Object-oriented programming (with C++)

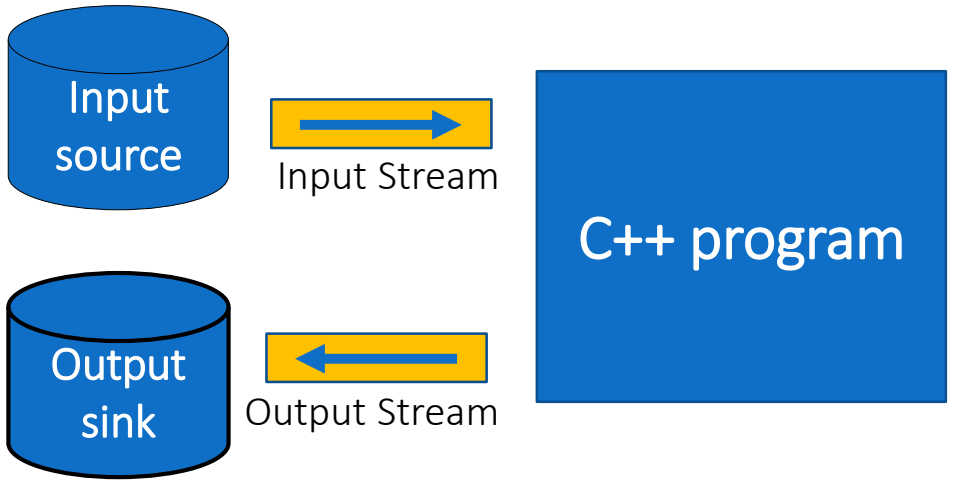
Lecture #2: Overview of basic structures in C++

Non Object-Oriented Extensions to c

- Major improvements over C
  - Stream I/O
  - Declarations
  - Parameter passing by reference
  - Default argument values

Stream I/O in C++

- Input and output in C++ can be handled by streams.
- A stream is a sequence of bytes that may be either *input* to a program or *output* from a program



- **cin**, **cout**, and **cerr** are three standard devices for input (keyboard), output and error output (tied to the screen)
- **iostream** should be included (using namespace std)

I/O Stream in C++

- Buffered (***cin, cout***)
- Unbuffered (***cerr***)
- Buffered characters should be flushed
- Unbuffered characters can be seen immediately

I/O Stream in C++

- The directive **`#include <iostream.h>`** declares 2 streams: **`cin`** and **`cout`**
- **`cin`** is associated with standard input.
  - Extraction: **`operator>>`**
- **`cout`** is associated with standard output
  - Insertion: **`operator<<`**

Simple example (cout)

```
// A Simple Hello-World Program
#include <iostream>
main() {
    /* the output statement */
    std::cout << "Hello World! \n";
}//ex1-hello.cpp
```

Simple example (cout)

```
#include <iostream>
using namespace std;
main() {
    int a = 15, b = 10;
    cout << "a = " << a << ", b = " << b << "\n";
    float f = 3.14;
    cout << "f=" << f;
    cout << 5 << ", " << 26.5 << ", " << (a+b) << '\n';
    char ch ='A'; int n = 65;
    cout << ch << ", " << int(ch) << ", " << char(n) << endl;
}//ex2-cout.cpp
```

Simple example (cin)

```
#include <iostream>
using namespace std;
main() {
        int a, b;
        float f; char ch;
        cout << "Enter two integers, one float, and a char: ";
        cin >> a >> b >> f >> ch;
        cout << "a = " << a << ", b = " << b << ", f = " << f << ", ch = " << ch << endl;
        char name[80];
        ch = '\0';
        int i = 0;
        cout << "Enter your name (with '#' at the end): \n";
        while (1) {
          cin >> ch; // ch = cin.get()
          if (ch == '#') break;
          name[i++] = ch;
        }
        name[i]= '\0';
        cout << name << endl;
}//ex3-cin.cpp
```

Simple example

```
#include <iostream>
#include <fstream>
int fileSum();
int fileSum()
{
        std::ifstream infile("Ex2file");
        int value, n;
        int sum=0;
        //Read the number of integers
        infile>>n;
        //Read members
        int i=0;
        while (i<n)
        {
                infile>>value;
                sum=sum+value;
                i++;
        }
        return sum;
        infile.close();
}
```

Why use I/O streams

- Streams are sub-classable; istreams, ostreams,… are real classes and hence sub-classable. We can define types that look and act like streams, yet operate on other objects. Examples:
  - Stream that listens to external port
  - Stream that writes to a memory area.

Why use I/O streams

- Streams may be faster: **`printf`** interprets the language of '%' specs, and chooses (at runtime) the proper low-level routine. C++ picks these routines statically based on the actual types of the arguments.
- Streams are extensible; I/O mechanism is extensible to new user-defined data types.

Declarations in C++

- In C++, declarations can be placed anywhere (except in the condition of a **`while`**, **`do/while`**, **`for`** or **`if`** structure.)
- An example

```
cout << “Enter two integers:”;
int a, b;
cin >> a >> b;
cout << “The sum of” << a << “ and ” << b
     << “ is ” << x + y << endl;
```

Parameter passing by reference

- Parameters: variables in the method definition (the member + the type)
  - int f(int x) {…}

- Arguments: the actual value of this variable that gets passed to function. The member in the calling
  - f(m)

Parameter passing by reference

- In C, all function calls are call by value.
  - Call be reference is simulated using pointers.

- *Reference parameters* allows function arguments to be changed without using return or pointers.

Call by value

```
#include <iostream>
using namespace std;
int incrementByValue(int);
main()
{
        int x = 5;
        cout << "x = " << x << " before incrementByValue\n"
                << "Value returned by sqrByVal: "
                << incrementByValue(x)
                 << "\nx = " << x << " after incrementByValue\n\n";
}
int incrementByValue(int a)
{
        return a += 10;
        // caller's argument not modified
}
```

Call by Reference

```
#include <iostream>
using namespace std;
int incrementByRef(int &);
main()
{
        int z = 9;
         cout << "z = " << z << " before incrementByRef\n";
                incrementByRef(z);
                cout << "z = " << z << " after incrementByRef\n";
}
int incrementByRef(int &a)
{
        return a += 10;
        // caller's argument modified
}
```

Constant Reference Parameter

```
#include <iostream>
using namespace std;
void incrementByRef(const int &);
main()
{
        int z = 9;
         cout << "z = " << z << " before incrementByRef\n";
                incrementByRef(z);
                cout << "z = " << z << " after incrementByRef\n";
}
int incrementByRef(const int &a)
{
        return a += 10;
        // caller's argument modified
}
```

Default Arguments

- Parameters can be assigned default values
- Parameters assume their default values when no actual parameters are specified for them in a function call

Example

```
#include <iostream>
using namespace std;
int sum (int lower, int upper=10, int inc=1) {
  int sum = 0;
  for (int k = lower; k <= upper; k+= inc)
        sum += k;
  return sum;
}
main()
{
        cout << "Sum of integers from 1 to 10 is " << sum (1);
}//ex6-defaultarg.cpp
```

Pointers and Dynamic Memory Allocation

- Overviews
- Pointer Conversions
- Allocating Memory
- Arrays and Dynamic Allocation

Typical ways to declare a pointers

- Pointers
  - A **pointer** is a variable whose value is the address of another variable.

- Typical ways to declare a pointers:
  - char *p=message;
  - char * const p=message;
  - const char *p=message;
  - const char * const p=message;

Pointers to Constants

- **Pointers to Constants**: The object cannot be modified when this pointer is used for access

```
int x = 0;
const int * p = &x;
*p = 30; // Error!
x = 10; // OK!
```

Constant Pointer

```
char * const aStr="John Smith";
char msg[10]="John Smith";

char * const sp=msg;
sp++; //Error;
strcpy(sp, "Nam"); //OK
```

Pointer and function

- Cannot pass a pointer to a constant to a function with a parameter that is a pointer to a non-constant

```
char * aFunction(char *, const char *);
main()
{
    const char *a;
    const char *b;
     aFunction(a,b); //Error!
}
```

Pointer and function

- Pointer arguments:
  - Pass-by-reference.

Examples

Pointer conversions

- Pointers to Array Elements
- Void Pointers
- References to Pointers

Pointers to Array Elements

- Pointer is related to a type or a class
- The pointer pointing to an array can be incremented or decremented (pointer arithmetic)

```
float flist[] = { 10.3, 13.2, 4, 9.6 };
float *fp = flist;
fp++;
cout << *fp;
```

Pointers to Array

```
float flist[] = {10.3, 13.2, 4, 9.6};
float * fp = flist;
int * ip = (int *) fp;
ip++;
cout << *ip;
```

Void Pointers

- To obtain flexibility of types

```
void * memcpy (void *dest, const void * src, size_t nbytes);

const unsigned ArraySize = 500;
long arrayOne[ArraySize];
long arrayTwo[ArraySize];
memcpy (arrayOne, arrayTwo, ArraySize * sizeof(long));
```

Void Pointers

- Casting required

```
int *p;
void *v = p;
p = (int *) v; //cast needed
```

Allocating Memory

- Static and Dynamic Allocation
  - Allocating memory for objects is handled by the compiler– **Static Allocation**
  - Allocating memory for objects at run time and you can control the exact size and the lifetime of these memory locations – **Dynamic Allocation**
  - **Automatic memory allocation:** occurs for (non-static) variables defined inside functions.

Static and Dynamic Allocation

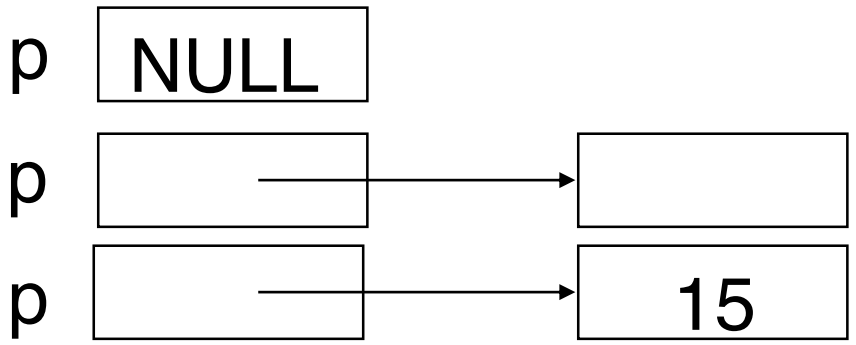
- De-allocating memory refers to releasing a block of memory whose address is in a pointer variable – deleting a pointer.

- Fragmentation: is the condition where gaps occur between allocated objects

The *new* Operator

- The ***new*** operator is used to allocate memory dynamically

```
int *p;
p = new int;
*p = 15;
int * array = new int[50];
```



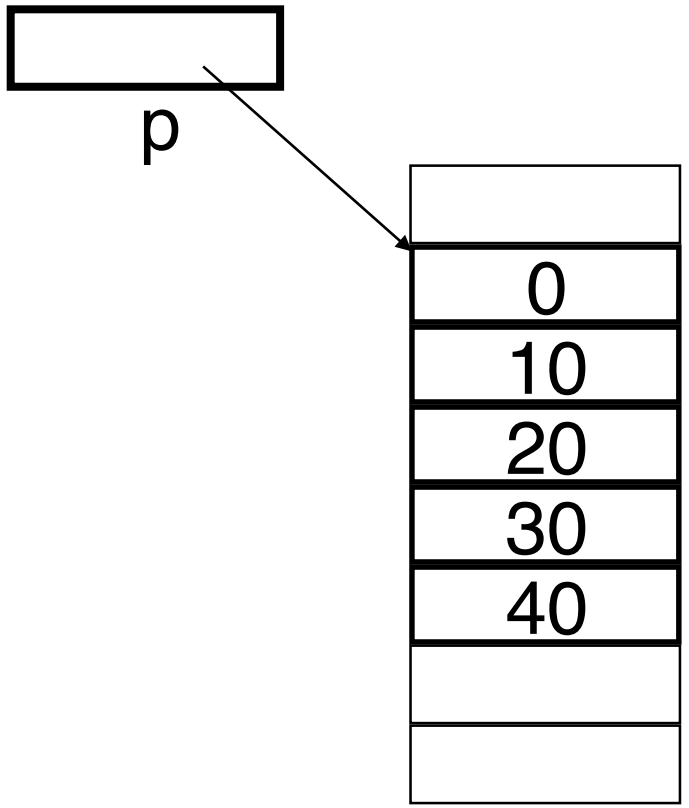
The *delete* Operator

- The ***delete*** operator is used to deallocate memory space (created dynamically)

```
delete  p;
delete [] array;
```

One-dimensional arrays

```
int *p = new  int[5] ;
for (int j=0; j < 5; ++j)
   *(p + j) = 10 * j;
for (int j=0 ; j < 5; j++ )
   cout  <<  p[ j ];
delete [] p;
//ex7_new1DArray.cpp
```

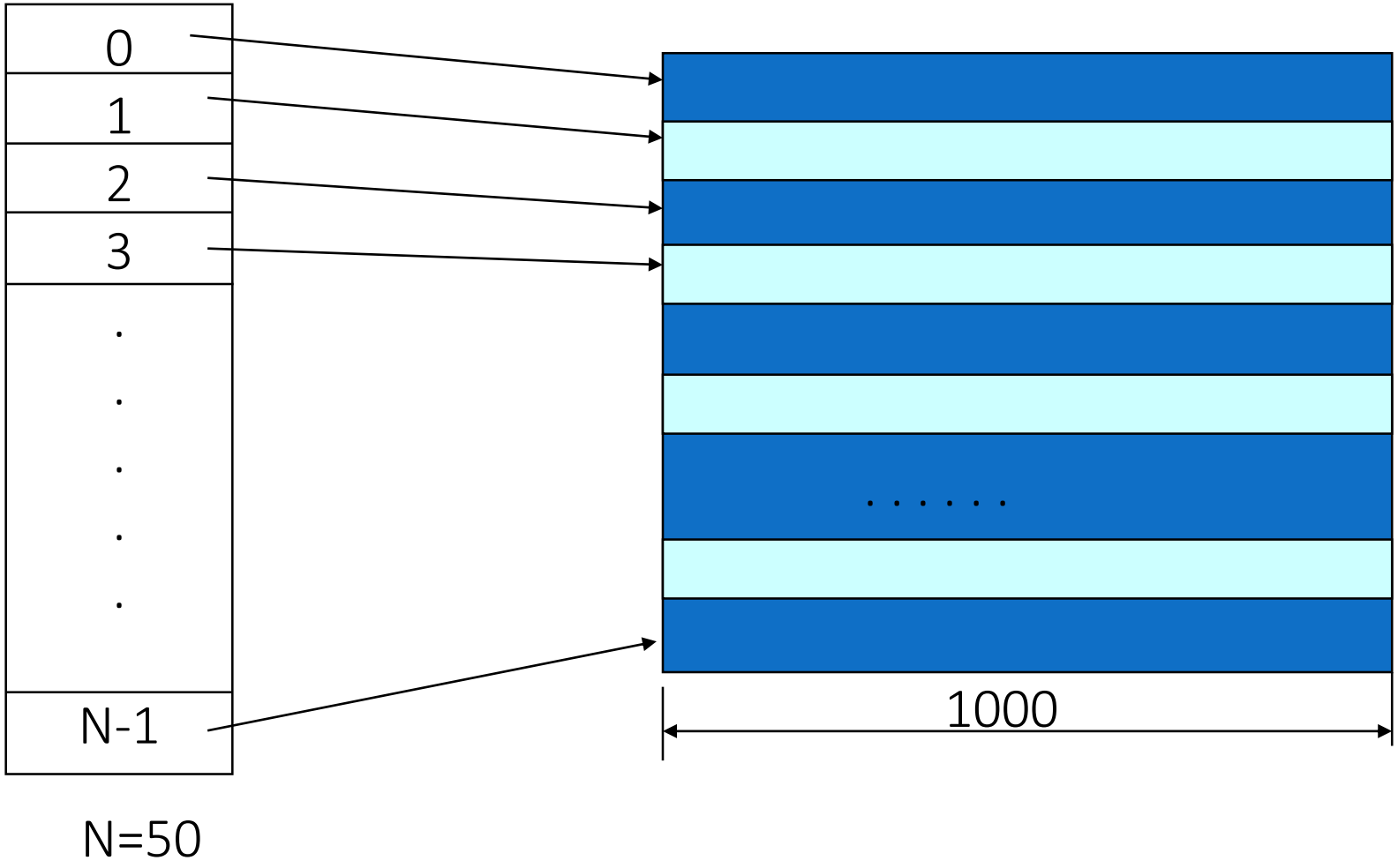


Two-Dimensional Arrays

- Array of Pointers

```
const unsigned NumRows = 50;
const unsigned RowSize = 1000;
int * samples[NumRows];
for (unsigned i = 0; I<NumRows; I++)
    samples[i]= new int[RowSize];//See the graph
```

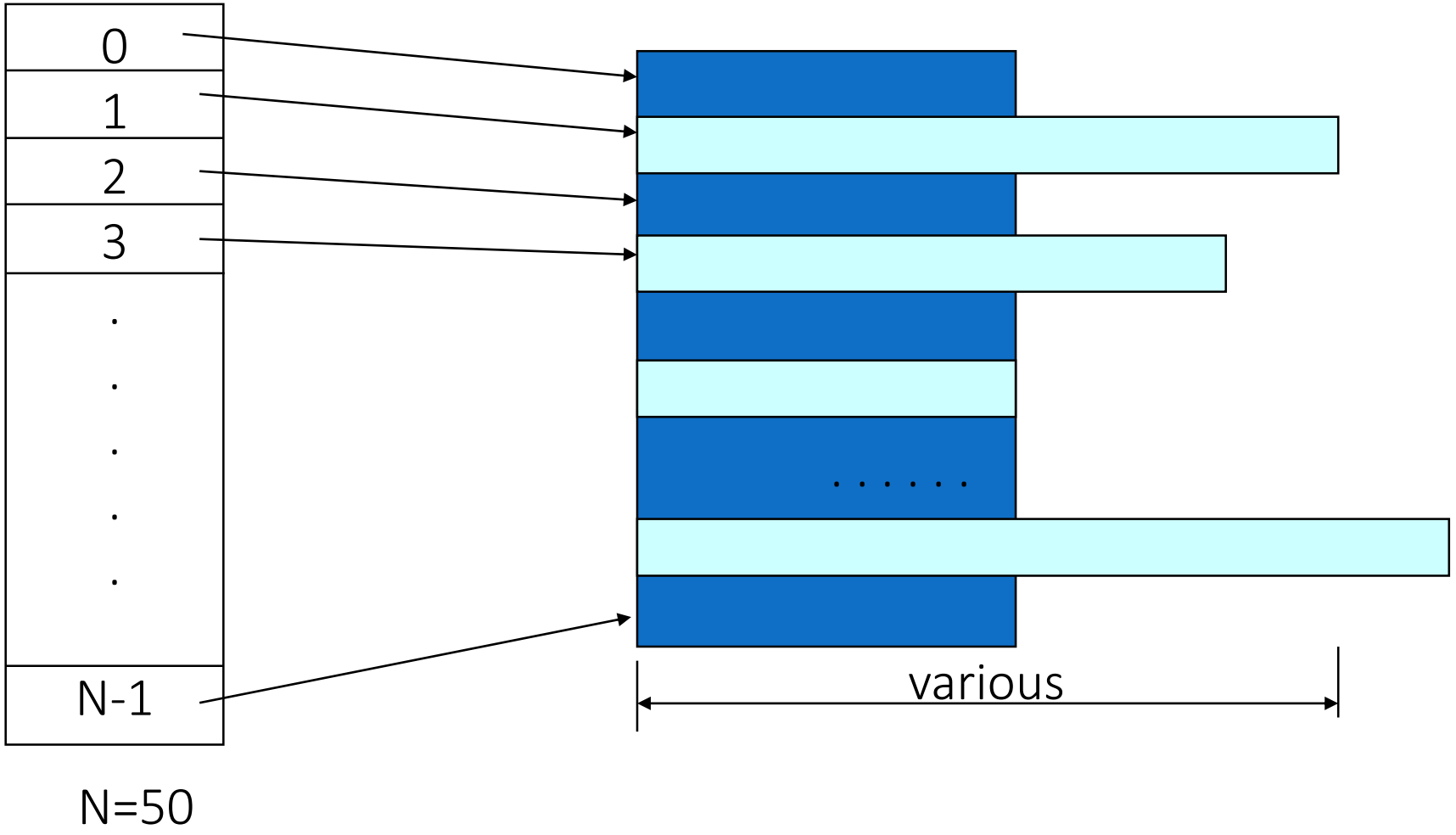
Array of Pointers



Accessing

```
for (unsigned i = 0; i<NumRows; i++)
    for (unsigned j = 0; j<RowSize; j++)
        samples[i][j]=0;
```

Ragged Array



Ragged Array

- Applications
  - To store an array of strings (e.g. our names)
  - Efficient storage for graph

