Chapter 3

Imperative Programming;
Fundamental Programming Concepts

References :

[1] รังสิพรรณ มฤคทัต, กระบวนทัศน์ในการเขียนโปรแกรม (บทที่ 3)

[2] Tucker & Noonan, Programming Languages: Principles and Paradigms (Chapters 4, 5, 9, 11, 12)

[3] Dietel & Dietel, Java: How to Program

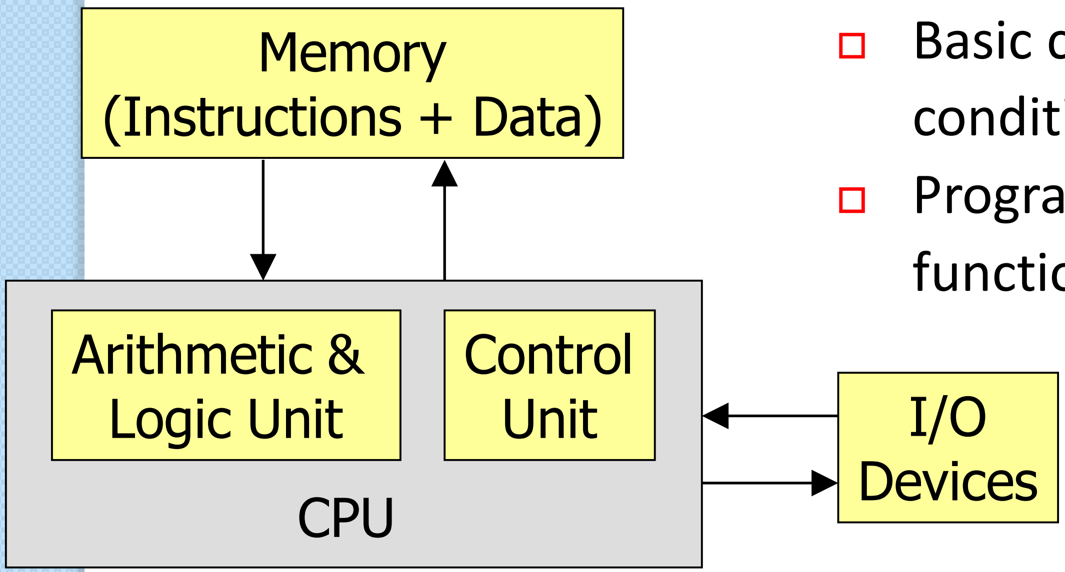
Chapter Objectives

At the end of this chapter, you should be able to:

- Identify scopes and lifetimes of variables in Java programs
- Identify memory locations of variables in Java programs and their relationship with lifetimes
- Explain runtime stack, method call, and argument passing
- Explain heap and object creation
- Write Java programs with arrays

Imperative Programming

- Or Structural / Procedural programming
- Based on von Neumann machine
  - Program as a sequence of commands (imperare = command in Latin) that load, update, and store data in the memory
  - CPU reads and executes the commands one-by-one



- Basic commands : assignments, conditional, branching (& loops)
- Program reusing with library functions

Review of Fundamental Concepts

- For imperative & other paradigms
- Topics
  - Variables & binding, scope, lifetime
  - Type system
  - Data types
  - Statements : conditional, jump, loops
  - Function call, parameter passing
  - Memory management

Variables

- A variable can be characterized by 6 attributes : name, address, value, type, scope, lifetime
  - L-value : address of variable ➔ when var appears on LHS of assignment statement, it refers to memory cell (i.e. address)
  - R-value : value of variable ➔ when var appears on RHS of assignment statement, it refers to content of memory cell
    - x = y + 100;
  - Type : determine range of values a variable can have & set of legal operations for the values of that type, e.g.
    - A[k] ➔ k must be integer
  - Scope & Life time

R. Marukatat

Name Binding

- Bind a name and its attributes (type, value, etc.)
  - Static binding : occur at compile time and remain unchanged throughout program execution
  - Dynamic binding : occur at run time or can change during program execution, e.g. variable references via pointer, calls to virtual functions
- Each attribute may be bound to a name at different times, e.g.
  - Type is bound to a var at compile time
  - But values are bound at run time

Scope

- Range of statements in which a variable is visible
  - There may be > 1 variables (different memory addresses) with the same name
  - Given instruction **a = b + 10** ➔ program must know which variables **a** and **b** refer to, in order to check type and read previous values
- Static or lexical scope
  - Var’s scope can be determined prior to program execution, from source code
  - C, C++, Pascal, Java

- C, C++, Java
  - Local variable is visible within the block it is declared & inner blocks
    - C/C++ : unless inner block declares var of the same name
    - Java : inner block cannot declare var of the same name
  - In C++ and Java, if a variable is declared in the initialization part of a **for** loop, its scope is only within that loop
  - Function's parameter is local to the function
  - Disjoint scopes : local vars in a function cannot be seen by other functions
- Java allows nested classes but not nested methods
  - Members of the outer class can be seen by the inner class

R. Marukatat

Dynamic scope

- Var's scope is only known at runtime, from execution history
- If var declaration/previous reference is not found in current function ➔ search in its caller and so on
- Problems : reliability (variable can be accessed by anyone), type checking task, slow variable access

- Early versions of Lisp used dynamic scope only
- Late versions of Lisp also support static scope

Example : dynamic scope

```
1    void funcA ( )  {
2        x = x + n;
3    }
4    void funcB ( )  {
5        float x = 100.5;
6        funcA ( );
7    }
8    void main( )  {
9        int x = 10, n = 20;
10       funcB ( );
11       funcA ( );
12   }
```

main calls funcB (line 10)	
8 : call main	
9 :	declare x, n
10 : call funcB	
5 :	declare x
6 : call funcA	
2 : x = 120.5	use x, n

main calls funcA (line 11)	
8 : call main	
9 :	declare x, n
11 : call funcA	
2 : x = 30	use x, n

R. Marukatat

Lifetime

- Time interval during which a variable/object holds a block of memory
  - Local variable : single method execution
  - Object : allocation until de-allocation

