Programming Methodology

Spring 2009

Variables

교육과학기술부
국가지정연구실
National Research Lab

Topics

Components of a variable

- Name
- Assignment
- l-value and r-value
- Scope & Life time

Arrays

Constant variables

Programming language constructs

```
int foo(char* l) {
    int j, k;
    float x, y, z;
    …
    j = 0;
    while (j < n) {
        x = 2.0 * y + z;
        …
    }
    …
}

main() {
    int m;
    …
    m = foo(list);
    …
}
```

types

variables

relational

arithmetic

expressions

statements

blocks

function call

parameters

procedures/functions

Variables and assignments

- Components of a variable
  1. name : identifier composed of letters
  2. memory location : bound to the variable
  3. current value : stored in the location
  4. data type : static/dynamic binding
  5. scope : static/dynamic
  6. life time : interval during which a location is bound to the variable
- An assignment changes the contents of some components of a variable.

```
float x;
x = 1.5;          //the type is a real number of value 1.5
…
x = 10;           //now, the type is a real number of value 10.0
```

location (l-value) and value (r-value)

- The ***location*** of a variable is regarded as a kind of value owned by the variable. → *We call the location the **l**-value of the variable.*
- To make a distinction, the ***real value*** of the variable is called the ***r***-value. → ***Ex:*** `char z = 'o';` *// l-value of z is the address of z, and r-value is 'o'*
- A rule of assignments:

"The left-hand side of an assignment should have the l-value, and its right-hand side should have the r-value."

```
float x;           // create real variable x with the l-value in which the r-value is undefined
float y = 4.1;     // create real variable y with the r-value defined by storing 4.1 to l-value of y
x = y;             // store the r-value of y into the l-value of x
y = y * 3.0;       // store the r-value of y times 3.0 into the l-value of y
```

- If different variables have the same l-value, it is called *aliasing*.

→ ***Ex:***

```
char d;
char & c = d;      // c and d share the same location
```

Properties of l-values and r-values

- **referencing**: the operation of getting the l-value of a variable

  Ex: `char* p = &c;`

- **dereferencing**: the operation of going from a reference to the r-value it refers

  → In C++, the right-hand side is always dereferenced **once**.

```
char c, d;
char *p, **q, **r;
c = d;        // OK! dereferenced once
q = r;        // OK! dereferenced once
d = p;        // error! dereferenced twice
p = c;        // error! no dereferencing necessary
r = c;        // error! no dereferencing sufficient
```

Properties of l-values and r-values

- Some expressions, such as *id*, *array reference* and *dereference*, have both l-values and r-values.
- All other expressions have r-values only.
  - ➔ So they can **NOT** appear on the left-hand side of assignments.

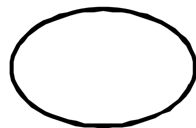
```
4.5 = 10.1;                  // illegal - integers have no l-values
"jane" = y;                  // illegal - strings have no l-values
y + 1 = a[i];                // illegal - arithmetic expressions have no l-values
x = 10.1;                    // legal - id expressions have l-values
a[i] = y + 1                 // legal - array references have l-values
employee.name = "jane"       // legal
*p = foo(x,y)                // legal - dereference expressions have l-values
foo(x,y) = 1.1;              // maybe legal, maybe illegal
```

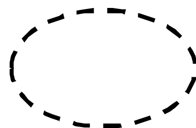
gnu c++ says error: 'non-lvalue in assignment'

