

Programming Methodology

Object-Oriented Programming

Concepts of object-oriented programming

block (already discussed earlier) vs. module
data abstraction vs. object abstraction
parametric polymorphism

Language features employed in existing object-oriented languages

construction/destruction of abstract objects, type inheritance,
virtual functions and memory managements

Object-oriented problem solving

Modules

3



- A **module** is similar to a **block** in that both are a collection of declarations and statements.
- A module is different from (or we may say, more sophisticated than) a block because a module can **export** a subset of the declarations to outside the module.

Cf: all the declarations of a block are visible only inside the block.

- The exported declarations in a module is called the **interface** of the module.



Programming Methodology

Modules

4



- The declarations specified in an interface can be accessed by other modules or objects.
- The remaining declarations are hidden from others.
- In this sense, a module serves as a **black box**.
 - A module interacts with the rest of the program through an high-level interface while hiding low-level implementation details.

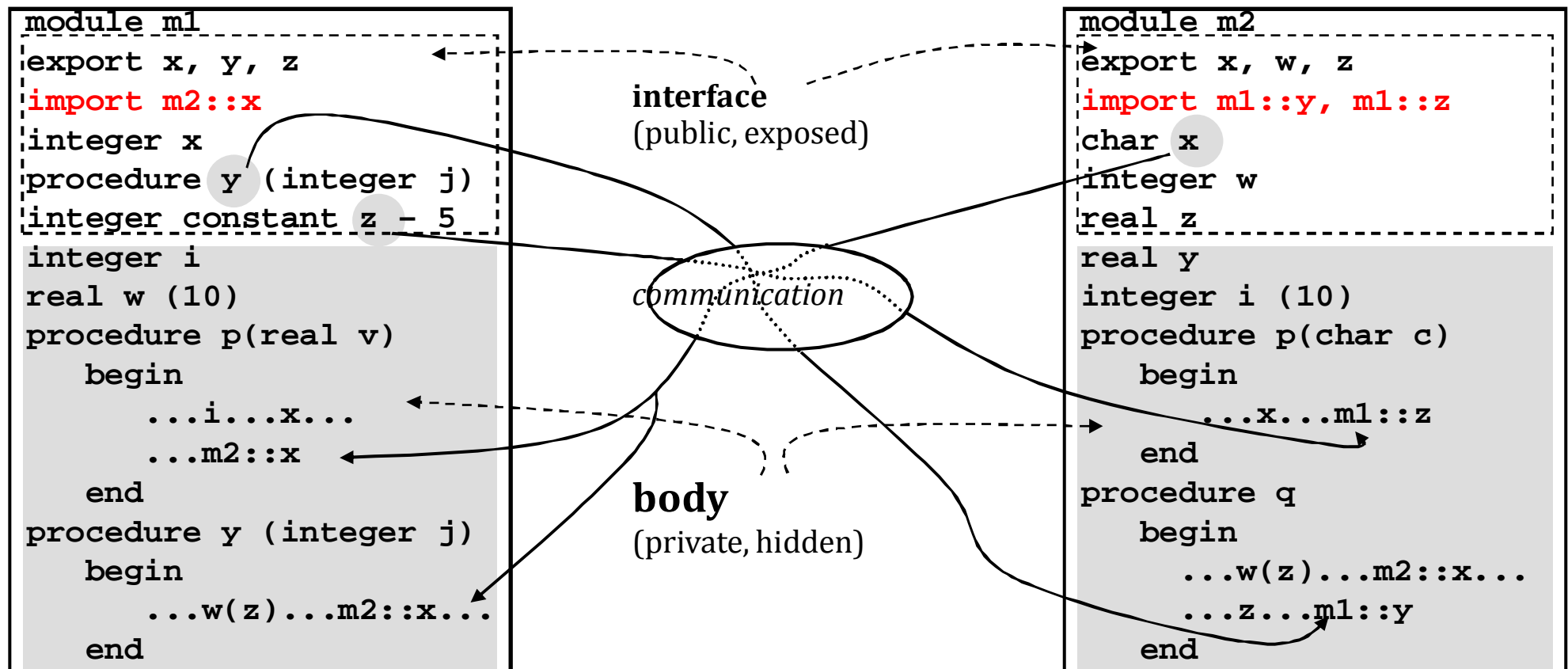
```
C++  
class demo {  
    public:  
        ...           //the interface open to the outside  
    private:  
        ...           //the rest of the module hidden from the outside  
};
```

Naming control in modules

5



- Many newer languages (esp. object-oriented) provide modules.
 - *modules (Modula), classes (C++, SmallTalk), packages (Ada), clusters (CLU)*
 - In some languages, **import list** is not explicitly specified since it is deducible.



Advantages of modules over blocks

6



- Globals/non-locals are needed for communication bet'n blocks.
- In modules, globals are discouraged because modules can communicate through parameters specified in the interface.
 - Thus, data sharing is explicit in modules, which solves the problems of side effects, indiscriminate access and screening

```
int many;
f() {
  .. many ..
}
g() {
  .. many ..
}
h() {
  int mary;
  .. g() .. many
  ... f() ..
}
```

*no error detected,
but any semantic
error?*

blocks

```
module p
export f, g, many;
procedure f()
procedure g()
int many;

f() {
  .. many ..
}
g() {
  .. many ..
}
```

modules

```
module q
import p.f, p.g;

h() {
  int mary;
  ..p.g()..many
  ... p.f() ...
}
```

naming error detected

modules

Advantages of modules over blocks

7



- Modules provide natural unit for *separate compilation*.
 - Only the change in the interface of a module affect other modules.
 - A module with objects imported from other modules can be compiled without knowing the detailed implementation of the imported objects.
 - What's the advantage of separate compilation in terms of efficiency?
- Modules can be data **objects** or variables.
 - ex: class objects in C++

Modules in the Modula language

8



- ❑ Modula is a Pascal-like language with modules.
- ❑ No explicit export statement.
- ❑ All declarations in a definition module are exported.
- ❑ A reference to an imported object is qualified with the name of the imported module in a importing module.

```
module main;  
  from m1 import x, f;  
  from m2 import x, w;  
  var x : real;  
  begin  
    m1.f(m2.x);  
    x = m1.x * 3.5 + m2.w;  
    ...  
  end main.
```


Modules in the Modula language

```
definition module m1;
  var x : integer;
  procedure f(var j :character);
  const z = 5;
end m1.
implementation module m1;
  from m2 import x;
  var i : integer;
  var w : array [1..10] of
integer;
  procedure p(var v :real);
  begin
    ... i ... x ...
    ... m2.x ...
  end p;
  procedure f(var j: character);
  begin
    .. w[z] .. m2.x ..
  end f;
end m1.
```

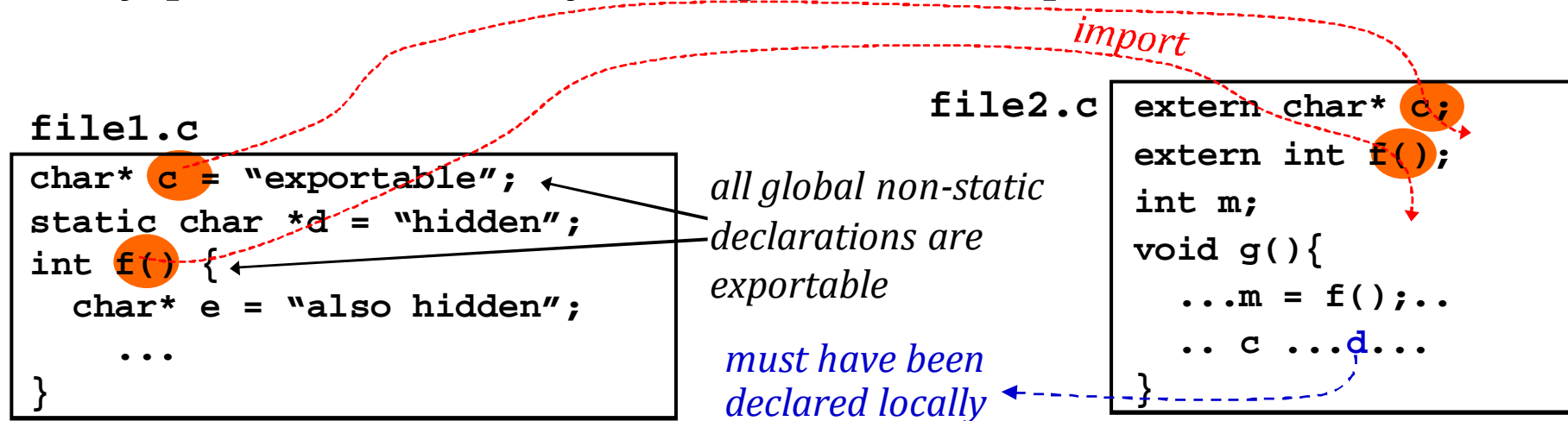
```
definition module m2;
  var x : character;
  var w : integer;
  var z : real;
end m2.
implementation module m2;
  from m1 import y, z;
  var y : real;
  var a : array [1..10] of
integer;
  procedure p(var c :character);
  begin
    ... x ... m1.z ...
  end p;
  procedure q;
  begin
    .. a[w] .. y ..
    .. z ... m1.y ..
  end q;
end m2.
```

Primitive form of modules in C

10



- Files in C language can be thought of as modules because they provide a facility to export and import declarations.