- Scope ➔ spatial concept
- Lifetime ➔ temporal concept

Scope *vs.* Lifetime

```
void func_1 ( )  {
    int c  = x + 20;     ×
}

void func_2 ( )  {
    int a = x + 20;      ×
    int x = 100;
    int b = x + 20;      ✓
    func_1 ( );
    int d = x + 20;      ✓
}
```

- x's scope is within func_2
- Its lifetime starts at the beginning of func_2 (but can be used after declaration point), extends over the time func_1 is called, and continues after func_1 returns
- Its lifetime ends when func_2 returns

R. Marukatat

Type System

- A well-defined system of associating types, values, and possible operations with variables and objects
- Statically typed language (C, C++, Java)
  - Object's type is determined at compile time
  - At run time, object has 1 type throughout its lifetime
- Dynamically typed language (Lisp)
  - Object's type can change during program execution
  - Type checking at run time → high runtime overhead

- Strongly typed language
  - Always detect type error (either at compile time or run time) before it actually occurs
- Weakly typed language
  - Allow some ambiguity when checking type, e.g. union in C
  - So programs may be unreliable or give wrong results

- Java ➔ statically typed + strongly typed
- C, C++ ➔ statically typed + weakly typed

Data Type

- Can be broadly divided into 2 groups
  - Elementary type : data is stored in fixed memory space
  - Structured type : memory space has variable size, depending on structure of data to be stored

Comparing Elementary Type

Type	C, C++		Java	
Byte	Not exist		byte	8 bits
Integer	short int long	16 bits 32 bits 64 bits	short int long	16 bits 32 bits 64 bits
Real	float double	32 bits 64 bits	float double	32 bits 64 bits

R. Marukatat

Type	C, C++	Java
Character	char ASCII code : 8 bits	char Unicode : 16 bits
Boolean	Not exist	boolean Value = true/false, not 0/1
Pointer	*	Not exist
Reference	& (only in C++)	Exist but used implicitly

Type Conversion

- Type conversion may be implicit (type coercion) or explicit

double d; int j, k;	
k = d / j;	implicit narrow casting : double to int
k = (int) d / j;	explicit narrow casting
d = j / k;	implicit widen casting : int to double
d = (double) j / k;	explicit widen casting

- Some information may be lost by narrow casting
- When the expression (RHS) is evaluated, all data types are checked → type of expression = biggest data type

R. Marukatat

Type Conversion in Java (Casting)

- Conversion between primitive data types is done by type casting
- All primitive data types, except boolean, can be casted from or to the others
  - Widen casting : can be either implicit or explicit
  - Narrow casting : explicit casting only

Type Conversion in Java (Wrapper)

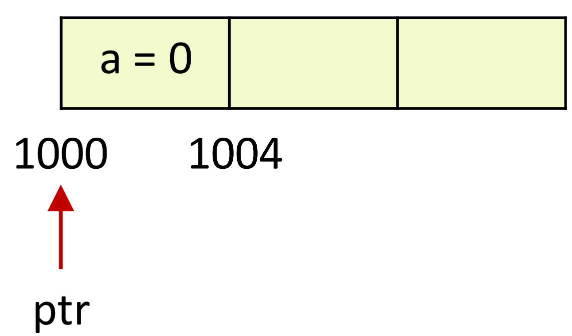
- Conversion between number (primitive) and string is done via wrapper class
  - Wrapper class wraps primitive data, allowing it to be manipulated as an object
- If "+" is placed between number and string, the number is automatically converted to string for string concatenation
  - String word = "AAA" + "BBB" // "AAABBB"
  - String word = "AAA" + 123 // "AAA123"
  - String word = 123 + "" // "123"

R. Marukatat

Pointer

- Variable that holds memory address of, or points to, another variable or object

```
int a = 0, b, *ptr;
ptr  =  &a;          // referencing (ptr points to a)
b    =  *ptr;        // de-referencing (content of var pointed by
                     // ptr, which is a)
ptr++;               // point to a's address + 4
```



Reference in Java

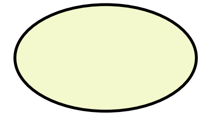
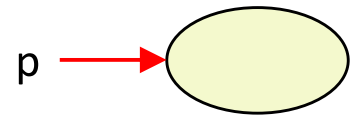
- Replace pointer
- Reference are used implicitly, details & mechanisms are hidden from programmer
  - Same syntax for primitive and reference variables
- Arrays & objects are handled via reference variables
  1. Reference var declaration

     ```
     MyClass ref;                // ref holds null
     ```

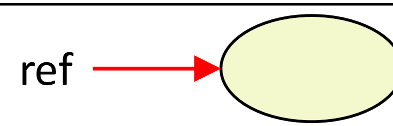
  2. Object allocation

     ```
     ref = new MyClass();        // create object in heap
                                 // ref holds its address
     ```

Object and pointer in C/C++

MyClass ob;	
MyClass *p;	p → null
MyClass *p = new MyClass(); p = &ob;	

Object and reference in Java

MyClass ref;	ref → null
MyClass ref = new MyClass();	

Array in Java

- Depending on language, array can be allocated in static area, run-time stack, or heap
- Array in Java is treated as an object. It is allocated in heap. Each array's element is assigned initial value automatically
  - Number initial value = 0
  - Char initial value = '\u0000'
  - References to object initial value = null
  - Boolean initial value = false
- Index of the first element is zero

R. Marukatat

Array of primitive data / strings

```
int [ ] num = new int [5];                      // allocate array
String [ ] words = new String [3];

int [ ] num = {1, 2, 3, 4, 5};                  // allocation & initialization
String [ ] words = {"AA", "BB", "CC"};
```

