Chapter 2

C Pointer Arithmetic and Correspoinding Assembly Code

Variable, Constant, and Variable Name

- Variable Declarations
- int x;
  - define integer **<u>variable</u>** “x” (allocate memory area to store an integer value and name the area “x” – associate label “_x” (“x”) with the address of the area)
  - x (in LHS): denotes that the target variable of the assignment is “x”
    - x = 3; // 3 is stored in variable “x”
  - x (in RHS): denotes the value of variable “x” (the contents of memory area named “x”)
  - &x: denotes the address of variable “x” (the address of the memory area named “x”)

Variable, Constant, and Variable Name (cont)

- int *px;
  - define pointer to integer **<u>variable</u>** “px” (allocate memory area to store a pointer to integer and name the area “px”)
  - px (in LHS): denotes that the target variable of the assignment is “px”
    - px = &x; // the address of variable “x” is stored in variable “px”
  - px (in RHS): denotes the value of variable “px” (the contents of the memory area named “px” --- the address of variable “x”)
  - &px: denotes the address of variable “px” (the address of the memory area named “px”)
  - *px：***<u>dereference</u>*** of px
    - *px (LHS)
      - *<u>In this case</u>*、it assigns the RHS value in the memory area addressed by the value of “px”. *<u>However, it is not always so</u>*
    - *px (RHS)
      - *<u>In this case</u>*、 it denotes the contents of the memory area addressed by the value of “px” 。*<u>However, it is not always so</u>*

Variable, Constant, and Variable Name (cont)

- 3
  - denotes **constant** 3
    - no memory area is allocated; in common implementations, value 3 is embedded in an machine instruction (immediate addressing))
    - therefore, both “3 = x” and “&3” are illegal

Pointers and arrays

- int a[10];
  - reserve a memory area to store 10 integers and name the starting address of the area "a"
  - "a" is a **constant** and there is no memory area named "a" (therefore, &a is illegal)
      - try printf("%d, %d\n", a, x);
  - the type of "a" is "pointer to integer"
  - "a[i]" is a variable (the contents of the i^{th} integer area in the allocated area)
  - "&a[i]" is the address of variable "a[i]"
- C/C++ do not have an array (arithmetic)
  - exception： declaration of an array, such as "int a[10];", that allocates a memory area
- "a[i]" is shorthand for "*(a+i)" ("a" is a pointer to [some object]、" i" is an integer)
  - when the compiler encounters "a[i]," it executes "*(a+i)"
  - you can write "i[a]" in place of "a[i]"
    - (a[i] == *(a + i) == *(i+a) == i[a])
  - after declaring "int *pa=a;", you can write "pa[i]" to access the same area as "a[i]"

Pointer declaration: the right spiral rule

- Declaration of pointers
  - types that appear in the right side of the variable name
    - [ ] [ ] ... [ ] : array[ ][ ]…[ ] of (read "[ ][ ]…[ ]" at once)
    - (…) : function that takes … and returns
  - types that appear in the left side of the variable name
    - * : pointer to
- interpretation of a pointer declaration
  1. find the variable name and start from there
  2. draw a half right arc to the first C symbol and read it
  3. draw a half right arc to find the next C symbol and read it
  - repeat Step 3 until C data type (int、float, etc) is found

Pointer declaration: the right spiral rule (cont)

- char *x[3];
  - x is of type “array [3] of pointer to char”
- char (* x) [2];
  - x is of type “pointer to array [2] of char”
- char * (* (x [ ]))( );   [== char *(*x[ ])( );]
  - x is of type “array [ ] of pointer to function that returns pointer to char”
- int * (* (* x [2][3])[4]) ( );
  - x is of type “array[2][3] of pointer to array[4] of pointer to function that returns pointer to int”

Pointer arithmetic

- Consider the following type declaration

  TYPE *ptr;   // “ptr” is a pointer to an instance of TYPE (“ptr” is a variable)

  TYPE a[10];   // “a” is a pointer to an instance of TYPE (“a” a constant)

- Only two types of operations may be applied to pointer variables/constants
  - pointer + integer (similarly, pointer – pointer)
  - *pointer   // dereferencing
- Type “array[i][j][k]...[z]” is equivalent to “pointer to array[j][k]....[z]”
  - since there is no “array arithmetic in C”, when the type under analysis is [i][j][k]…[z], it must be converted to “pointer to array[j][k]…[z]” **(RULE1)**

Pointer arithmetic (cont)

Apply the following derivation rules using “T” (type) and “V” (value) on an expression containing a pointer

1. pointer+integer

ptr: T suppose “pointer to TYPE”

V suppose “VAL”

then

ptr + i (i is an integer) is

T pointer to TYPE

the type does not change when an integer is added

V VAL+i*sizeof(TYPE)

when added 1, ptr addresses the next element in “the array”

Multi-dimensional arrays

Consider declaration “int a[3][4];”