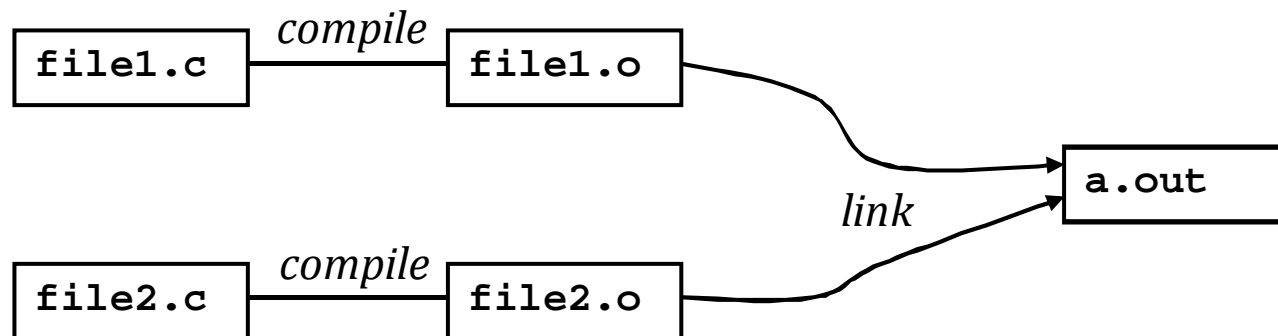
- The default is to put all global declarations in a file into the interface of the file.
 - All global declarations are exported by default.
 - Names exported by files have to be unique since file names are not part of exported names unlike ordinary modules.

Primitive form of modules in C

11



- To hide a declaration within a file, it must be declared static.
- Declarations from other files can be *imported* by extern declarations.
- Similarly to modules, files in C can also be *compiled separately* with ease.



Modules in C++

12



- The class type can be used to implement modules.

```
class Complex {
public:
    Complex(float r1, float im) { r = r1; i = im; } // constructor
    float real_part() { return r; }
    float imaginary_part() { return i; }
    Complex &operator+(const& Complex);
    Complex &operator-(const& Complex);
    ...
private: float r, i;
};

...
Complex object1(7.6,3); // object1 is 7.6+3.0i
Complex object2(5,1.1); // object2 is 5.0+1.1i
float r = object1.real_part(); // 7.6 is returned
object1 = object1 + object2; // object1 ← object1.operator+(object2) = 12.6+4.1i
```

Modules in C++

13



- The **public** part is the interface of a class module, and the **private** part is the body.
- Therefore, the declarations in the public section are exported, and the variables **r** and **i** are not exported.
- Crucial differences between classes and files in C++?
 - Different data objects can be created for each class.
 - Class names are part of the exported names.
e.g.) `object1.real_part()`, `object1.r`

Abstraction

14



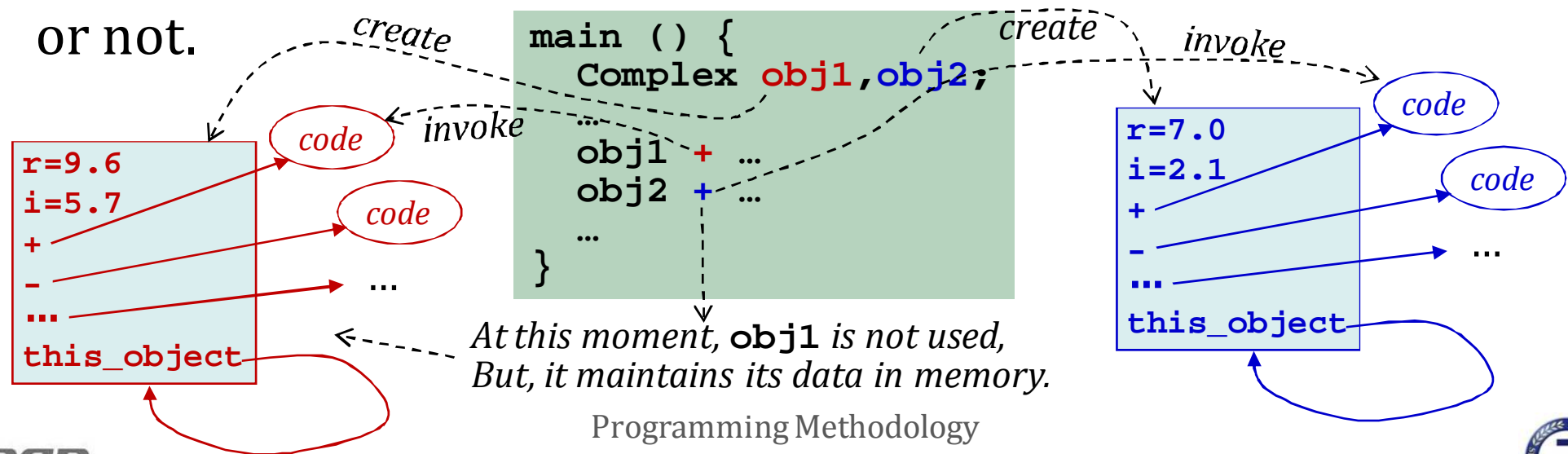
- **Abstraction** of a process or object consists of
 1. its high-level and essential properties that are exposed
 2. the remaining low-level details that are hidden.
- The forms of abstraction in programming languages
 1. procedural abstraction
 2. data abstraction (type abstraction)
 3. object abstraction

Function/Procedure vs. Module/Object

15



- Function/procedure = code
- Module/object = code + data
- A function itself is a passive entity
 - ▣ It has no life when it is not invoked.
 - ▣ No activation record or any other data structures maintaining its status
- A module is an active entity maintaining its data structures until it is explicitly destructed whether it is currently invoked or not.



Procedural abstraction

16



- Procedure **blocks** are procedural abstractions.
- *Task: "Prints the names of all employees living in L.A. in alphabetical order"*

```
struct ER { char* name; char* addr; ... ER* next; }
main() {
    ER* full_record = read_record_file("employ records");
    ER* nw_record = get_employees_living("L.A.", full_record);
    ER* sorted_record = sort_names(nw_record);
    print_records(sorted_record);
}
```

- Local variables and algorithms used in a block are hidden within the block, and only parameters and name of the procedure are exposed.
 - *For example, the procedure sort names can use any sorting algorithm (ex: quick/merge/radix sorting) and data structures or the algorithm without affecting the caller **main**.*
- Advantages of procedural abstractions are that they provide **program partitioning** and **information hiding**.
 - *Why are they advantageous?*

Program partitioning

17



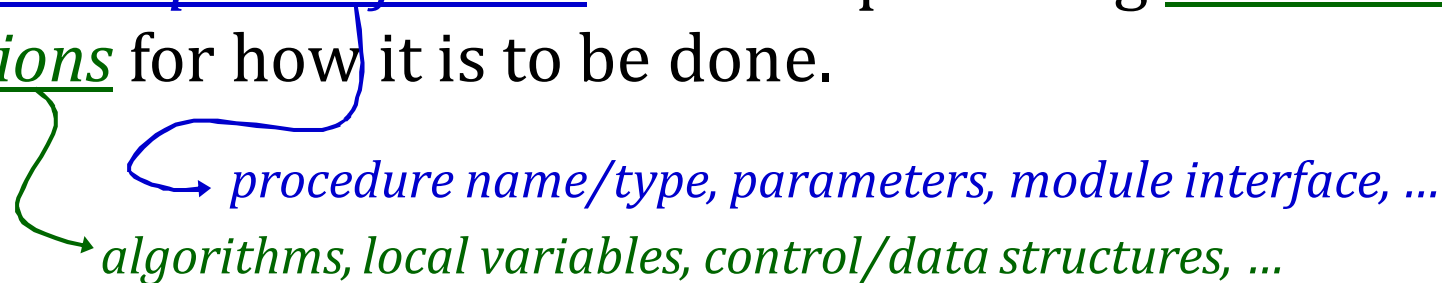
- allows the programmer to focus on one section of a program at a time without the overall detailed program continually intruding.
- abstracts away many of the details of each program section, facilitating the construction of comprehension of a large program.
- usually makes programs smaller.
 - ex) calls to the same subroutine
 - Advantages of smaller programs?
 - *easier to manage since difficulty of program writing and debugging increases more than linearly with the program size.*

Information hiding

18



- can be achieved by allowing a program to specify the high-level description of a task without providing low-level design decisions for how it is to be done.