Summary

Object-oriented programming

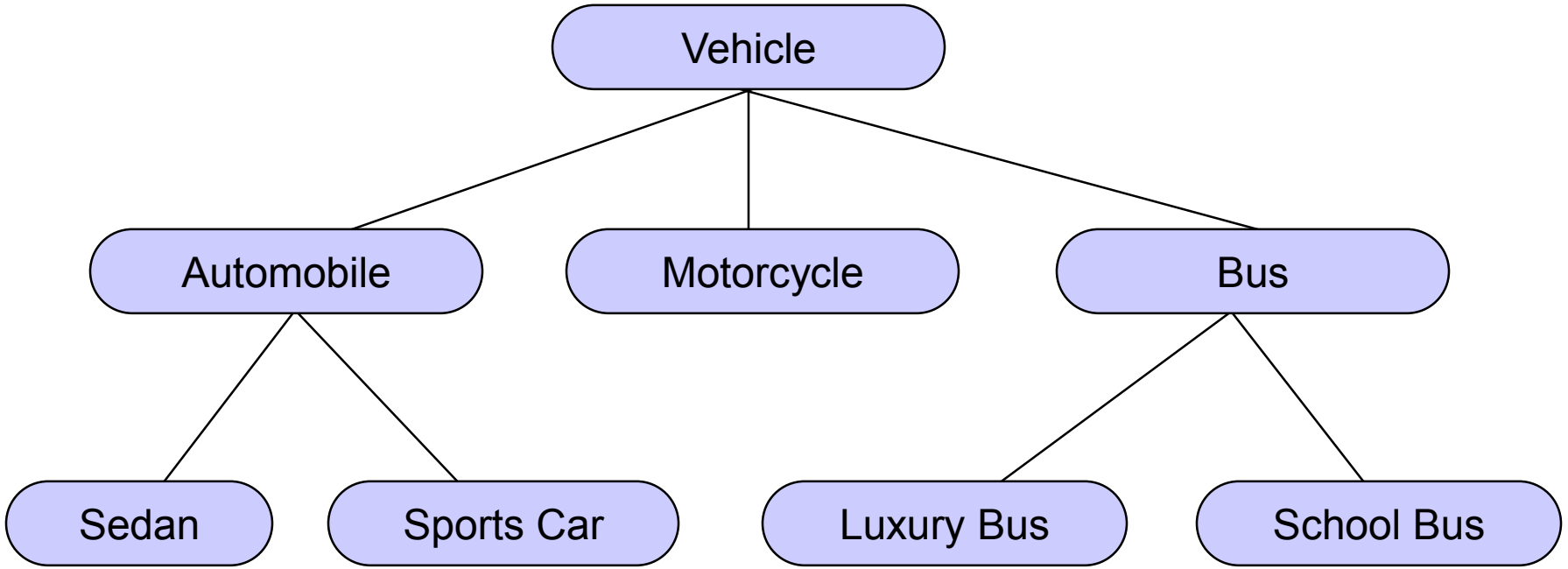
Lecture #3: Introduction to OOP

Solving a Programming Problem

- Analysis
- Design
- Coding
- Management
- Programming paradigms
- Programming languages

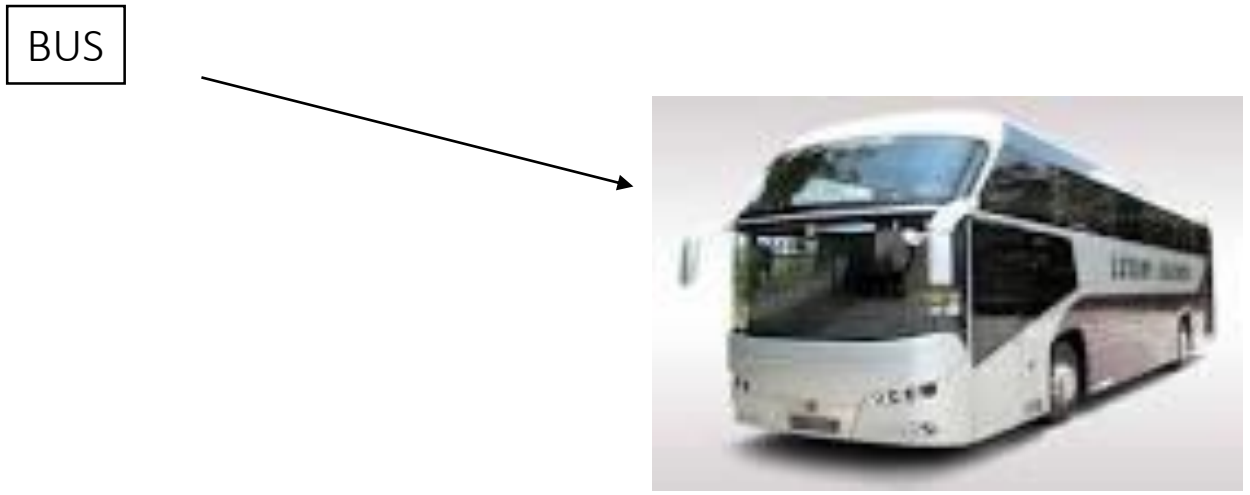
Thinking Methodology

- Induction
  - From specialization to generalization
  - E.g., to create the (abstract) concept “Bus” from different bus



Thinking Methodology

- Deduction (infer)
  - From generalization to specialization
  - E.g., from the concept “Bus” we have learned, we deduce that a vehicle is or is not a bus.



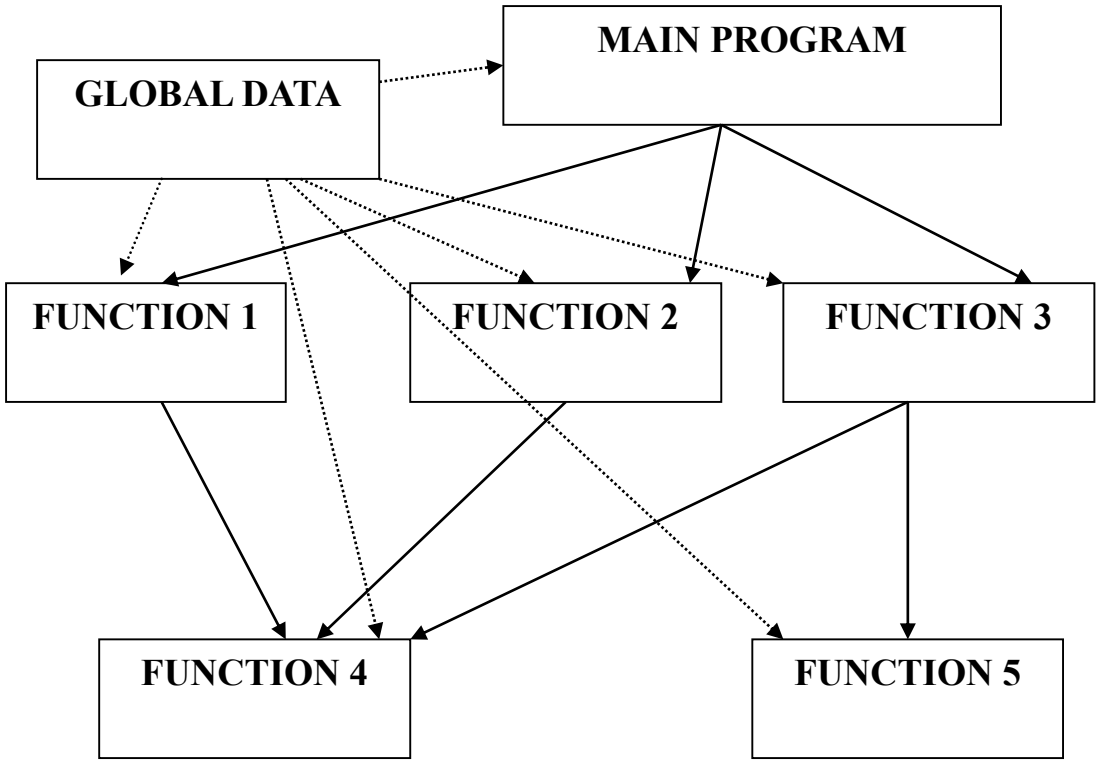
Concepts and Relationships

- A rectangle uses lines
- A circle is an ellipse
- A wheel is part of automobile
- A set creates its elements

Approaches in Software Development

- Top Down approach
  - A Single module will be split into several smaller modules
  - General to Specific
- Bottom Up approach
  - Lot of small modules will be grouped to form a single large module
  - Specific to General
- If the requirements are clear at the first instance we can go for Top down approach
- In circumstances where the requirements may keep on adding, we go for Bottom up approach

Structured programming



- Using function
- Function & program is divided into modules
- Every module has its own data and function which can be called by other modules.

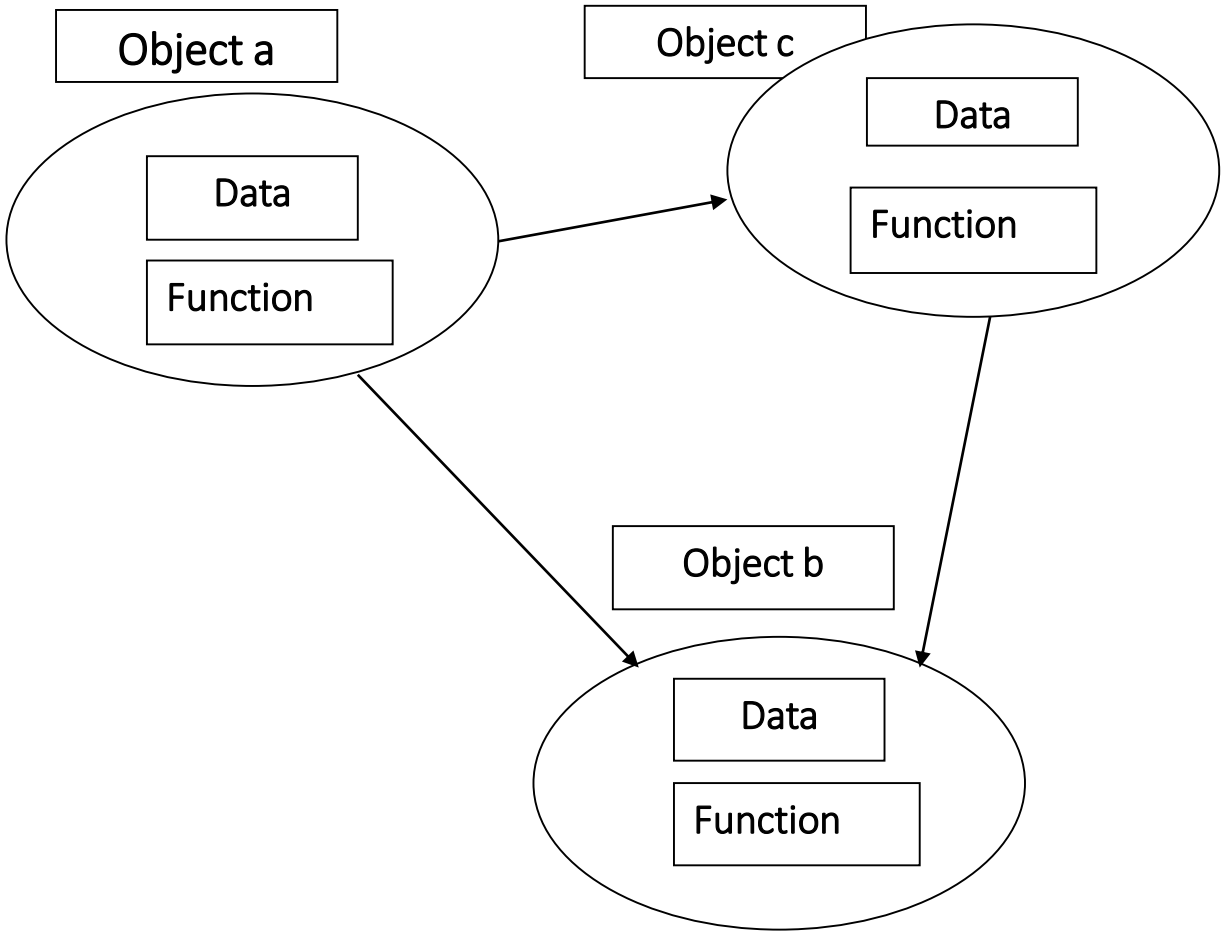
OOP Significant

- In real world scenario, for developing a software, we need requirements.
- But practically speaking, requirements keep on changing and request to add new features will keep on accumulating.
- So its better if we follow the Object Oriented Programming Strategy (bottom up approach)

Object-Orientation

- A thinking methodology
  - Everything is an object
  - Any system is composed of objects (a system is itself an object)
  - The evolution and development of a system is caused by the interactions of the objects inside/outside a system

Object-Orientation



- Objects have both data and methods
- Objects of the same class have the same data elements and methods
- Objects send and receive *messages* to invoke actions

Key idea in object-oriented:

- *The real world can be accurately described as a collection of objects that interact.*

Object-Orientation

- Everything is an object
  - A student, a professor
  - A desk, a chair, a classroom, a building
  - A university, a city, a country
  - The world, the universe
  - A subject such as CS, IS, Math, History, …

Object-Orientation

- Systems are **composed of** Objects
  - An education system
  - An economic system
  - An information system
  - A computer system

Object-Orientation

- The development of a system is caused by interactions
- E.g., VGU is defined by the interactions among:
  - students
  - professors
  - staff
  - board governance
  - state governance
  - …

Inside VGU

Outside VGU

Object-Orientation

- Object-Orientation is a design methodology (OOD)
  - Objects are the building blocks of a program
  - Objects represent real-world abstractions within an application

Design Methodology

- Object-orientation supports
  - Induction: objects → a class
    - *There are tools doing this automatically*
  - Deduction: a class → objects
    - *Usually done by programmers*

Design Methodology

- Object-orientation supports
  - Top-down: from a super-class to sub-classes
  - Bottom-up: from sub-classes to a super-class

Why OOP?

- An OOP language should support
  - Easy Representation of
    - *Real-world objects*
    - *Their States and Abilities*
  - Interaction with objects of same type
  - Relations with objects of other type
  - Polymorphism and Overloading
- Convenient type definitions

Why OOP?

- Save development time (and cost) by reusing code
  - once an object class is created it can be used in other applications
- Easier debugging
  - classes can be tested independently
  - reused objects have already been tested

Design principles of OOP

- Four main design principles of Object-Oriented Programming(OOP):
  - *Abstraction*
  - *Encapsulation*
  - *Polymorphism*
  - *Inheritance*

Abstraction

- Focus only on the important facts about the problem at hand to design, produce, and describe so that it can be easily used without knowing the details of how it works.

Analogy:

- When you drive a car, you don't have to know how the gasoline and air are mixed and ignited. Instead you only have to know how to use the controls.

Abstract Data Types (ADT)

- In ADTs: to define the interface to data abstraction without specifying implementation details
- ADT includes the following properties
  - It exports a type
  - It exports of set of operations
  - Axioms and preconditions define the application domain of the type

ADT in C++

- Showing only the essential features and hiding the unnecessary features
- The access modifiers in C++ or any OOP language, provides abstraction
- Functions also provide abstraction
- If a variable is declared as private, then other classes cannot access it
- The function name that is used in function call hides the implementation details from user. It shows only the outline of functionality that the function provides.

Encapsulation

- The process of bringing together the data and method of an object is called as encapsulation
- The data and method are given in the class definition
- Classes provides us with this feature - Encapsulation

Encapsulation

- Allows for modularity
- Controls access to data
- Separates implementation from interface
- Extends the built-in types

Example

- Complex numbers:
  - Real part
  - Imaginary part

  } Both parts are represented by real numbers

  - Operations: addition, substraction, multiplication or division to name a few
- To represent a complex number it is necessary to define the data structure to be used by its ADT.

Example (Complex numbers)

- One can think of at least two possibilities to do this:
  - Both parts are stored in a two-valued array where the first value indicates the real part and the second value the imaginary part of the complex number. If x denotes the real part and y the imaginary part, you could think of accessing them via array subscription: $x=c[0]$ and $y=c[1]$.
  - Both parts are stored in a two-valued record. If the element name of the real part is r and that of the imaginary part is i, x and y can be obtained with: $x=c.r$ and $y=c.i$.

⇒ In the first version, x equals $c[0]$. In the second version, x equals $c.r$. In both cases x equals "something"". It is this "something" which differs from the actual data structure used. But in both cases the performed operation "equal" has the same meaning to declare x to be equal to the real part of the complex number c: both cases archieve the same semantics.

⇒ the ADT definition says that for each access to the data structure there should be an operation defined => contradicted!

Example (Complex numbers)

- Once you have created an ADT for complex numbers, say *Complex*, you can use it in the same way like well-known data types such as integers.
  - *Complex* a;

Example (Complex numbers)

- If you think of more complex operations the impact of decoupling data structures from operations becomes even more clear.
  - For example the addition of two complex numbers requires you to perform an addition for each part. Consequently, you must access the value of each part which is different for each version.
  - By providing an operation “add” you can *encapsulate* these details from its actual use. In an application context you simply “*add two complex numbers*” regardless of how this functionality is actually archieved.

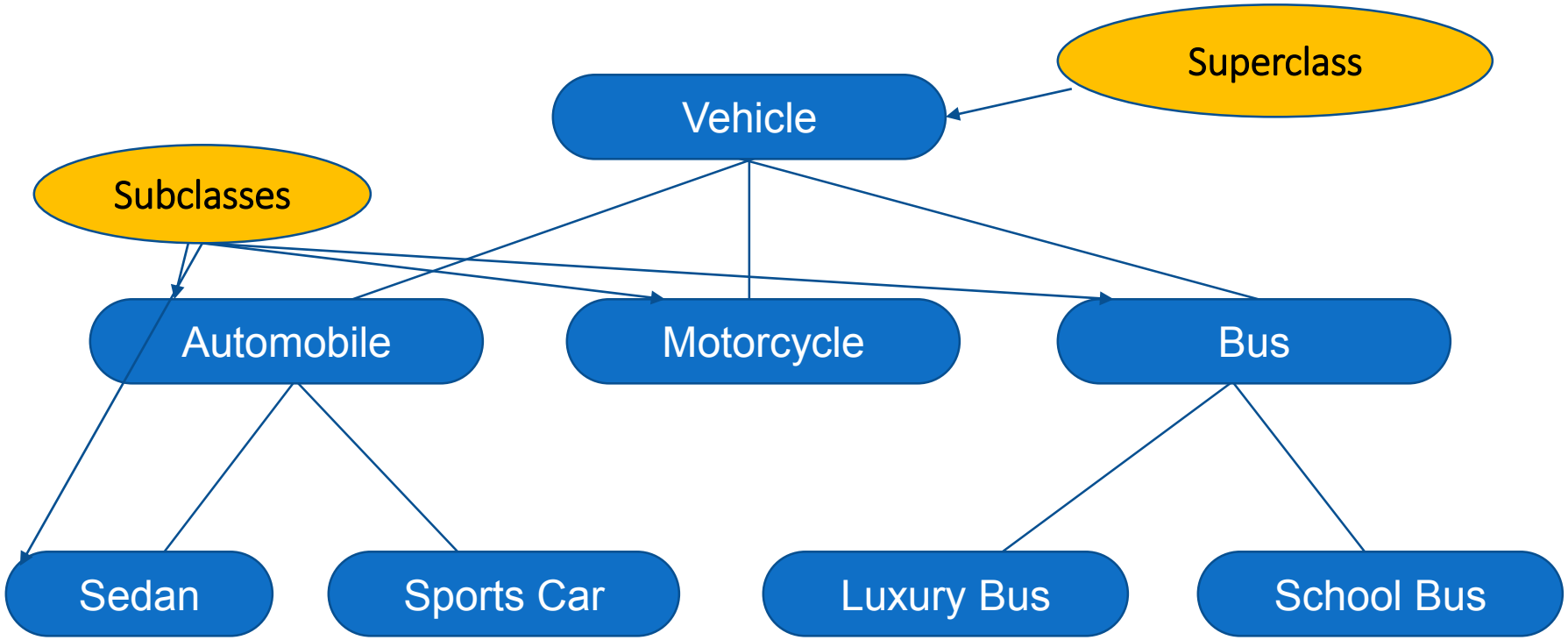
Inheritance

- Allows code reusability, by which a new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify some of those entities and add new entities.

Inheritance Terminology

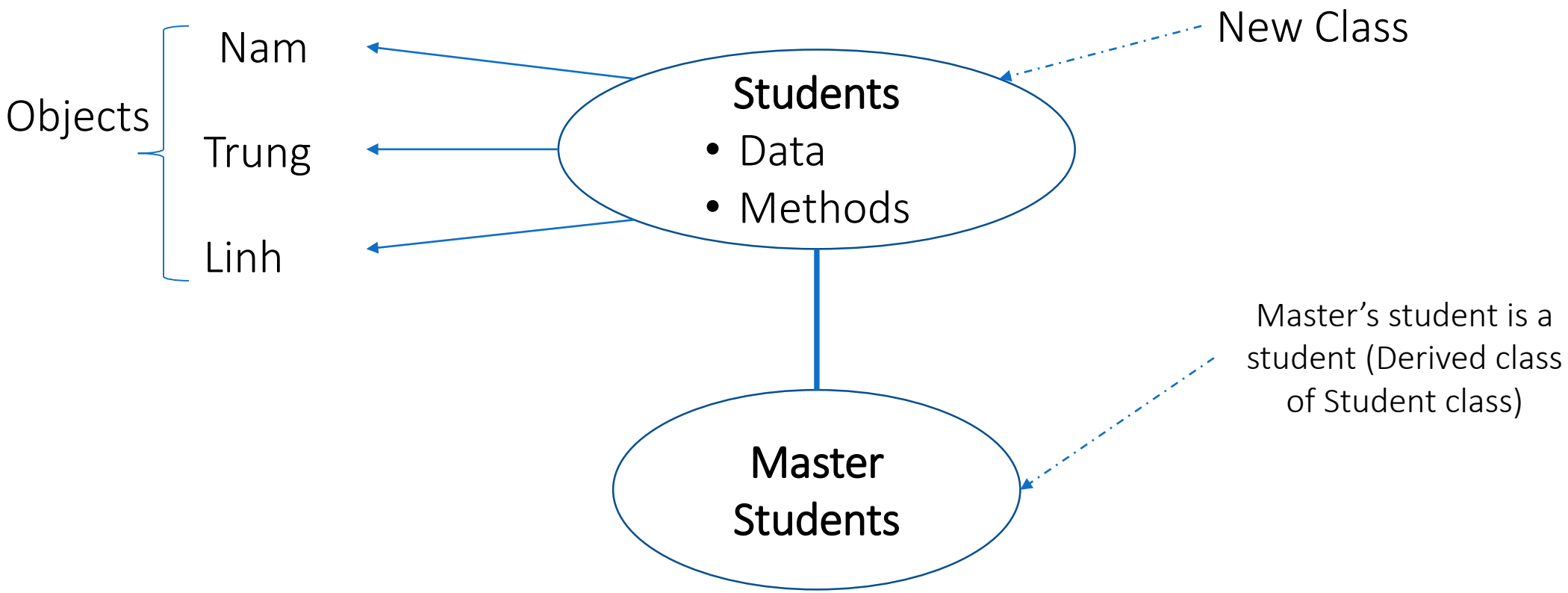
- **Class:** the abstract data type in Object-Oriented languages.
- **Object:** A class instance.
- **Derived class (subclass):** A class that is defined through inheritance from another class.
- **Parent class (superclass):** A class from which a new class is derived.
- **Methods:** the subprograms that define the operations on objects of a class.
- **Messages:** the calls to methods. Messages have two parts--a method name and the destination object.
- **Message Protocol (Message Interface):** the entire collection of an object's methods.

