60-141-02 LECTURE 4: POINTERS

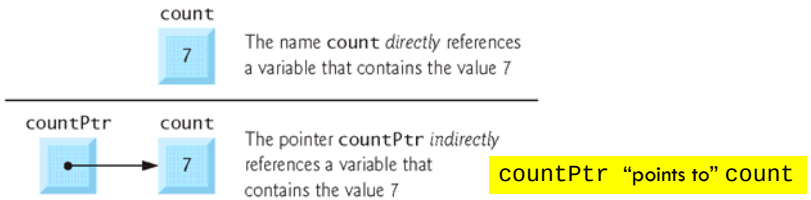
Edited by Dr. Mina Maleki

Outline

- Pointer Concepts & Mechanisms
- Pointer Variable Declaration and Initialization
- Pointers Operations and Arithmetic
- Pointers and Arrays
- Pointers and Functions
- The Const Qualifier

Definition

- Pointers are variables whose values are *memory addresses*.
- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an *address* of a variable that contains a specific value.
- In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value



Definition …

- Pointers are also a variable, and has its own memory address!
- Pointers allow direct access to memory locations, provided you supply or query the correct address.
- Pointers enable programmers to:
  - simulate call by reference
  - create and manipulate dynamic data structures that can grow/shrink at execution time (linked lists, stacks, queues, trees, …)
- CAUTION: Accessing invalid or protected memory spaces lead to sudden program and even system crashes. Errors reported by the compiler or at run time such as 'Segmentation fault, core dumped, and bus error' usually hint to a faulty pointer value.

Pointer Declaration

5

- Like all variables, pointers must be defined before they can be used.
- When a pointer variable is declared, its name must be preceded by an asterisk:

  <data_type> *<pointer_variable_name>;

  ```
  int *ptr;
  ```

  - `ptr` is a pointer variable capable of pointing to ***objects*** of type `int`.
  - We use the term *object* instead of *variable* since `ptr` might point to an area of memory that doesn't belong to a variable.

Pointer Declaration …

6

- Pointer variables can appear in declarations along with other variables:

  ```
  int i, j, a[10], b[20], *p, *q;
  ```

- The asterisk (*) notation does not distribute to all variable names in a declaration.
  - definition of two pointers: `int *x, y;` OR `int *x, *y;`
- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

  ```
  int *p;     /* points only to integers   */
  double *q;  /* points only to doubles    */
  char *r;    /* points only to characters */
  ```

Hint: Include the letters *Ptr* in pointer variable names to make it clear that these variables are pointers.

Pointer Initialization

7

- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to **NULL, 0** or an **address.**
- **NULL** (pointer to nothing) `int *yPtr = NULL;`
  - A pointer with the value `NULL` points to *nothing*.
- The integer **0** `int *yPtr = 0;`
  - The only legal integer you may assign to a pointer
  - Equivalent to initializing a pointer to NULL
- A valid **address** (best to query it using **&**)

```
int y = 5;
int yPtr = &y;
int x, *xPtr = &x;
```

Initialize pointers to prevent unexpected results.

Basic Pointer Operators

8

- **&** Address or referencing operator (ampersand)
  - A unary operator that returns the address of its operand.
  - Example: `ptrNum = &Num;`
    - means: the pointer ptrNum will have the address of variable Num. So ptrNum now 'points' to Num.
- * Indirection or dereferencing operator (asterisk)
  - Returns the value of the object to which its operand (ie. a pointer) points. `printf( "%d", *ptrNum);`
    - means: since ptrNum points to Num, we want the value of Num (an integer).
  - Note that this '*' symbol was also used in declaring the pointer variable!

Basic Pointer Operators …

- As long as `ptrNum` points to `Num`, `*ptrNum` is an ***alias*** for `Num`.
  - `*ptrNum` has the same value as `Num`.
  - Changing the value of `*ptrNum` changes the value of Num.
  - `*ptrNum` and `Num` are equivalent.
- Applying & to a variable produces a pointer to the variable.
- Applying * to the pointer takes us back to the original variable:

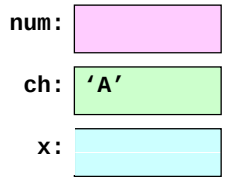
```
j = *&i;    /* same as j = i; */
```