- can reduce program complexity.
 - *With information hiding, when a design decision is changed, only the block is affected, facilitating testing and refinement of the program.*

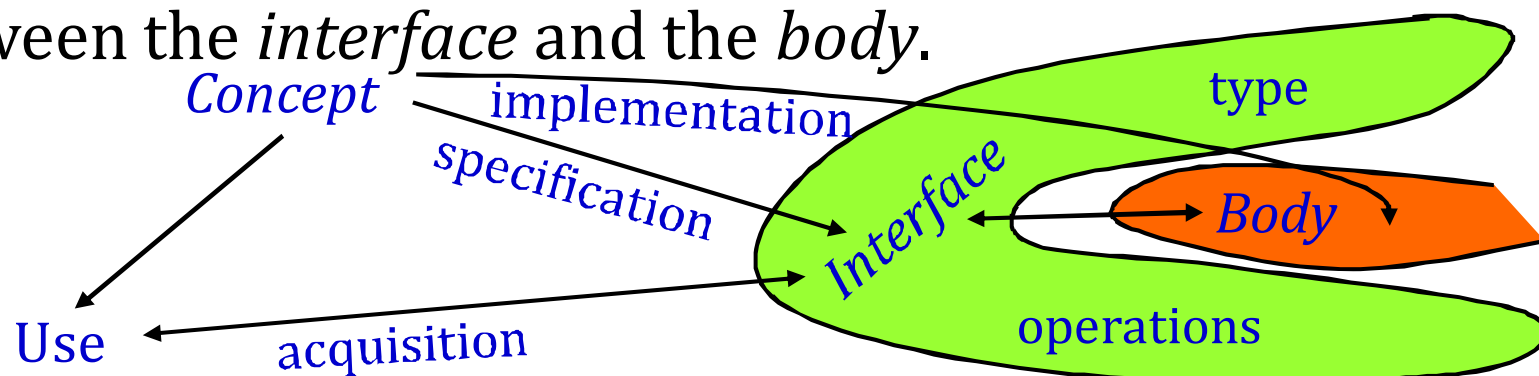
Data abstraction

data objects + operations on the objects

19



- A data abstraction is a user-defined **abstract data type** which encompasses the *representation* of a data type and a set of *operations* for objects of that type.
 - Like procedural abstractions, data abstractions provide **program partitioning** and **information hiding**.
- A crucial ingredient of an abstract data type is separation between the *interface* and the *body*.

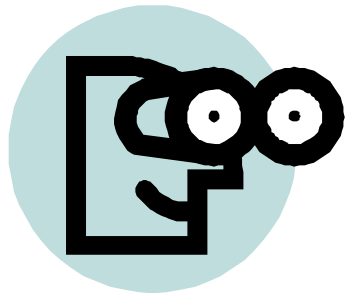


- An interface is like a contract between the users and the designers.
 - a high-level and short specification of the data type and description of the operations provided
- The body implements the specification defined by the interface.

Data abstraction



ADT int



interface

representation: ..., -3, -2, -1, 0, 1, 2, 3, ...

operations: +, -, *, /, &, |, <, >, ==, ...

implementation

representation: 16-bit binary

...
1111111111111111
0000000000000000
0000000000000001
0000000000000010
...

operations: binary add/subtract/multiply/...

```
0000001000100111
+ 0000101010010101
-----
0000110010111100
```

Examples of abstract data types



□ Binary Tree

abstract view: an object which can be queried for its label and for its left and right children associated with operations: *insert, delete, root, left, right ...*

concrete view: a record containing a data field and pointers to its children records with operations: *allocation, deallocation, pointer assignments*

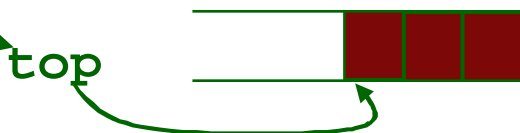


□ Stack

abstract view: an ordered list in which all insertions and deletions are made at one end, called the top (the opposite end is called the bottom), associated with operations: *push, pop, empty?, top_elem, clear ...*

concrete view 1: an array with an additional integer that holds the index of the top. associated with operations: *array assignments*

concrete view 2: a linked list with a pointer that points to the top element associated with operations: *allocation, deallocation, pointer assignments*



Modules for data abstraction

22



- To provide program partitioning and information hiding, data abstractions are typically implemented with **modules**.

Why? → a module provides facilities (e.g., *interface*) to hide the details of data structure representations.

- Example: Stack

- A data abstraction for a stack can be implemented with an abstract data type **Stack** with a module (*a class in C++*).
- Since **Stack** is a data type, it can have objects of that type by declarations. *e.g.*) **Stack stack1, stack2;**
- Programs use the public operations **pop**, **push** and **is_empty**, without being aware of the underlying design decisions such as whether a linked list or an array is used to implement **Stack**.

Modules for data abstraction

23



- Procedural abstractions are provided by languages with block structure.
 - *languages mostly before 80's*(Fortran, C, Pascal)
- Data abstractions are provided by languages that supports modules.
 - *languages in 80's or later*(Ada, Modula-2, CLU, C++)

Moving toward data abstraction



```

struct Element {
    ElemType data;
    Element* next;
};
struct Stack {
    Element* top;
    int num_of_elems;
};
main() {
    Element* d, e;
    Stack s;
    s.top = 0;
    s.num_of_elems = 0;
    . . .
    e = new Element;
    e->data = data;
    e->next = 0;
    s.top = e;
    . . .
    d = s.top;
    s.top = s.top->next;
    data = d->data;
    . . .
    if (s.num_of_elems > 0)
        . . .
}
    
```

```

Element* pop(Stack* stack) {
    Element* t = stack->top;
    stack->top = t->next;
    return t;
}
void push(Stack* stack,
          Element* elem) {
    elem->next = stack->top;
    stack->top = elem;
}
main() {
    Element* d, e;
    Stack s;
    s.top = 0;
    s.num_of_elems = 0;
    . . .
    e = new Element;
    e->data = . . . ;
    . . .
    push(&s, e);
    . . .
    d = pop(&s);
    data = d->data;
    . . .
    if (s.num_of_elems > 0)
        . . .
}
    
```

```

Class Stack {
public:
    Stack() // constructor
        { initialize }
    void push(ElemType* data)
        { . . . }
    ElemType* pop()
        { . . . }
    Boolean is_empty()
        { . . . }
private:
    Element* top;
    int num_of_elems;
};
main() {
    Stack s; // automatically
            // initialized
            // by the constructor
    . . .
    s.push(data);
    . . .
    data = s.pop();
    . . .
    if (s.is_empty())
        . . .
}
    
```

initialize

push

pop

check if it is empty

Original code for stack operations

Procedural abstraction for stack operations

Data abstraction for stack operations

Type security with data abstraction

25



- **Subtypes:** improve *type security* by constraining the set of legal operations on a piece of data. → *But the facilities have limitations.*

```
subtype day_type is integer range 1..31;
var d1, d2, d3 : day_type;
    i : integer;

d1 := -11; // Error detected
d2 := i; // Possible error, but maybe undetectable at run-time.
d3 := d2 + 20; // Possible error, but maybe undetectable at run-time.
```

- **Data abstraction:** offer better security by providing facilities that define a set of legal operations according to semantics of the data type.

```
class Day_Type {
public: int operator+(int c) { 0<c+d<32 ? return c+d : error; }
       int operator=(int c) { 0<c<32 ? d=c : error; return *this; }

private: int d; // private variable for storing the value
}
int i;

Day_type day1 = i; // Error detectable at run-time
Day_type day = day1 + 20; // Error detectable at run-time
```

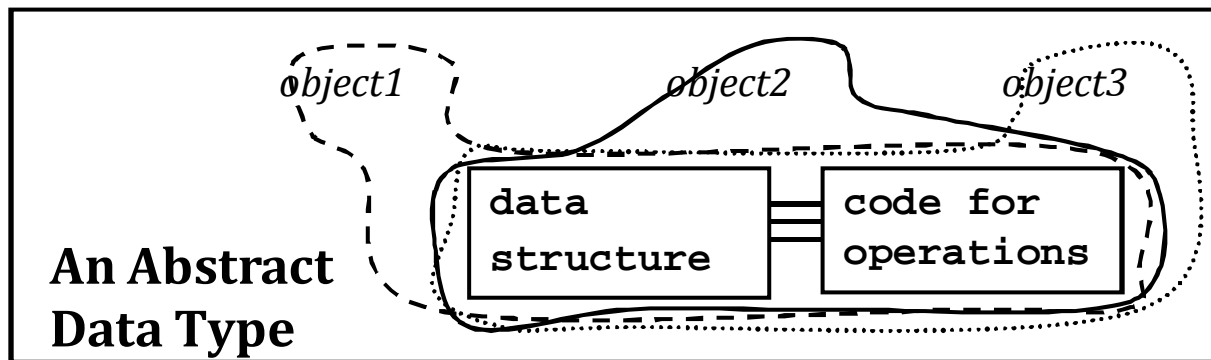
Limitations of data abstraction

The code that implements the operations on the type

26



- In data abstraction, all objects of the same abstract data type use the same **representation** (= *data structure* + code).



e.g.) Data type: **Stack**

- *data structure* → an array
- *code* → array assignments to implement push, pop, and top operations

- In the development of large software, **reuse** of existing representations is essential to increase the productivity.
- However, there may not be a single representation that is the most efficient under all situations.
- Thus, an existing abstract data type usually requires some *modification* in its representation.