Inheritance



What properties does each vehicle inherit from the types of vehicles above it in the diagram?

Inheritance



Polymorphism

- The ability of objects to respond differently to the same message or function/method call
  - *Poly – Many*
  - *Morph – Form*
  - *Polymorphism => co exist in more than one form*
- C++ supports function overloading and operator overloading to implement polymorphism

Object-oriented programming

Lecture #5: Objects and Classes

Outline

- Objects
- Messages
- Classifications
- Classes
- Instances
- Meta-classes

Object-based Programming

- To view the world as a collection of autonomous, interacting objects
  - Examples: people, animals, plants, buildings, rooms, stairs, ...

Real-World Object Properties

- Active, autonomous behaviors
  - Not directly controlled by the outside
- Communicative
  - Can send/receive “messages” to/from other objects
- Collaborative
  - long-term relationships between objects will arise

Real-World Object Properties

- Nested
  - Complex objects have other objects as components (which in turn may have object components ...)
- Uniquely named/identifiable
- Creation/Destruction
  - May be created and destroyed

Examples

- A person
  - Thinks (most of the time)
  - Communicates with others
  - Collaborates and works with others
  - Has a name (and a NRIC)
  - Is born/then dies

Examples

- A computer
  - Autonomous: it can do many things
  - Communicative (with people, with other computers)
  - A serial No.
  - Built/Destroyed

Virtual Objects

- Objects that exist in programs only:
  - Virtual objects consist of the above features
  - Virtual objects are much more precise in their names, borders, interactions, etc.
  - Virtual objects are the basic components for your object-oriented programs

Examples of Virtual Objects

- A bank account
  - Has a balance; responds to messages for deposits, withdrawals, and balance queries
- Set
  - Elements can be added, deleted, and queried

Examples of Virtual Objects

- E-Ticket
  - Records that customer has paid for service in advance of flight(s)
- Payment
  - A transaction in which money is exchanged

Virtual Object Example: a Set

- aSet understands the messages
  - aSet.add(anElement)
  - aSet.remove(anElement)
  - aSet.contains(anElement)
  - aSet.size()
- Messages have both an effect (causing internal changes or induced messages) and a return value
- Users only need to know external view of aSet to use it

Sets in non-OOP Languages

- A Set itself is a (passive) structure
- Operations on a Set are active but stateless functions, i.e., they do not remember anything from one call to the next
  - procedure add(s:Set, e:Element)

Sets in non-OOP Languages

- There is no encapsulation
  - The programmer could directly manipulate the data, possibly putting the data in a compromised state

What are Objects?

- Booch:
  - Something that “has state, behavior, and identity”
- Martin/Odell
  - “Anything, real, or abstract, about which we store data and those methods (operations) that manipulate the data”
- Peter Müller:
  - *An **object** is an instance of a class. It can be uniquely identified by its **name** and it defines a **state** which is represented by the values of its attributes at a particular time.*

Definition: Message

- A message is a request to an object to invoke one of its methods. A message therefore contains
  - the name of the method and
  - the arguments of the method
- Consequently, invocation of a method is just a reaction caused by receipt of a message. This is only possible if the method is actually known to the object

The Interface

- The operation “name” list open to other objects (other objects can send messages)
- If an object has a function but does not show it to the public, this is useless for the public (called private function)
- In C++, only public functions (belong to the interface) can be called by other objects (outsiders)

Properties of Objects

- Encapsulation (Data & Operations)
- Information Hiding
- Data Abstraction (with classes)
- Abstract Data Type (with classes)

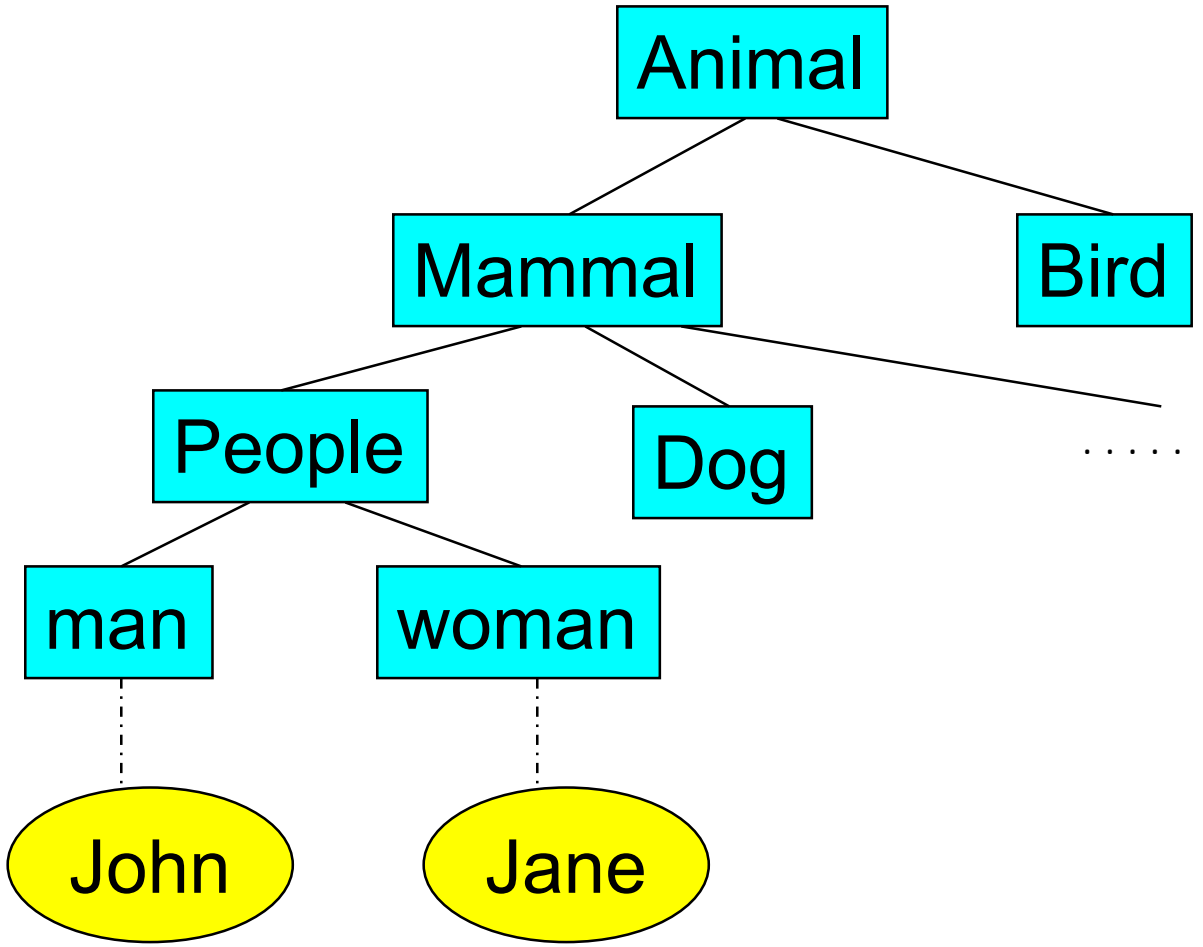
Advantages of Objects

- Same as Abstract Data Types
  - Information hiding
  - Data abstraction
  - Procedural abstraction
- Inheritance provides further data abstraction
  - Easier and less error-prone product development
  - Easier maintenance

Classification

- Classification is not unique to object-oriented systems. On the contrary, classification is applied in many other domains.

Classification



Class

- A class is either a classification or an abstraction of objects
- Class is not only a concept, but also a practical mechanism of programing
- OOP is actually programming with classes

Class

- “Class -- a definition of **an implementation** (methods and data structures) shared by a group objects.”
- “Class -- **a template** from which objects may be created. It contains a definition of the state descriptors and methods for the objects.”

Class

- Classes serve two distinct purposes
  - Factories that create new objects (code plus specifications)
  - Classification of objects (specification only: e.g. sizable class specifies any object with a size method)

Intentional Notion of Class

- New objects are instances of a class. Their states and behaviors are determined by the class definition
- This is so-called intentional notion of a class
- The intentional notion determines the structure of instances of that class

Extensional Notion of Class

- A class consists of object-warehouse and object-factory
- Object warehouse means that a class implicitly maintains a class extent
- A class extent consists of all instances of the class
- Object-factory means that there exists a constructor for each class, to generate new instances of that class

Understanding Classes

- A class provides a definition of the structure of instances of that class
- A class defines the names and attributes (state) and methods (behavior) of an object belonging to the class

Examples of Class

```
class Integer {
Ds:
    int I
Ops
    setValue(int n)
    Integer addValue(Integer j)
}
```

Examples of Class

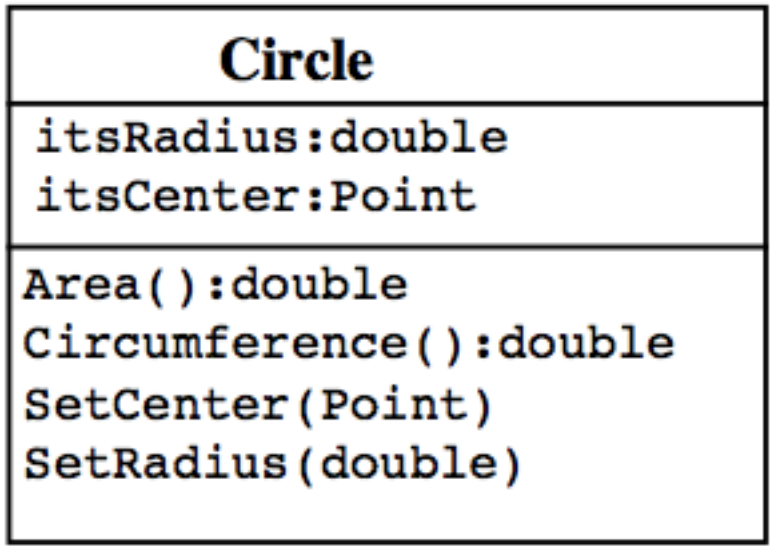
```
class Horse {
Ds:
  Age
  Weight
  Color
Ops:
  Drag
  Run
  Ride
}
```

Relationships among Classes

- Link (use-a)
  - A link relationship exists between two classes that need to communicate (an instance of one class sends a message to an instance of another class)
- Composition (has-a)
  - A composition relationship exists when a class contains data members that are also other class objects

UML: Representing Class

- UML stands for Unified Modeling Language



Instantiation

- The mechanism of creating new objects from a class definition is called instantiation
- Every class has such a mechanism

Instantiation

- Static instantiation and Dynamic instantiation
  - (static means) at compile time (by compiler);
  - (dynamic means) at run time
    - *Dynamic instantiation requires run-time support for allocation and de-allocation of memory*

Making Objects from Class Templates

- aSet = new Set()

/* Set is the class; this makes an object */

- anotherSet = new Set()

/* Same factory, but different contents */

- Each time an object is created, it is new and unique

Object

- Object ::= <Oid, Cid, Body>
  - Oid is the identification of the object;
  - Cid is the identification (or name) of the class of this object;
  - Body is the actual space for memory
- Note: the operations of the object are implemented in the class

Example of Class in C++

```
class Student {
private:
    unsigned numCoursesRequired;
    unsigned age;
public:
    Student(unsigned nCourses);
    void attendLecture();
    void selfStudy();
    void play();
};
```

The Relations between Class and Object

- An object is an instance of a class
- What if we treat class as an object???
- Then, what is the class of a class?
  - A meta-class

Meta-class

- A meta-class is the class of a class
- The attributes of a meta-class can be used to described a class (e.g. # of instances of a class)

Meta-class

- Explicit support of meta-classes means that objects, classes and meta-classes are treated uniformly
- Classes can be created at run-time by explicitly sending a message to a special meta-class
- Not all OOP languages support meta-class

Object-oriented programming

Lecture #6: Objects and Classes in C++

Outline

- Class/Object Definition
- Member Access
- Member Function
- Object Copy: Deep vs. Shallow

Classes and Their Instances

- *class class_name{ Member_List }; //Class*

  Member_List ::= MemberVariables | MemberFunctions

- *class_name identifier; //Object*

Class Definition

```
class Point {
  private:
    int x,y; //coordinates
  public:
    Point(int xVal = 0, int yVal = 0) { x = xVal; y = yVal; }
    int getX() { return x; }
    int getY() { return y; }
};
```

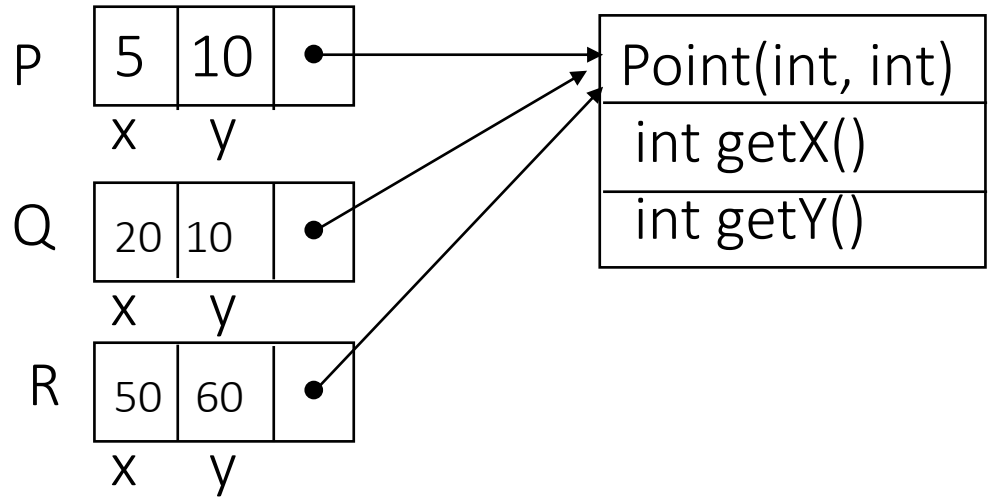
Member Function

- A function that has its definition or its prototype within the class definition
- (Many) used interchangeably with **Methods**

Class Members

- A class object contains a copy of the data defined in the class
  - The functions/methods are shared
  - The data are not

```
Point P(5,10);
Point Q(20,10);
Point R(50,60);
```



Point Objects

//Lec6_ex1-Point.cpp

Controlling Member Access

```
class class_name {
    public:
     //public members
    protected:
     //protected members
     private:
     //private members
};
```

Access Rules

Type of Member	Member of the Same Class	Friend	Member of a Derived Class	Non-Member Function
Private (Default)	X	X		
Protected	X	X	X	
Public	X	X	X	X

Access Rules

- For public member:
  - You can get to the member using the “.” (dot) operator
  - p.getX();
- For the other two member types:
  - NO!

Point (ex2)

```
class Point {
  private:
    int x;
  public:
    int y;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int xVal) { x = xVal; }
    void setY(int yVal) { y = yVal; }
};
```

Point (ex2)

```
Point p;
p.setX(300);
p.setY(500);
cout << “x = ” << p.getX() << endl;
cout << “y = ” << p.y << endl;
//Lec6_ex2-Point.cpp
```

=> What the printed values?

Example about friend class

- Lec6_ex3_FriendClass.cpp

```
#include <iostream>
class A {
    private:
        int a;
    public:
        A() { a=10; }
        void seta(int value);
    friend class B;     // Friend Class
};
void A::seta(int value) { a=value; }
class B {
    private:
        int b;
    public:
        void showA(A& x) { std::cout << "A::a=" << x.a;   }
};
```

```
int main() {
    A a;    B b;
    a.seta(15);
    b.showA(a);
    return 0;
}
```

Inline Functions

- In-line: functions are defined within the body of the class definition.
- Out-of-line: functions are **declared** within the body of the class definition and **defined** outside
- *inline* keyword: Used to define an inline function outside the class definition.

```
class Point {
 ...
 public:
    void setX(int valX) {x=valX;}; // set x
    void setY(int valY);  //set y
    void delta(int, int); //adjust the coordinators
};
 void Point::setY(int valY) { y=valY; }
 inline void Point::delta(int dx, int dy){ x +=dx; y+=dy; }
```

Omitting the name is allowed

In-line

Out-of-line

Also In-line

Constant Member Function

- To guarantees that the state of the current class object is not modified by the function

```
class Point {
 public:
 …
    int getX() const { return x; }
};
```

- Another example: *Lec6_ex4_ConstantMember_Function.cpp*

Constructors

- Called to create and object
- Declared with the name: *classname(….);*
- **Default constructors:** no parameters
- **Alternate constructors:** with parameters

Constructors

- **Overloading Constructors**: with different parameters
- Point() and Point(int, int) are overloaded constructors

Constructors

- **Copy Constructor:** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously
- If a copy constructor is not defined in a class, the compiler itself defines one

Constructors (E.g.)

- Point p; //default
- Point a(p), b = p; //copy
- Point c(20,30); //alternate
- Point figure[3]; //default
- Point figure[2] = {p,c}; //copy

Pointers to Class Objects

- Student *p;
- p = new Student;
- if (!p) //could not create *Student*
- Point * figure = new Point[10];

// call Point() 10 times

Destructors

- Called when an object is to be deleted
- Declared with the name: *~classname();*

(no parameters)

- Example:
  - ~Point() { cout << “Destructor called.”; }

Deep Copy Operation

- Deep copy -- copying the contents of an object

```
char name[] = “John Smith”;
char * cp = new char[30];
strcpy(cp,name);
```

- Shallow copy -- copying the pointer of an object

```
char name[] = “John Smith”;
char * cp = name;
```

Student Copy

```
#include <string>
class Course {
private:
    string name;
public:
    Course() { }
    Course( const string & cname );
};
```

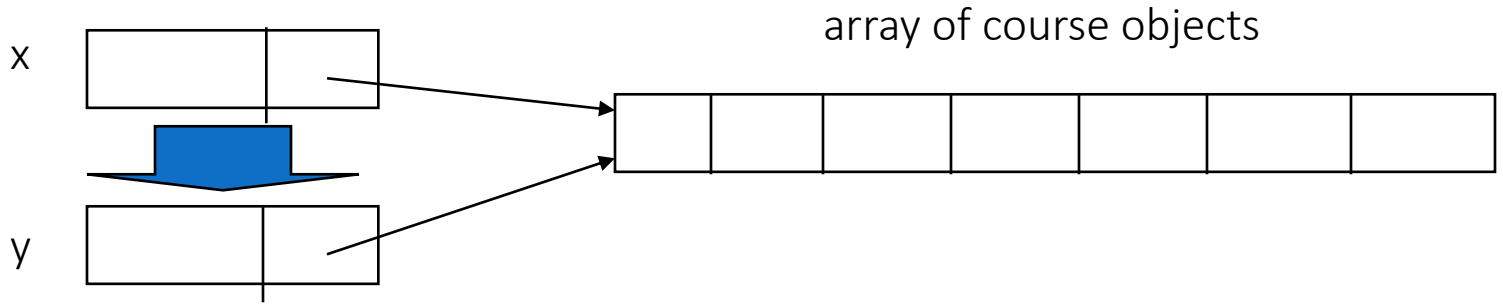
Student Copy

```
class Student {
private:
  Course * coursesTaken;
  unsigned numCourses;
public:
  Student( unsigned nCourses );
  ~Student();
};
```

```
Student::Student( unsigned nCourses ) {
  coursesTaken = new Course[nCourses];
  numCourses = nCourses;
}
Student::~Student() {
  delete [] coursesTaken;
}
```

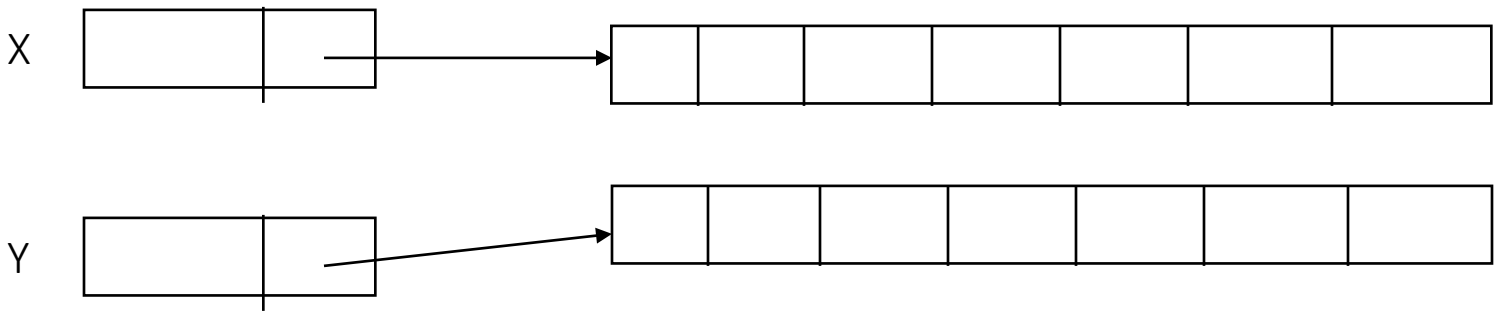
Student Copy

```
int nCourses = 7;
Student x(nCourses);
Student y(x);
```



Student Copy

- If you'd like the following:



Student Copy

```
Student( const Student & s) {
    numCourses = s.getNumCourses();
    courseTaken = new Courses[numCourses];
    for(int i = 0; i < numCourses; i++)
        courseTaken[i] = s.getCourse(i);
}
```

Composite Classes

- A class which contain data members that are objects of other classes
- When a class contains *instances*, *references*, or *pointers* to other classes, we call it a composite class

Composite Classes

- E.g.: A Circle contains a Point; a Figure contains an array of Points

Pre-defined Order

- The constructors for all member objects are executed in the order in which they appear in the class definition. All member constructors are executed before the body of the enclosed class constructor executes.
- Destructors are called in the reverse order of constructors. Therefore, the body of the enclosed class destructor is executed before the destructors of its member objects.