- The & and * operators are complements of one another.
  - `&*aPtr == *&aPtr`

Steps to Pointers

- ***Step 1***: Declare the variable to be pointed to

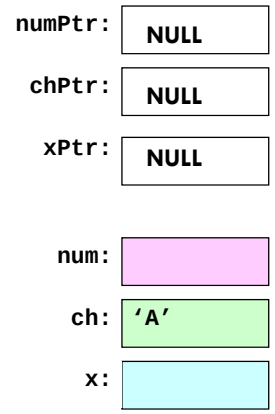
```
int    num;
char   ch = 'A';
float x;
```



Steps to Pointers …

- ***Step 2***: Declare the pointer variable

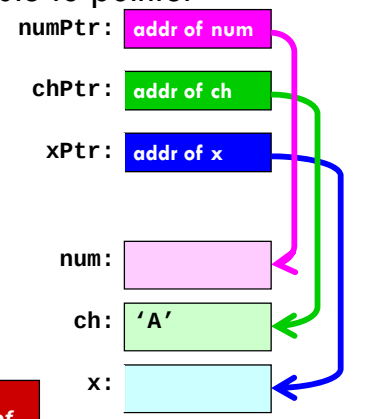
```
int    num;
char   ch = 'A';
float x;

int*  numPtr = NULL;
char  *chPtr = NULL;
float * xPtr = NULL;
```



Steps to Pointers …

- ***Step 3***: Assign address of variable to pointer

```
int    num;
char   ch = 'A';
float x;
int*  numPtr = NULL;
char  *chPtr = NULL;
float * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;
```

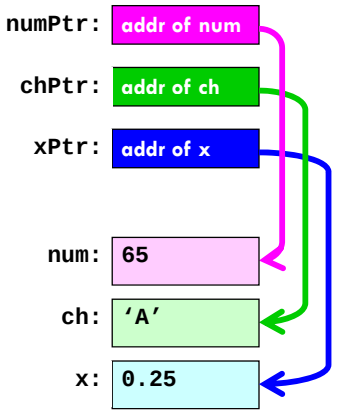


A pointer's type has to correspond to the type of the variable it points to

Steps to Pointers …

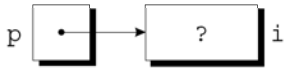
- ***Step 4***: De-reference the pointers

```
int   num;
char  ch = 'A';
float x;

int*  numPtr = NULL;
char  *chPtr = NULL;
float * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;

*xPtr = 0.25;
*numPtr = *chPtr;
```

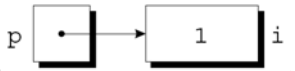


Example

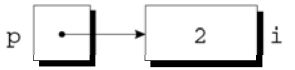

```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);    /* prints 1 */
printf("%d\n", *p);   /* prints 1 */
*p = 2;
```



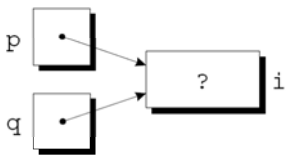

```
printf("%d\n", i);    /* prints 2 */
printf("%d\n", *p);   /* prints 2 */
```

Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Example:

```
int i, j, *p, *q;
p = &i;
q = p;
```

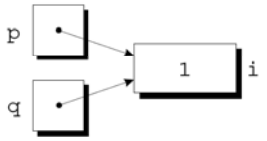
q now points to the same place as p:



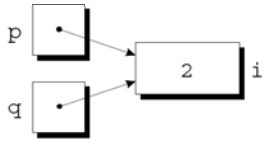
Pointer Assignment …

- If p and q both point to i, we can change i by assigning a new value to either *p or *q:

```
*p = 1;
```




```
*q = 2;
```

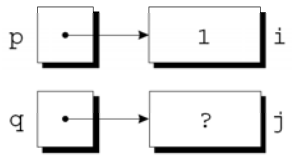


- Any number of pointer variables may point to the same object.

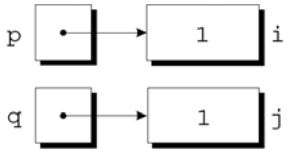
Pointer Assignment …

- Be careful not to confuse `q = p;` with `*q = *p;`
- The first statement is a pointer assignment, but the second is not.