Graphical notation for variables

name

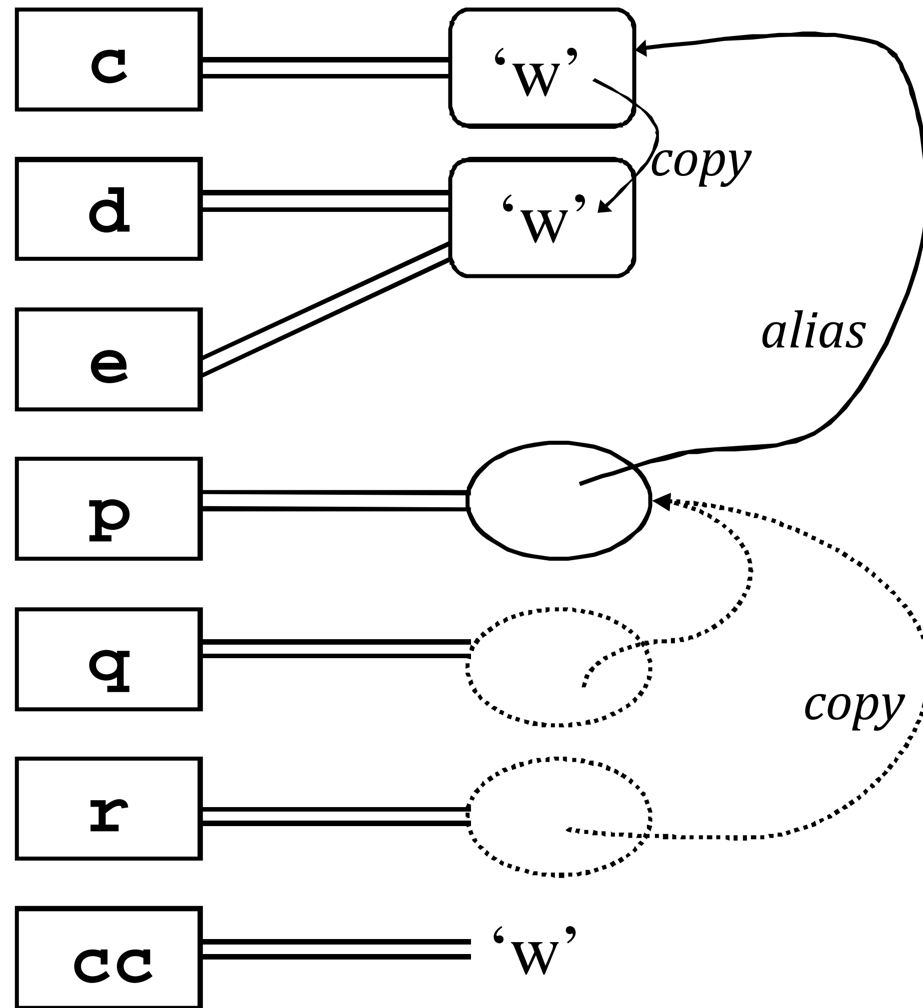
Location that can contain an ordinary data value


Location that can contain a reference (address) value


Location that can contain a reference-reference value

Graphical notation for variables

```
char c = 'w';

char d = c;

char& e = d;

char* p = &c;

char** q = &p;

char** r = q;

const char cc = 'w'
```



Name conflicts

- In languages, a name can be used to mean several different objects. ➔ Ex) Ford: : a name of a man, a car, or a company …
- Using the same name to represent different objects in different places causes a **name conflict** and potentially ambiguity of the language.
- But, name conflicts are often necessary to improve naturalness of a language and readability/writability.

Ford drives a car.
They work for the Ford. → *How do we know which Ford means which in each sentence?*
She drives a Ford.

Ford1 drives a car.
They work for the Ford2. → *OK. no more naming conflict! So, the meaning is clearer.*
She drives a Ford3. *But do you like it?*

Scope

- Ambiguity due to naming conflicts can be resolved by associating names with the **environments** where each name is defined.
  - → Jane in Ohio, Jane in Maine, Jane in Virginia
- In a programming language, a **scope** is a program environment in which names are defined or declared.
  - → procedures and blocks (Pascal, C/C++), lambda expressions (Scheme)
- Through a declaration within some scope, a name is bound to a variable with certain attributes, and the variable is called a **bound variable**.
- A variable which is not bound in the scope is called a **free variable**.