Exercise

- To write a program to test this order.

Exercise

Object-oriented programming

Lecture #7: Inheritance

Additions to Last Lecture

- Constructors: if no constructor is defined, the compiler will create a default one for us (same for destructor)
- Access Modifiers:
  - The access modifiers work on **class level**, and not on **object level**
  - That is, two objects of the same class can access each others private members

Example (Date)

```
class Date {
   private:
     int month, day, year;
   public:
     Date(int mo, int dy, int yr) {
                  month = mo; day = dy; year = yr; }
     Date( Date & d) {
        month = d.month; day = d.day; year = d.year; //WHY
     }
};
```

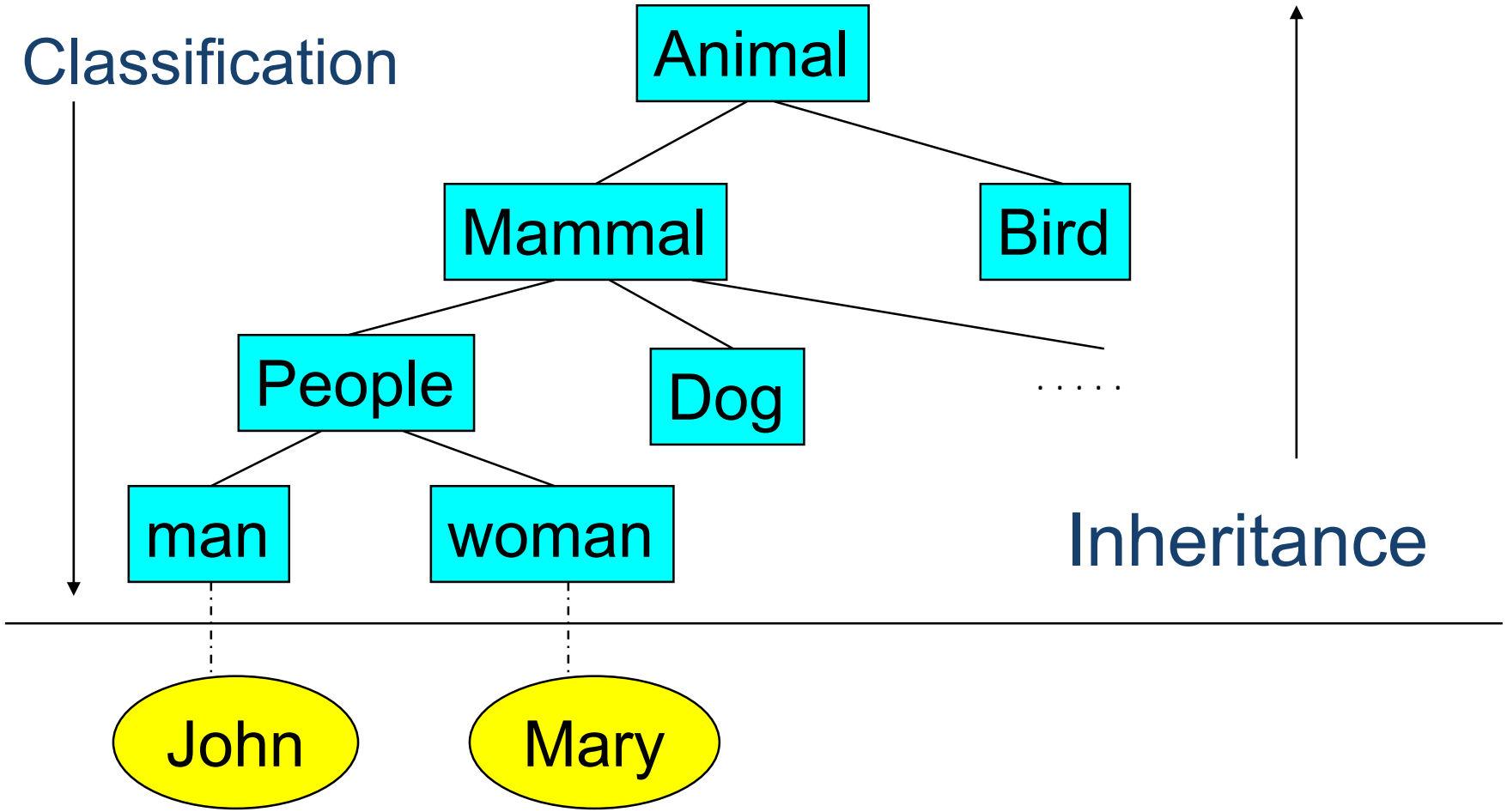
WHY???

- The private modifier enforces Encapsulation principle.
- The idea is that 'outer world' should not make changes to Person internal processes because Person implementation may change over time (and you would have to change the whole outer world to fix the differences in implementation - which is nearly to impossible).
- When instance of Person accesses internals of other Person instance - you can be sure that both instances always know the details of implementation of Person. If the logic of internal to Person processes is changed - all you have to do is change the code of Person.

Outline

- Classification
- Sharing
- Inheritance
- Subclass
- The Categories of Inheritance
- Abstract Class

Classification/Inheritance



Classification

- Classification arises from the universal need to describe uniformities of collections of instances
- Classification is a basic idea for understanding the concept inheritance

Classification/Inheritance

- Commonality
  - The base class captures the common information (attributes) and features (operations) of the derived classes
- Customization
  - An existing class is used to create a customized version of the class
- Common Design Interface
  - A base class may define the design requirements for its derived classes by specifying a set of *member* functions that are required to be provided by each of its derived classes

Sharing

- Sharing is ubiquitous in real-world, thus it is important in object-orientation
- Inheritance is a technique that promotes sharing
- Inheritance means that new classes can be derived from existing classes
- A subclass (derived class) inherits the attributes and the operations of a superclass (base class); but a subclass can define additional operations and attributes

Sharing

- Can be seen a a specialization mechanism
  - Instances of a subclass are specializations (additional state and behavior) of the instances of a superclass
- Can also be seen as generalization mechanism
  - Instances of a superclass generalizes the instances of a subclass

Sharing

- Is also the basic idea of inheritance
- The ability of one class to share the behavior of another class without explicit redefinition
- An approach which allows classes to be created based on an old class

Three Dimensions of Sharing

- Static or dynamic sharing:
  - At times, sharing patterns have to be fixed. This can be done at object creation (static) or when an object receives a message (dynamic)
- Implicit or explicit sharing:
  - The programmer directs the patterns of sharing (explicit) or the system does it automatically (implicit)
- Per object or per group sharing:
  - Behaviors can be specified for an entire group of objects or can be attached to a single object

Purposes of Inheritance

1. Reusing existing design reduces software development and testing costs
2. Modular Design: A new class inherits the data and functionalities of an existing class

What is Inheritance?

- **Inheritance** is a mechanism for expressing similarity
- **Inheritance** is a natural property of classification
- **Inheritance** is a mechanism that allows (a class) A to inherits properties of (a class) B

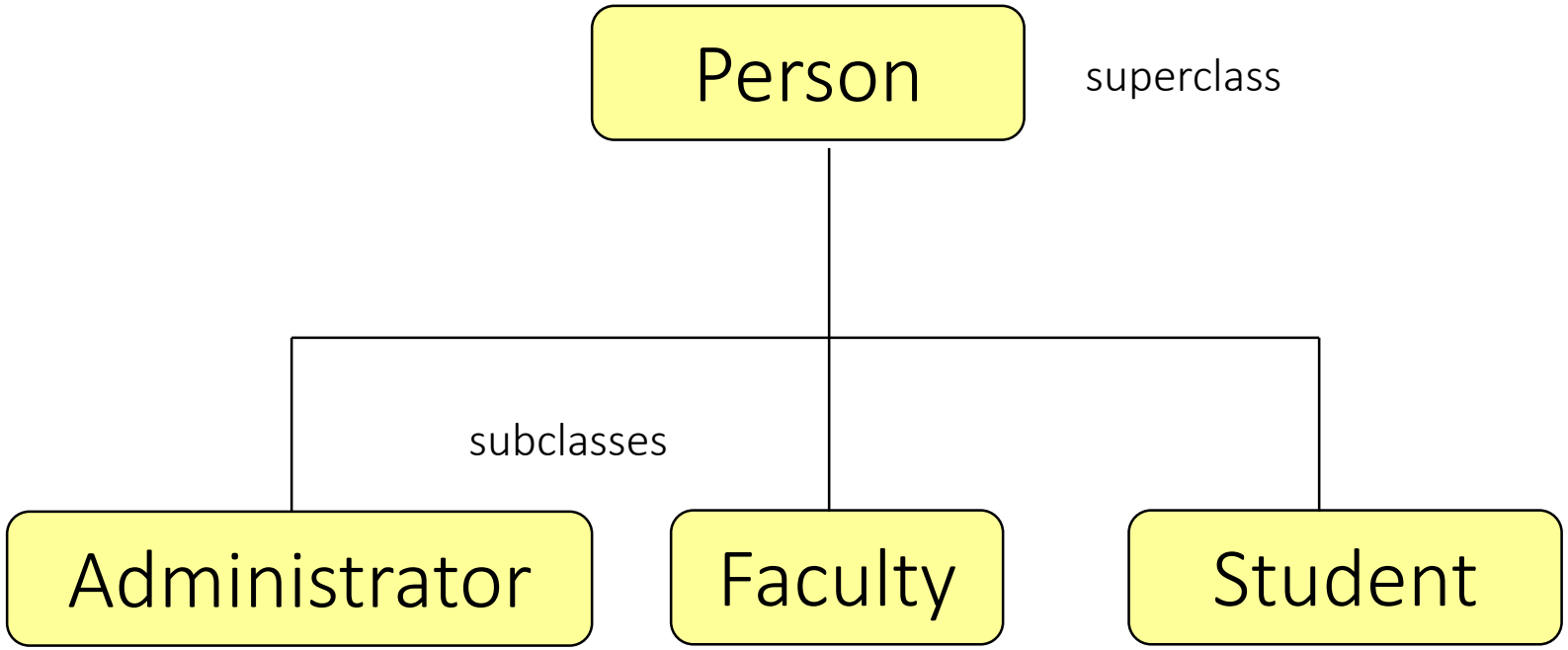
Inheritance in OOP

- Assume “A inherits from B”. Then objects of class A have access to attributes and methods of class B without the need to redefine them

Superclass/Subclass

- If class A inherits from class B, then B is called **superclass** of A. A is called **subclass** of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share similar (signature) behavior as objects of the superclass

Example


Person
superclass
subclasses
Administrator
Faculty
Student

Subclasses

- Subclasses refer to not only inheritance of specification but also inheritance of implementation and so can be viewed as reusability mechanism

Important Aspects of Subclasses

- Modifiability
  - The degree of modifiability determines how **attributes** and **methods** inherited from a superclass can be modified in a subclass. In this context, an approach of distinguishing modifiability of the object states (attributes) and the object behaviors (operations) are used

Attributes

1. No redefinition: modification is not allowed
2. Arbitrary redefinition: redefinition without constrained is allowed
3. Constrained redefinition: the domain of attributes is constrained
4. Hidden redefinition: definitions of attributes are hidden in subclass to avoid conflicts

Operations

1. Arbitrary redefinition: all changes to operations are allowed
2. Constrained redefinition: Parts of the signature of methods in the subclass have to be subtypes of the parts of the methods of the superclass

- This is important in overriding and overloading of methods

Naming Conflicts

- Conflicts between a superclass and a subclass
  - When attributes or methods defined in a subclass has the same names as attributes or methods defined in the superclass
  - This is resolved by **overriding**

Categories of Inheritance

- Whole/Partial Inheritance
  - Whole Inheritance is when a class inherits all the properties and operations from its superclass
  - Partial Inheritance is when only some properties are inherited while others are suppressed

Categories of Inheritance

- Default Inheritance: inherited properties and constraints can be modified
- Strict inheritance: doesn't allow the user to modify inherited properties or constraints

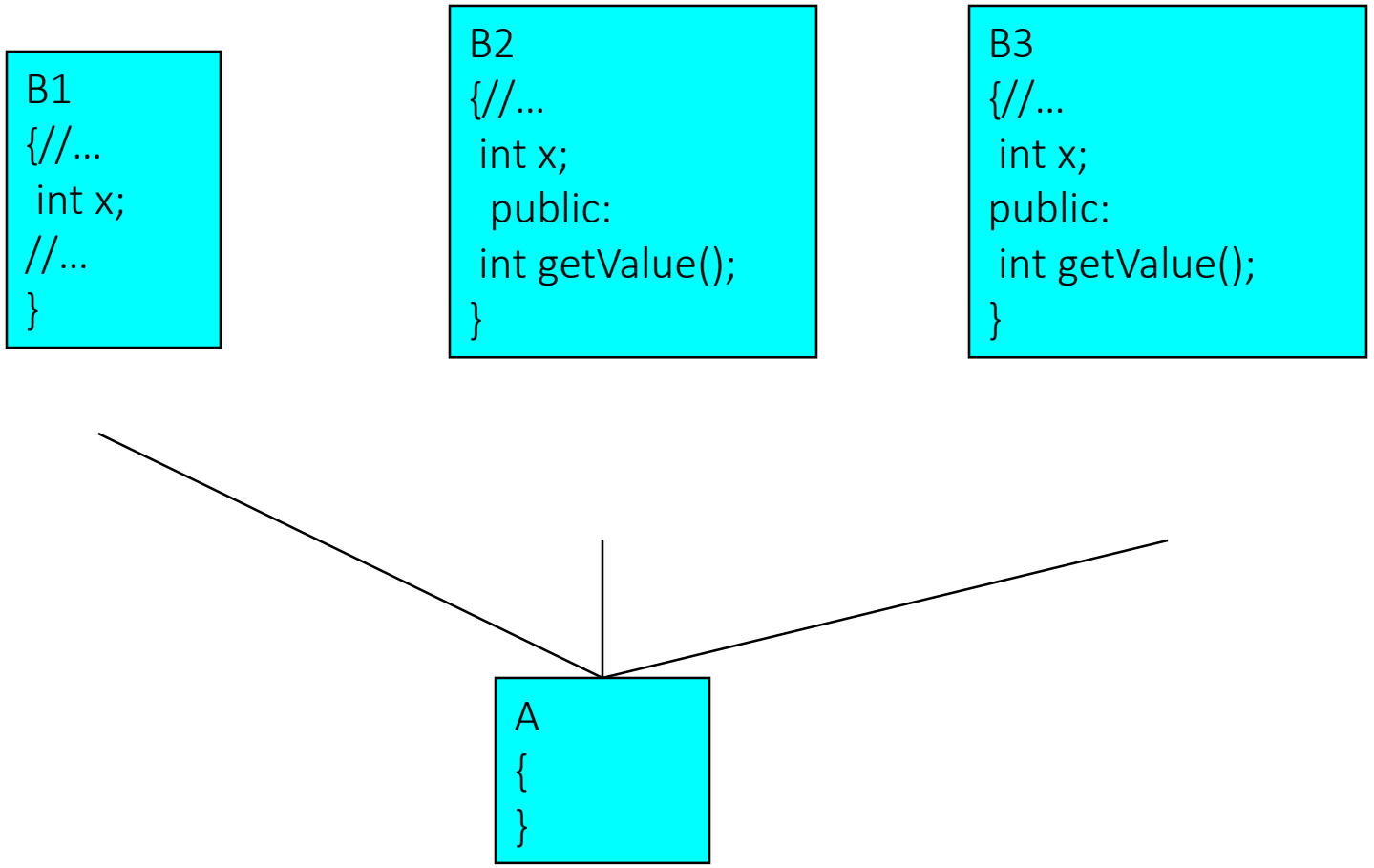
Categories of Inheritance

- Single (simple) inheritance:
  - A class can inherit from one superclass only. This means the inheritance hierarchy forms a tree
- Multiple inheritance
  - A class can have more than one superclass

Multiple Inheritance

- **Naming Conflicts:** if attributes or methods defined in one superclass have the same name as attributes or methods defined in another superclass

Naming Conflicts



Conflict Resolution

1. Using the order of the superclass

- If there are more attributes or methods with the same name in different superclasses, the ones occurring in the first class in the list of superclasses are inherited (done by the compiler)
- If the order is from left to right, then x means x of B1, and getValue() means the getValue() of B2

Conflict Resolution

2. Determined by the user

- The user can inherit conflicting attributes or methods, but has to explicitly rename conflicting attributes or methods in the subclass (done by the user)
- B1::x; or B2::x; or B3::x;
- B2::getValue(); or B3::getValue();

Abstract Class

- A class without any instance
- Mammal is an abstract class: only for classification and inheritance. Never make objects of the class.

Abstract Class

- It is also possible to have a classification which does not have any (executable) code behind it

Abstract Class

- Class A is called an **abstract class** if it is used only as a superclass for other classes. Class A only specifies properties. It is not used to create objects. Derived classes must define the properties of A.

Reuse

- Many take inheritance as a reusing tool
- In this case, we can take inheritance as a code reusing tool

Inheritance/Classification

- Classification implies inheritance (whole and simple inheritance)
- Inheritance may effect clarity of classification (for multiple inheritance and for part inheritance)

Declaring Derived Classes

```
class derived_class_name :
access_specifier base_class_name
{ /*...*/ };
```

Object-oriented programming

Lecture #8: Inheritance in C++

Outline

- Base and Derived Classes
  - Single Inheritance
  - Declaration of Derived Classes
  - Order of Constructor and Destructor Execution
  - Inherited Member Accessibility
- Multiple Inheritance
- Virtual Base Classes

Base and Derived Classes

- A **base class** is a previously defined class that is used to define new classes (now)
- A derived class inherits all the data and function members of a base class (in addition to its explicitly declared members)

Single Inheritance

- Implement an “is-a” relationship
- The derived class only has one base class

Declaring Derived Classes

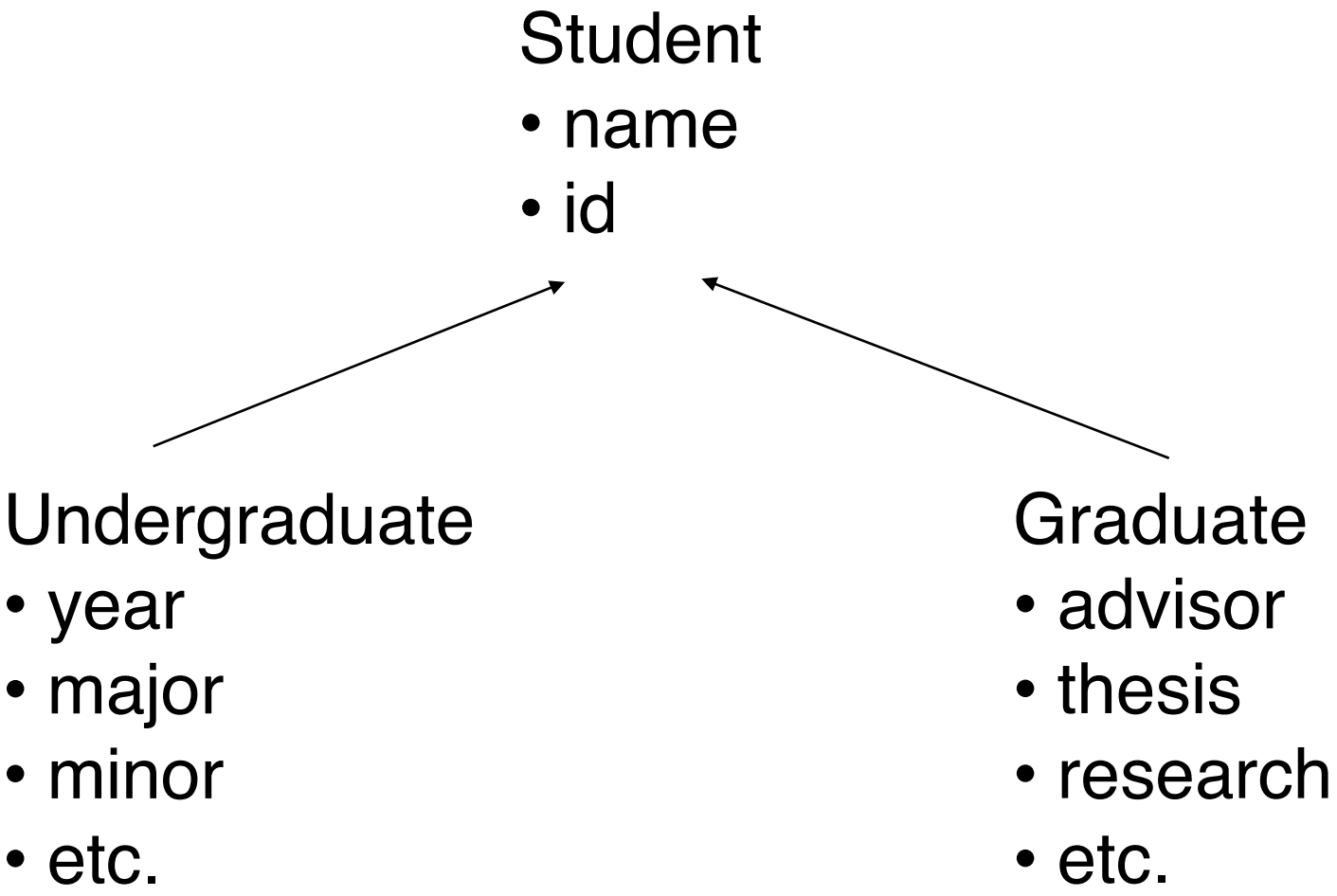
class class_name : access_specifier$_{opt}$ base_class {

    Member_list

};

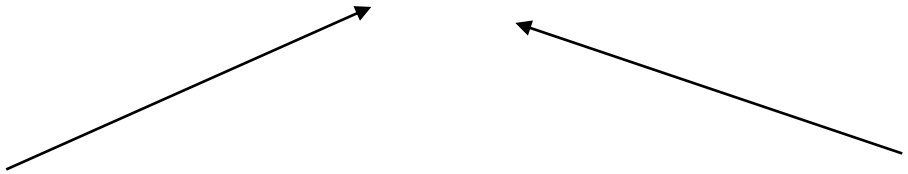
access_specifier ::= public|protected|private (default)

Example 1


Student
• name
• id
Undergraduate
• year
• major
• minor
• etc.
Graduate
• advisor
• thesis
• research
• etc.

