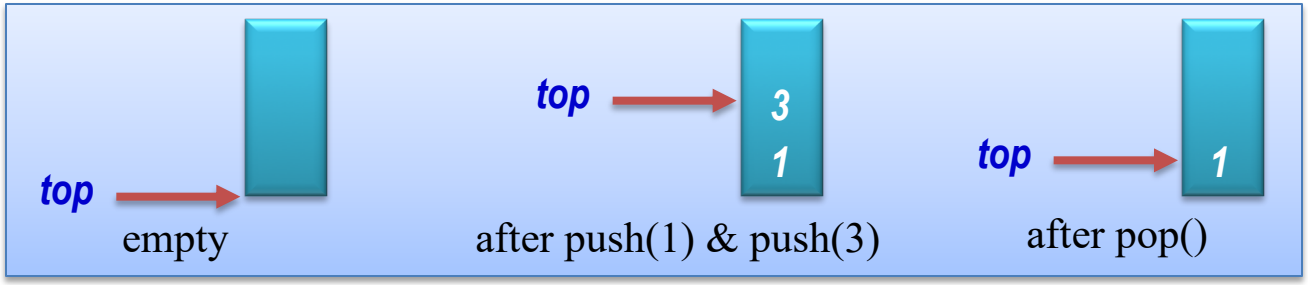
- Class String

```
class String {
  // Data members
  private:
    char value[];              // represents chars in the string
    int len;                   // represents string's length

  // Methods
  public:
    String subString(int, int)  {      /* hidden */      }
    void concatString(String)   {      /* hidden */      }
    void setChar(char, int)     {      /* hidden */      }
    bool containChar(char)      {      /* hidden */      }
    int length()                {      /* hidden */      }
}
```

Encapsulation Example (2)

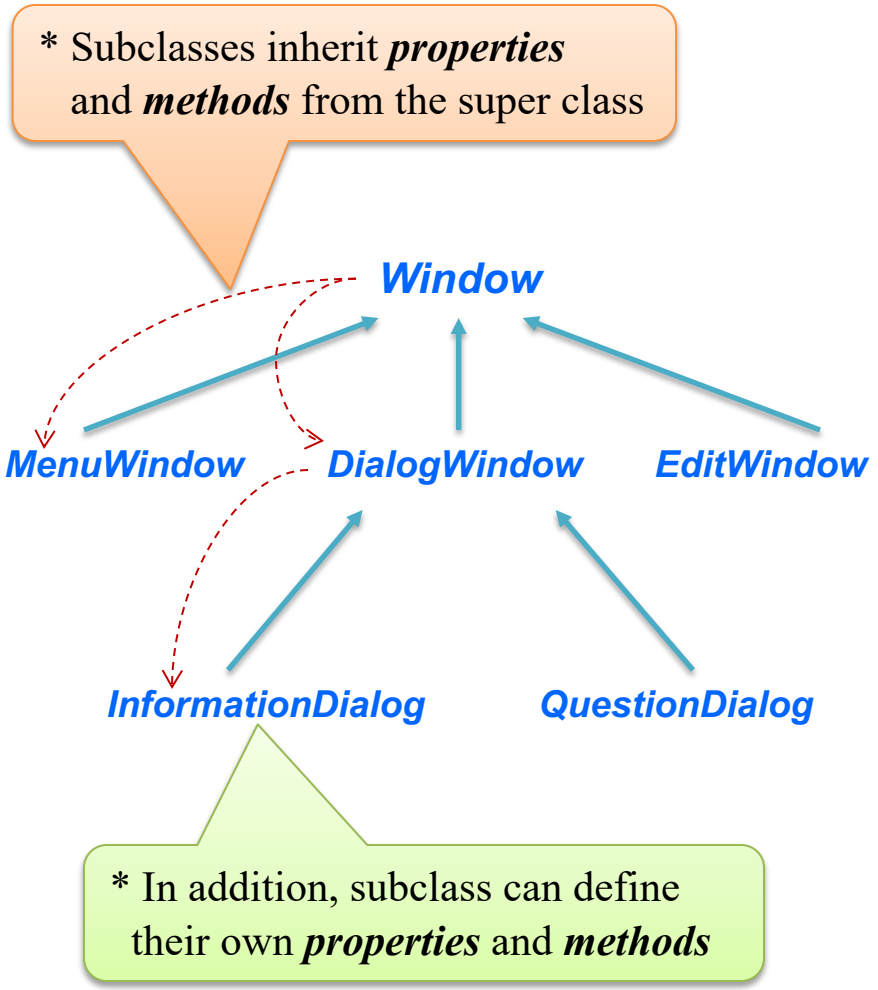
- Class IntStack
  - Stack
    - A data structure to store arbitrary objects using LIFO (Last-In First-Out) mechanism
  - Basic operations
    - **push**(object ***o***): inserts data object ***o*** into the Stack if it is not full
    - object **pop**(): removes and return the most recently inserted object from the Stack if it is no empty

```
class IntStack {
  // Data members (hidden)
  private:
    int stack[10];   // int stack
    int top = -1;    // empty

  // Exposed methods
  public:
    void push(int o) { top++; … }
    int pop() { … top--; }
}
```