- a memory area for 3*4 integers is allocated
- “a” is a constant and its value is the starting address of the allocated area
- a[0][0] is a variable and denotes the value of the element in the 0th (the first) row and the 0th column of the array
- then, what is “a[0]” ? Execute the following program:

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
int main( ) {
   printf("%x\t%x\t%x\n", a, a[0], a[0][0]);
   printf("%x\t%x\t%x\n", a+1, a[0]+1, a[0][0]+1);
   printf("%x\t%x\t%x\n", a+1, *a+1, **a+1);
   return 0;
}
```

output:
4227136 4227136 1
4227152 4227140 2
4227152 4227140 2

Multi-dimensional arrays (cont)

- from the execution, what do you observe ?   what is the difference between “a” and “a[0]” ?
- Clearly understand that “ *pointer” is not necessarily the contents of the location addressed by “pointer”

Pointer arithmetic (important)

2. *pointer (***<u>dereference: remove the leftmost "pointer to" in the type</u>***)
   1. Case 1:
      - ptr: T "pointer to a simple (non-array) object type"
        V suppose "VAL"
      - *ptr T the simple object type
        V the contents of the memory area addressed by "VAL"
   2. Case 2:
      - ptr: T "pointer to [i][j][k]...[z] of object type"
        （an array object type）
        V suppose "VAL"
      - *ptr T [i][j][k]...[z] of object type
        = pointer to [j][k]...[z] of object type
        (C has "pointer arithmetic" but does not have "array arithmetic"; when the array notation appears in the derivation, transform it to the equivalent pointer notation)
        V VAL (the same value as ptr)
        if the pointer points to an array object, the value does not change after dereferencing

Pointer representation and array representation

- a[i] = *(a + i), where a is a pointer (to some object) and i is an integer
- a[i][j] = (a[i])[j] = *(a[i] + j) = *(*(a + i) + j)
- a[i][j][k] = ((a[i])[j])[k] = *((a[i])[j] + k) = *(*(a[i] + j) + k) = *(*(*(a + i) + j) + k)
- Thus、to analyze a[i][j][k], apply the “+int” and “* (dereference)” rules in the following order:
  1. a
  2. a+i
  3. *(a+i) (= a[i])
  4. a[i] + j
  5. *(a[i] + j) (= a[i][j])
  6. a[i][j] + k
  7. *(a[i][j] + k) (= a[i][j][k])

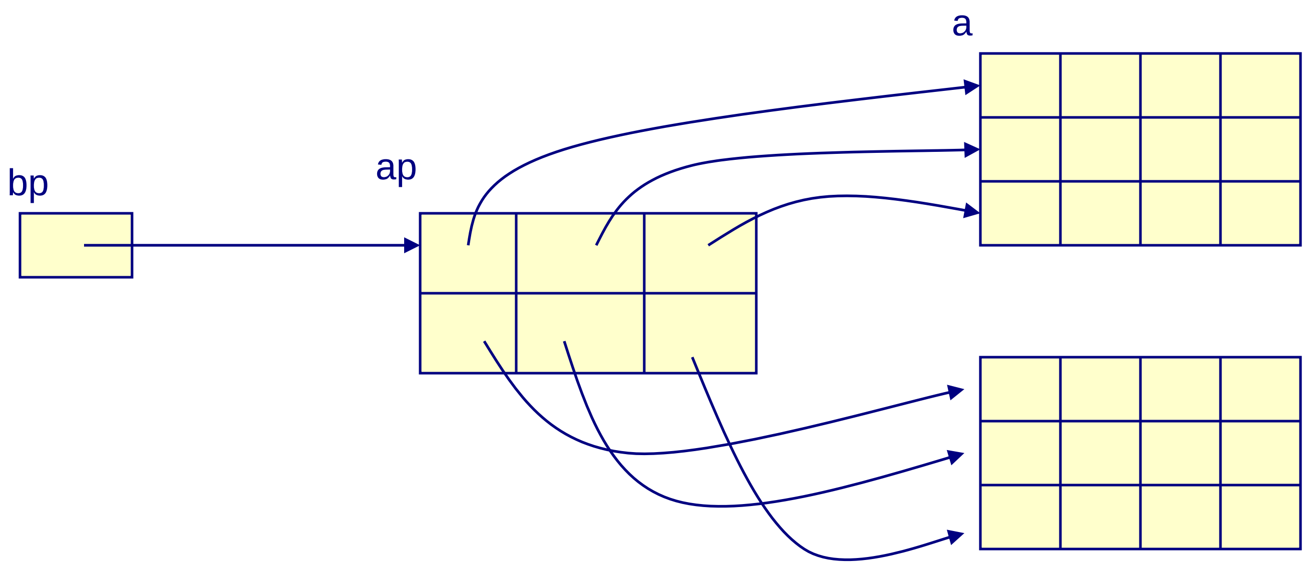
Exercise on pointer arithmetic

- Consider the following declaration

int a[2][3][4] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

int *ap[2][3] = {{a[0][0], a[0][1], a[0][2]}, {a[1][0], a[1][1], a[1][2]}};

int *(*bp)[3] = ap;



Exercise on pointer arithmetic (cont)

- Derivation of a[1][2][3]

a:	T	[2][3][4] of int=ptr to [3][4] of int