Example 2

Publication:
- publisher
- date



Magazine:
- circulation
- # of issues per year

Book:
- ISBN
- author

Example 2

// Lec8_ex2-Publication.cpp

Different Views of an Employee

- Full-time or part-time
- Permanent or Temporary

(note different behavior, e.g. Social Insurance)

- How to define its base class?
- How to define derived classes based on this base class

Order of Constructor and Destructor Execution

- Base class constructors are always executed first
- Destructors are executed in exactly **the reverse order** of constructors

Example 3

// Lec8_ex2-Employee.cpp

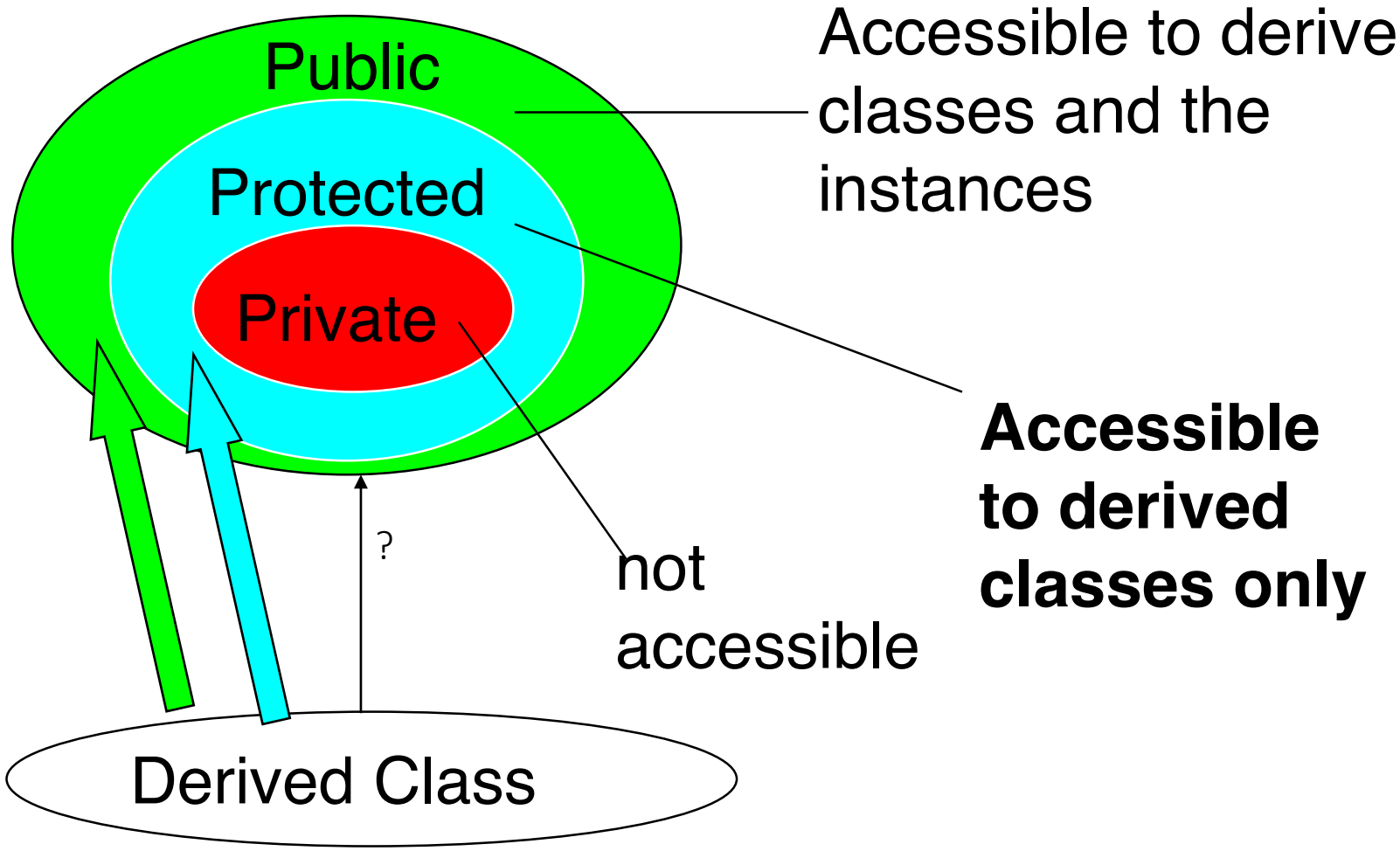
Overriding

- A function in the derived class with the same function name will override the function in the base class
- We can still retrieve the overridden functions by using the scope resolution operator “::”

Types of Class Members

- private
- protected
- public

Types of Class Members



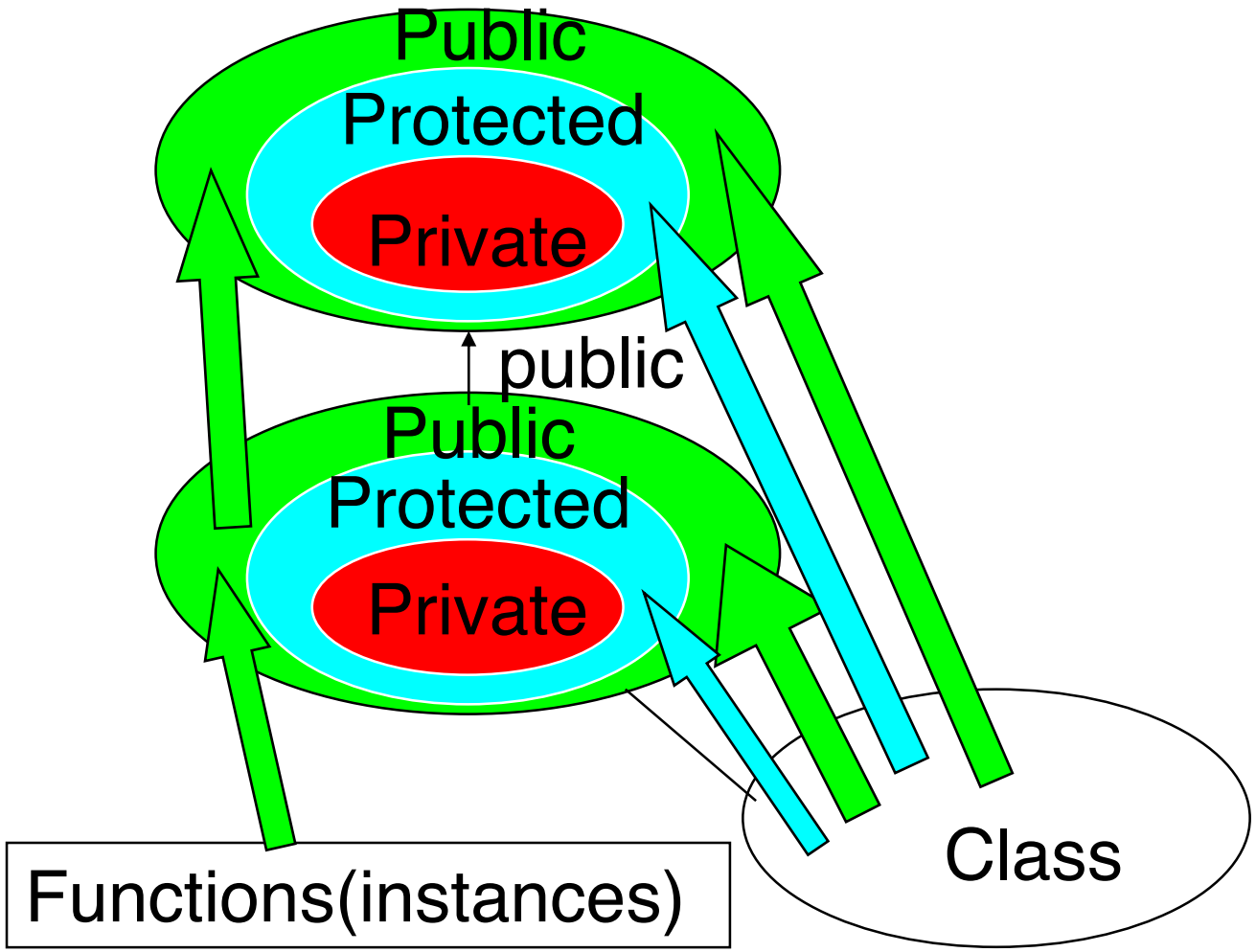
Types of Inheritance

- public
- private
- protected

public Inheritance

- public and protected members of the base class become respectively public and protected members of the derived class

public Inheritance



Example: Lec8_ex3-public-inher.cpp

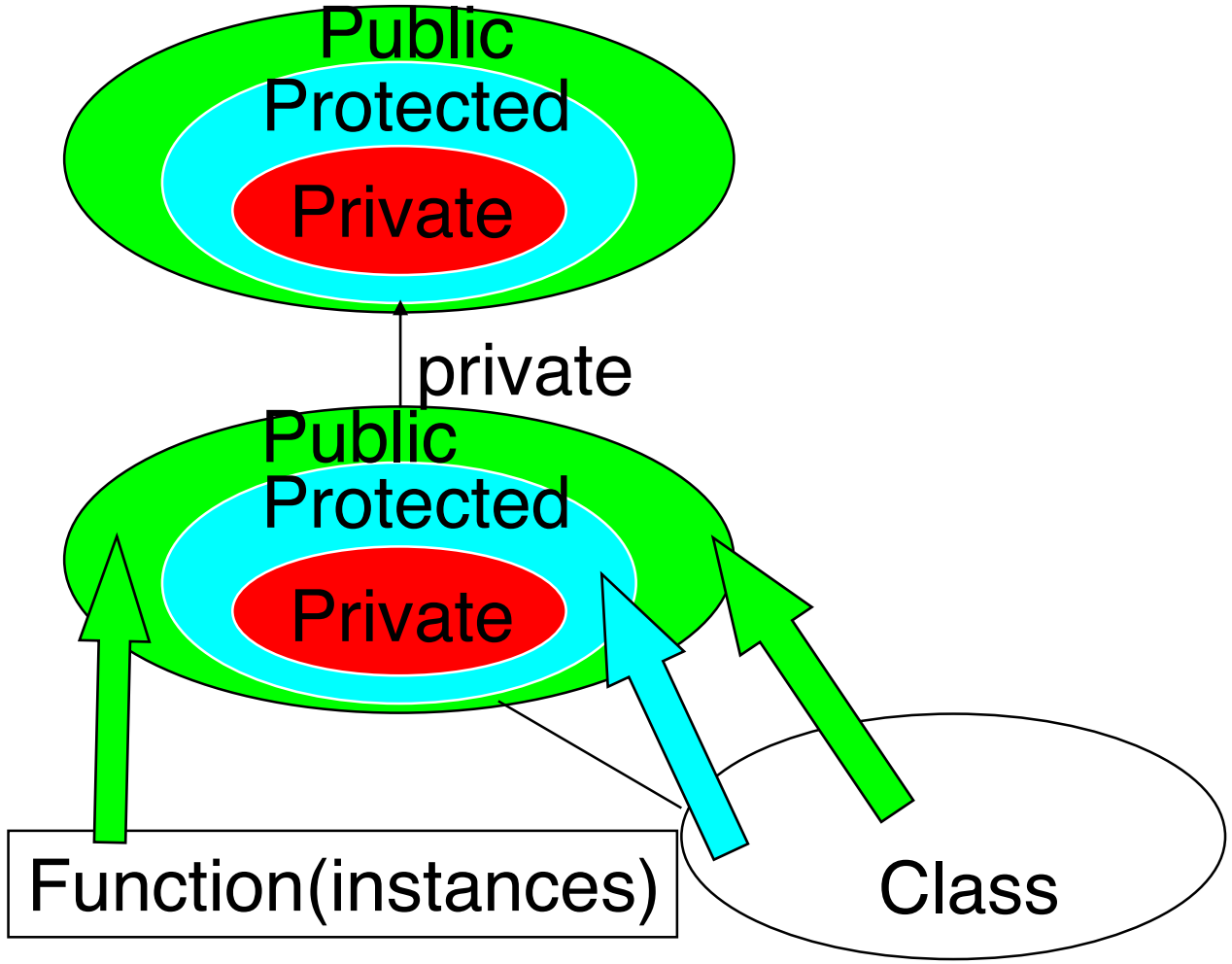
```
int main() {
    ..........
    p = aStack.removeFirst();
    ..............
return 0;
}
```

```
****** Creating an Item

Item::Item50
******* Creating a Stack

**** Stack::push() 50
Item::setPtr
List::putFirst() 50
**** Stack::pop()
List::removeFirst()
aStack.pop() 50

Item::setItem100
**** Stack::push() 100
Item::setPtr
List::putFirst() 100
Calling removeFirst() from aStack
List::removeFirst()
aStack.removeFirst() 100
```

private Inheritance

- public and protected members of the base class become private members of the derived class

private Inheritance



Example: Lec8_ex4-private-inher.cpp

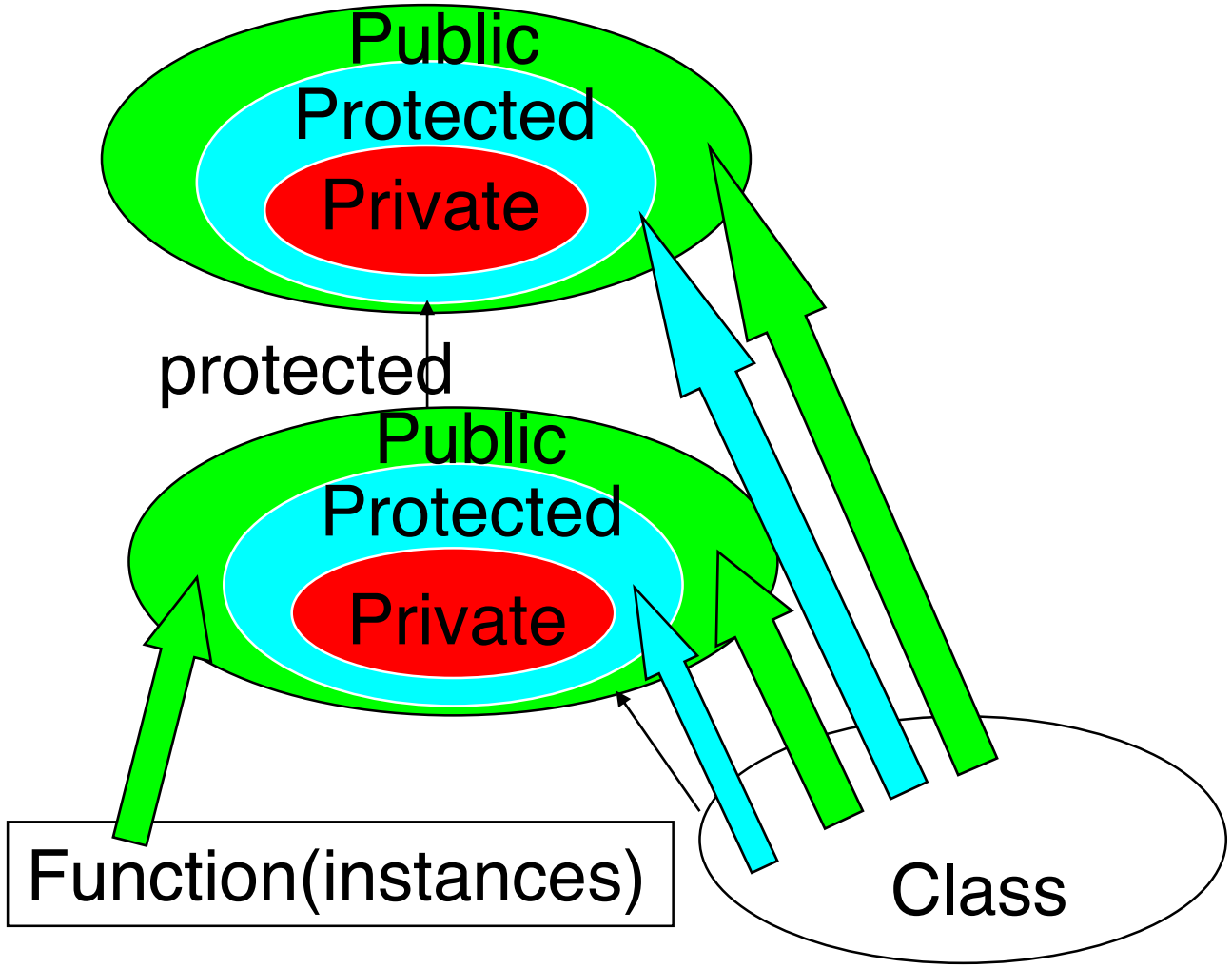
```
int main() {
      ……
      p = aStack.removeFirst(); //Error
      ……
return 0;
}
```

```
****** Creating an Item

Item::Item50
******* Creating a Stack

**** Stack::push() 50
Item::setPtr
List::putFirst() 50
**** Stack::pop()
List::removeFirst()
aStack.pop() 50

Item::setItem100
**** Stack::push() 100
Item::setPtr
List::putFirst() 100
Calling removeFirst() from aStack
List::removeFirst()
aStack.removeFirst() 100
```

protected Inheritance

- public and protected members of the base class become protected members of the derived class

protected Inheritance



Example: Lec8_ex5-protected-inher.cpp

```
int main() {
    ..........
    p = aStack1.removeFirst(); //Error
    ............
return 0;
}
```

```
Item::Item50
Item::setPtr
List::putFirst() 50
Stack1::push() 50
Stack1::pop()
List::removeFirst()
 aStack1.pop50

Item::setItem100
Item::setPtr
List::putFirst() 100
Stack1::push() 100
```

Constructors in Derived Classes

- When an object of a derived class is created, the constructor of the (derived) class must first call the constructor of the base class
- See Lec8_ex6-ctor_derived.cpp

Constructor- Initializers

```
class_name::class_name(param-list) : ctor-initializer {
// function body
}
```

- ctor-initializer is used to transfer the parameters to the constructors of the base-class
- See Lec8_ex7-ctor_init.cpp

Why using ctor-initializer?