Object abstraction as a solution

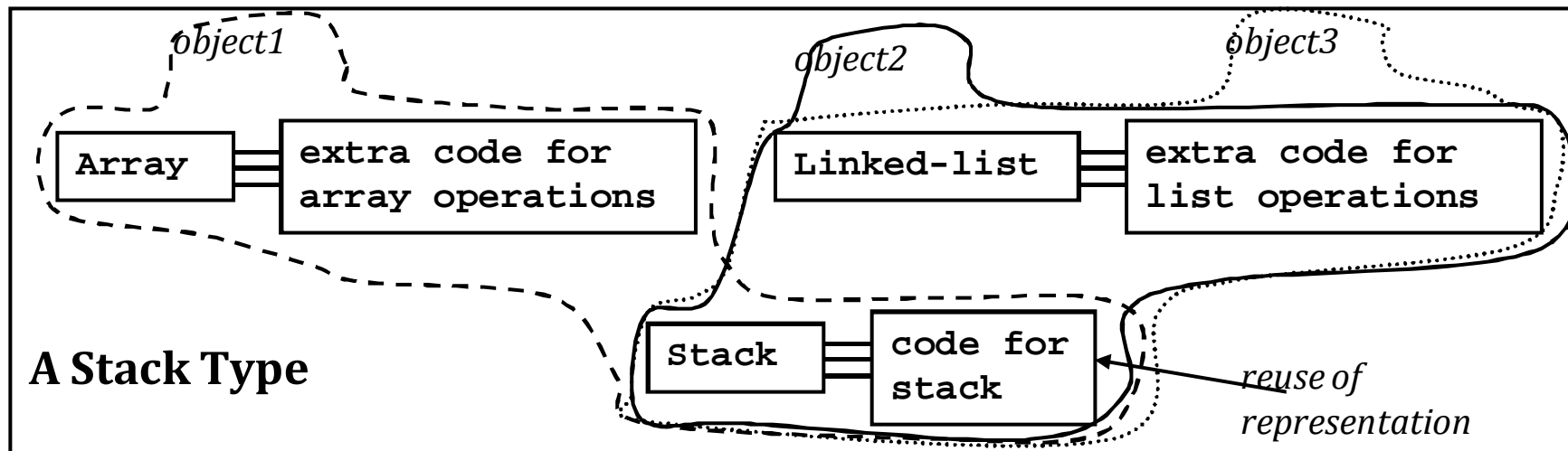
27



- Different situations may prefer the autonomy to choose their own versions of representation **derived** from the same **base** representation that is common to them.

Ex) Someone may want the **Stack** type to be implemented with *arrays* while others want it to be implemented with *linked lists*.

- In **object abstraction**, each object can have a representation different from what other objects of the same type have.
→ *multiple representations*



Object-oriented programming

28



- The language that supports object abstraction is called a **object-oriented programming** language.
- Object-Oriented programming languages
 - Simula 67 : Class concept was first introduced
 - Smalltalk : programming using window system
 - Objective C, **C++** : start from C language
 - Flavor, CLOs (Common List Objective System) : start from Lisp language
 - Turbo Pascal : start from Pascal language
 - Actor
 - Ada 95 : OO extension of the modular language Ada 83

Object-oriented programming

29



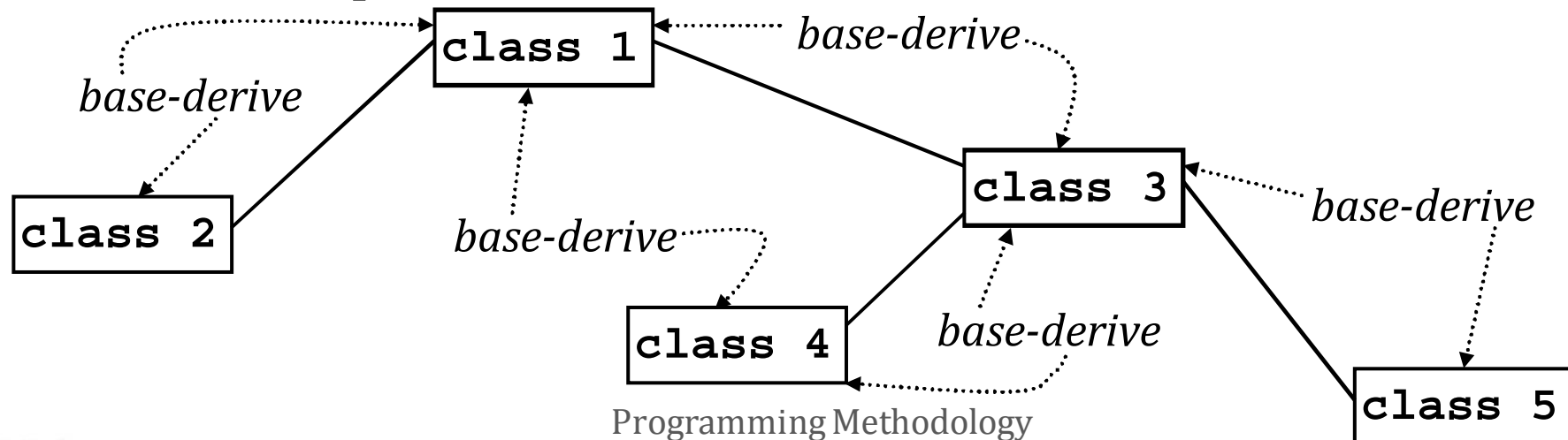
- OO programming treats an overall system as a collection of interacting *objects*.
- Objects are instances of a *data type* (= a **class** in C++ or SmallTalk term).
- The objects interact by sending *messages* to each other.
- Each message is associated with a *method* (or *member function*) in C++ or SmallTalk term.
- Methods are defined by the code in the data type.
- To support object abstraction, a language should provide **data abstraction + type inheritance**.
→ *for multiple representations*

Type inheritance

30



- One data type D inherits the data and operations of other data types $B_1 \sim B_n$.
 - B_i 's are called **base** types and D is their **derived** type.
 - Example of derived types: Subtypes in Pascal and Ada
- In object abstraction, *abstract data types* (= **classes** in C++ terms) can be placed in a hierarchy.
- This hierarchy establishes a *base-derived* class relationship between the parent class and the child class.



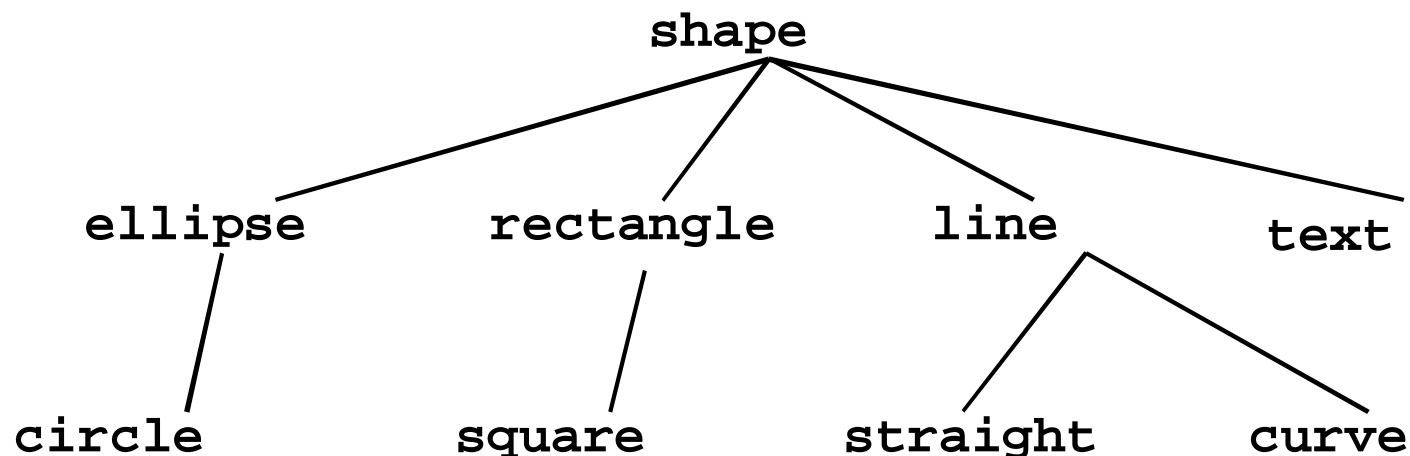
Programming Methodology

Type inheritance

31



- Type inheritance in object abstraction
 - Objects in a child (derived) class can use the representation defined in its parent (base) class.
- Through the inheritance, a class can contain code that can be refined in different ways in different derived classes.
- This provides an effective way to **reuse** code.
- Ex: a hierarchy for graphic objects



An example of type inheritance

32



- We have two classes, each of which represents a record of an employee and that of a manager in a corporate.
- If they are represented in C++, then

```
class Employee {
    char* name;
    int salary;
    int position;           // secretary, president, janitor...
    . . .
    Employee* next;       // points to the next coworker
};

class Manager {
    char* name;
    int salary;
    . . .
    char* department;    // managed by this manager
    Employee* men;       // under this manager
    Employee* next;
};
```