Inheritance

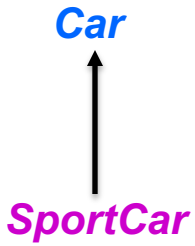
- Inheritance hierarchy
  - Set of parent-child (i.e., ***is-a***) relationships among classes
    - Parent class a.k.a. ***super class*** or ***base class***
    - Child class a.k.a. ***subclass*** or ***derived class***
  - A class should be within an inheritance hierarchy (otherwise called stand-alone)
- Code reuse with inheritance
  - Inheritance promotes a form of code reuse by ***specialization*** (or ***extension***) of a super class into subclasses



Types of Inheritance

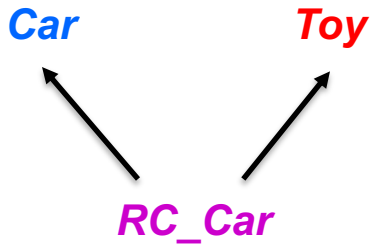
- Single inheritance
  - A child class has exactly one parent
  - Every OOP language must support at least single inheritance

```
class SportCar : public Car {
      // members inherited from Car
      // members newly defined in SportCar
}
```



- Multiple inheritance
  - A child class may have multiple parents
  - C++ supports multiple inheritance (c.f., no multiple class inheritance in Java*)

```
class RC_Car : public Car, public Toy {
      // members inherited from both Car and Toy
      // members newly defined in RC_Car
}
```



* Java has no multiple inheritance. But we can simulate with multiple interfaces in Java.

Polymorphism

- Definition of polymorphism
  - In a dictionary
    - The ability to appear in many forms ("다형성")
  - In computer science
    - To allow a program code to work with various types
  - In OOP
    - The ability of an interface to be implemented in multiple ways
  - In C++
    - Actually refers to runtime binding of a method
      - I.e., in C++, a method is ***polymorphic*** if its binding occurs at runtime rather than compile time

Polymorphism (Cont.)

- A powerful tool for programmers
  - A programmer can invoke an object's polymorphic method without exactly knowing which class type or subtype is involved
  - E.g., a **`Window`** class hierarchy
    - Imagine base class **`Window`** and its 20 or so derived classes
    - Each class has a polymorphic **`display`** method with the same signature, which performs some operation specific to the class (window type)
    - With polymorphic methods, programmers can invoke 20 or so different (differently implemented) methods with the same interface according to the situation

Example of Polymorphism

- Various versions of the **`display`** method of **`Window`** classes, e.g.,
  - Class **`Window`** has a **`display`** method that draws basic window frame on the screen
  - Class **`MenuWindow`** has a **`display`** method that draws a list of choices in addition to the basic window frame

- A programmer can use a single **`display`** method for any type of **`Window`** object
  - E.g., **`w->display();`**
    - If **`w`** points to a **`Window`** object, its **`display`** is invoked
    - Else if **`w`** points to a **`MenuWindow`** object, its **`display`** is invoked

❖ Note that the system determines which version of **`display`** to be invoked at runtime (not compile time)