SOAR RESEARCH GROUP

Scope

```
{                          // Beginning of a new environment/scope
  int a[10]; int i; char c;       // declarations of three names
  … c …                    // a reference to 'c': bound to a character variable
  … a[i] …                 // references to 'a' and 'i': bound to integer variables
  … x …                    // a reference to the free variable 'x' within this scope
}                          // End of a scope
… c …                      // The name no longer represents a character variable.
                           // Then, what is it now?
```

→ *A binding of a name is* ***visible*** *and* ***effective*** *inside the scope where the name is declared.*

- The idea of a scope is to limit the boundary of a declaration of a name.
- Outside the boundary of a declaration, that declaration and binding should not be visible.

Example: scopes in mathematics

- The notion of a scope also appears in mathematics.

Let x be the amount of salary that Susan receives every month, and y be the price of the new car she wants to have. Then, the time that it takes for her to buy the car is ***y/x*** *months.*

Now, let x be the amount of money that John wastes every day, and y be the total amount of money he currently has. Then, the time that it takes before he goes bankrupt is ***y/x*** *days.*

scope 1

scope 2

declarations

references

- Existential, ∃, and universal, ∀, quantifiers provide a formal notion of scope.

➔ $(\forall x\,(\exists y\, f(x,y)) \vee (\forall z\, g(x,y,z)))$

declarations

scope of x

scope of y

scope of z

free variable

→ A declaration of x makes x a bound variable.

SOAR RESEARCH GROUP

Principle in binding of variables

"A bound variable in an expression can be renamed uniformly to another variable that does not appear in the expression without changing the meaning of the expression."

- Example:

$$\forall x\ (\exists y\ f(x,y)) = \forall x\ (\exists v\ f(x,v))$$
$$= \forall y\ (\exists v\ f(y,v))$$
$$= \forall y\ (\exists x\ f(y,x))$$
$$\neq \forall y\ (\exists y\ f(y,y))$$

The declaration $\forall y$ in $\forall y\ (\exists y\ f(y,y))$ is vacuous and invisible to $f(y,y)$ in $(\exists y\ f(y,y))$ because $\exists y$ supercedes $\forall y$.

Using the principle

- What does this mean to a programming language?

```
code 1 → int x; { int * y;    … x + *y; … }
code 2 → int x; { int * v;    … x + *v; … }
code 3 → int y; { int * v;    … y + *v; … }
code 4 → int y; { int * x;    … y + *x; … }
code 5 → int y; { int * y;    … y + *y; … }
```

➔ code 1 =

useful for detecting programming assignment copies?