	V	&a[0][0][0] (since "a" is an array name, it denotes the starting address of the array area)
a+1:	T	ptr to [3][4] of int
	V	&a[0][0][0]+1*sizeof([3][4] of int)=&a[1][0][0]
*(a+1)=a[1]	T	[3][4] of int = ptr to [4] of int
	V	&a[1][0][0] (Case 2)
a[1]+2	T	ptr to [4] of int
	V	&a[1][0][0]+2*sizeof([4] of int)=&a[1][2][0]
*(a[1]+2)=a[1][2]	T	[4] of int = ptr to int
	V	&a[1][2][0] (Case 2)
a[1][2]+3	T	ptr to int
	V	&a[1][2][0]+3*sizeof(int)=&a[1][2][3]
a[1][2][3]	T	int
	V	the contents of &a[1][2][3] =24 (Case 1)

Exercise on pointer arithmetic (cont)

- derivation of ap[1][2][3]

ap	T	[2][3] of ptr to int = ptr to [3] of ptr to int
	V	&ap[0][0]
ap+1	T	ptr to [3] of ptr to int
	V	&ap[0][0]+1*sizeof([3] of ptr to int) = &ap[1][0]
ap[1]	T	[3] of ptr to int = ptr to ptr to int
= *(ap+1)	V	&ap[1][0] (Case 2)
ap[1]+2	T	ptr to ptr to int
	V	&ap[1][0]+2*sizeof(ptr to int) = &ap[1][2]
ap[1][2]	T	ptr to int
= *(ap[1]+2)	V	the contents of &ap[1][2]=&a[1][2][0] (Case 1)
ap[1][2]+3	T	ptr to int
	V	&a[1][2][0]+3*sizeof(int) = &a[1][2][3]
ap[1][2][3]	T	int
= *(ap[1][2]+3)	V	the contents of &a[1][2][3]=24 (Case 1)

Exercise on pointer arithmetic (cont)

- Derivation of bp[1][2][3]

bp	T	ptr to [3] of ptr to int
	V	&ap[0][0] (since bp is a simple variable, its value is the contents of the variable)
bp+1	T	ptr to [3] of ptr to int
	V	&ap[0][0]+1*sizeof([3] of ptr to int)=&a[1][0]
bp[1]	T	[3] of ptr to int = ptr to ptr to int
= *(bp+1)	V	&ap[1][0] (Case 2)
bp[1]+2	T	ptr to ptr to int
	V	&ap[1][0]+2*sizeof(ptr to int) = &ap[1][2]
bp[1][2]	T	ptr to int
= *(bp[1]+2)	V	the contents of &ap[1][2] =&a[1][2][0] (Case 1)
bp[1][2]+3	T	ptr to int
	V	&a[1][2][0]+3*sizeof(int) = &a[1][2][3]
bp[1][2][3]	T	int
= *(bp[1][2]+3)	V	the contents of &a[1][2][3] =24 (Case 1)

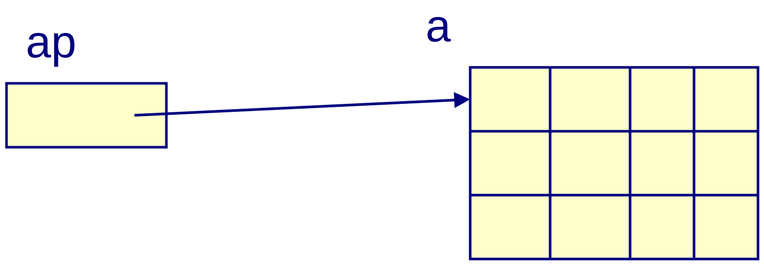
Assignments to pointer variables

- In an assignment statement to a pointer variable, the type of the right hand side must match the type of the left hand side
- Recall the previous declaration of a, ap, and bp
- int *ap[2][3] = {{a[0][0], a[0][1], a[0][2]}, {a[1][0], a[1][1], a[1][2]}};
  - each element of ap is of type “pointer to int”
  - from the derivation of a[1][2][3], the type of a[1][2] is also “pointer to int”
- int *(*bp)[3] = ap;
  - the type of bp is “pointer to [3] of pointer to int”
  - the type of ap is “[2][3] of pointer to int”, which is equivalent to “pointer to [3] of pointer to int”

Example of Incorrect Pointer Declaration

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
int **ap = a;      // warning "type mismatch"

access ap[1][2]
```