- Without it, the default constructor for the base class would be called, which would then have to be followed by calls to access functions to set specific data members

Destructors

- Destructors are called implicitly starting with the last derived class and moving in the direction of the base class

Compatibility Between Base and Derived Classes

- An object of a derived class can be treated as an object of its base class
- The reverse is not true

Nested Class Scope

- A public or protected base class member that is hidden from the derived class can be accessed using the scope resolution operator "::"
- For example: ***base-class::member***
- On the contrary, the base class cannot access the members of its derived classes

Implicit Conversion of Derived Pointers to Base Pointers

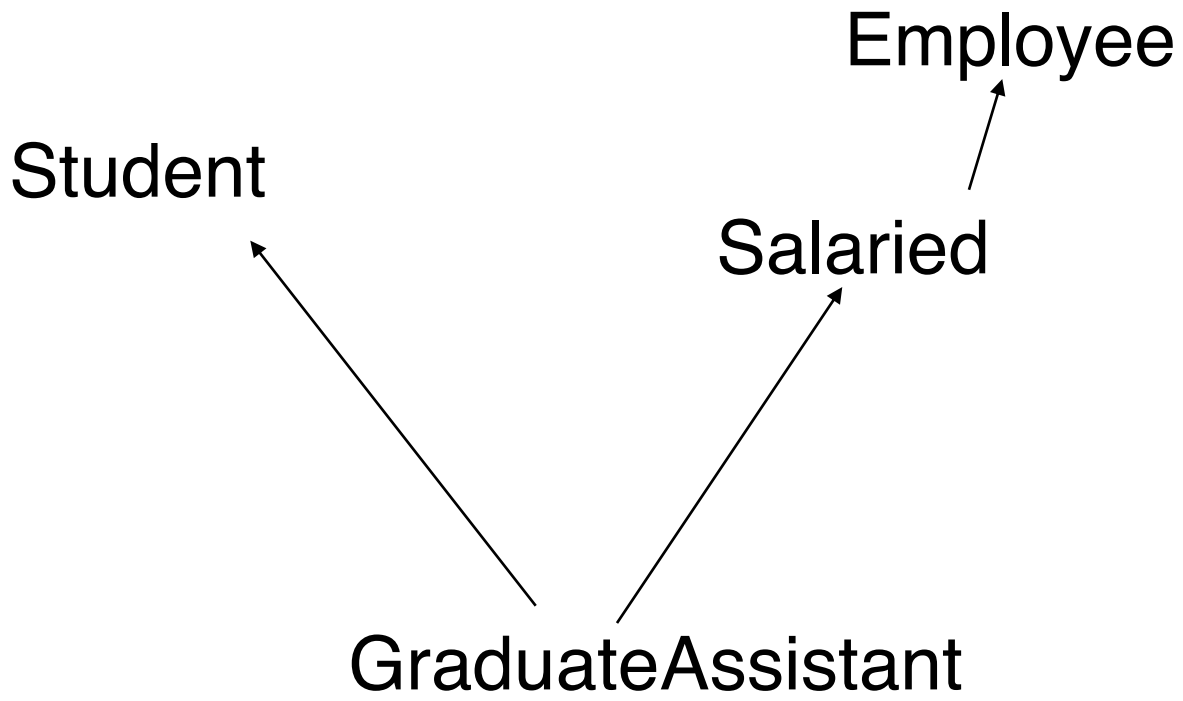
- A base type pointer can point to either a base object or a derived object

```
// Assume Point3D is derived from Point
Point3D * cp = new Point3D;
Point3D *cp1;
Point * p;
p = cp; //OK
cp1=p; //Error
cp1=(Point3D*) p;//OK
```

Casting Base Pointers to Derived Pointers

- A base pointer cannot be ***implicitly*** converted to a derived pointer
- This conversion is risky, because the derived is expected to contain more (attributes & behaviors) than the base object
- Forcing class users to use explicit casting often leads to poor code

Multiple Inheritance

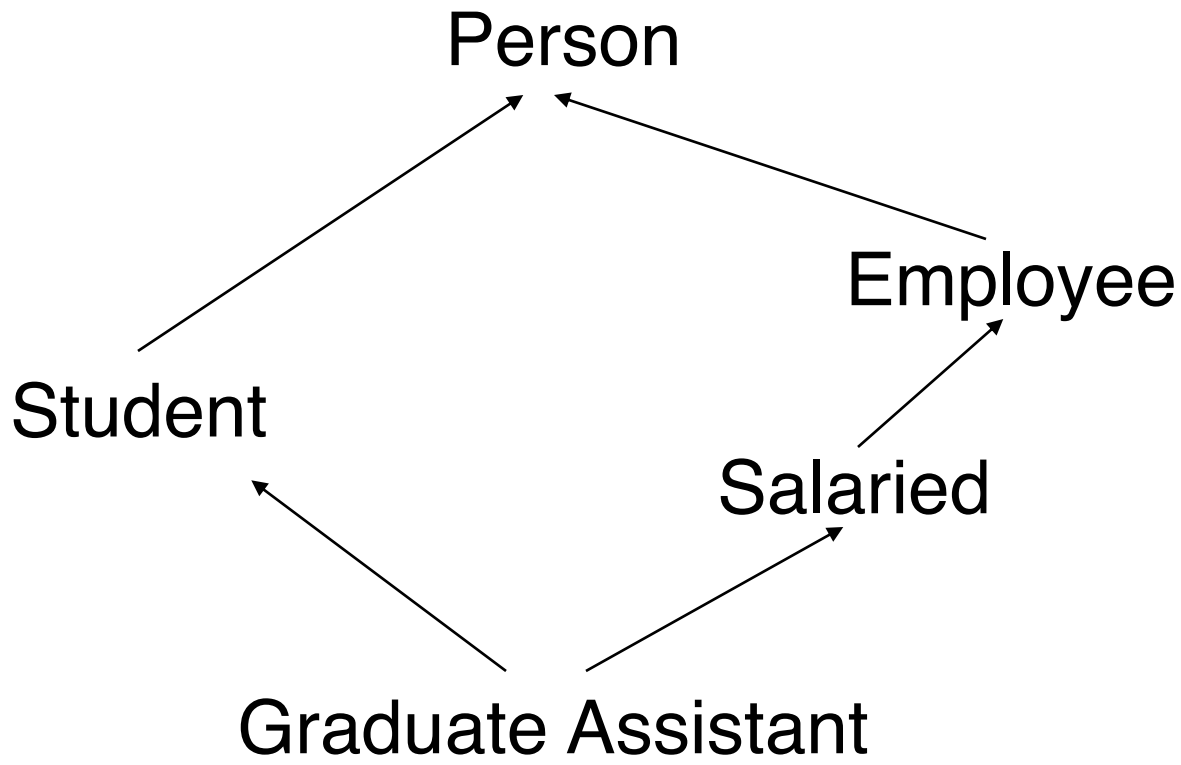


Example: Lec8_ex8_GradAssistant.cpp

Example

- Question: If we want to set a GradAssistant's age by calling setAge(), which setAge() should we use?
  - Student::setAge() or Salaried::setAge()
- Solution: Abstract (Virtual) base classes

Virtual Base Classes

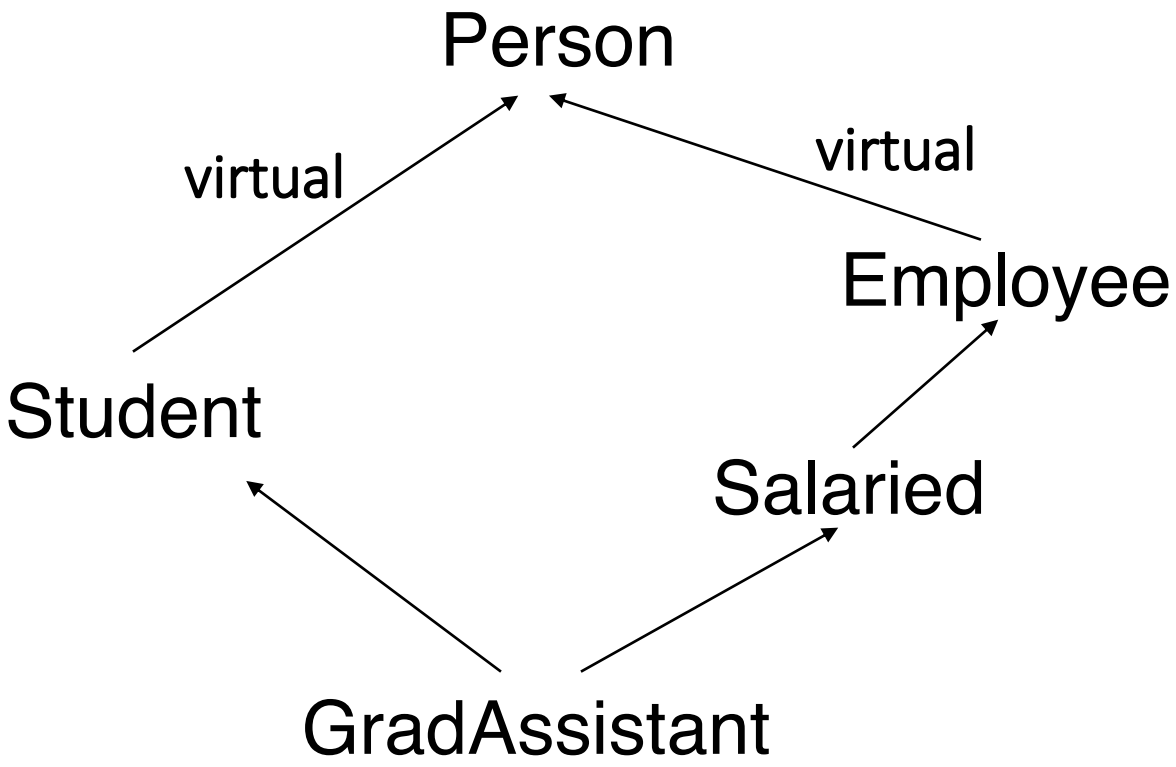


Example: Lec8_ex9_GradAssistant.cpp

Virtual Base Classes

- The function call to getAge() in GradAssistant::display() is ambiguous unless Person is inherited as a virtual base class
- Adding “virtual” lets the complier decide which function and which variable should be accessed

Virtual Base Classes



Virtual Base Classes

- When we use virtual inheritance, we are guaranteed to get only a single instance of the common base class. In other words, the **GradAssistant** class will have only a single instance of the **Person** class, shared by both the **Student** and **Employee** classes. By having a single instance of **Person**, we've resolved the compiler's immediate issue, the ambiguity, and the code will compile fine

Object-oriented programming

Lecture #9: Polymorphism

Outline

- Definition and Different Types of Polymorphism
- (Inclusion) Polymorphism and Dynamic Binding

Literal Meaning

- Polymorphism:
  - *poly* ~ many/different
  - *morph* ~ forms/shapes

- *Definition: the ability to assign a different meaning or usage to something in different contexts*

Types of Polymorphism

- Overloading
- Coercion
- Parametric Polymorphism
  - A concept originated from functional programming
  - Now popular in Java as a form of generic programming
- **Inclusion Polymorphism or *Overriding***
  - In the context of Object Oriented Programming (OOP), it is *the* Polymorphism

Overloading

- We *overload* when we want to do "essentially the **same thing**", but **with different parameters**

```
int add(int a, int b) { return a + b; }
float add(float a, float b} { return 1.0 + a + b; }
```

Coercion

- An object or a primitive is (automatically) cast into another object or primitive type (more than just overloading)

```
int add(int a, int b) { return a + b; }
float add(float a, float b) { return 1.0 + a + b; }
int main() {
    std::cout << add(1, 1.0) << endl;
}
```

Parametric Polymorphism

- Provides means to execute the same code for any type
- In C++ parametric polymorphism is implemented via templates.

Inclusion Polymorphism

Encapsulation

- Bundling data and associated functionalities
- Hide internal details and restricting access

Inheritance

- Deriving a class from another, affording code reuse

Abstraction

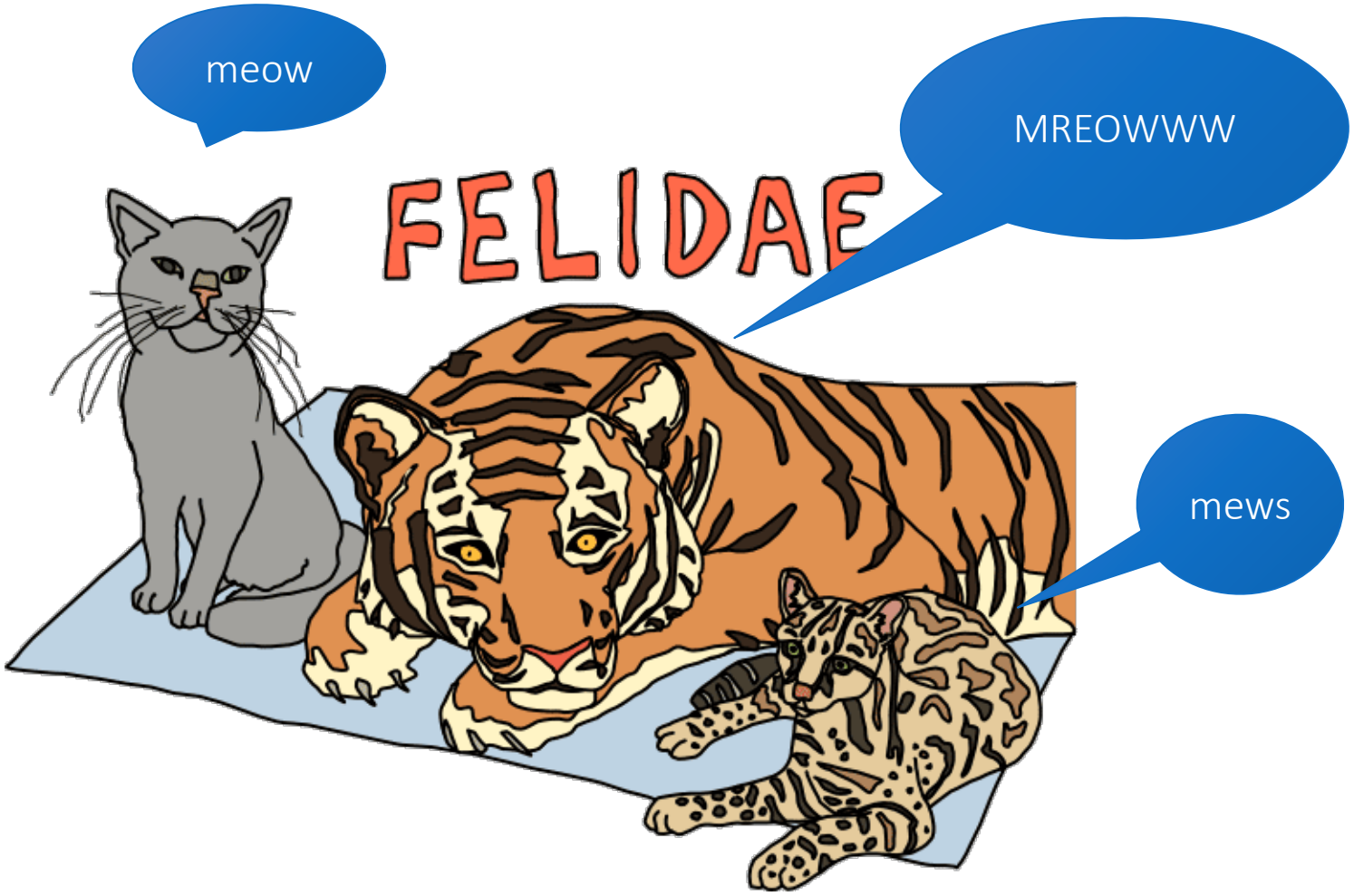
- Hiding the complexity of the implementation
- Focusing on the specifications and not the implementation details

Polymorphism or Overriding

Polymorphism

- Different types of objects respond differently to the same function call (and same parameter types)
- This is achieved by *overriding*
- We *override* an inherited function when we want to do something *slightly different* than what in the base class
- Note that this highly connected to the concept of inheritance

Illustration for (Inclusion) Polymorphism



Binding

- Connecting a method call to a method body is called *binding*
- When binding is performed before the program is run, it is called *static*/*early* binding
  - e.g. in C compilers
- Binding occurs at run-time, based on the type of object, is called *dynamic*/*late* binding
- Polymorphism is a concept; Dynamic binding is a description of how that concept is implemented

Static Binding in C++

```
// file cats.h

class Felid {
public:
  void meow() {
    std::cout << "Meowing like a regular cat! meow!\n";
  }
};

class Cat : public Felid {
};

class Tiger : public Felid {
public:
  void meow() {
    std::cout << "Meowing like a tiger! MREOWWW!\n";
  }
};

class Ocelot : public Felid {
public:
  void meow() {
    std::cout << "Meowing like an ocelot! mews!\n";
  }
};
```

Static Binding in C++

```
#include <iostream>
#include "cats.h"

void do_meowing(Felid *felid) {
  felid->meow();
}

int main() {

  Felid* felidae[] = { new Cat(), new Tiger(), new Ocelot() };

  for (int i = 0; i < 3; i++)
    do_meowing(felidae[i]);

}
```

```
Meowing like a regular cat! meow!
Meowing like a regular cat! meow!
Meowing like a regular cat! meow!
```

Dynamic Binding in C++

```
// file cats.h

class Felid {
public:
  virtual void meow() {
    std::cout << "Meowing like a regular cat! meow!\n";
  }
};

class Cat : public Felid {
};

class Tiger : public Felid {
public:
  void meow() {
    std::cout << "Meowing like a tiger! MREOWWW!\n";
  }
};

class Ocelot : public Felid {
public:
  void meow() {
    std::cout << "Meowing like an ocelot! mews!\n";
  }
};
```

Dynamic Binding in C++

- In C++, methods are non-virtual by default. They can be made virtual by using ***virtual*** keyword
- This design decision is due to the fact that C++ was made to be *almost as efficient as* C
  - Dynamic binding is more costly than static binding
  - "If you don't use it, you don't pay for it"
- If a method cannot be overridden, it then can be statically bound (and/or made inline)
  - More compiler optimization techniques can be applied to such methods

Dynamic Binding in C++

- Virtual Function
  - A non-static member function prefaced by the *virtual* keyword
  - It tells the compiler to generate code that selects the appropriate version of this function at run-time

Importance of Polymorphism

- When we have a collection of generic reference variables and each is assigned to a different type of object
  - We can step through the collection and for each element, calling a polymorphic function (e.g. `meow()`)
  - The runtime system picks out the appropriate method bodies that apply to the different types of objects

Importance of Polymorphism

- When new derived class (subclass) is added, we do not need to change the existing code. Example:
  - We add a new class `Cheetah`
  - This should not affect the code defining the previous classes and the method `doMeowing()`

Virtual Destructors

- See Lec9_ex1-VirtualDes.cpp
- Calling the wrong destructor could be disastrous, particularly if it contains a *delete* statement
- Destructors are not inherited
- Always make base classes' destructors virtual when they're meant to be manipulated polymorphically

Abstract Classes

- In C++, an interface describes the **behavior or capabilities** of a class **without committing to a particular implementation of that class**.
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data
- A class is made abstract by declaring at least one of its functions as **pure virtual** function. *A pure virtual function is specified by placing "= 0" in its declaration*

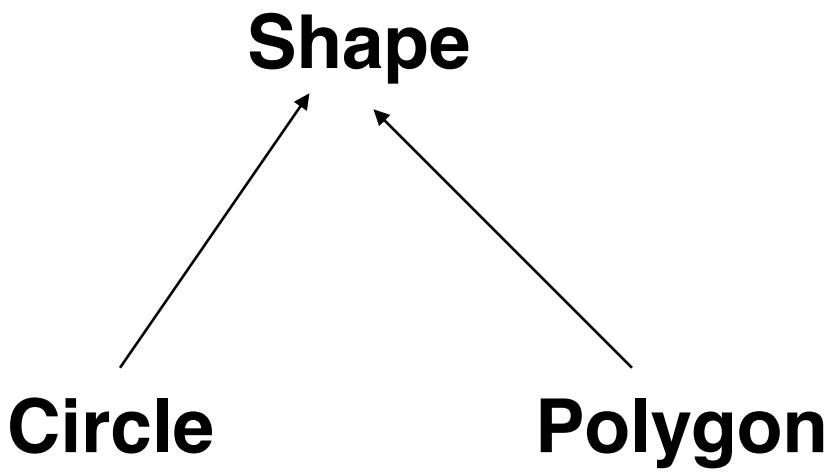
Pure Virtual Function

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Abstract Classes

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit. Abstract classes often represent concepts for which objects cannot exist
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error
- In contrast, classes that can be used to instantiate objects are called **concrete classes**

Example: Lec9_ex2-Shapes.cpp



An Abstract Derived Class

- If a pure virtual function is not defined in a derived class, the derived class is also considered an abstract class
- When a derived class does not provide an implementation of a virtual function the base class implementation is used
- It is possible to declare pointer variables to abstract classes

“this” Keyword

- Within a member function of a class, “this” is the name of an implicit pointer to the current object which receives the message associated to this function
- “this” is really a ***polymorphic*** word which can mean any object in the C++ language

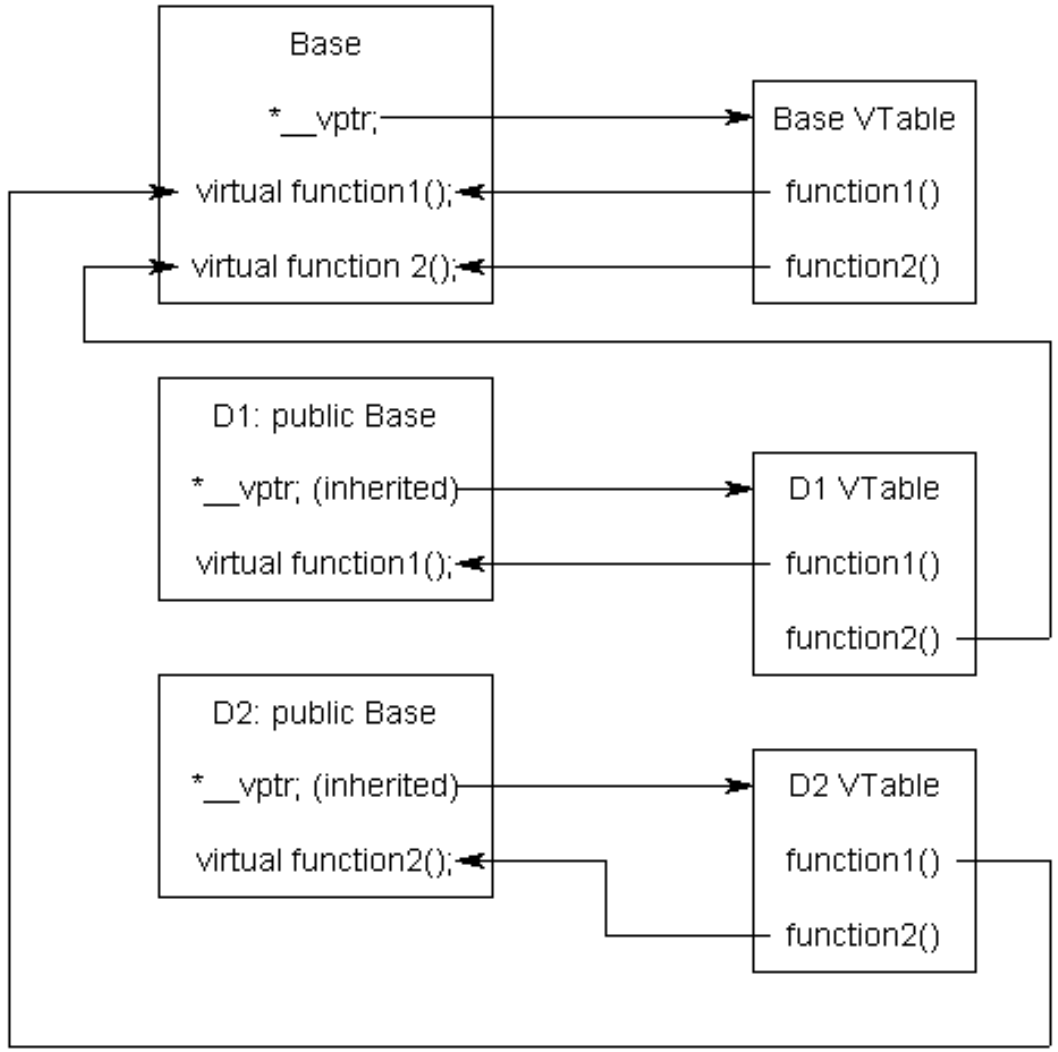
Summary

- Four types of Polymorphism
  - Runtime polymorphism
  - Parametric polymorphism (compile time polymer.)=>Templates
  - Ad-hoc polymorphism (Overloading)
  - Coercion polymorphism (Casting)
- (Inclusion) Polymorphism or Overriding
  - Why important?
  - Do not confuse with overloading
  - Requires dynamic binding
  - Static binding as default, virtual keyword

Virtual Tables

- C++ (and Java) use the virtual table (`vtable`) mechanism to implement dynamic binding
  - A class with virtual member functions has a virtual table which contains the address of its virtual functions
  - An object of such a class has a pointer (`vptr`) to point to the virtual table of the class
  - Dynamic binding is done by
    1. *Following the `vptr` to reach the virtual table of the class*
    2. *Looking up the virtual table for the entry point of the appropriate function at run-time*

Virtual Tables



Object-oriented programming

Lecture #10: Overloading

Outline

- Function Overloading
- Operator Overloading

Function Overloading

- The ability to give several functions the *same name*, provided the parameters for each of the functions differ in either:
  - number
  - type
  - order

Function Overloading

- (A form of polymorphism, as we discussed in Lecture #9)
- In C++ (and in many programming languages), a function is identified by not only the **name** but also the **number**, the **order** and the **types of the parameters**, which is called the **signature**

Examples

- void swap (unsigned long &, unsigned long &)
- void swap (double &, double &)
- void swap (char &, char &)
- void swap (Point &, Point &)

Each is a different function!!!