Array of objects

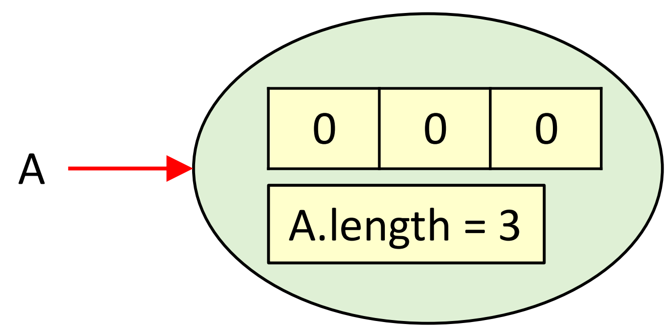
```
MyClass [ ] mems = new MyClass [5];             // allocate array
for (int i=0; i < mems.length; i++)             // length = size of array
    mems[i] = new MyClass( );                   // allocate each member
```

R. Marukatat

Array in C/C++

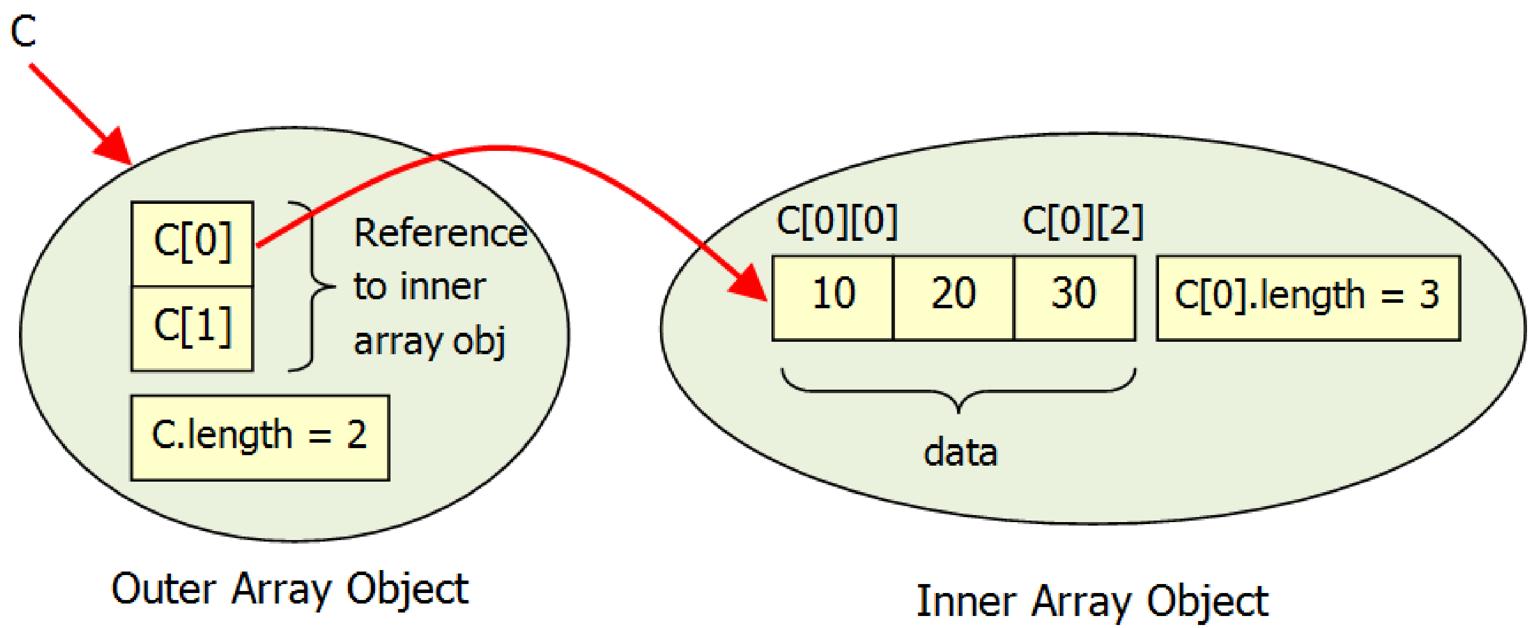
int A[3];	No initialization. Members contain junk values junk | junk | junk
int *p = new int[3];	p → junk | junk | junk

Array in Java

int [ ] A = new int[3]; int A [ ] = new int[3];	Members are automatically initialized to 0 Length is a special field in array object 

2-Dim array

```
int [ ][ ] C = new int [2][3];                    // allocate array
for (int row=0; row < C.length; row++)
    for (int col=0; col < C[row].length; col++)
        C[row][col] = (col + 1) * 10;
```



Character in Java

- Use 16-bit Unicode (\u) instead of ASCII code
  - Represent more alphabet & symbol than ASCII
  - \u0000 - \u007F ASCII characters
  - \u0E00 - \u0E7F Thai characters
  - Escape character : similar to C, C++ (\b, \t, \n, \r)
- From ASCII to Unicode : add 0 in 1st byte
- From Unicode to ASCII : drop 1st byte, keep 2nd byte

Unicode characters can be encoded (e.g. using UTF8, UTF16) before actually stored in memory. So \u0000 - \u007F chars use only 1 byte

String in Java

- Java string is an object, a sequence of characters, rather than array of characters as in C / C++
  - Do not have null character at the end of the string
  - E.g. consider a string "hello"

C++ (ASCII)	68 65 6C 6C 6F **00**
Java (Unicode)	0068 0065 006C 006C 006F

  - Print character 'e'

C++	cout << **str[1]** << endl;
Java	System.out.println(**str.charAt(1)**);

Structured Data Type

- Record (struct / class) = group of heterogeneous vars
  - Members may have different types and are not necessarily allocated in consecutive memory locations
  - Record's size = sum of all members' sizes
- Variant record (union) = group of alternative vars
  - Members may have different types
  - All members share the same memory space ➔ one member can be stored/accessed at a time
  - Union's size = size of the largest member
  - Usually cause type safety problems