An example of type inheritance

33



- To the language, **Employee** objects and **Manager** objects are completely different.
- But, they have many fields in common.
- In fact, a manager is also an employee in real life.
- So, it would be ideal if a manager object is treated like an employee object with *extra fields*.
- This can be represented more efficiently in C++ as follows:

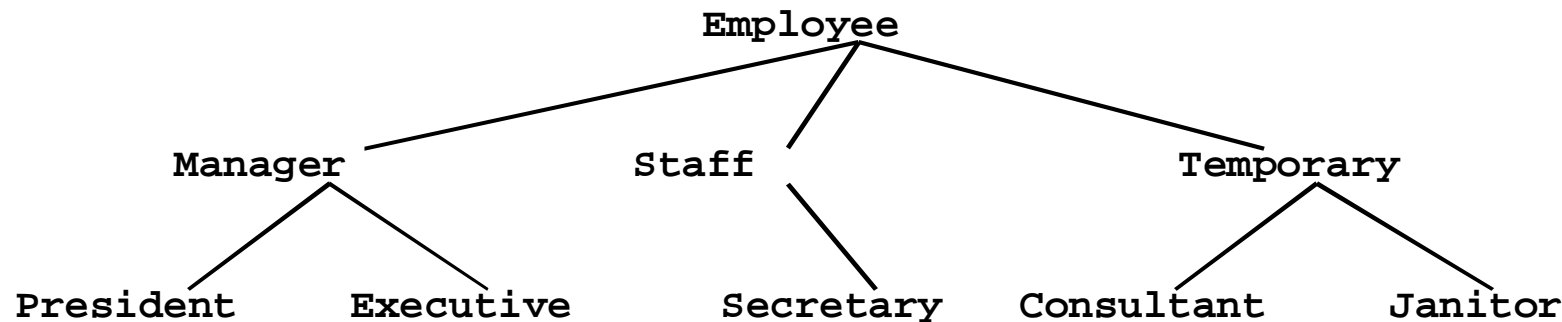
```
class Manager: public Employee {  
    char* department;           // extra field  
    Employee* men;             // extra field  
};                               // The representation is greatly  
                                // simplified with code reuse
```

A hierarchy of employees

34



- Using the base-derived class relationship, the language knows a manager object is derived from the base class **Employee**.
→ So, the data and code defined in **Employee** is reused in **Manager**.
- We can build a hierarchy of employees in a corporate.



- The hierarchy can be represented in the language in terms of base-derived class relationships.

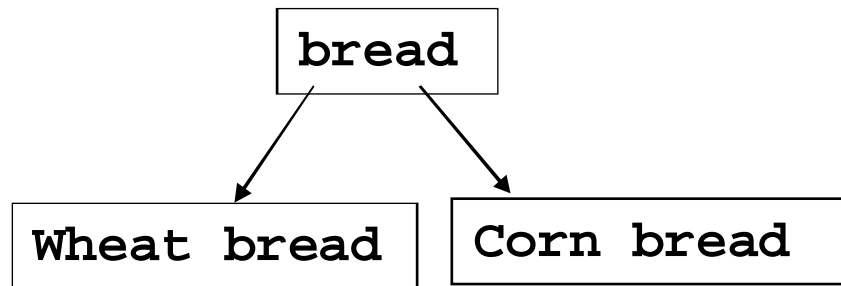
```
class Staff: public Employee { . . . };  
class Temporary: public Employee { . . . };  
class President: public Manager { . . . };  
class Executive: public Manager { . . . };  
class Secretary: public Staff { . . . };  
class Consultant: public Temporary { . . . };  
class Janitor: public Temporary { . . . };
```

Inheritance types

35

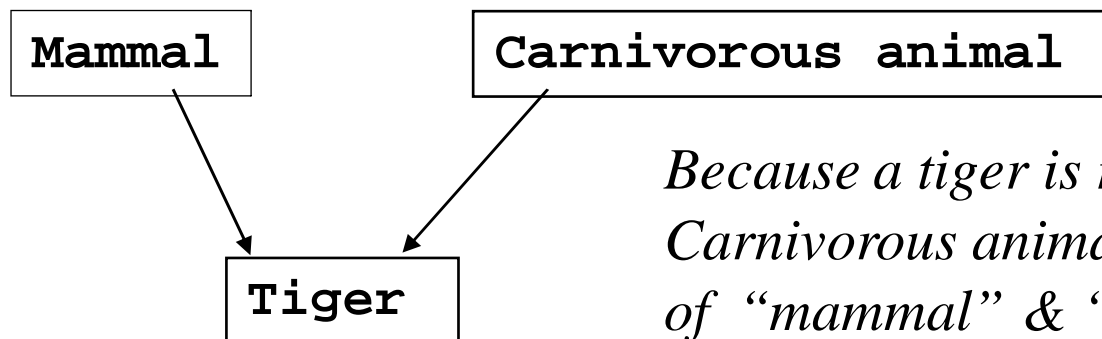


- Single inheritance → *convenient to manage because of its level of tree formation, but it doesn't often reflect real world as it is*



Wheat bread and corn bread are kind of bread so they inherits and uses class of “bread”

- Multiple inheritance → *more flexible in terms of reflection of real world, but it needs very much cautions because of occurrence of problems like collision between inherited forms*



Because a tiger is not only mammal and but also Carnivorous animal, it inherits and uses classes of “mammal” & “carnivorous animal”

Dynamic binding in OO programming

36



- A derived object can be assigned to its base object.
 - That is, a reference variable of a class can point to objects of any class derived from that class.
 - By doing so, a reference variable of a type (base class) is used for different types (derived class) at run-time. → **dynamic binding**

```
void insert_employees() {
    Employee e1, e2, e3, e4, *eptr;
    Manager m1, m2, m3, *mptr;
    Employee* employee_list = 0;           // The list is initially empty
    eptr = &m1;                             // dynamic binding - simply copy the reference of m1
    mptr = &e1;                             // illegal - due to e1's lack of the extra field in m1
    mptr = (Manager*) &e2;                 // forced to be legal, but dangerous
    e2 = m1;                               // both are illegal because member-wise copy
    m3 = e4;                               // is impossible due to their different sizes
    e3.next = employee_list;              // insert the employee e3 to the list
    ...
    employee_list = &e3;
    m1.next = employee_list;              // insert the manager m1 to the list
    employee_list = &m1;                  // dynamic binding!
}
```

→ Note that all other variables in C++ are *statically bound*.

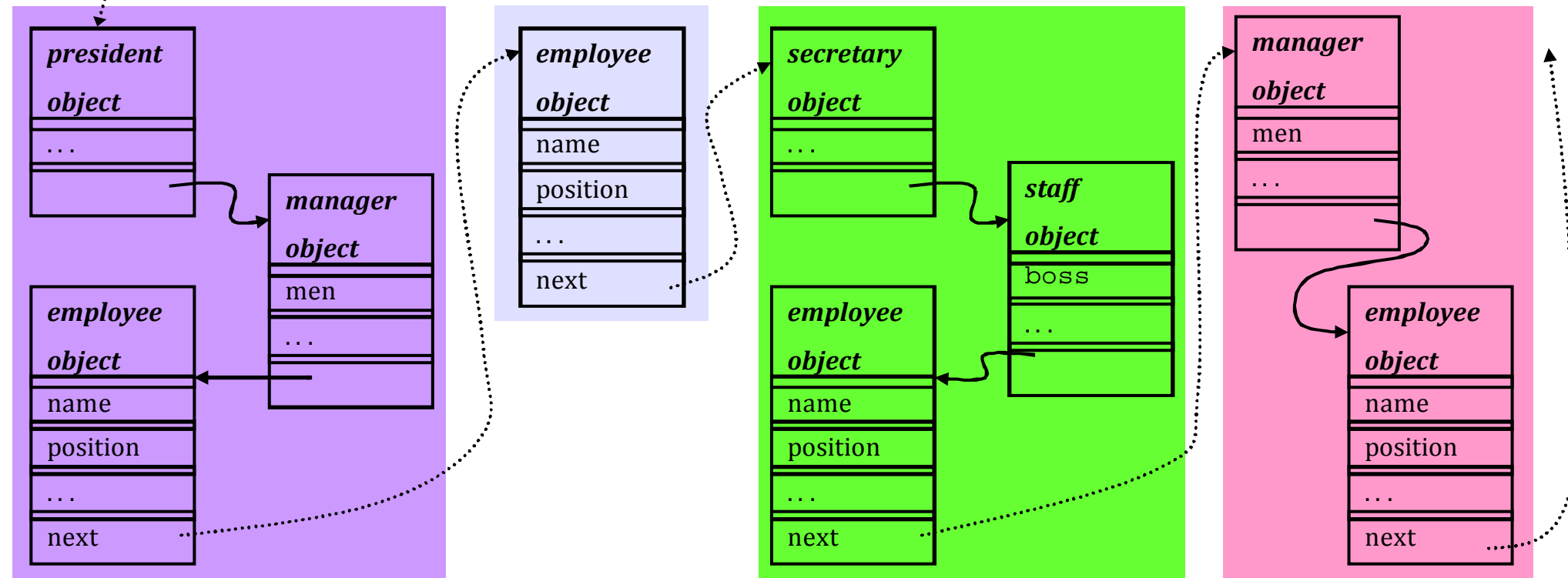
Why dynamic binding?