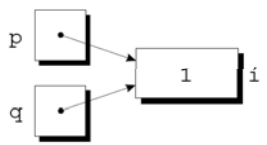
```
p = &i;
q = &j;
i = 1;
```



`*q = *p;`



`q = p;`



Common Mistakes

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```

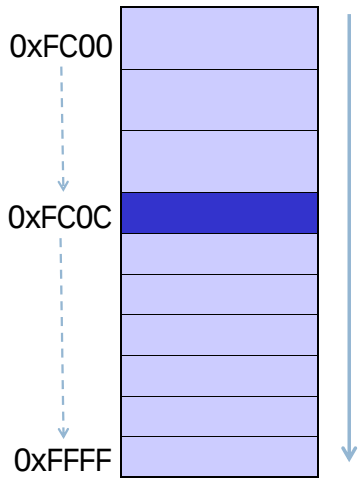
- Assigning a pointer of one type to a pointer of another type

```
int *p; char *c = NULL;
p = c;    /*** WRONG ***/
```

Memory organization

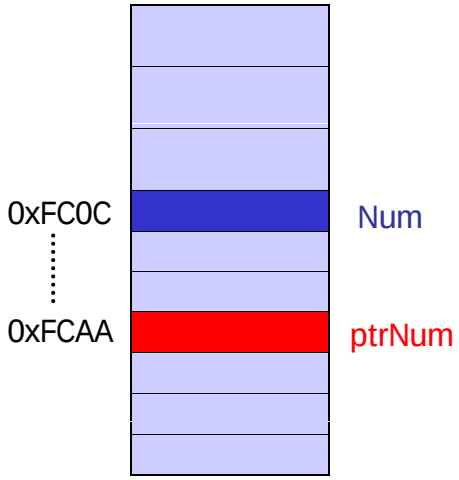
Memory Organization

- Memory 'cells' or 'blocks' are sequentially organized
- Each memory cell has a UNIQUE address
- A memory address is represented by a hexadecimal number (0x indicates hex)
- Addresses increase in one direction (positive direction)



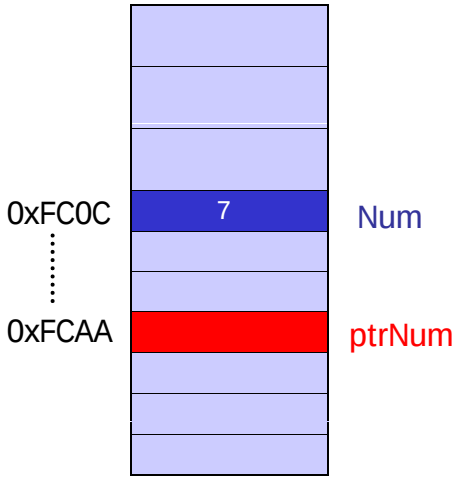
Memory Organization …

```
int Num;
int *ptrNum;
```



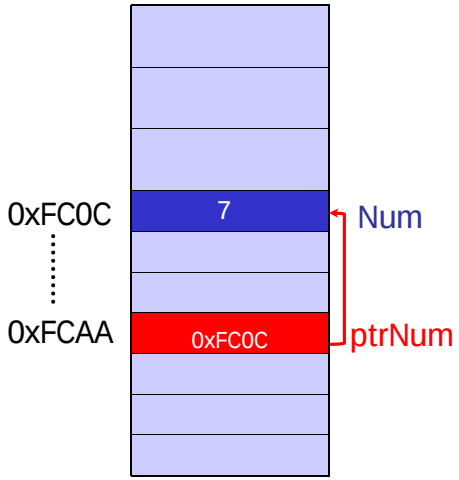
Memory Organization …

```
int Num;
int *ptrNum;
Num = 7;
```



Memory Organization …

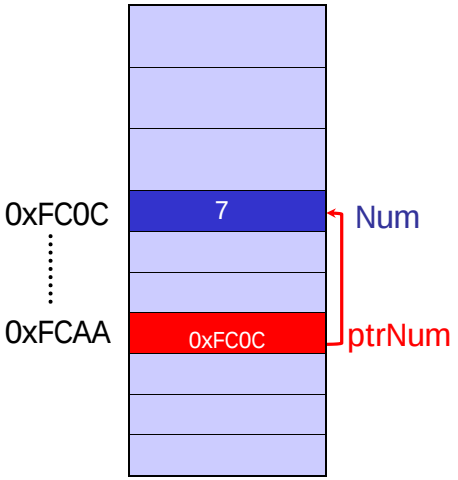
```
int Num;
int *ptrNum;
Num = 7;
ptrNum =&Num;
```