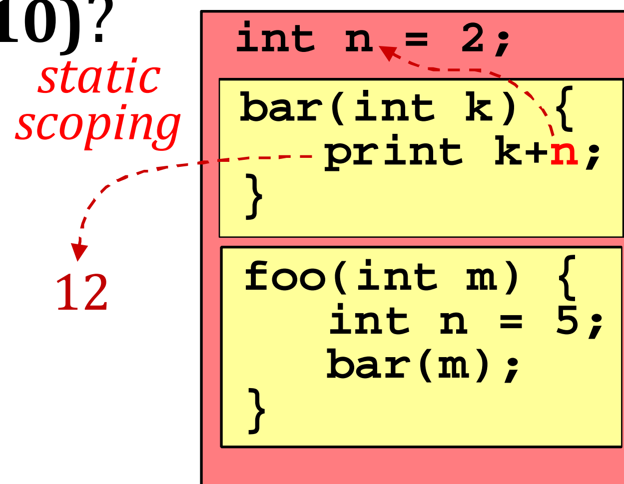
Static vs. dynamic scopes

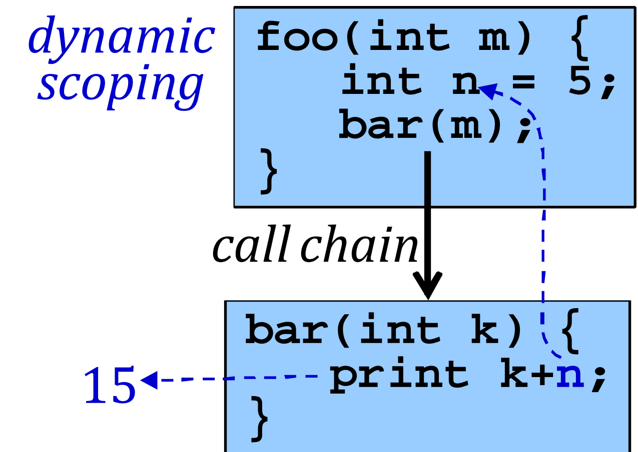
- Each language has its own rules that tell us where to find the declaration for a name in a program. The rules are called the **scope rules** (or scope regime).
- Scope rules can be categorized largely into two kinds:
  - static scope (Fortran, Pascal, C/C++, Scheme, Common Lisp)
  - dynamic scope (pure Lisp, APL, SmallTalk)
- static scope
  - A scope of a variable is the region of text for which a specific binding of the variable is visible.
  - Therefore, the connection between references and declarations can be made lexically, based on the text of the program.
  - At compile time, a free variable is bound by a declaration in textually enclosing scopes/environments.

Static vs. dynamic scopes

- dynamic scope
  - The connection between references and declarations cannot be determined lexically since, in general, a variable is not declared until run-time, and may even be redeclared as the program executes.
  - At run-time, a free variable in a procedure is bound by a declaration in the environment from which the procedure is called.
  - rarely adopted by most existing languages since it generally has more disadvantages (difficult to read, more expensive …) than static scoping
- Ex: Output of **foo(10)**?

```
int n = 2;

bar(int k) {
    print k+n;
}

foo(int m) {
    int n = 5;
    bar(m);
}
```





Life time

- Life time of a variable is the interval of time for which a specific binding of the variable is active. → *cf: scope*
- During the life time of a variable, a variable is bound to memory storage.
- Let **x** be a **simple/automatic** variable declared inside a scope **S**.
  - The life time of **x** begins when program execution enters the scope **S**, and ends when execution leaves the scope.
  - Only when the binding of **x** is visible, **x** is a **live** variable.
- A **global** variable is kind of a simple variable whose scope is global.

Life time

- Let **x** be a **static** variable declared inside a scope **S**.
  - The life time of x begins when program execution starts, and ends when program execution terminates.
  - Even when the binding of **x** is not visible, **x** remains a live variable.

C++

```
extern int omnipresent;  // Global, nonstatic variable
int f() {
  static int die_hard;  // Static variable, as it says
  int short_lived;        // Automatic variable
  … f() …
}
main() {
   … f() …
}
```

  - What are the life time and scope of **die_hard**?
  - What are the life times and scopes of **short_lived** and **omnipresent**?
  - What happens to them when **f** is recursively called?

Topics

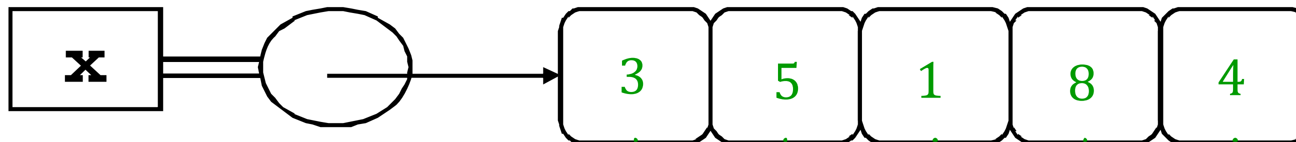
- Components of a variable
  - Name
  - Assignment
  - l-value and r-value
  - Scope & Life time
- **Arrays**
- **Constant variables**

Array variables

- In some languages like Pascal, when you declare

```
x, y : array [1..5] of integer;
```

the compiler creates not only the storage for 5 integers, but also a pointer to its beginning address.