Example of Incorrect Pointer Declaration (cont)

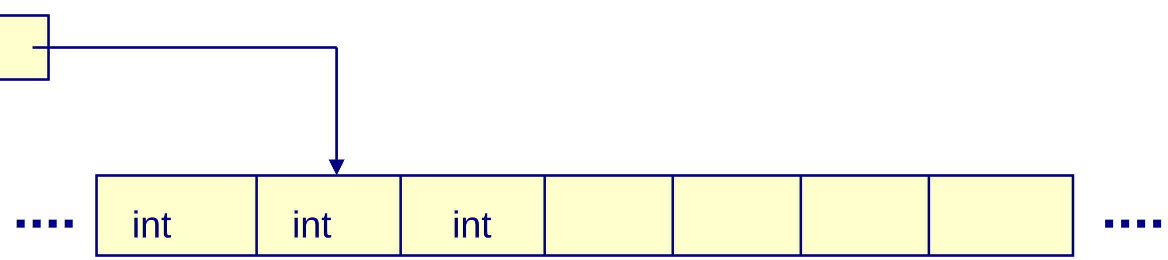
ap	T: ptr to ptr to int
	V: &a[0][0]
ap+1	T: ptr to ptr to int
	V: &a[0][0]+1*sizeof(ptr to int) // assume sizeof(int) == sizeof(ptr) == 4
	= &a[0][1]
ap[1]	T: ptr to int
	V: the contents of &a[0][1]=2
ap[1]+2	T: ptr to int
	V: 2 + 2*sizeof(int) = 10
ap[1][2]	T: int
	V: the contents of address 10 (segmentation fault)

Output:
4202496
4202500
2
10
Segmentation fault (core dumped)

Declaration of pointer variables

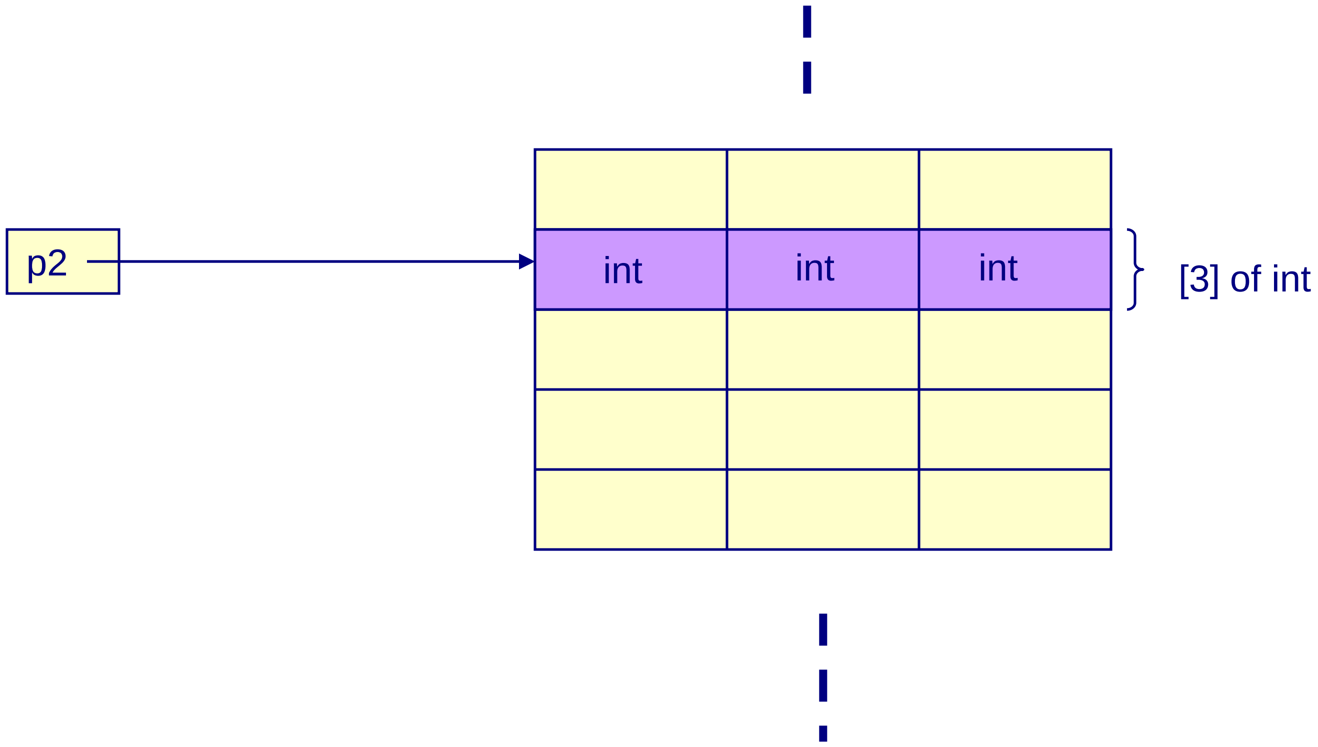
- Consider a declaration “int *p1”
- By applying the pointer arithmetic on “p1”, you can access any element in the virtual one-dimensional integer array starting at the address pointed to by “p1”
- Thus, the image of the object pointed to by “p1” is not