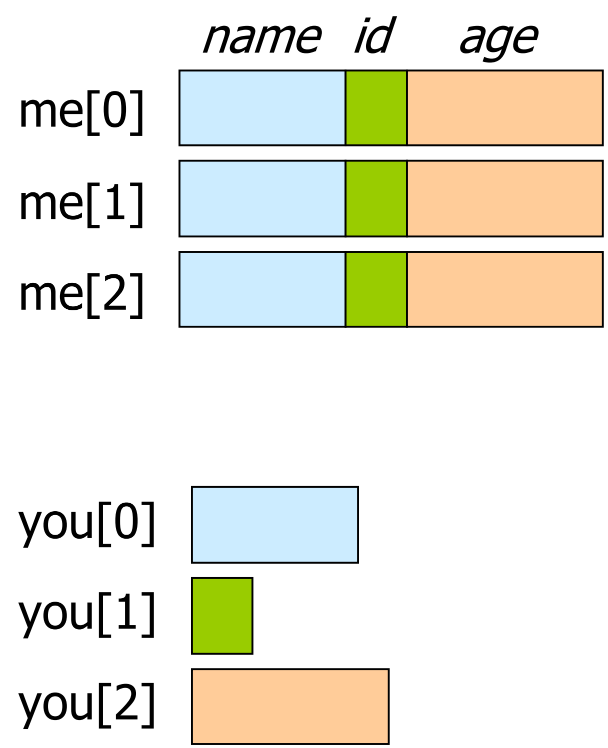
Record *vs.* Union

```
struct mydata
{
      char    name [20];
      int     id;
      float   age;
};

struct mydata me [3];
for (i=0; i<3; i++) {
      cin >> me [i].name;
      cin >> me [i].id;
      cin >> me [i].age;
}
```

```
union yourdata
{
      char    name [20];
      int     id;
      float   age;
};

union yourdata you [3];
cin >> you [0].name;
cin >> you [1].id;
cin >> you [2].age;
```



Operators in decreasing precedence (1)

Operation	C	C++	Java
Scope resolution	Not exist	::	Not exist
▪ Member selection ▪ Subscript ▪ Method Call ▪ Size of object or type	. -> [ ] ( ) sizeof	. -> [ ] ( ) sizeof	. [ ] ( ) Not exist
▪ Unary negation ▪ Complement ▪ Bitwise complement ▪ Increment, decrement ▪ Address of ▪ De-reference ▪ Allocate ▪ De-allocate ▪ Cast	- ! ~ ++ -- & * Not exist Not exist ( )	- ! ~ ++ -- & * new delete ( )	- ! ~ ++ -- Not exist Not exist new Not exist ( )

Operators in decreasing precedence (2)

<table>
<tr><th>Operation</th><th>C</th><th>C++</th><th>Java</th></tr>
<tr><td>Multiply, divide, modulo</td><td>* / %</td><td>* / %</td><td>* / %</td></tr>
<tr><td>Add, subtract</td><td>+ -</td><td>+ -</td><td>+ -</td></tr>
<tr><td>Left shift, right shift</td><td><< >></td><td><< >></td><td><< >></td></tr>
<tr><td>▪ Less than, less or equal
▪ Greater, greater or equal</td><td>< <=
> >=</td><td>< <=
> >=</td><td>< <=
> >=</td></tr>
<tr><td>Equal, not equal</td><td>== !=</td><td>== !=</td><td>== !=</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>&</td><td>&</td></tr>
<tr><td>Bitwise XOR</td><td>^</td><td>^</td><td>^</td></tr>
<tr><td>Bitwise OR</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>&&</td><td>&&</td></tr>
<tr><td>Logical OR</td><td>||</td><td>||</td><td>||</td></tr>
<tr><td>Assignment</td><td>=</td><td>=</td><td>=</td></tr>
</table>

Statements

- Similar to C/C++, with some changes

Statements	C	Changes in Java
Expression	See previous slide	Some different operators
Conditional	if-else, switch-case	Switch can test string value Arrow-case (no fall through)
Loop	for, while, do-while	for-each loop
Arbitrary jump	goto	Not exist as it causes unreliability in the program
Others	return, break, continue	Same as C
Function call	Similar form but different mechanisms	

Memory Management

Static memory

- Store values whose memory requirements are known at compile time & remain constant throughout program execution, e.g. global and static variables

Run-time stack

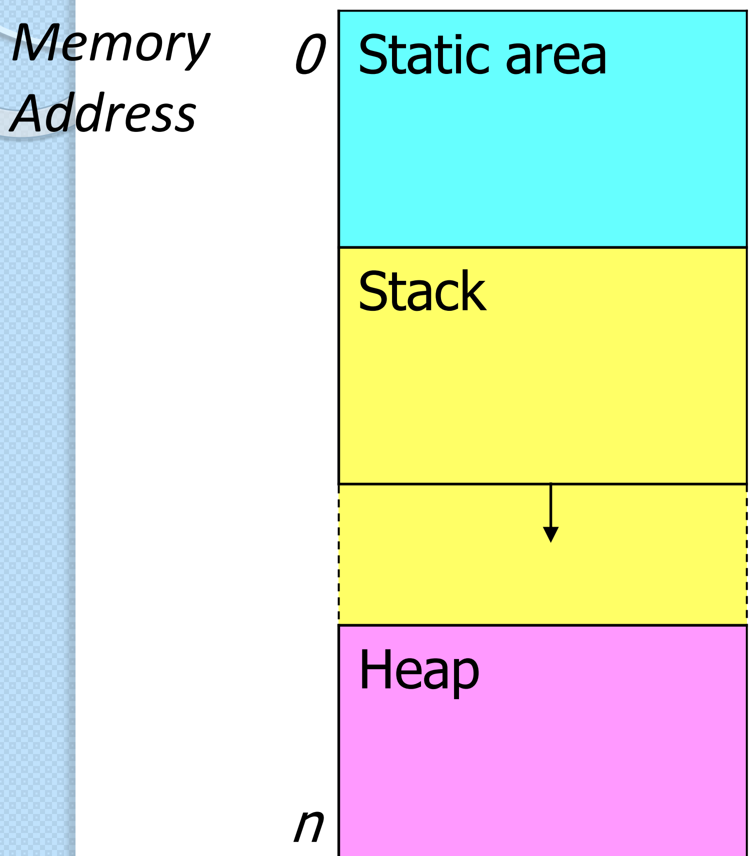
- Used for passing execution control between active functions
- Store parameter-argument linkage and local variables