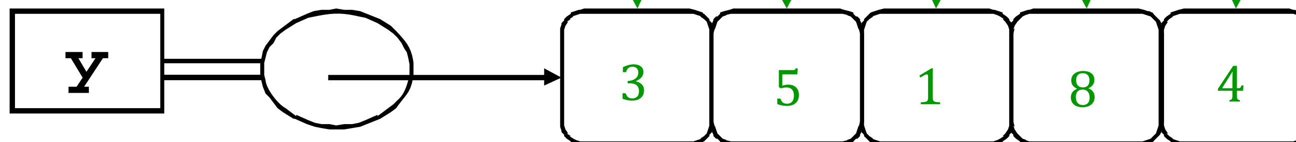


When we have an array assignment

```
y = x;
```

the whole contents of the array **x** are copied to the location of the array **y**.

copy




```
3 3
3 10
0 10
```

```
write (x[1],y[1]);
y[1] = 10;
write (x[1],y[1]);
x[1] = 0;
write (x[1],y[1]);
```

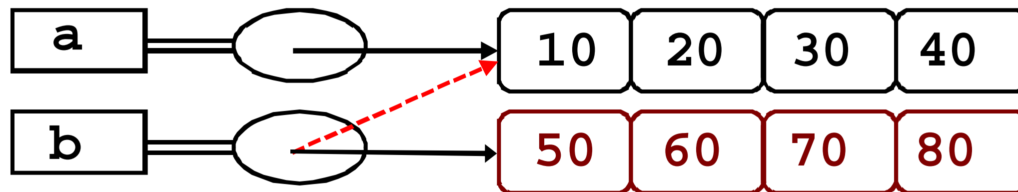
Array variables

- But in Java/C++, an array is really a pointer.

Java arrays

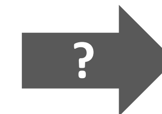
```
int[] a = { 10, 20, 30, 40 };
int[] b = { 50, 60, 70, 80 };
int[] c = new int[4];
```

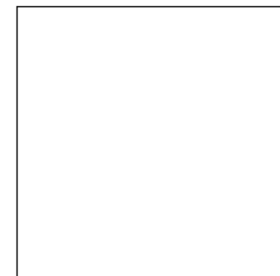
- So, arrays and pointers can be mixed. Thus, after **b = a**, we should have …



→ *What happens to this array?*

```
b = a;                  // legal   - Java arrays
c = a;                  // legal
System.out.println(a[1] + b[1] + c[1]);
b[1] = 1;
System.out.println(a[1] + b[1] + c[1]);
a[1] = 99;
System.out.println(a[1] + b[1] + c[1]);
```





Array variables

- In C++, the result is slightly different since C++ has explicit pointer types while Java does not.

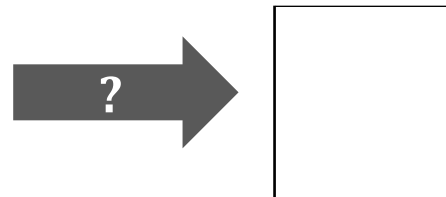
C++ arrays

```
int a[] = { 10, 20, 30, 40 }; // static array
int b[] = { 50, 60, 70, 80 }; // static array
int *c = new int[4];    // array pointer (dynamic array)
```

- In C++, array assignment is illegal. (*element-wise copies needed*)

```
b = a;                  // illegal
c = a;                  // legal
cout << a[1] << c[1];
c[1] = 99;
cout << a[1] << c[1];
```

g++: *"ISO C++ forbids assignments of arrays"*
Visual C++: *"left-hand side must have l-value"*



- Static arrays use stack memory; their pointers cannot change dynamically.
- **c = a**: r-value of **a** (address of the array) is store to l-value of pointer **c**.
- So in this respect, Java arrays act more likely as C++ array pointers.

Multidimensional arrays

- Declaration of 2-dimensional and 3-dimensional arrays

```
float a[3][4];
int b[4][5][2];
```

- The elements of **a** are shown as a table.