- but the following infinite length of one-dimensional integer array

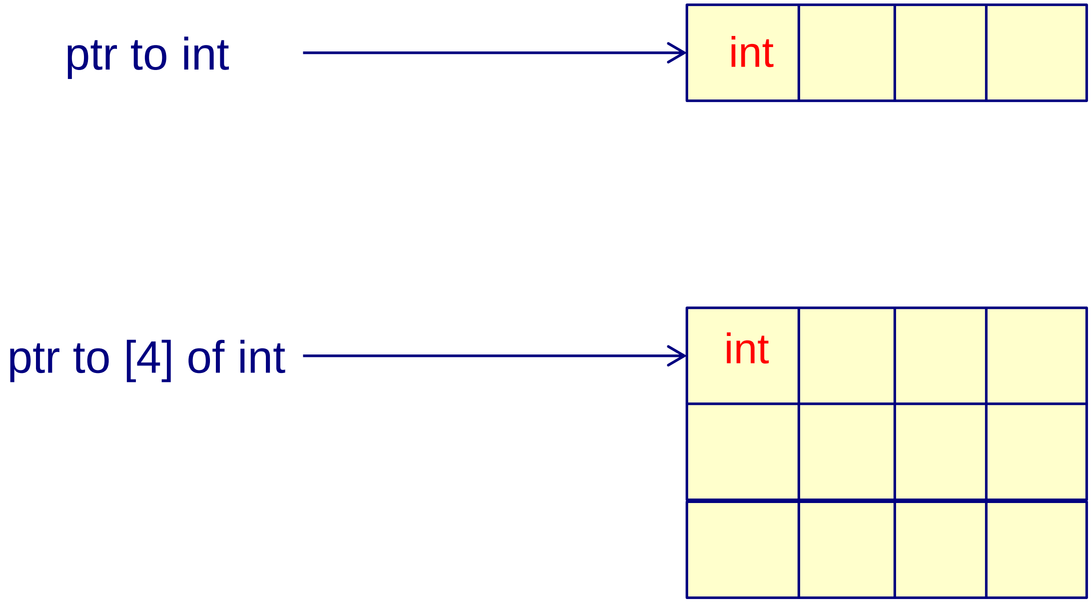


Declaration of pointer variables (cont)

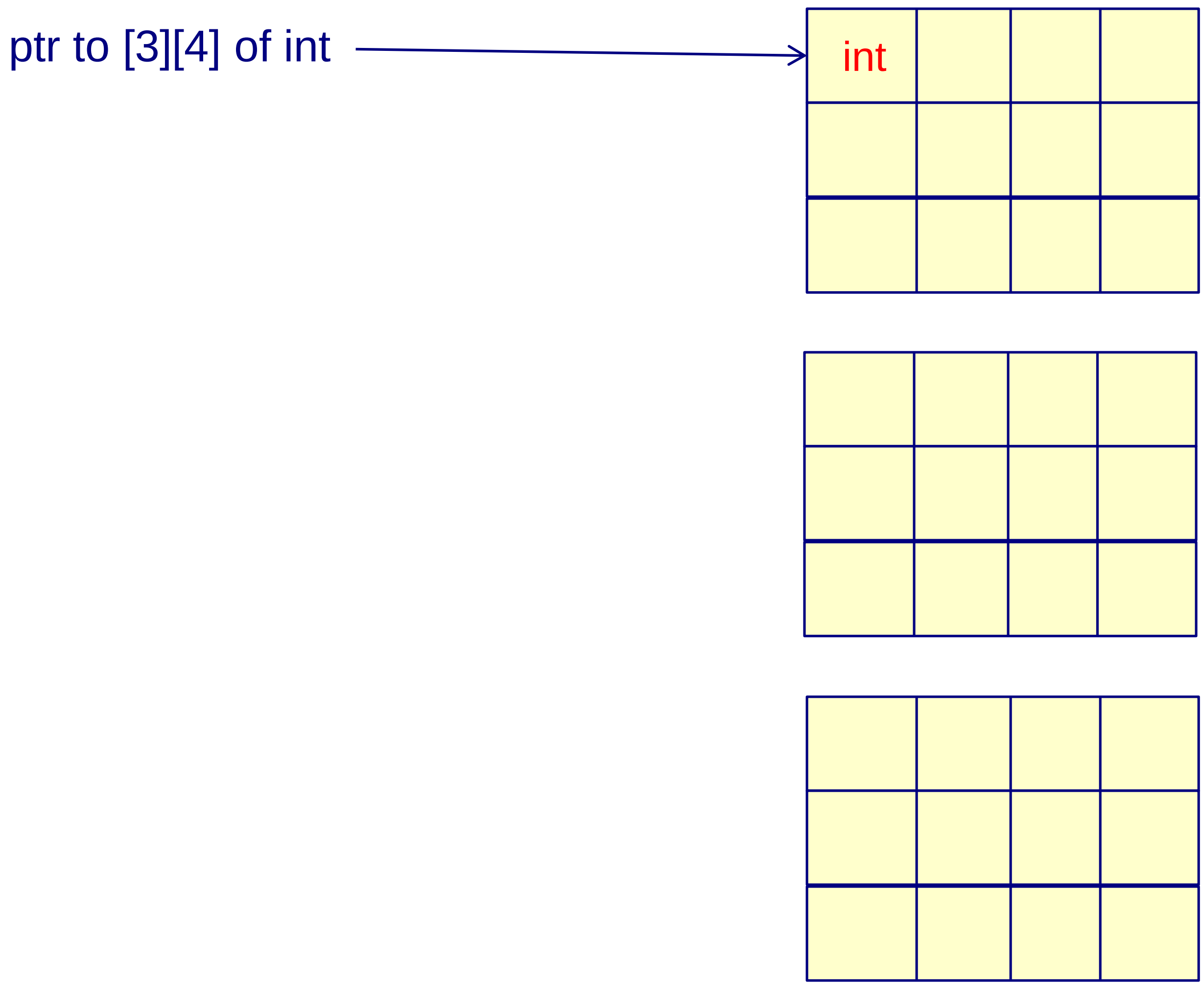
- Similarly, "int (*p2)[3]" declares pointer variable "p2" that points to the following infinite length of one-dimensional array [3] of int