37



- Using dynamic binding, managers and all other people in a corporate can be treated as employees in the language.

```
void list_names(Employee* employee_list) {  
    for (Employee* e = employee_list; e != 0; e = e->next)next  
        cout << e->name; // e is dynamically bound to employee  
                        // all derived classes of Employee  
}
```



Need more for dynamic binding

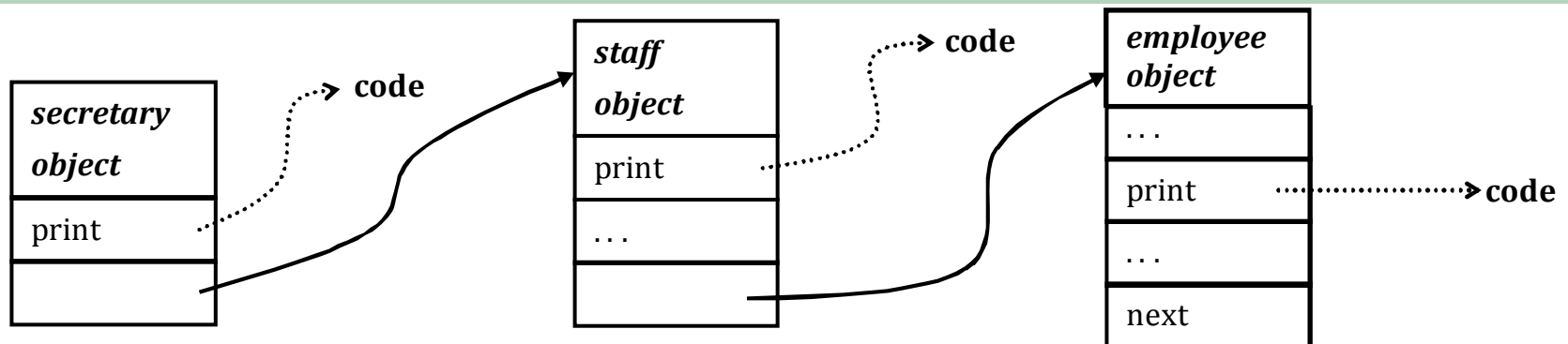
38



- Suppose all objects have the **print** function to print the specific information for each object.

```
class Employee {
    void print() { /* print name, salary, position, ... */ }
};
class Manager: public Employee {
    void print() { /* print name, salary, ..., department, men */ }
};
class Staff: public Employee {
    void print() { /* print name, salary, ..., boss */ }
};

void print_employees(Employee* employee_list) {
    Employee* e = employee_list;
    for (; e != 0; e = e->next)
        e->print(); // ambiguous! → Which print will be invoked for each object?
} // also tedious to define the code for all base prints if the secretary print is only needed
```



Dynamic binding of methods

39



- To choose the right *member function* (or called *method*) **print** for each object, **print_employees** should check the type of the object before it is printed. → *This is awkward!*

```
for (; e != 0; e = e->next)
    switch (e->position) {
        case MANAGER: ((Manager*) e)->print(); break;
        case STAFF: ((Staff*) e)->print(); break;
        . . .
    }
```

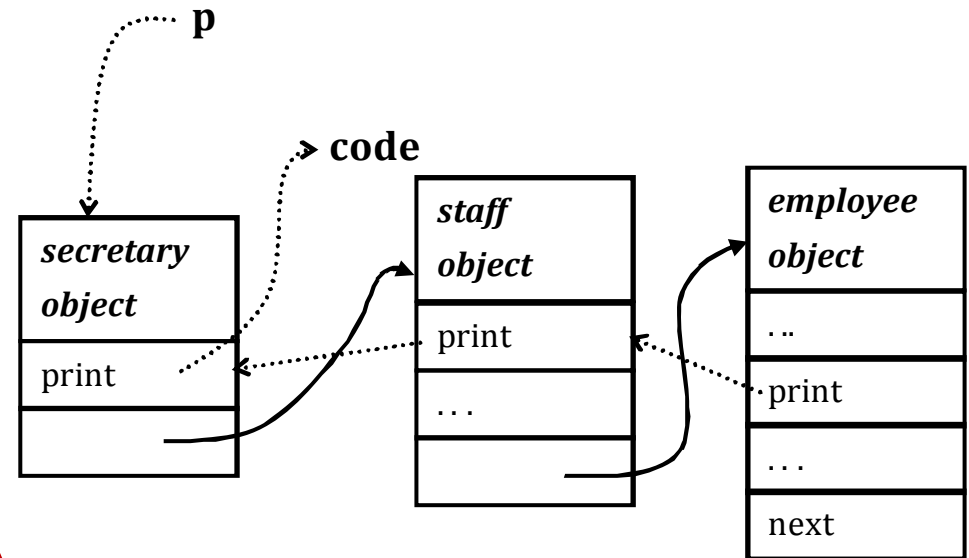
Dynamic binding of methods

40



- To solve the problem of dynamically choosing the specialized *methods* of each object, C++ provides **virtual functions**.

```
class Employee {  
    . . .  
    virtual void print() = 0;  
};  
class Staff: public Employee {  
    . . .  
    virtual void print() = 0;  
};  
class Secretary: public Staff {  
    . . .  
    virtual void print() { . . . }  
};  
.  
.  
Employee* p = new Secretary;  
p->print();  
// Which print is to be used is determined at run-time
```



no need to define if no objects of these types will be actually printed!

// Which print is to be used is determined at run-time

Multiple representations

41



- Type inheritance and dynamic binding enable an abstract data type to have multiple representations.

```
class Stack
{
public:
    Stack();
    virtual void push
        (Element* data);
    virtual Element* pop ();
private: . . .
};
```

```
class Array_Stack: public Stack
{
public:
    Array_Stack (int size);
    void push (Element* data);
    Element* pop ();
private: . . .
};
```

```
class List_Stack: public Stack
{
public:
    List_Stack ();
    void push (Element* data);
    Element* pop ();
private: . . .
};
```

```
int main()
{
    Element* x, y, z;
    // cf: Stack in Ada
    Stack* st1 = new Array_Stack(99);
    Stack* st2 = new List_Stack;
    . . .
    st1->push(x); // insert x to the array
    st2->push(y); // insert y to the linked list
    z = st1->pop();
    ...
}
```

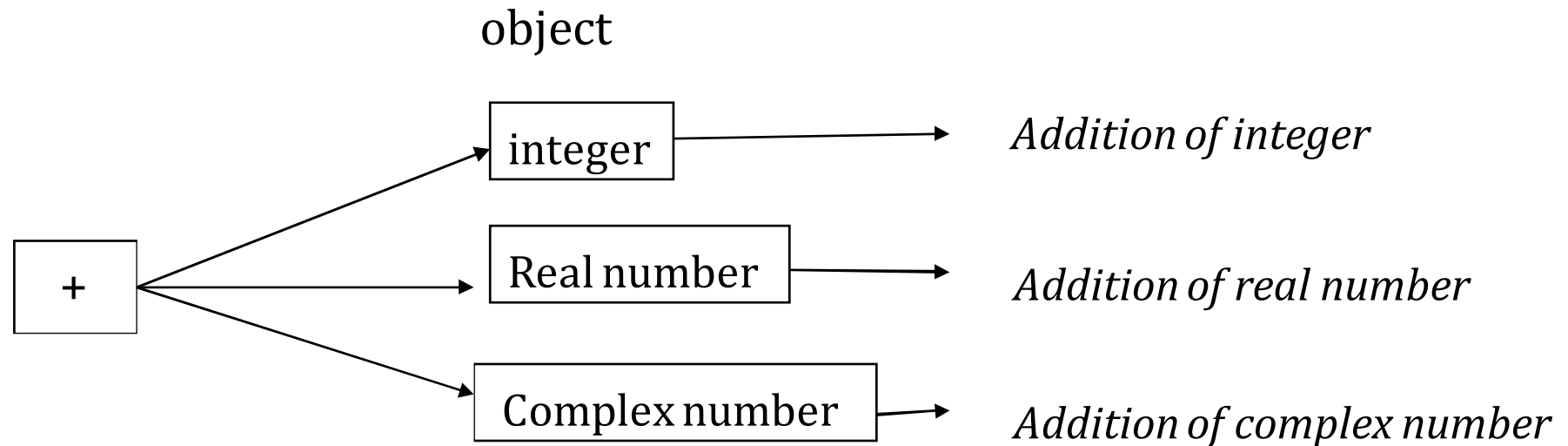
Polymorphic objects in OO languages

42



□ Ad-hoc polymorphism

- Although Object-Oriented programs are different to each other, they send the same messages to the related objects so as to provide the functionality (called *polymorphism* ← read the **type** system) of performing the same operation.
- Overloaded operators



Polymorphic objects in OO languages

43



- Universal polymorphism

- **inclusion** polymorphism → type inheritance (*subtypes, derived classes*)

Ex) **Manager** class objects in C++ ← *derived (sub) type objects*

→ *Type expression for **Manager** objects* = $\left\{ \begin{array}{l} \mathbf{Manager} \\ \mathbf{Employee} \end{array} \right.$

Ex) **Employee::print()** work on objects of all its derived classes

- **parametric** polymorphism → **template**

- Recall ...