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

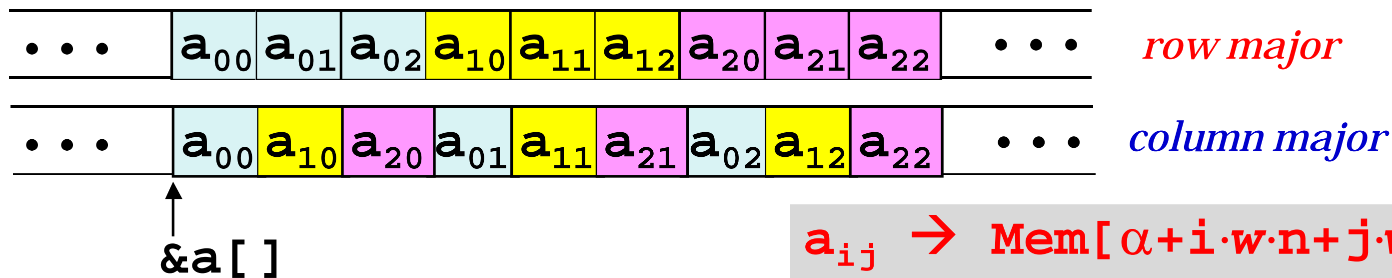
- Two schemes to represent **a** or **b** in programming languages
  - Mapping onto an 1-dimensional array

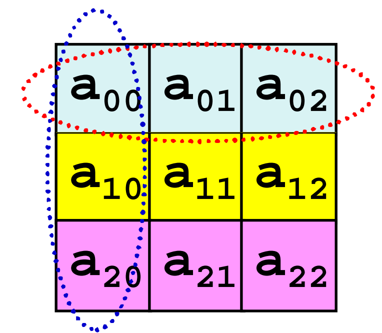
    → adopted by many languages like *Fortran* (up to 7-dimensions possible)
  - Using the array-of-arrays

    → adopted by *Java/C/C++*

Mapping onto an 1-D array

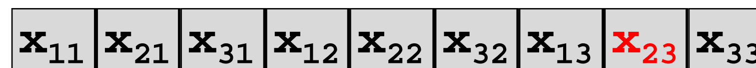
- The elements in a matrix/cube are mapped sequentially to an 1-dimensional contiguous space in the memory.
- Row/Column-major mapping
  → converting an $m \times n$ 2-D array into an 1-D array with length $m \cdot n$ by collecting elements by rows/columns





a_{ij} → Mem[α+i·w·n+j·w]
a_{ij} → Mem[α+j·w·n+i·w]

- Example: Fortran90
  → *column major*



•••

```
integer x(3,3) = (/ 4,5,9,0,9,6,2,8,3 /)
… x(2,3) …
```

→ *offset from the beginning = 7*
→ *byte address = **x**+28*

Using array-of-arrays

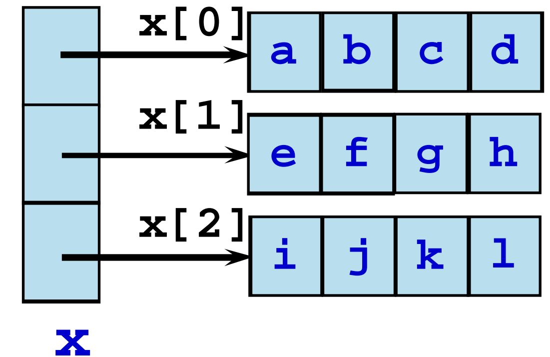
- Java and C++ basically have only 1-D arrays.
- Conceptually, their multidimensional arrays are simulated with arrays of arrays. (*Maybe this is why we write* `a[i][j]` *instead of* `a[i,j]`)

  Ex: a 2-D array is viewed as an 1-D array of rows each of which is also represented as an 1-D array.