Declaration of pointer variables (cont)



Declaration of pointer variables (cont)



Declaration of pointer variables (cont)

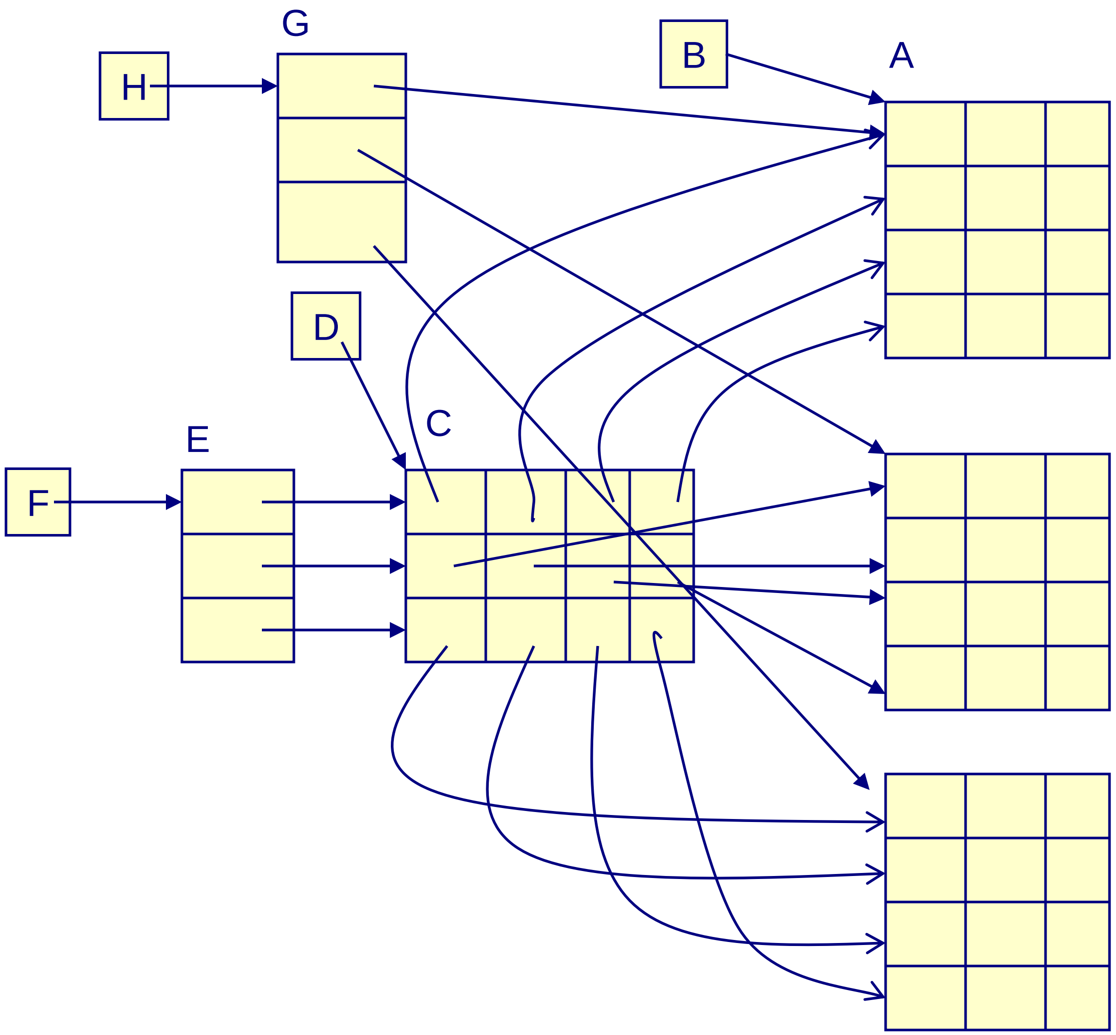
Consider the previous declaration of a, ap, and bp