Heap

- Store values allocated at run time
- Dynamic allocation / de-allocation causes fragmented heap, so garbage collection is needed to manage available space

R. Marukatat

Program's Address Space (Run Time)

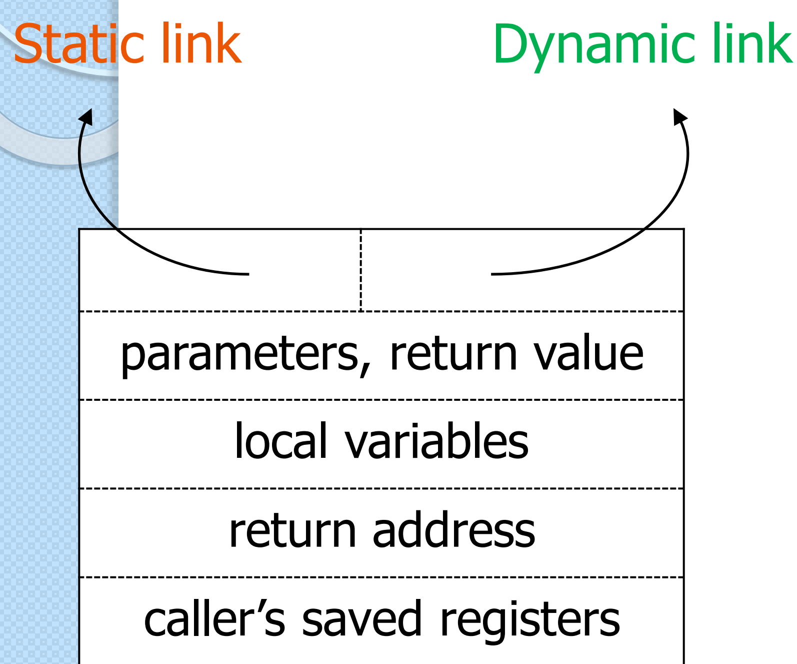


- Size of static area must be known before run time
- Heap size is estimated at the beginning of program execution
- If stack grows beyond its estimated boundary, e.g. non-terminating recursive call ➔ stack overflow

Based on Tucker & Noonan [2], Ch.11
(others may show different orders of static area, stack, heap)

R. Marukatat

Stack Frame (Activation Record)



Stack frame of
a called function

Based on Tucker & Noonan [2], Ch.9

- Pushed on run-time stack when the function is called
- Static link : to static parent (block of program that defines this function) ➔ to access non-local vars
- Dynamic link : to stack frame of caller ➔ record execution chain
- In dynamic scoping language, var's scope is checked by traversing dynamic links

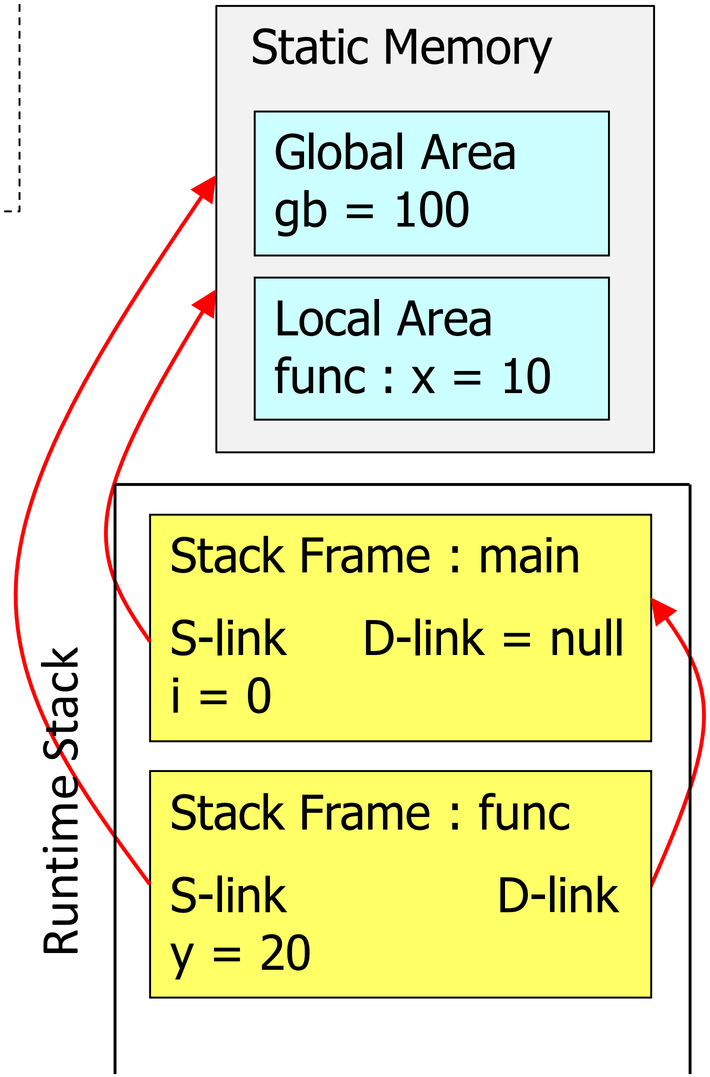
R. Marukatat

```
int gb = 100;

void func( )  {
   static int x = 10;
   int y = 20;
   gb++;  x++;  y++;
   cout  <<  gb <<  “  ”
          <<  x   <<  “  ”
          <<  y   <<  endl;
}

void main( ) {
   for (int i=0; i < 3; i++)  {
      func( );
      gb = gb + 100;
   }
}
```

gb	x	y
101	11	21
202	12	21
303	13	21



Functions / Methods

- Method declaration usually consists of
  - Method name
  - Return value
  - List of formal parameters
    - Arguments (actual parameters) = values being passed to corresponding parameters when the method is called
  - Modifiers or prefixes
- Argument passing
  - By value
  - By reference or by address
- There is only passing by value in Java