Example 1

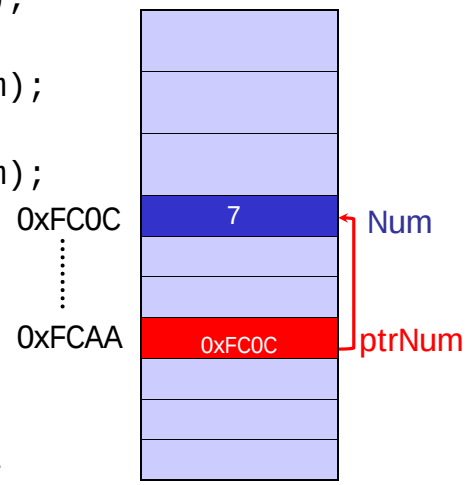
```
int Num;
int *ptrNum;
Num = 7;
ptrNum = &Num;
```

- `printf("%d",Num);`
  `7`
- `printf("%p",&Num);`
  `0xFC0C`
- `printf("%p",ptrNum);`
  `0xFC0C`
- `printf("%d",*ptrNum);`
  `7`



Example 1 …

- `printf("%p",&ptrNum);`
  `0xFCAA`
- `printf("%p",&*ptrNum);`
  `0xFC0C`
- `printf("%d",*&ptrNum);`
  `0xFC0C`
- `printf("%d",*&Num);`
  `7`
- `printf("%d",&*Num);`
  `Error`



because * next to an integer refers to the multiplication binary operator!

Example 2


```
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main( void )
{
   int a; // a is an integer
   int *aPtr; // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a; // set aPtr to the address of a

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
} // end main
```

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Exercise: Trace the following code


```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```

name	address	memory
	510	
x	511	?
y	512	?
p1	513	?
p2	514	?

Exercise: Trace the following code

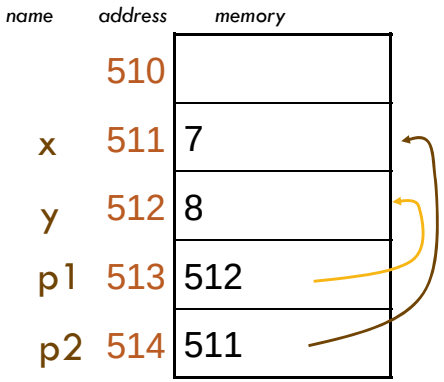

```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```

name	address	memory
	510	
x	511	7
y	512	8
p1	513	?
p2	514	?

Exercise: Trace the following code

29

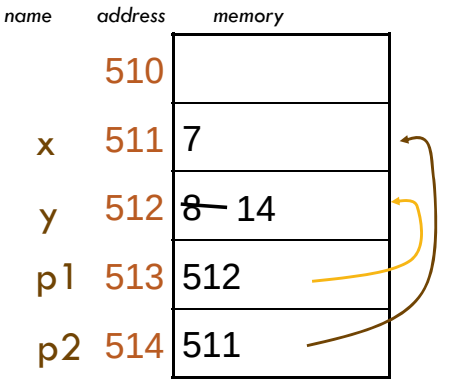
```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```



Exercise: Trace the following code

30

```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```

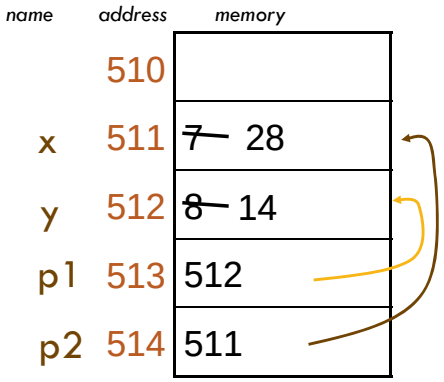



```
*p1 = x + *p2;
y = x + x;
```

Exercise: Trace the following code

31

```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```

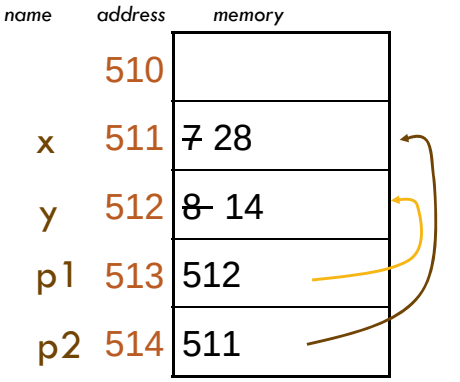