- The type of ap is “array[2][3] of pointer to int”
  - ap is an array with [2][3] elements, and each element is of type “pointer to int”
  - each element points to one row of array “a” (each row consists of 4 elements)
- The type of bp is “pointer to [3] of pointer to int”
  - bp itself is a simple (non-array) pointer variable that points to “[3] of pointer to int”
  - bp points to ap, that is, “[2][3] of pointer to int” -- a two-dimensional array of pointer to integer with column length 3
  - each element of the two-dimensional array (of type “pointer to integer”) points to one row of “a”

Exercise on pointer declaration

Declare the pointer variables so that the following relation holds

A[i][j][k] ==
B[i][j][k] ==
C[i][j][k] ==
D[i][j][k] ==
E[i][j][k] ==
F[i][j][k] ==
G[i][j][k] ==
H[i][j][k]



Allocation of arrays in heap

- Library function “void *malloc(SIZE in bytes)” allocates “SIZE bytes” in the heap area and returns the base (starting) address of the allocated area (the return type is ptr to void)
- If an area for an imaginary array is allocated in heap and its starting address is set to a pointer that is appropriately declared, any element in the imaginary array is accessed via the pointer
  - Since malloc( ) returns the “pointer to void” type, the return value must be cast to an appropriate type

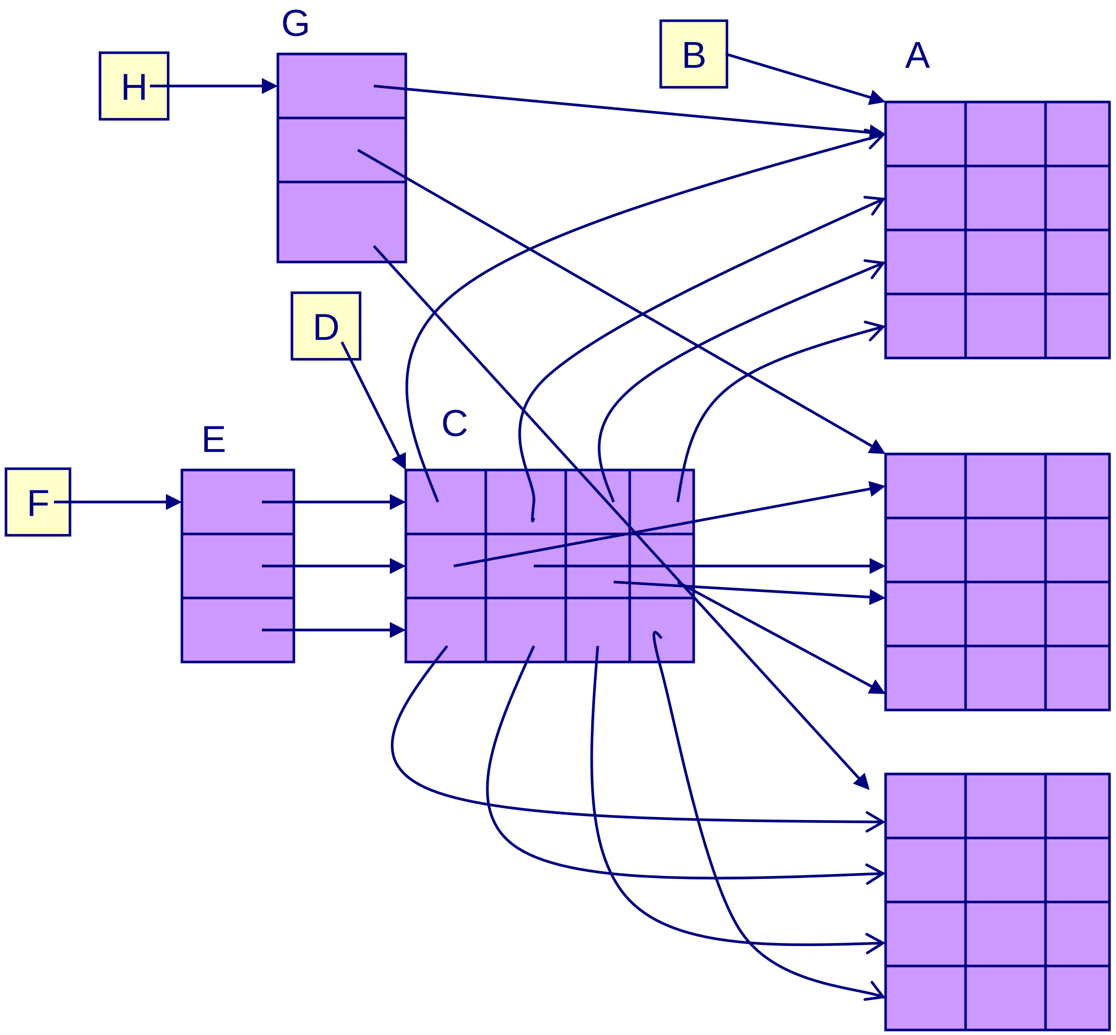
Allocation of arrays in heap (cont)