Passing Primitive Values in C / C++ / Java

Function_1

var

100

address_1

Function_2

param

100

address_2

- Function_1 sends variable's content
- Function_2 copies argument's value to its variable param

Both functions handle different variables

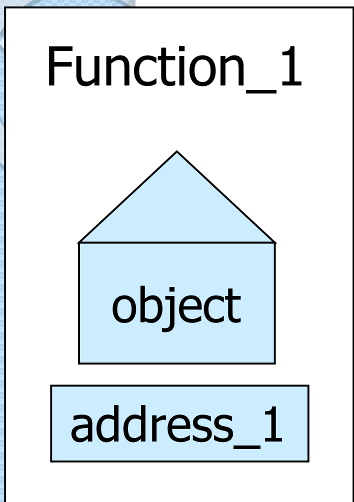
```
int var = 100,  *p;
p = &var;               // C, C++

Function_2 (var);
Function_2 (*p);        // C, C++
```

```
void Function_2 (int param)
{
    param = param + 50;
}
```

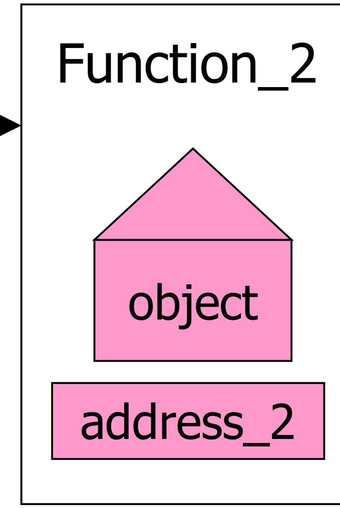
Passing by values for all languages

Passing by Values in C / C++



- Function_1 sends object
- Function_2 copies argument's value to its object param

Both functions handle different objects

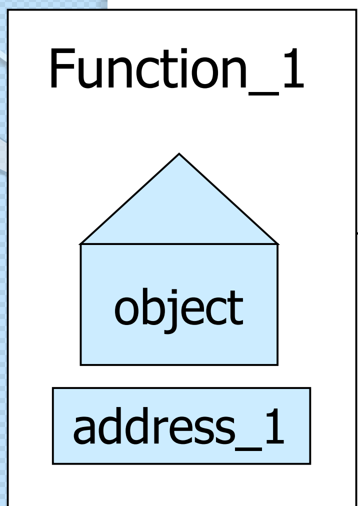



```
MyHouse  ob,  *p;
p = &ob;

Function_2 (ob);
Function_2 (*p);
```

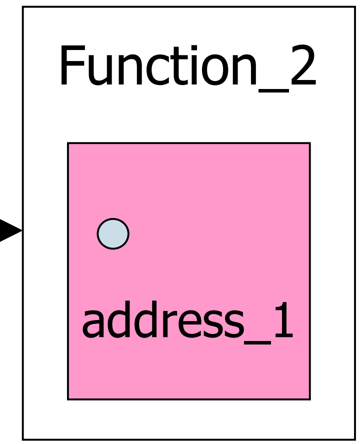
```
void Function_2 (MyHouse param)
{
    param.action();
}
```

Passing by References in C / C++



// **C** : send object's address
Function_2 (**&ob**);
Function_2 (**p**);

// **C++** : send object
Function_2 (**ob**);
Function_2 (***p**);

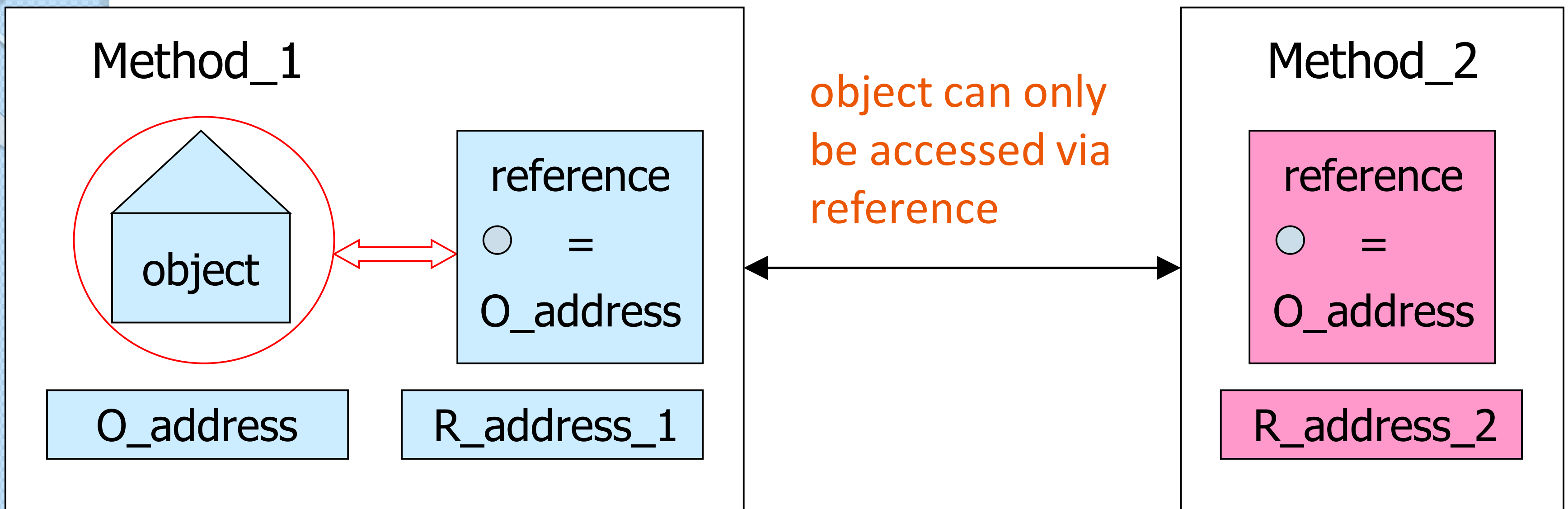


// **C** : copy argument's value (i.e. address) to pointer param
// Simulate the effect of, but not purely passing by reference
void Function_2 (MyHouse ***param**) { param**->**action() }