- unlike ad hoc polymorphic functions, universal polymorphic functions typically allow the *same code* to be used regardless of the types of the parameters, and

- they exploit a *common structure* among different types.

Ex) **Employee::print()** assumes all objects have **Employee** structure

Parametric polymorphism in C++

44

```
template<class T> class List {           // T is a type variable
    T* list;
    int size;
public:
    List() { list = 0; }                // 0 is polymorphic that can be applied to the unknown type T
    ~List() { delete [] list; }        // delete is polymorphic
    create(int new_size) { list = new T[new_size]; size = new_size; }
    int size() { return size; }
    T& operator[](int i) { return list[i]; }
    void insert(T elem, int pos) { list[pos] = elem; }
    . . .
};
main() {
    List<float> flist;                   flist.create(100);
    List<Complex> clist;                 clist.create(9);
    List<int>  ilist1;                   ilist1.create(200);
    List<int>  ilist2;                   ilist1.create(130);
    List<List<int>> list_ilist;          // a list of lists of integers
    . . .
    flist[29] = 3.43e+20;
    clist1[0] = Complex(3.1, 4.2); // create a complex object and copy it to the list of complex type
    clist1.insert(1, Complex(2.1,9.0)+clist1[0]);
    for (int j = 0; j < 200; j++)
        ilist1[j] = j * 10;
    list ilist[0] = ilist1;
    list ilist[1] = ilist2;
}
```



Parametric polymorphism in C++

45



```
template<class S> List<S>& merge(List<S>& l1, List<S>& l2) {
    // merge the two lists of type S (type variable), and return the merged list
    List<S>* Slist = new Slist;
    Slist->create(l1.size()+l2.size());
    int i;
    for (i = 0; i < l1.size(); i++)
        (*Slist)[i] = l1[i];
    for (int j = i; j < Slist->size(); j++)
        (*Slist)[j] = l2.[j-i];
    return *Slist;
}

. . .
main() {
    List<char> charlist1; charlist1.create(50);
    List<char> charlist2; charlist2.create(70);
    List<Employee> elist1; elist1.create(33);
    List<Employee> elist2; elist2.create(26);

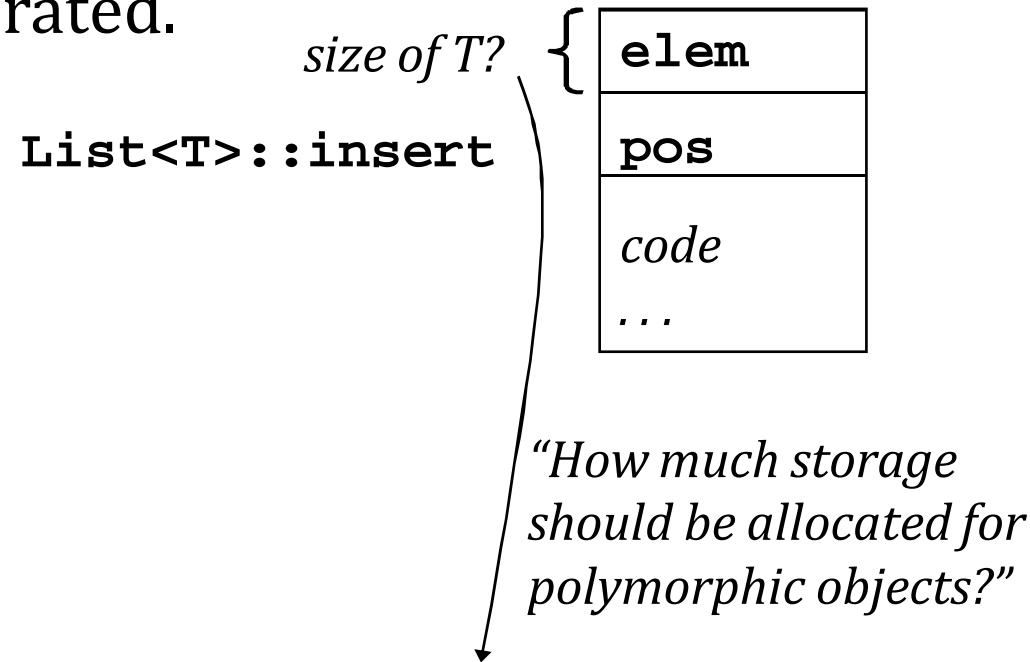
    . . .
    List<char> clist3 = merge(charlist1, charlist2);
    List<Employee> elist3 = merge(elist1, elist2); // merge two employee records
    ...
}
```

Implementing parametric polymorphism

46



- In many languages (C++, Ada), different instantiations of code are to be generated.



4 or 8 bytes if T = int
8 bytes if T = float
... if T = Complex
... if T = Employee

In short...

- ❑ Object-oriented programming associates the object-oriented design concept in software engineering with the programming language.
- ❑ It is used in software system design and implementation.
- ❑ Its primary goal is to improve programmers' productivity and reduce software complexity and management cost as increasing software extensibility and reusability.
- ❑ Key concepts of OO programming
 - ❑ Module (class, package, cluster)
 - Abstract data types and operations
 - Information hiding
 - ❑ Inheritance
 - ❑ Polymorphism

Imperative vs. Object-Oriented

48

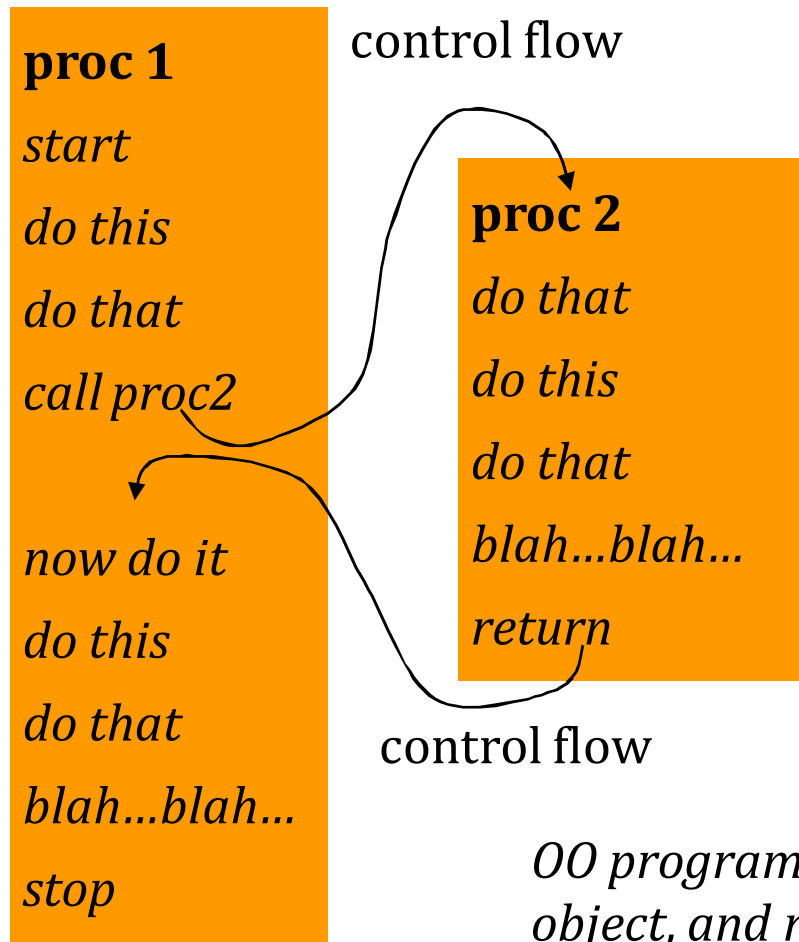


- a procedure
 - a collection of *imperative* orders/instructions with data
 - not first-class valued
 - operations are performed by procedures in imperative programs
 - data is merely the storage where the result of computation is stored
- an object
 - a variable with its own data and methods
 - data represents the current state of the object
 - methods are the operations on the data defined for the object
 - objects in object-oriented programs interact with other objects by exchanging messages
- mapping problem space to program space
 - imperative/procedural programming: bottom-up
 - object-oriented programming: generally top-down