Poor Practice

```
class Student {
   public:
     unsigned credits (); // get the credits
     unsigned credits (unsigned n); // set the credits with n
};
```

Functions with the same names should have similar functionality

Overloading Constructors

```
class Point {
    int x, y;
    public:
      Point (int xx = 0, int yy = 0) {
        x = xx; y = yy;
        cout << “Point Constructor.\n”;
      }
};
```

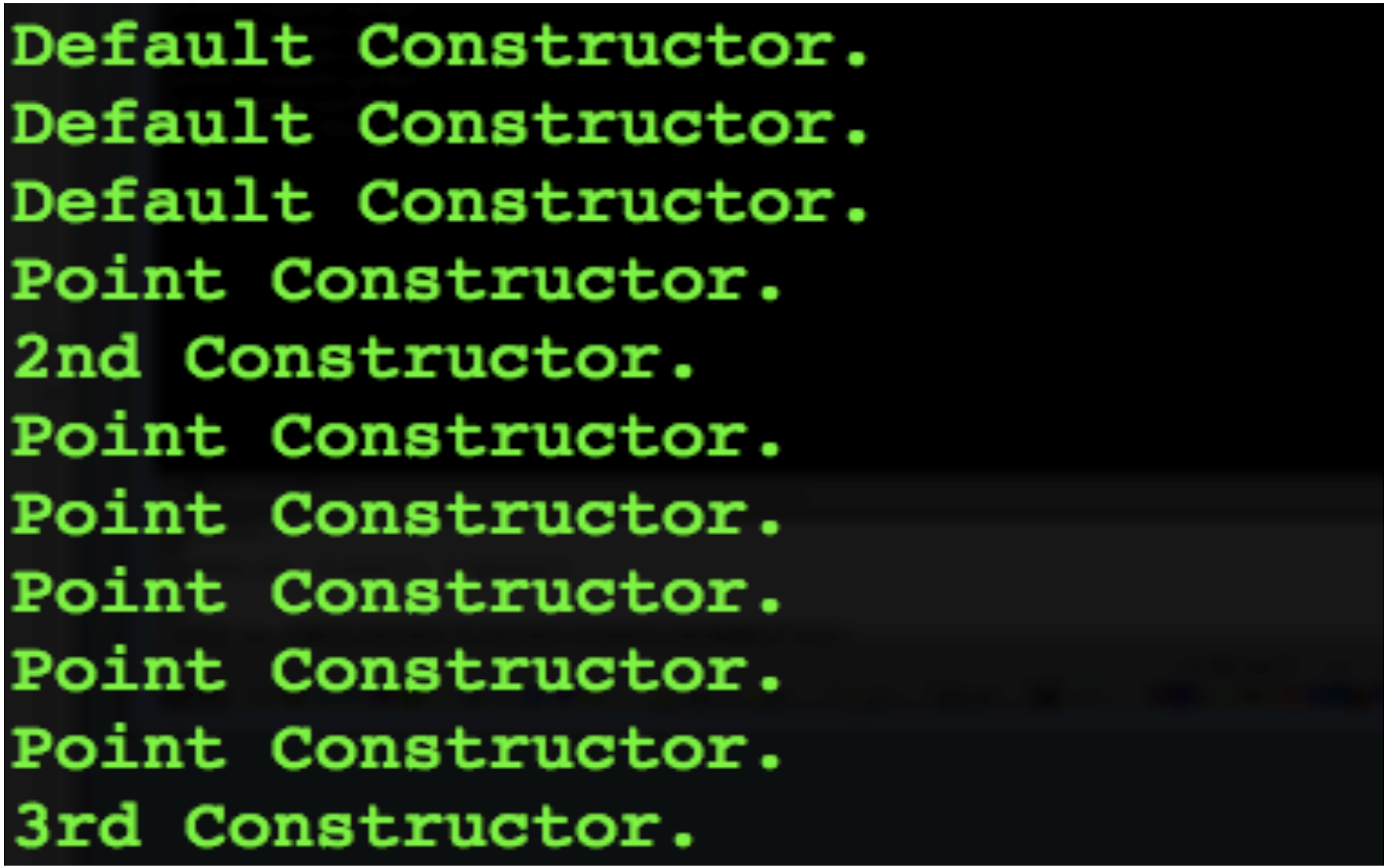
Overloading Constructors

```
class Figure {
   public:
     Figure() { cout << “Default Constructor.\n”; }
     Figure(const Point & center) {
       cout << “2nd Constructor.\n”;
     }
     Figure(const Point vertices[], int count) {
       cout << “3rd Constructor.\n”;
     }
};
```

Overloading Constructors

```
int main() {
    Figure fig1[3];
    Point center(25, 50);
    Figure fig2(center);
    const int VCount = 5;
    Point verts[VCount];
    Figure fig3(verts, VCount);
    return 0;
}
```

Overloading Constructors



Coercion (revisited)

```
void calculate (long p1, long p2, double p3, double p4);
long a1 = 12345678;
int a2 = 1;
double a3 = 2.3455555;
float a4 = 3.1;
calculate(a1, a2, a3, a4); // OK
Student s;
calculate(s, 10, 5.5, 6) // Incompatible
```

Overloading Resolution

- Best-matching function principle:
  - For each argument, the compiler finds the set of all functions that best match the parameter
  - If resulted in zero or more than one function, an error is reported

Example

```
void display (int x); // version 1
void display (float y); // version 2
int i;
float f;
double d;
display(i); // version 1
display(f); // version 2
display(d); // do not know which one!!!
```

Another Example

```
void print (float a, float b) { cout << "version 1\n”; }
void print(float a, int b)  { cout << "version 2\n”; }
int main() {
  int i = 0, j = 0; float f = 0.0; double d = 0.0;
  print(i, j); // version 2
  print(i, f); // version 1
  print(d, f); // version 1
} // ex4-coercion.cpp
```

Another Example

```
print (d,3.5);   // error
print (i,4.5);   // error
print (d,3.0);   // error
print (i,d);     // error
```

Explicit Casting makes it work

```
print (d,float(3.5)); // version 1
print (i,int(4.5));     // version 2
print (d,float(3.0)); // version 1
print (i,int(d));                 // version 2
```

Operator Overloading

- Refers to the technique of ascribing new meaning to standard operators such as +, >>, =, ..., when used with class operands
- In fact, it is a way to name a function
- **Using the same name with some normal operators, make the program more readable**

Operator Overloading

- Define an overloaded operator in class AClass

```
class AClass {
   public:
     int operator +(AClass &a) { return  1; }
   };
   int main() {
     AClass a, b; int i;
     i = a+b; //i = a.operator +(b);
} // Lec10_ex5-overloading-op.cpp
```

Example of Operator Overloading

(Already in C++)

- The '+' symbol has been overloaded to represent:
  - integer addition
  - floating-point addition
  - pointer addition

Operators Allowing Overloading

- Unary Operators
  - new, delete, new[], delete[],
  - ++, --, (), [], +, -, *, &, !, ~,
- Binary operators
  - +, -, *, /, %, =, +=, -=, *=, /=, %=, &, |, ^, ^=, &=, |=, ==, !=, >, <, >=, <=, ||, &&, <<, >>, >>=, <<=, ->, ->*

Operators that do not Allow Overloading

- ‘.’ member access
- ‘.*’ member access-dereference
- ‘::’ scope resolution
- ‘?:’ arithmetic-IF

Restrictions

- Neither the precedence nor the associativity of an operator can be changed
- Default arguments cannot be used
- The “arity” of the operator cannot be changed
- Only existing operators may be overloaded

The Time Class

//Lec10_ex6-time

The Time Class

- When the compiler sees ++**a,** it generates a call to **Time::operator++();** When it sees **b**++ it calls **Time::operator++(int);**
- All the user sees is that a different function gets called for the prefix and postfix versions. However, the two functions calls have different signatures, so they link to two different function bodies. The compiler passes a dummy constant value for the **int** argument (which is never given an identifier because the value is never used) to generate the different signature for the postfix version.

“=” Operator and Copy Constructor

- The “=” (Assignment) can also be overloaded.

Copy Constructor

```
Transcript::Transcript( const Transcript & T ) {
      count = T.count;
      courses = new string[MAXCOURSE];
      for(unsigned i = 0; i < count; i++)
        courses[i] = T.courses[i];
      cout <<"copy constructor."<<endl;
}
```

Assignment operator

```
Transcript & Transcript::operator =( const Transcript & T )
{
    if( this != &T ) { // not the same object?
        delete [] courses;
        courses = new string[MAXCOURSE];
        count = T.count;
        for(int i = 0; i < count; i++)
          courses[i] = T.courses[i];
    }
    cout << "= operator.” << endl;
    return *this;
}
```

Assignment Constructor

Lec10_ex7-transcript.cpp

Object-oriented programming (with C++)

Lecture #11: Templates and Friends

Outline

- Function Template
- Class Template
- Friends

Templates

- Templates give us the means to define a **family** of functions or classes that share the same functionality but which may differ with respect to the data type used internally
- A function template is a framework for generating related functions
- A class template is a framework for generating the source code for any number of related classes

Function Template

- A function can be defined in terms of an **unspecified** type
- The compiler generates separate versions of the function based on the type of the arguments passed in the function calls

Function Template

template <class T>

return-type function-name(T param)

// one parameter function

- T is called a template parameter

One Parameter Function Template

```
template <class T>
void display(const T &val) { cout << val; }
```

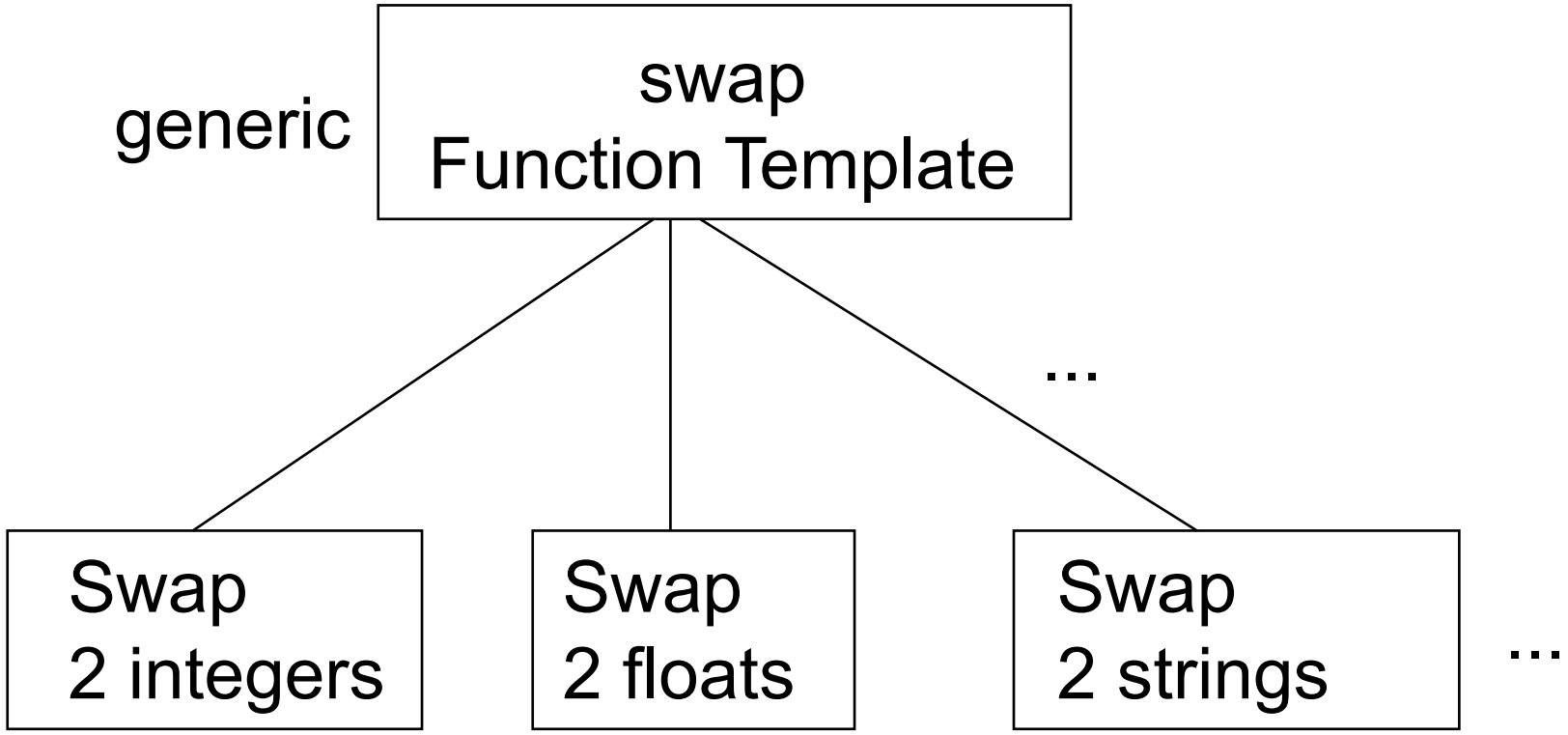
- One parameter function with an additional parameter which is note a template parameter:

```
template <class T>
void display(const T & val, osteam &os) { os << val; }
```

- The same parameter appears multiple times

```
template < class T>
void swap(T & x, T & y) {}
```

Swap


generic
swap
Function Template
...
Swap
2 integers
Swap
2 floats
Swap
2 strings
...

Swap

```
#include <iostream>
template <class T>
void swap(T & x, T & y) {
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

Swap

// Lec11_ex1-swap.cpp

Multiple Parameter Function Template

```
template <class T1, T2>
void arrayInput(T1   array, T2 & count) {
  for (T2 j= 0; j < count; j++) {
      cout << “value:”;
      cin >> array[j];
  }
}
```

Multiple Parameter Function Template

```
const unsigned tempCount = 3;
float temperature[tempCount];
const unsigned stationCount = 4;
int station[stationCount];
arrayInput(temperature, tempCount)
arrayInput(station, stationCount);
```

Table Lookup

```
template <class T>
long indexOf( T searchVal, const T * table, unsigned size )
{
  for (unsigned i = 0; i < size; i++)
    if (searchVal == table[i])
      return i;
  return -1;
}
```

Table Lookup

```
int main() {
  const unsigned iCount = 10, fCount = 5, sCount = 5;
  int iTable[iCount] = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 };
  foat fTable[fCount] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
  cout << indexOf( 20, iTable, iCount )   << endl;
  cout << indexOf( 2.2f, fTable, fCount ) << endl;
  string names[sCount] = { "John","Mary","Sue","Dan","Bob" };
  cout << indexOf( (string) "Dan", names, sCount ) << endl;
  return 0;
}
```

Note

- In a function template, if an operator is used, you must be sure every class relevant to the template will support the operator (such as “==”, etc.)
- We can use operator overloading to ensure compatibility

Student Comparison

```
class Student {
    public:
      Student( long idVal ) {  id = idVal; }
      int operator ==( const Student & s2 ) const {
        return id == s2.id;
      }
    private:
      long id;  // student ID number
};
```

Student Comparison

```
int main() {
    const unsigned sc= 5;
    Student sTable[sc] = { 10000, 11111, 20000, 22222, 30000 };
    Student s( 22222 );
    cout << indexOf( s, sTable, sCount ) << endl;     // print "3"
    return 0;
} //Lec11_ex3-student-comp.cpp
```

Overriding a Function Tempate

- When a function template does not apply to a particular type, it may be necessary to either
  - override the function template, or
  - make the type conform to the function template

Explicit Function Implemention

```
long indexOf( const char * searchVal, char * table[], unsigned size ) {
  for (unsigned i = 0; i < size; i++)
        if( strcmp(searchVal, table[i]) == 0 )
            return i;
  return -1;
}
```

Explicit Function Implementation

```
int main() {
    const unsigned iCount = 10, nCount = 5;
    int iTable[iCount] = { 0,10,20,30,40,50,60,70,80,90 };
    cout << indexOf( 20, iTable, iCount )  << endl;//2
    const char * names[nCount] =
      { "John","Mary","Sue","Dan","Bob" };
    cout << indexOf( "Dan", names, nCount ) << endl;//3
    return 0;
}//Lec11_ex4-expicit.cpp
```

Class Template

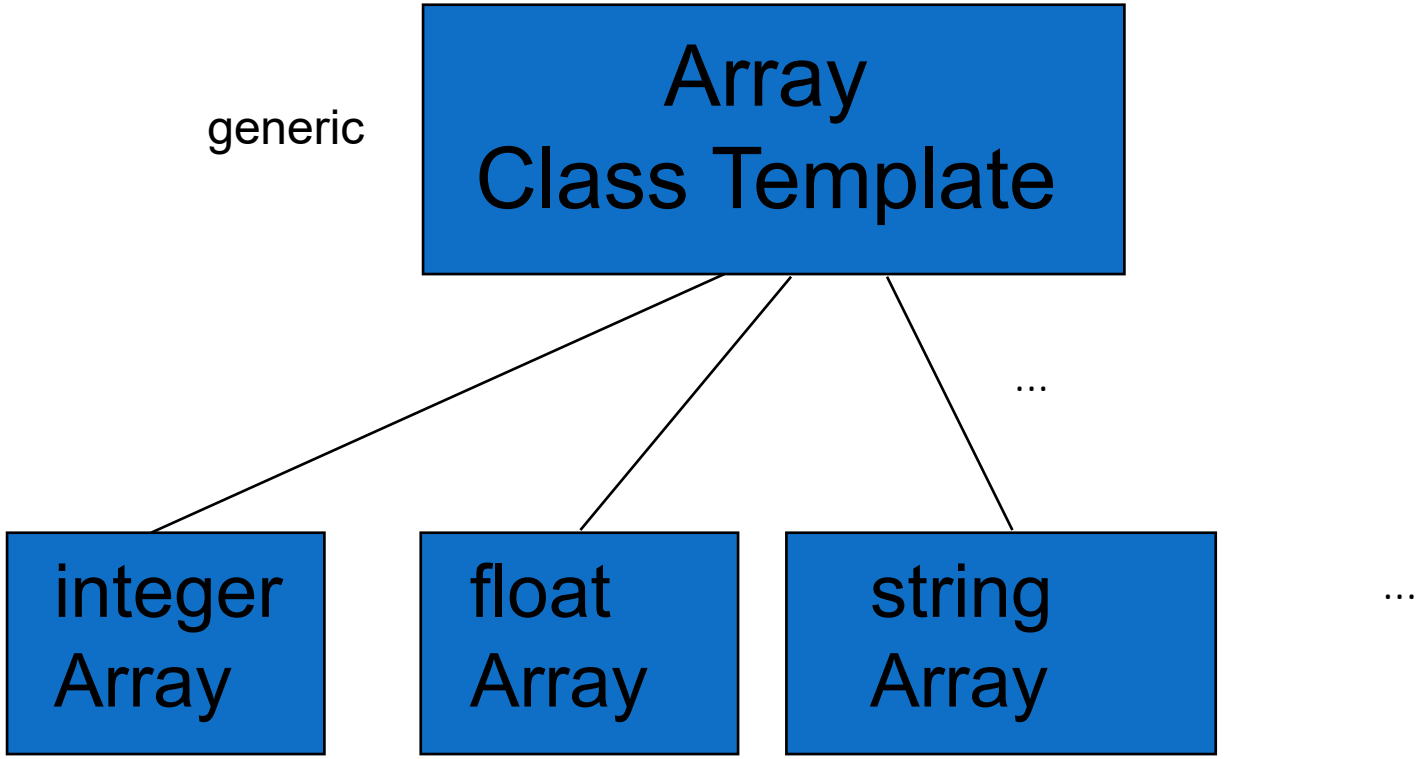
- Declare and define an object:

```
template <class T>
class MyClass {
// using T inside (parametrically)
};
MyClass <int> x;
MyClass <Student> a;
```

A Simple Example

```
template <class T1, class T2>
class Circle {
// ...
private:
  T1 x, y;
  T2 radius;
};
Circle <int, long> c1;
Circle <unsigned, float> c2;
```

Class Template



Array Class Template

```
template <class T>
class Array {
    public:
      Array( unsigned sz );
      ~Array();
      T & operator[]( unsigned i );
    private:
      T * values;
      unsigned size;
};
```

Array Class Template

```
template<class T> Array<T>::Array( unsigned sz ) {
  values = new T[sz]; size = sz;
}
template<class T> T & Array<T>::operator[] ( unsigned i ) {
  if( i >= size ) {
    cout << "ERROR: array index out of bound!!!\n”;  abort();
  }
  return values[i];
}
template<class T> Array<T>::~Array() {   delete [] values; }
```

Array Class Template

```
int main() {
  const unsigned numStudents = 2;
  Array<int>     ages( numStudents );
  Array<float>   gpas( numStudents );
  Array<string>  names(numStudents);
  for(int j = 0; j < numStudents; j++) {
        // do whatever you want
  }
  return 0;
} //Lec11_ex5-array.cpp
```

Templates and Inheritance

- A template is not a class
  - A template is not inherited
- A class based on a template is an ordinary class