// **C++** : copy only argument's address to reference param
void Function_2 (MyHouse **&param**) { param**.**action(); }

Both functions handle the same object

Passing by Value in Java




```
// ob, obs hold object's address
MyHouse  ob;
Method_2 (ob);

MyHouse [] obs = ...
Method_3 (obs);
```

```
// copy argument's value (i.e. address)
// to param, params
void Method_2 (MyHouse param)
{  param.action();  }

void Method_3 (MyHouse [] params)
{  for (MyHouse h : params)  h.action();  }
```

Garbage Collection

- Was first developed to facilitate dynamic run-time activities of functional languages
  - Lisp is the first language to have garbage collection
  - General idea : checking & collecting orphan nodes

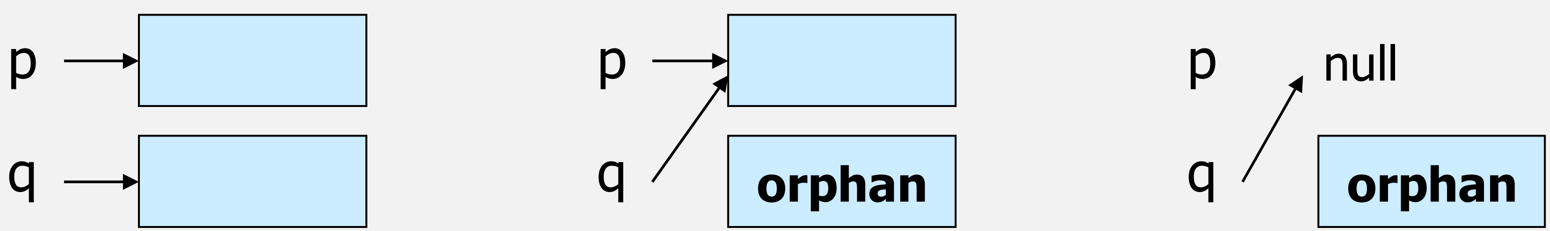
Widow & Orphan

```
class node { … }

node *p, *q;
p = new node();
q = new node();
```

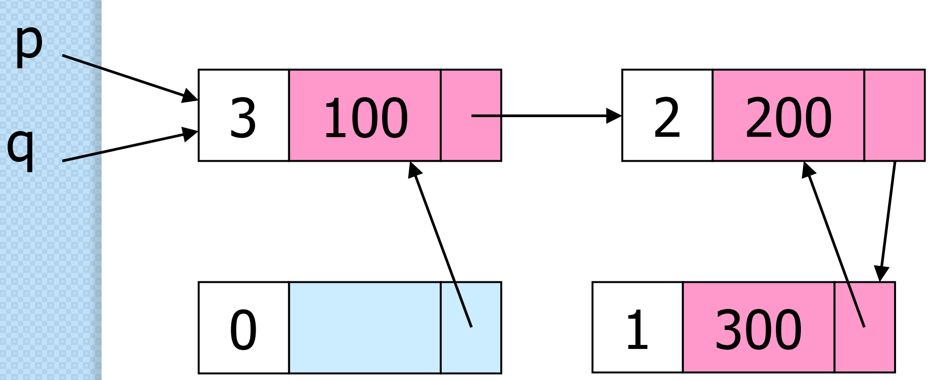
```
q = p;
```

delete(p);
// q = **widow**



Reference Counting

- Heap is a linked list : each node consists of data cell, pointers to other nodes, and RC which counts no. of references to this node
  - RC is updated through out program execution
  - Orphan node : RC = 0 ➔ when it is collected, RC of nodes previously pointed to by this node is decreased by 1



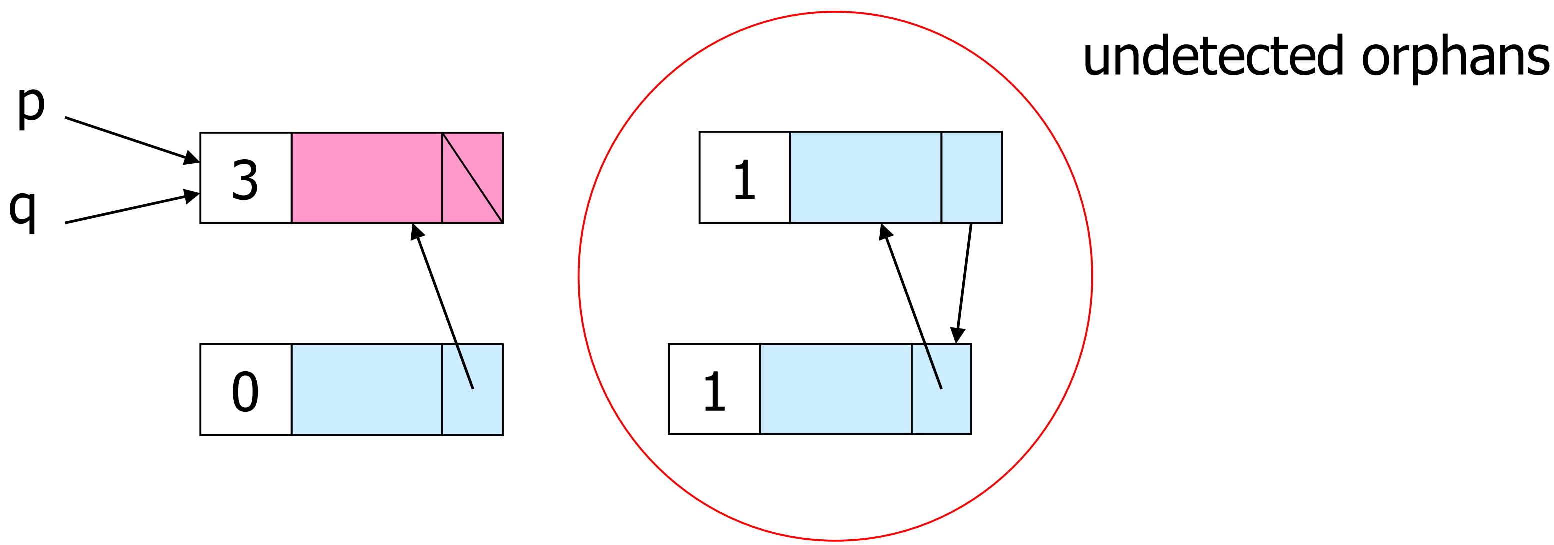

```
class NA {
  int data;
  NA *next;
}
NA *p = new NA(100);
NA *q = p;
p->next = new NA(200);
```