```
*p2 = *p1 + y;
x = y + y;
```

Question

32

```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print p2, *p2, &p2
print x, &x, y, &y
```



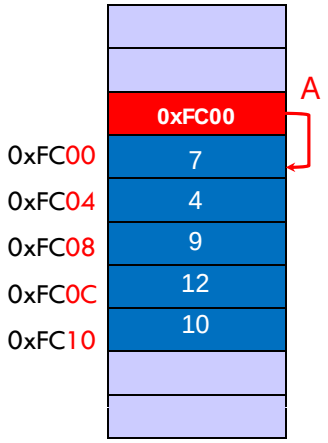
Pointers and Arrays

Relation between Pointers and Arrays
Array Access using Pointers
Pointer Arithmetic
Using Pointers for Array Processing
Arrays of Pointers

Relation between Pointers and Arrays

- Array basics:
  - arrays are a list of sequential memory cells
  - Each array element has a unique memory address
  - Every memory cell is equal to the next since all array elements are of the same type (int=4 bytes)
  - The name of the array can be used as a pointer to the first element in the array.

int A[ ] = {7,4,9,12,10}

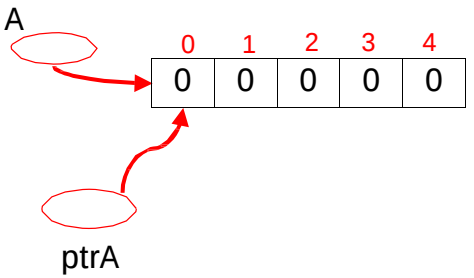


Relation between Pointers and Arrays…

- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- The relationship between pointers and arrays in C is a close one.
- Understanding this relationship is critical for mastering C.

Array Access


```
int A[5] = {0, 0, 0, 0, 0};
int *ptrA = A; or   int *ptrA = &A[0];
```

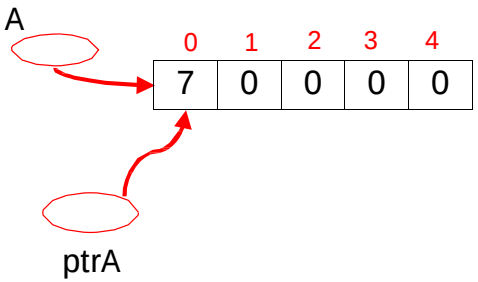


Array Access …

37

- We can now access `A[0]` through `ptrA`; for example, we can store the value `7` in `A[0]` by writing

```
*ptrA = 7;      //OR    A[0]=7
```



Pointer Arithmetic

38

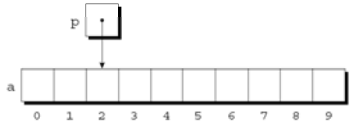
- If `ptrA` points to an element of an array A, the other elements of A can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `ptrA`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

Adding an Integer to a Pointer

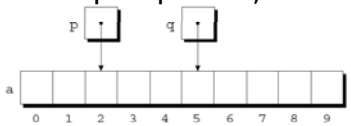
39

- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
- If `p` points to `a[i]`, then `p + j` points to `a[i+j]`.
- Example `int a[10], *p, *q, i;`

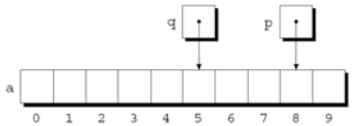
```
p = &a[2];
```




```
q = p + 3;
```




```
p += 6;
```

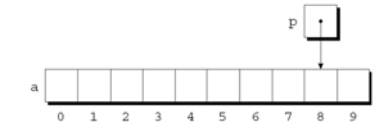


Subtracting an Integer from a Pointer

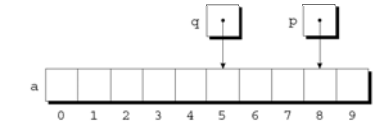
40

- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.
- Example:

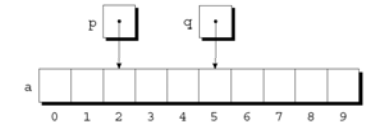
```
p = &a[8];
```




```
q = p - 3;
```




```
p -= 6;
```

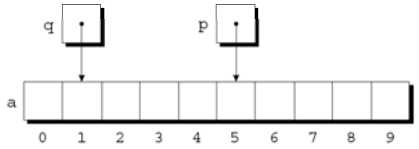


Subtracting One Pointer from Another

41

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.
- Example:

```
p = &a[5];
q = &a[1];
```



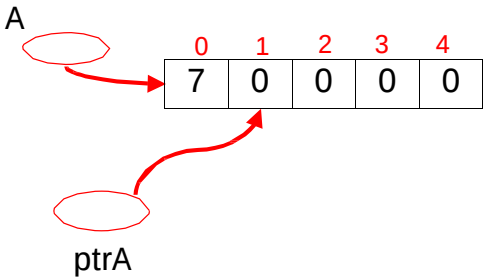

```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Example

42

```
int A[5] = {0, 0, 0, 0, 0};
int *ptrA = A;
ptrA++;
```

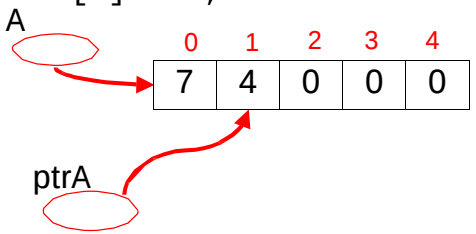
- In this scenario, *ptrA and A[1] refer to the same element in the array. *ptrA == A[1] == 0;