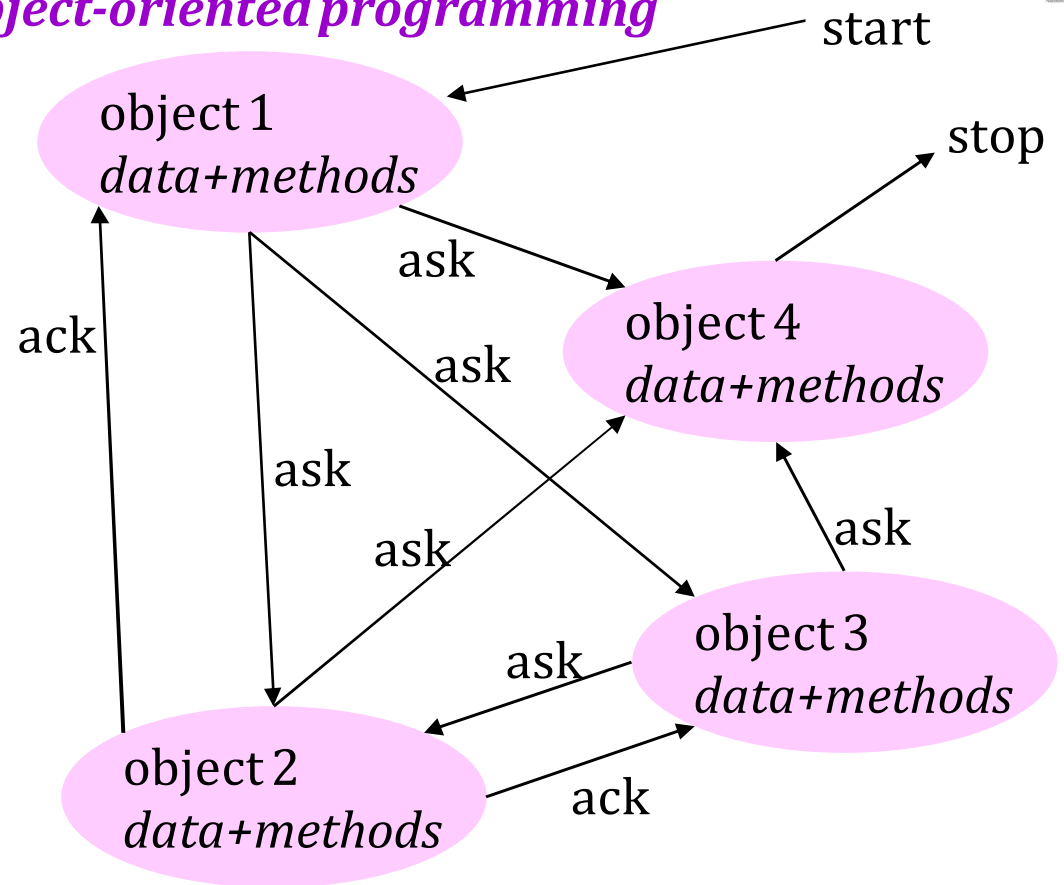
Imperative vs. Object-Oriented



Imperative programming



Object-oriented programming



OO programming regards all in problem area as individual object, and regards system operation for problem area as object operation by message transmission among the objects

Problem solving

50



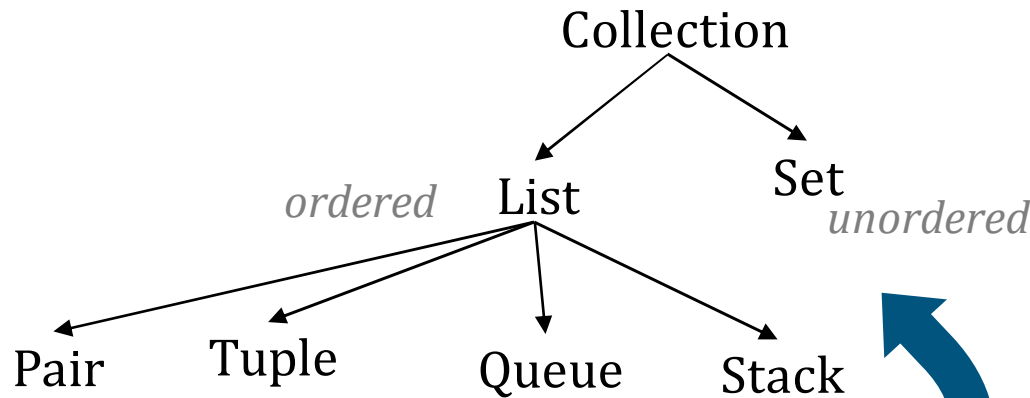
- Mapping problem space to program space
- imperative programming: bottom-up problem solving
 1. design and implement low-level structures: small blocks, loops, data structures, ...
 2. weave together the low-level structures into high-level structures: large blocks, subroutines, ...
- O-O programming: generally top-down problem solving
 1. partition a component in the problem space into several subcomponents
 2. each subcomponent is implemented with a object or a set of objects

Object-oriented problem solving

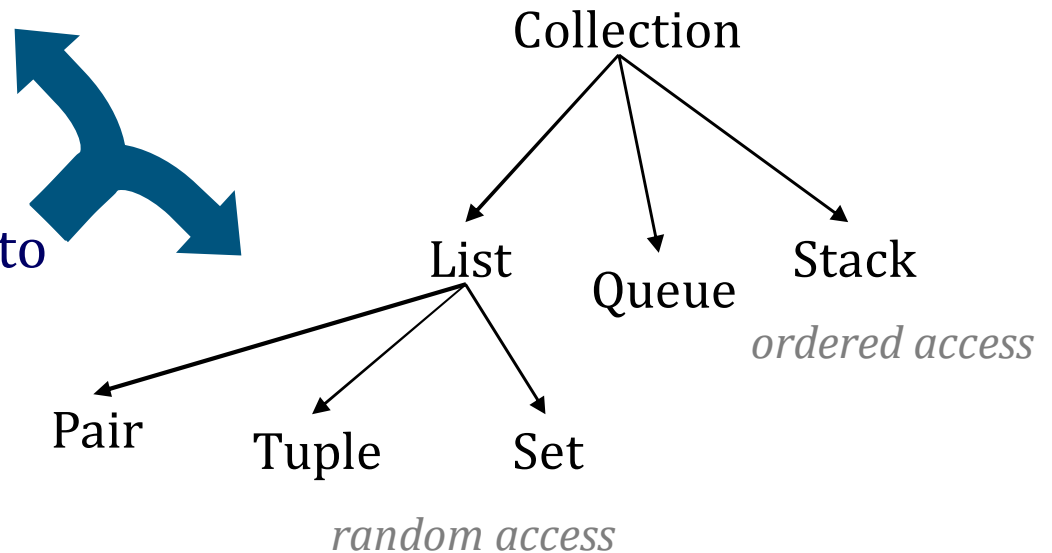
51



- Top level: partition a component in the problem space into several components



Depending on the user's approach to the problem, the way to partition a component may vary.



Object-oriented problem solving

52



- Lower level → *associate a component with characteristics that are common to all of its subcomponents, and define methods for it*
 - collection - a collection of elements, # of elements, empty?, print
 - list - insert, delete, list-print
 - stack - top, push, pop, print-top
 - queue - front, rear, insert, delete, print-front
 - set - insert, delete, union, difference, intersection
 - tuple - order, element-of-*n*th
 - pair - left-insert, right-insert, left-delete, right-delete, print-pair
- The original partitioning of a component determined at the top level will guide the relationship between data and methods and their implementation at lower (bottom) level.

Imperative programming example

53



Problem: “Design a database that maintains the information of employees!”

□ Fortran

```
integer ages(n), salaries(n)
character names(n),
character addresses(n)
...
do I = 1, n , 1
    names(i) = ...
    ages(i) = ...
...
enddo
...
get the index idx of "David"
from names
print *, ages(idx)
```

□ C

```
struct {
    int age, salary, ...;
    char* name, address, ...;
} database[n];
for (i = 0; i < n; i++) {
    database[i].name = ...
    database[i].age = ...
}
// Print the age of an employee "Peter"
idx = 1;
while (!strcmp("Peter",
               database[idx].name)
        && idx < n)
    idx++;
printf("%d", database[idx].age);
// easier and less error-prone than Fortran
// due to the composite data type struct
// but basically the approach is still the
// same: imperative programming
```

O-O programming example

54



□ C++

```
class{
  private:      ← information hiding
    int ages[n], salaries[n], ...;
    char* names[n], addresses[n], ...;
  public:
    void insert(char* name, char* address, ...);
    int age_of(char* name);
    ...
}database;
for(i = 0; I < n; i++)
  database.insert(...);
...
cout << database.age_of("David");
```

Conclusions about OO programming

55



- In OO programming paradigm, each object has some *state*. For computation, objects exchange *messages*.
 - The state of an object is mutated in response to incoming messages.
- OO programming provides programmers with a paradigm to build their programs in a *modular* pattern.
- A good modulation mechanism facilitates ...
 - work partitioning that helps avoid too much interaction bet'n users.
 - maintenance/debugging/refinement of existing programs.
- OO programming is an appropriate programming tool to model many real-world systems because
 - OO programming provides a natural mechanism to break down a program into separate objects.
 - A system in the real world usually comprises a set of physical objects.

OO is everywhere!



- It comes into the spotlight in a various field as computer science and business science.
- Object oriented programming language, that represents the object oriented concept well, is used.
- Object oriented operating systems regard resource and process as independent objects.
- Objected oriented database systems regard data as an independent object and process.
- Object oriented user interface simulation, etc