Advantages

- GC is invoked automatically upon pointer assignment
  ∴ overhead is distributed over program execution

Disadvantages

- Memory overhead to store RC
- Inaccessible circular chain cannot be detected
  E.g. if we execute p->next = null;



Mark & Sweep

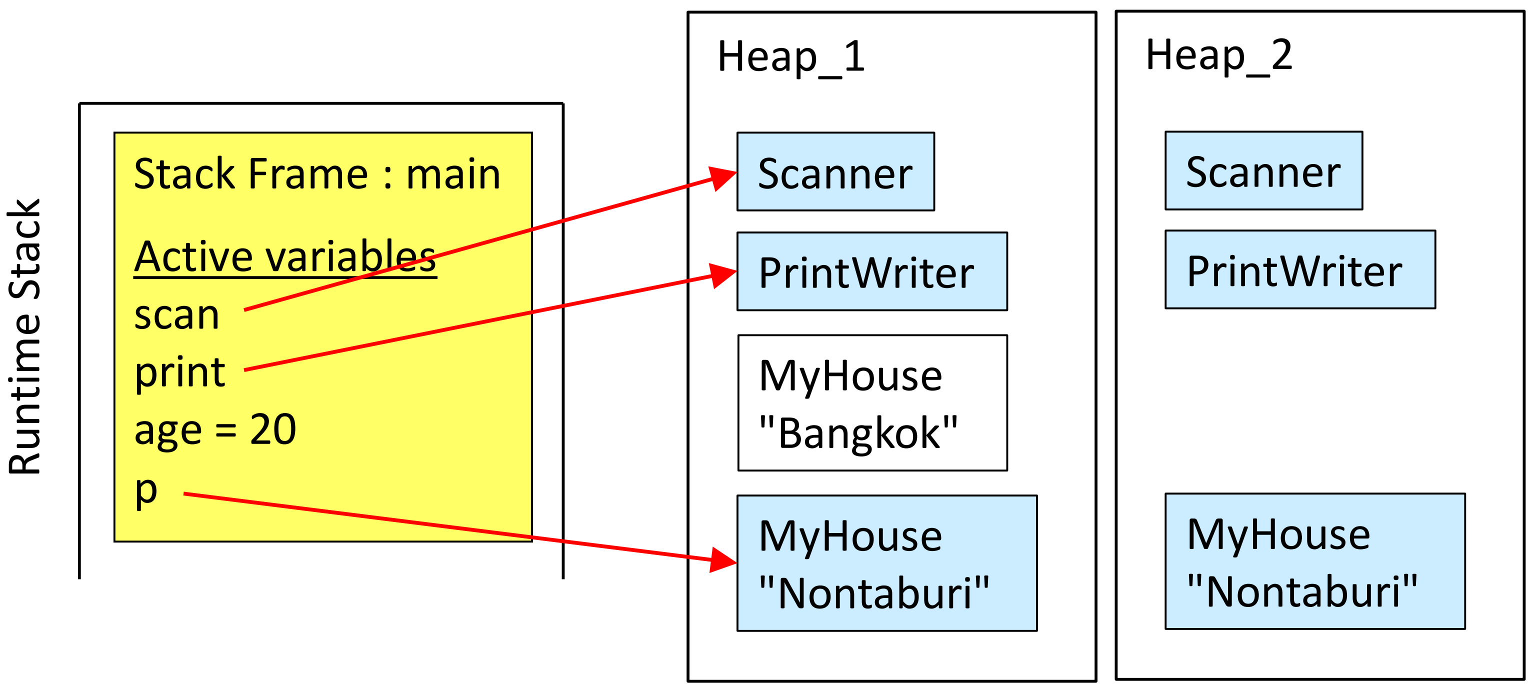
- Invoked once pointer assignment is executed but heap is full. Work in 2 steps
  - **Mark Step** : all active vars are checked → heap nodes that can be reached by them are marked
  - **Sweep Step** : all heap nodes are checked → unmarked nodes are returned to free list
- Advantages
  - Guarantee that all garbage can be reclaimed
- Disadvantages
  - Memory overhead to store accessible flag
  - High overhead since it has to check all vars & all heap nodes

Copy Collection

- Invoked once pointer assignment is executed but heap is full
- Only half of heap is used at a time
  - Suppose that we have heap_1 and heap_2
  - Use heap_1 as working heap until it is full
    - Copy accessible nodes to heap_2 (similar to Mark step)
    - Heap_2 has accessible nodes & free nodes left because orphan nodes from heap_1 is not copied
    - Use heap_2 as working heap until it is full
- Quicker than Mark & Sweep but require larger heap

```
public static void main(…) {
    Scanner scan = new Scanner(System.in);
    PrintWriter print = new PrintWriter(System.out);
    int age = 20;
    MyHouse p = new MyHouse("Bangkok");
    …
    p = new MyHouse("Nontaburi");
    MyHouse q = new MyHouse("USA");
}
```

garbage collector invoked !!



Garbage Collection in Java

- Garbage collection runs as a background process
  - Low-priority process, i.e. run when demand on processing time from other threads is low
- System.gc() can be called explicitly – invoke GC regardless of the current state of the heap