Example …

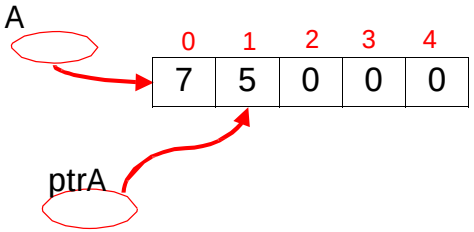
43

```
*ptrA = 4;      //OR  A[1] = 4;
```




```
*ptrA++;
```

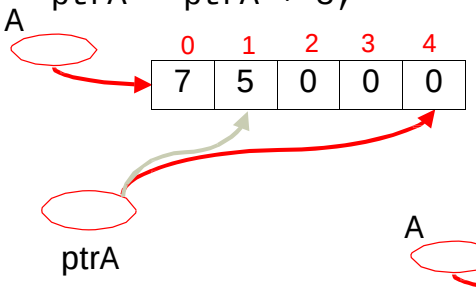
Increments the 'contents'



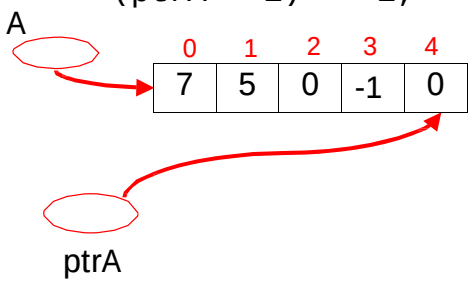
Example …

44

```
ptrA = ptrA + 3;
```

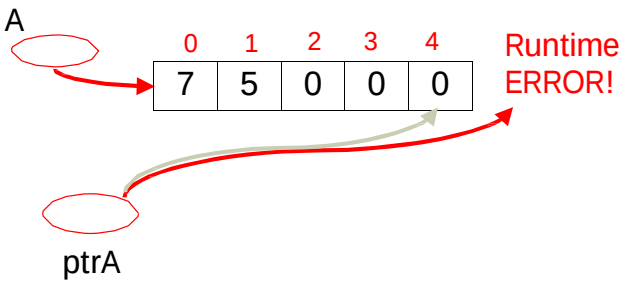



```
*(ptrA – 1) = -1;
```



Example …


```
ptrA += 1;
```



Combining the * and ++ Operators

- C programmers often combine the * (indirection) and ++ operators.
- A statement that modifies an array element and then advances to the next element:

```
a[i++] = j;
```

- The corresponding pointer version:

```
*p++ = j;
```

- Because the postfix version of ++ takes precedence over *, the compiler sees this as

```
*(p++) = j;
```

Combining the * and ++ Operators …

- Possible combinations of * and ++:

Expression	*Meaning*
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment

- The * and -- operators mix in the same way as * and ++.

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
  - Using relational operators is meaningful only for pointers to elements of the same array.
- The outcome of the comparison depends on the relative positions of the two elements in the array.
- After the assignments

```
p = &a[5];
q = &a[1];
```

the value of p <= q is 0 and the value of p >= q is 1.

Comparing Pointers …

- Common comparisons are:
  - check for null pointer if (p == NULL) …
  - check if two pointers are pointing to the same location

    if (p == q)
  - check if two values pointed by p and q are the same

    if (*p == *q)

Common Mistakes

- Performing arithmetic on a pointer that doesn't point to an array element
  - We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.
  - int *aPtr, *bPtr; → aPtr++; //WRONG
- Subtracting or comparing pointers unless both point to elements of the same array
- Running off either end of an array when using pointer arithmetic.

Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.

```
/* populate the array with 0's */
int A[5];              /* declare the array size 5 */
int *ptrA;             /* declare a pointer */
ptrA = A;              /* ptrA points to array A */

for (int i = 0; i < 5; i++)
{
     *ptrA = 0;
     ptrA ++;          /* same as A[i] = 0; */
}
```

Example 1


```
#include <stdio.h>
int main () {
   int i, A[4] = {10, 25, 34, 17};
   int *aPtr;
   aPtr = A;

   printf( "Array values using pointer\n");
   for (i = 0; i < 4; i++)
      printf("*(aPtr + %d):%d\n",i,*(aPtr + i) );

   printf( "Array values using A as address\n");
   for ( i = 0; i < 4; i++)
      printf("*(A+ %d) : %d\n", i, *(A + i) );

   return 0;
}
```

```
Array values using pointer
*(aPtr + 0) : 10
*(aPtr + 1) : 25
*(aPtr + 2) : 34
*(aPtr + 3) : 17
```

Example 1 …

53

```
#include <stdio.h>
int main () {
   int i, A[4] = {10, 25, 34, 17};
   int *aPtr;
   aPtr = A;

   printf( "Array values using pointer\n");
   for (i = 0; i < 4; i++)
      printf("*(aPtr + %d):%d\n“,i,*(aPtr + i) );

   printf( "Array values using A as address\n");
   for ( i = 0; i < 4; i++)
      printf("*(A+ %d) : %d\n", i, *(A + i) );

   return 0;
}
```

Array values using pointer
*(aPtr + 0) : 10
*(aPtr + 1) : 25
*(aPtr + 2) : 34
*(aPtr + 3) : 17

Array values using A as address
*(A + 0) : 10
*(A + 1) : 25
*(A + 2) : 34
*(A + 3) : 17

Example 1 …

54

```
#include <stdio.h>
int main () {
   int i, A[4] = {10,25,34,17};
   int *aPtr;
   aPtr = A;

   printf("Array values using array subscript \n");
   for ( i = 0; i < 4; i++)
      printf(“A[%d] : \n“, i, A[i] );

   printf("Array values using pointer subscript \n");
   for ( i = 0; i < 4; i++)
      printf(“aPtr[%d] : %d\n",  i, aPtr[i]);

   return 0;
}
```

Array values using array subscript
A[0] : 10
A[1] : 25
A[2] : 34
A[3] : 17

Example 1. A[i] == aPtr [i] == *(A+i) == *(aPtr+i)

55

```
#include <stdio.h>
int main () {
   int i, A[4] = {10,25,34,17};
   int *aPtr;
   aPtr = A;

   printf("Array values using array subscript \n");
   for ( i = 0; i < 4; i++)
      printf(“A[%d] : \n“, i, A[i] );

   printf("Array values using pointer subscript \n");
   for ( i = 0; i < 4; i++)
      printf(“aPtr[%d] : %d\n",  i, aPtr[i]);

   return 0;
}
```

Array values using array subscript
A[0] : 10
A[1] : 25
A[2] : 34
A[3] : 17

Array values using pointer subscript
aPtr[0] : 10
aPtr[1] : 25
aPtr[2] : 34
aPtr[3] : 17

Example 2

56

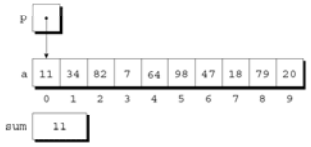
- A loop that sums the elements of an array a:

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
//OR for (p = a; p < a + N; p++)
  sum += *p;
```

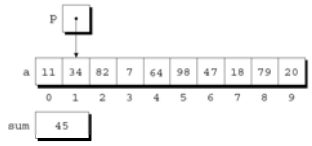
Example 2 …

57

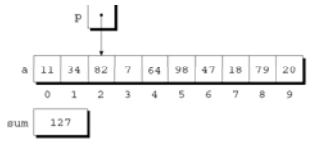
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



Example 3

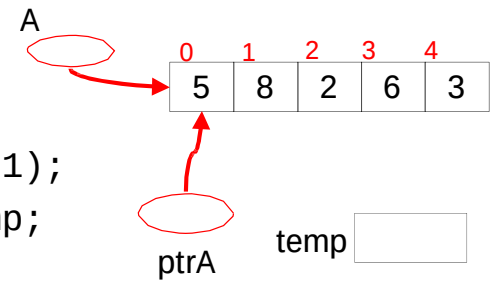
58