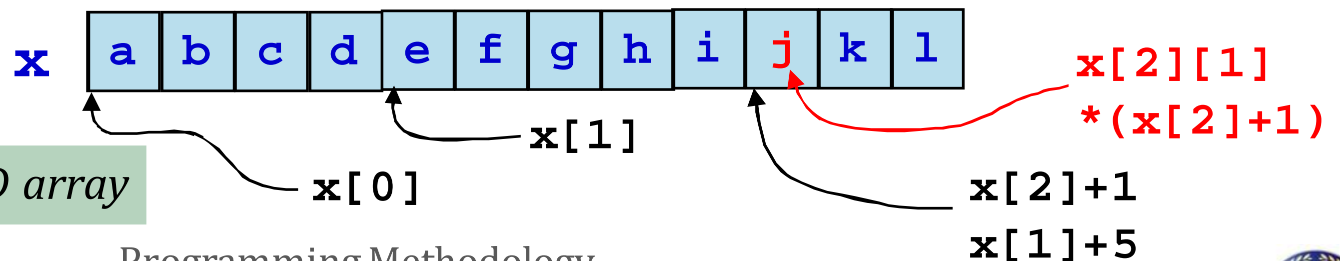
```
a, b, c, d
e, f, g, h
i, j, k, l
```

stored as four 1-D arrays

x = [x[0], x[1], x[2]]
x[0] = [`a, b, c, d`]
x[1] = [`e, f, g, h`]
x[1] = [`i, j, k, l`]



  - *x.length* = 3
  - *x*[0]*.length* = *x*[1]*.length* = *x*[2]*.length* = 4
- Physically, their static arrays are stored to 1-D memory in row-major order.



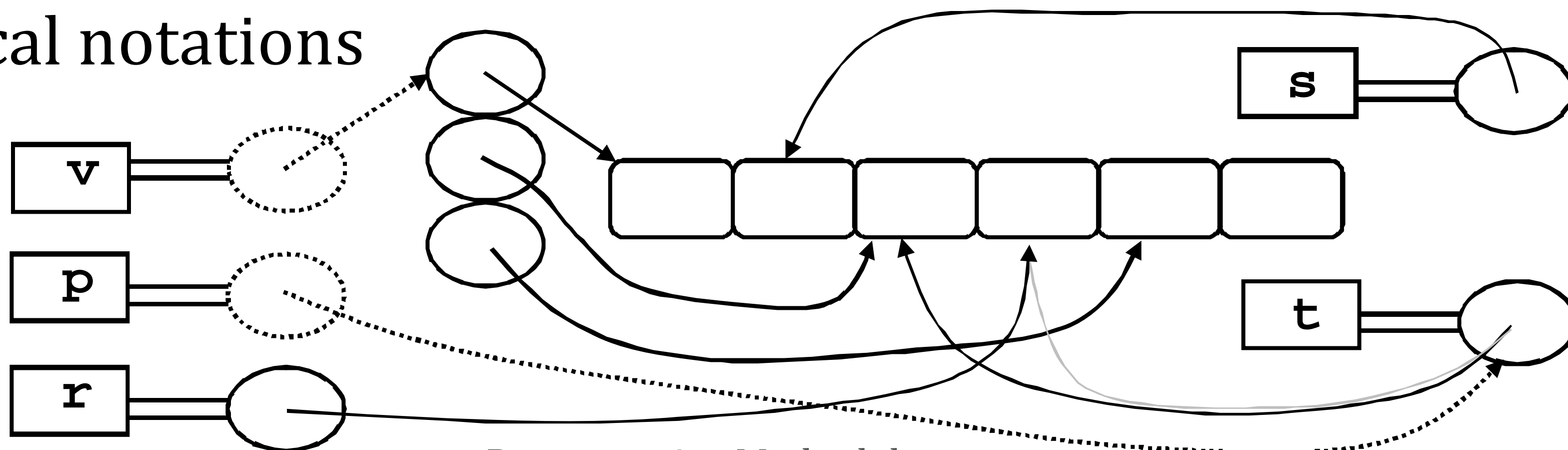
`int x[3][4];` *// static 2-D array*

Mixing arrays and pointers in C++

- Code

```
int v[3][2];
int *t = (int *)(v + 1);   //v[1]
int *s = (int *) v + 1;    //v[0]+1, *v+1 or &v[0][1]
int *u = v + 1;   // error: cannot convert int(*)[2] into int*
int **p = &t;
int **w = v;      // error: cannot convert int(*)[2] into int**
int **q = p + 1;
int *r = ++*p;
… p[0][1] …        // *(t+1)
… p[1][0] …       // runtime error: segment fault
… *q …            // runtime error: segment fault
```