```
class t1 : public Array<int> {...}//OK!!!
template <class T>
class Myclass : public aClass {
//OK!!!
};
```

Template and Static Members

- Remember
  - Template is not class
- Static members defined in a template are static members of the classes associated with a template

Friends

- Friend Classes
- Friend Functions

Friend Classes

- A friend class is a class in which all member functions have been granted full access to all the *(private, protected, and certainly public)* members of the class defining it as a friend (by instances)
- The friend class is declared inside the class granting friendship

Example

```
class C1 {
        friend class C2;
    //...
};
class C2{
        friend class C3;
    //...
};
```

Example

// Lec11_ex7-friend.cpp

Friend Functions

- A friend function is a function which has been granted full access to the private and protected members of an instance of the class
- The friend function is declared inside the class granting friendship

Example

//Lec11_ex8-friend-func.cpp

Properties of Friends

- Non-symmetrical
  - For example : If c2 is a friend of c1, but c1 is not necessarily a friend of c2.
- Non-transitive
  - For example: If c2 is a friend of c1 and c3 is a friend of c2, but c3 is not necessarily a friend of c1.

Properties of Friends

- Not inheritable
  - A friend of a base class is not inherited by derived classes

Example

```
class Employee {
      public:
        friend float calcPay(Employee &e);
      //..
};
class SalariedEmployee : public Employee{ //... };
```

Here, the function CalcPay() can not access the private members of SalariedEmployee!!!

Things to consider when using Friends

- Friends make loosely coupled classes tightly coupled, which may results in problems with modularity and error searching
- Friends diminish encapsulation and information hiding
- ***Limit the use of "friends"***

Object-oriented programming

Lecture #12: Container Classes

Lists

- A list consists of a set of sequentially organized elements
- A specific type of graph, where each node except the first has a single preceding node, and each node except the last has a single following node
- Contains 0-n nodes
- Implementation: array and linked lists

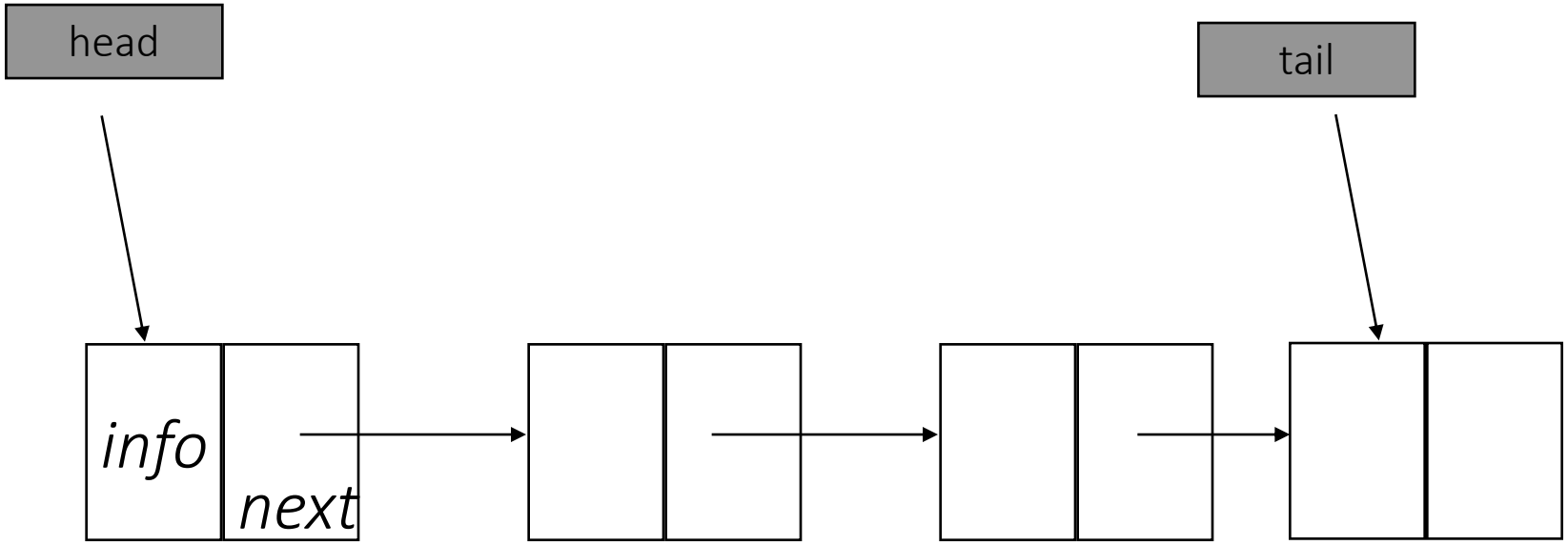
Linked Lists

- Each element in a list is called a **node**, and a connection between any two nodes is called a **link**
- Implementation: dynamic memory allocation (saves memory)
- Limitation: nodes are accessed sequentially

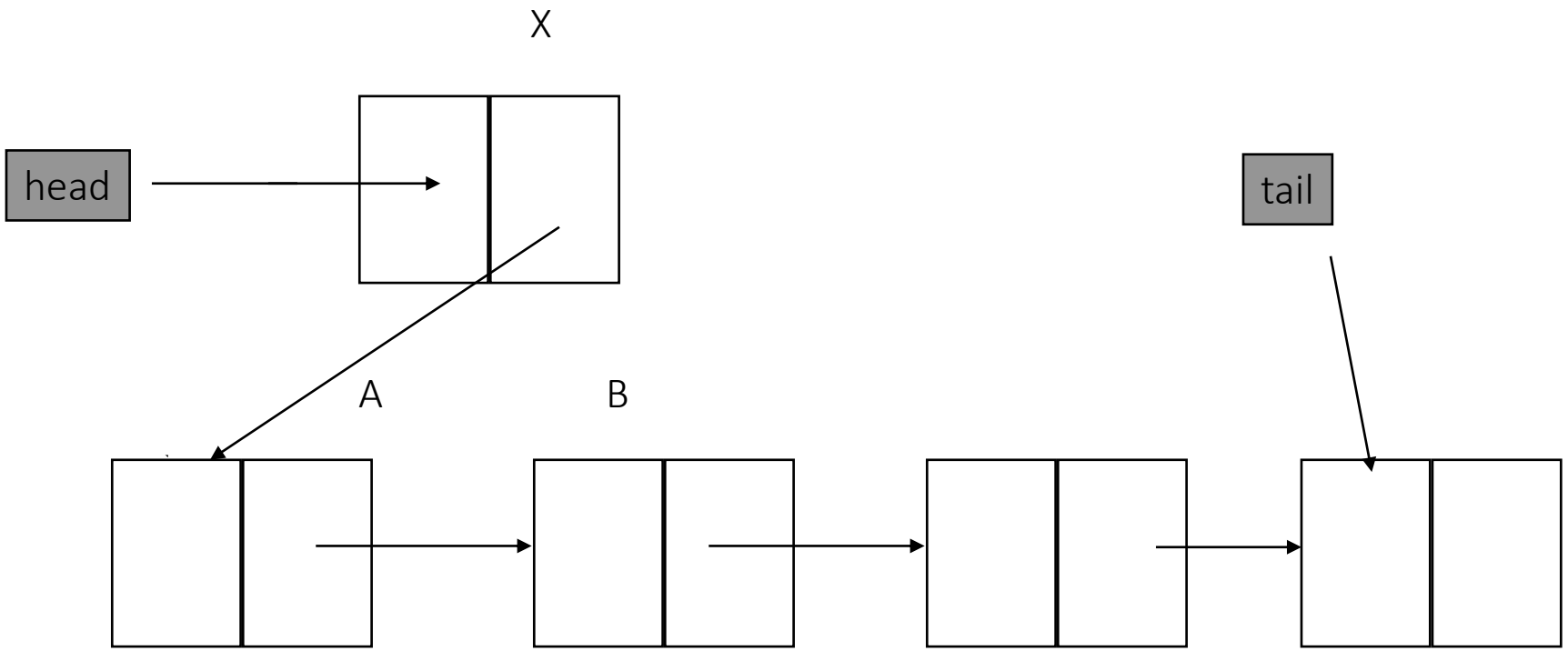
Operations on a List

- append: add a node at the end
- prepend: add a node at the beginning
- insert: insert a node in place
- find: find a specific node
- get: get a node at the current position
- replace: replace the content of a node
- isEmpty: find out if the list is empty
- remove: remove a node
- clear: remove all the nodes

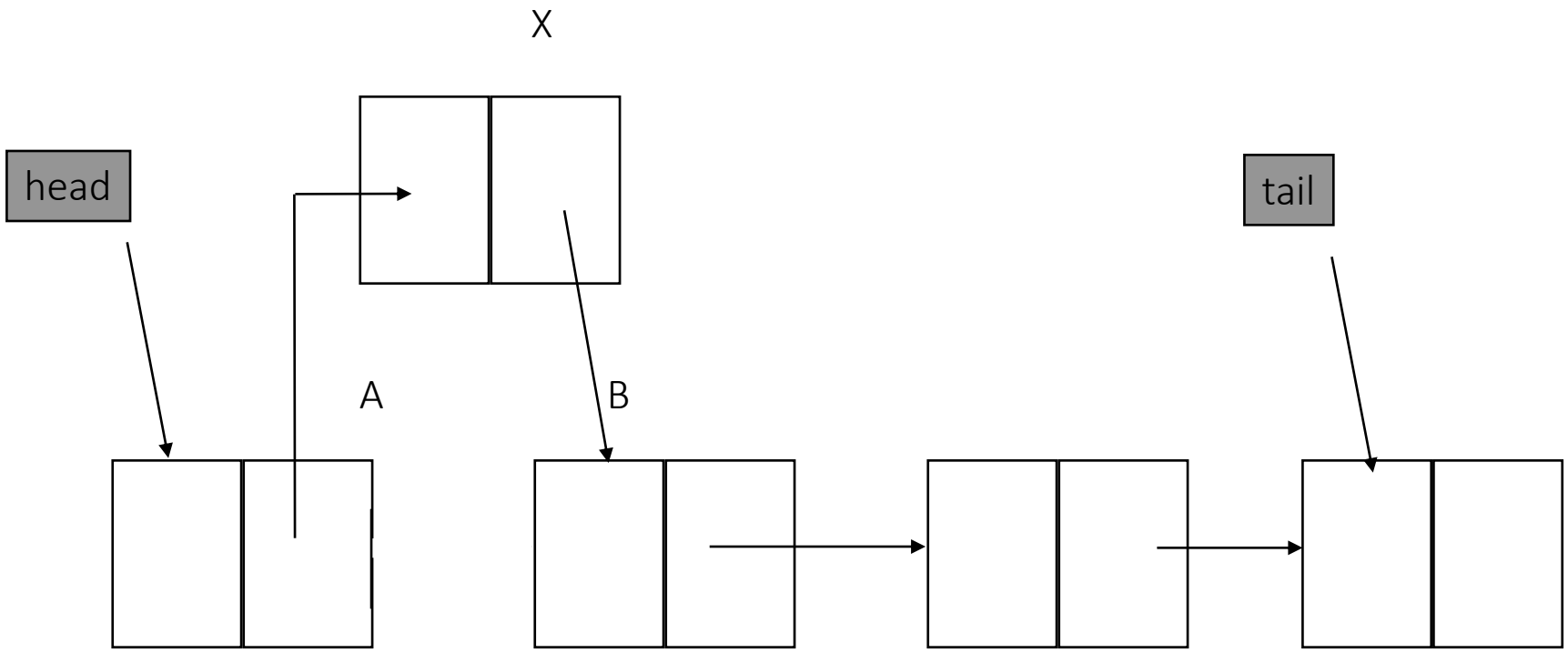
Generic Linked List



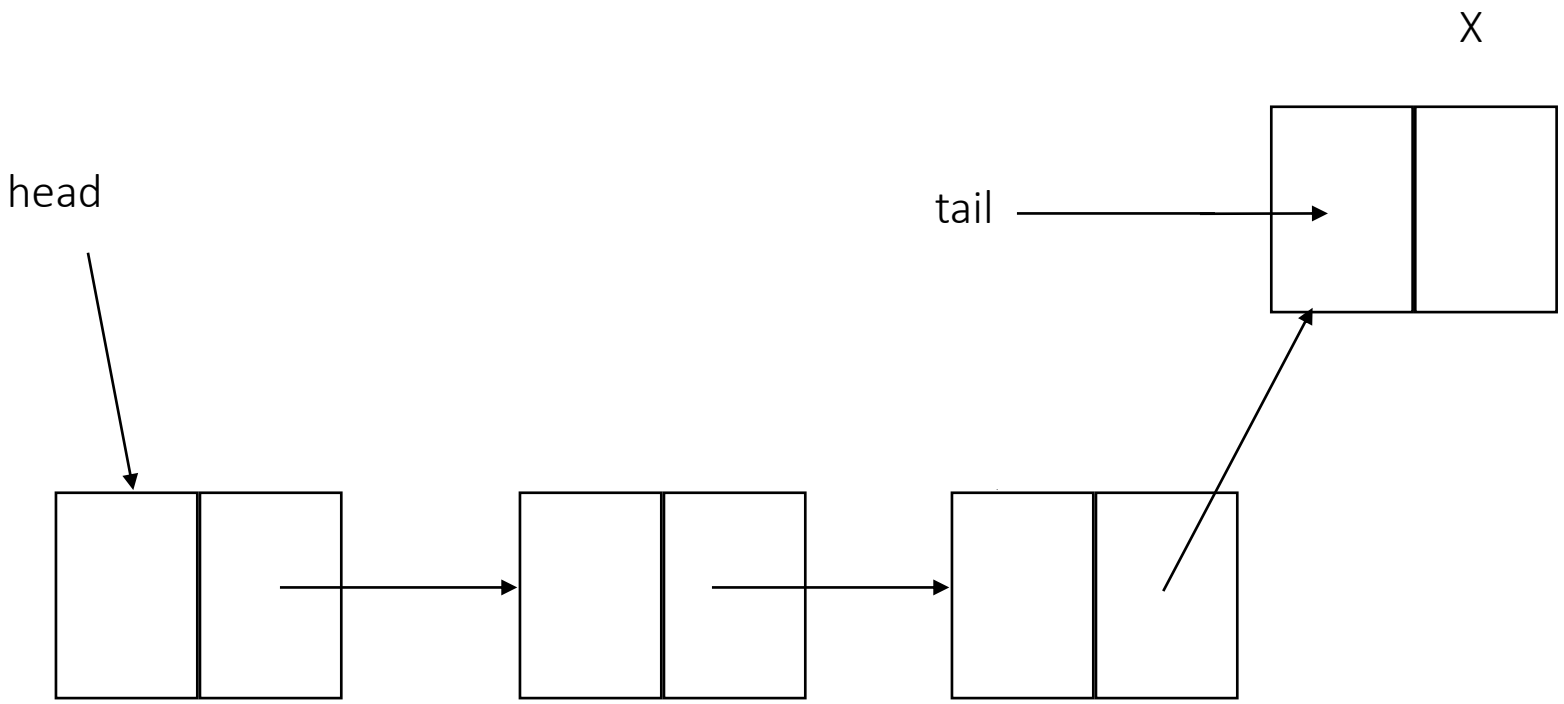
Prepend Operation



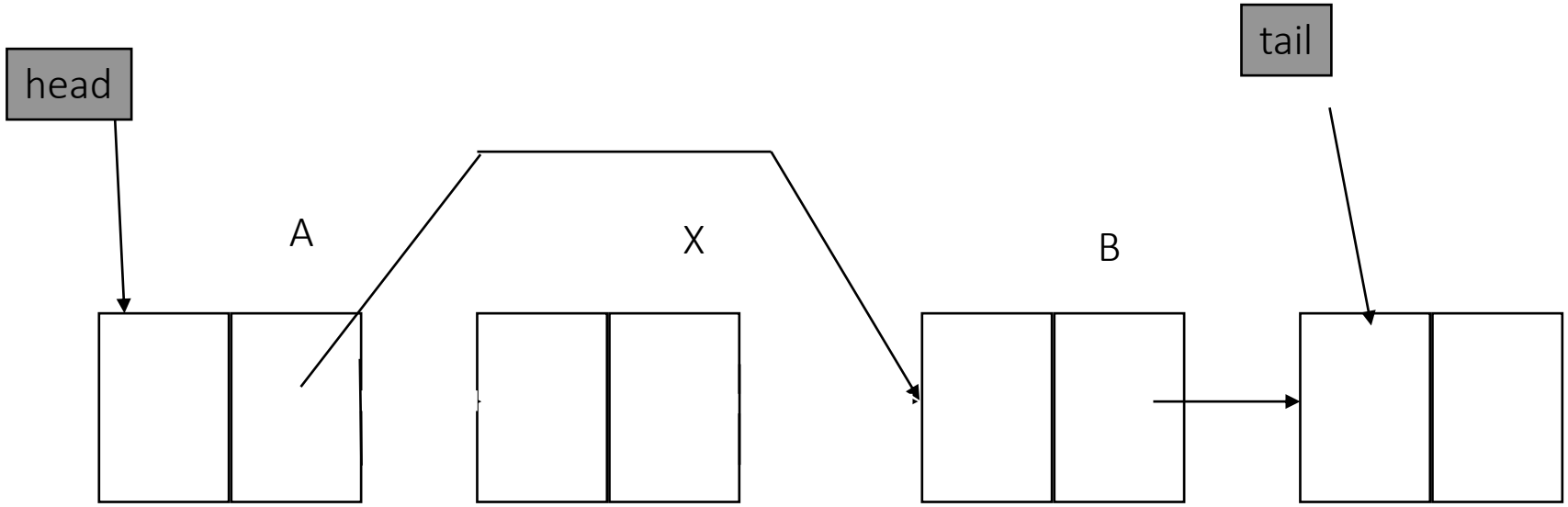
Insert Operation



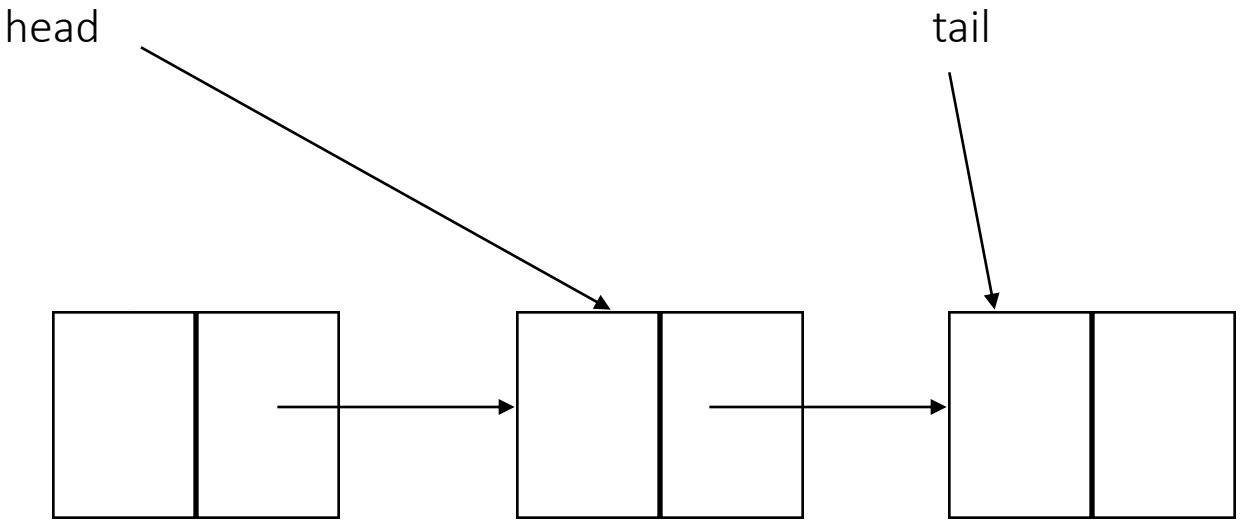
Append Operation



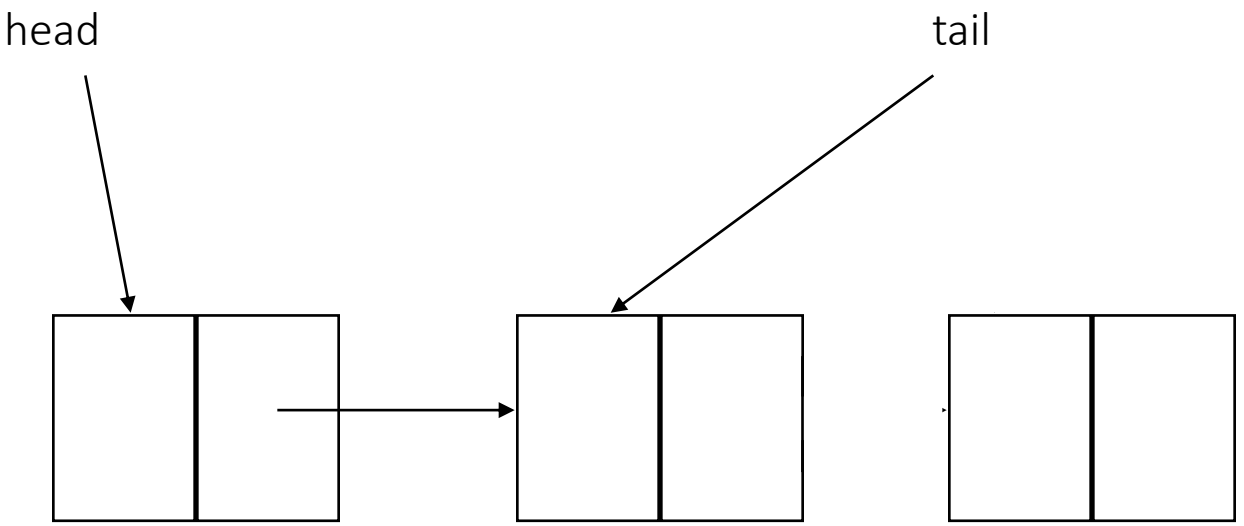
Remove a Node



Remove Head



Remove Tail



List with Different Types

- Using templates
- With the same structure, we can have integer, float, or string as a value of a node
- We can have a list with values in different types with the same base class

Exercises