- To specify a type in cast and sizeof operations,
  - first, declare the type for variable "x" (any name is ok)
  - then, delete "x" in the declaration
- Example:
  - type "ptr to [4] of ptr to int":
    1. declare the type for x: int *(*x)[4]
    2. delete x in the declaration: (int *(*)[4])
  - type "[3][4] of ptr to int":
    1. declare the type for x: int *x[3][4]
    2. delete x in the declaration: (int *[3][4])
- Allocate an imaginary array of type "[3][4] of ptr to int" in heap and declare a pointer variable "d" to point to the array (the type of "d" is "ptr to [4] of ptr to int")
  - int *(*d)[4] = (int *(*)[4]) malloc(sizeof(int *[3][4]));
- Note: Type declarations for the "sizeof" operation is not crucial. Only the size of instances of the type matters
  - You can declare malloc(3*4*sizeof(int *)), instead of malloc(sizeof(int *[3][4]))

Allocation of arrays in heap (cont)

Allocate purple colored arrays in heap and declare pointer variables correctly so that the following relation holds

B[i][j][k] ==
D[i][j][k] ==
F[i][j][k] ==
H[i][j][k]



Allocation of arrays in heap (cont)

Declarations:

```
int (*b)[4][3];

int *(*d)[4];

int ***f;

int (**h)[3];
```

```
// allocate array a and initialize it
 b = (int (*)[4][3]) malloc(sizeof(int [3][4][3]));
 for(i = 0; i < 3; i++)
   for(j = 0; j < 4; j++)
     for(k = 0; k < 3; k++)
             b[i][j][k] = i*12 + j*3 + k + 1;

 // allocate array c and initialize it
 d = (int *(*)[4]) malloc(sizeof(int *[3][4]));
 for(i = 0; i < 3; i++)
   for(j = 0; j < 4; j++)
     d[i][j] = b[i][j];

‘ // allocate array e and initialize it
 f = (int ***)malloc(sizeof(int **[3]));
 for(i = 0; i < 3; i++) f[i] = d[i];

 // allocate array g and initialize it
 h = (int (**)[3]) malloc(sizeof(int (*[3])[3]));
 for(i = 0; i < 3; i++) h[i] = b[i];
```

pointer to function

- Consider the following declaration
  - int (*ptf) (int, int)
    - ptf : a pointer to a function that takes (int, int) and returns int
    - that is, an area for variable ptr is allocated that stores a pointer to (address of) a function (in the text segment) that takes int and int as arguments and returns int
- In C, the name of a function denotes the starting address of the machine code of the function in the text segment
  - Suppose that function “int intAdd(int x, int y) { return x + y;}” is defined
  - By assignment “ptf = intAdd;”, the starting address of “intAdd” is set in ptf
  - To call “intAdd(3, 59);” through “ptr”, the following two ways are possible (both are end up with the identical assembly code):
    - int z = ptf(3, 59);
    - int z = (*ptf)(3, 59);

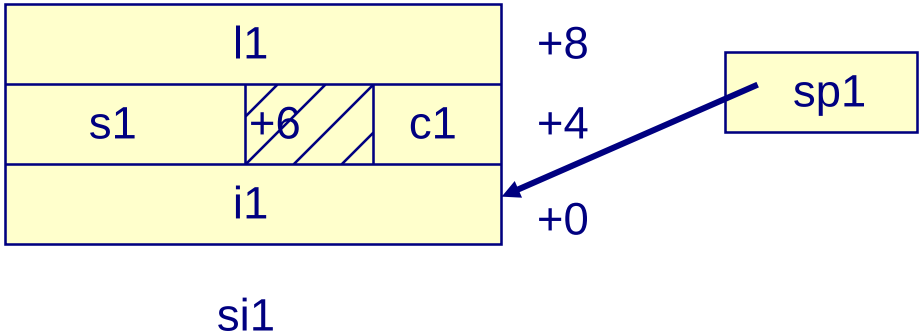
struct

- In struct, each field is converted to the offset relative to the starting address of the instance at compilation time
- The size of a struct instance is a multiple of the word size of the machine

In case of 4-byte、Little Endian machine

```
typedef   struct {
    int  i1;
    char c1;
    short s1;
    long l1;
} struct1;

struct1 si1;
struct1 *sp1 = &si1;
```




```
si1.l1 = 23;      // put 23 in the four-byte area starting &si1+8
sp1->c1 = 'a';  // put 'a' in the byte at the value of sp1 +4
sp1->s1 = 5;     // put 5 in the two-byte area starting the value
                 // of sp1 +6
si1.i1 = 31;     // the four-byte area starting &si1+0
```