- Graphical notations



Dynamic multidimensional arrays

- Create a dynamic array of dimension 4×5×9 for a pointer **z**.

```
int ***z;
```

- How about this?

```
z = new int[4][5][9];
```

→ *error: cannot convert* `int(*)[5][9]` *into* `int***`

- Explicit arrays of arrays to simulate a 3-D array.

create

```
z = new int**[4];
if (z != NULL)
  for (int i = 0; i < 4; i++) {
      z[i] = new int*[5];
      if (z[i] != NULL)
        for (int j = 0; j < 5; j++)
          z[i][j] = new int[9];
  }
…
z[i][j][k] = … // access it like ordinary 3-D array
…
```

clean-up

```
for (int i = 0; i < 4; i++) {
  for (int j = 0; j < 5; j++) {
      delete [] z[i][j];  z[i][j] = 0;
  }
  delete [] z[i];   z[i] = 0;
}
delete [] z;   z = 0;
```

Constant variables → ?

- Typical way to specify a name for a constant

```
#define Ten 10
…
x = Ten * y;     // x = 10 * y
```

- C/C++ provides a special construct '**const**' to declare a variable whose value is constant during program execution.

```
const int ten = 10;
…
x = ten * y;     // x = 10 * y
```

- A constant variable must be invariable.

```
ten = x + 1;      // error: const var ten cannot be assigned a value
```

cf:
```
Ten = x + 1;      // error: left hand side of '=' must have l-value
```

- It may be viewed as a static variable with a constant value.

→ *No dynamic memory alloc/dealloc is needed whether it is visible or not.*

SOAR RESEARCH GROUP

Constant variables

- Comparison with '**#define**'d names
  - Const vars have scopes, so their visibility can be controlled.
  - They can be bound to specific types: `const float`, `const char*`, …
  - They can be applied to diverse constructs.
    `const struct { … }`, `const int foo() { … }`
- **const** for pointer types

```
int x, y;              // ordinary variables
const int ten = 10;    // a constant variable
const int * tp = &ten;     // a pointer to a constant variable
tp = &x;               // error! How about tp = &ten again here?
int const * tq = &ten;     // a pointer to a constant variable
*tq = y;               // error! How about *tq = 20?
int * const tr = &x;       // a constant pointer to a variable
tr = &ten;             // error!  How about tr = &y?
*tr = 20;              // Is this OK?          How about *tr = y?
const int * const ts = &ten;  // a constant pointer to a constant variable
const int * const tt = &x;    // error!
ts = &x;               // error!  How about ts = &ten again or *ts = y ?
```

(The two declarations `const int * tp` and `int const * tq` are marked as equivalent: =)

Advantages over ordinary variables

- safer code
  (safeguard enforced
  by the compiler)

```
void foo(… &x) { … x = … }
…
int main() {
   const int ten = 10;        So this is error
   int tin = 10;
    … foo(ten)  … // foo should not change ten
   ten = … // Compile error! Did you mean tin?
```

- better quality code
  - Option 1: use static memory for **ten** instead of runtime stack
    ➔ efficient memory utilization especially for recursion
  - Option 2: use immediate addressing for **ten**

C code

```
const int ten = 10;
…
m = m + tin;
n = n + ten;
```

The compiler may propagate ten=10.

Assembly code

```
load r1,<m>
load r2,<tin>
add r1,r1,r2
load r3,<n>
add r3,r3,10
```