```
//Swap 2 array elements
int A[] = { 5, 8, 2, 6, 3};
int temp;
int *ptrA = A;
temp = *ptrA;
*ptrA = *(ptrA + 1);
*(ptrA + 1) = temp;
```

Example 3 …

59

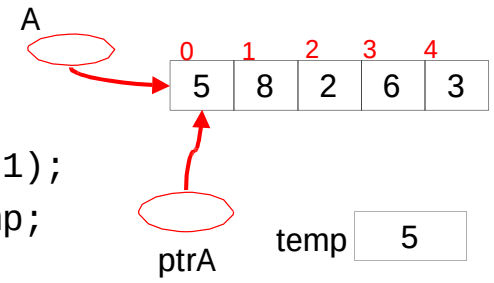
```
//Swap 2 array elements
int A[] = { 5, 8, 2, 6, 3};
int temp;
int *ptrA = A;
temp = *ptrA;
*ptrA = *(ptrA + 1);
*(ptrA + 1) = temp;
```



Example 3 …

60

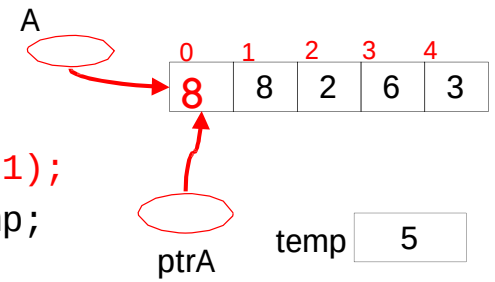
```
//Swap 2 array elements
int A[] = { 5, 8, 2, 6, 3};
int temp;
int *ptrA = A;
temp = *ptrA;
*ptrA = *(ptrA + 1);
*(ptrA + 1) = temp;
```



Example 3 …

61

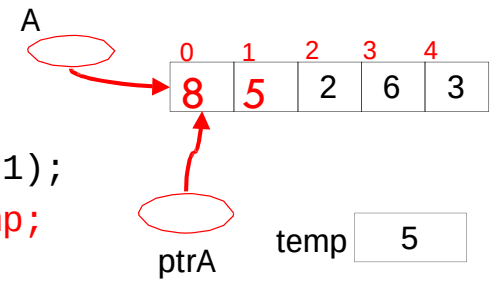
```
//Swap 2 array elements
int A[] = { 5, 8, 2, 6, 3};
int temp;
int *ptrA = A;
temp = *ptrA;
*ptrA = *(ptrA + 1);
*(ptrA + 1) = temp;
```



Example 3 …

62

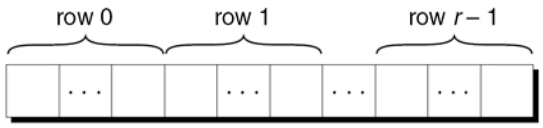
```
//Swap 2 array elements
int A[] = { 5, 8, 2, 6, 3};
int temp;
int *ptrA = A;
temp = *ptrA;
*ptrA = *(ptrA + 1);
*(ptrA + 1) = temp;
```



Pointers and Multidimensional Arrays

63

- Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays.
- C stores multi-dimensional arrays in row-major order.
- Layout of a 2D array with *r* rows:



- If p initially points to the element in row 0, column 0, we can visit every element in the array by incrementing p repeatedly.

Processing the Elements of a 2D Array

64

- Consider the problem of initializing all elements of the following array to zero:

```
int a[NUM_ROWS][NUM_COLS];
```

- The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

- If we view a as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
//OR for (p = a; p <= &a[NUM_ROWS-1][NUM_COLS-1; p++)
    *p = 0;
```

Processing the Rows of a 2D Array

- A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.
- To visit the elements of row `i`, we'd initialize `p` to point to element 0 in row `i` in the array `a`:

  `p = &a[i][0];`

  or we could simply write

  `p = a[i];`
- The expression `a[i]` is a pointer to the first element in row `i`.

Example


```
#include <stdio.h>
int MAX = 4;
int main () {
   char suit[][9] = {"Hearts", "Diamonds",  "Clubs",
     "Spades" };
   for (int i = 0; i < MAX; i++)
     printf("Value of names[%d] = %s\n", i, suit[i]);
   return 0;
}
```

fixed number of columns per row

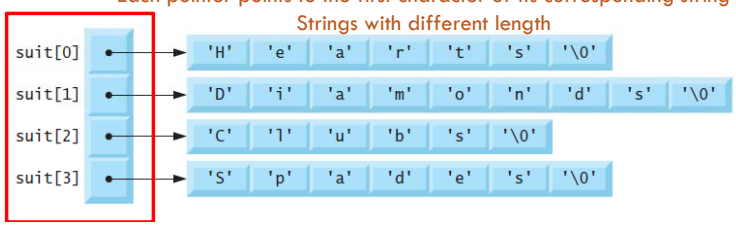
Suit[0]	'H'	'e'	'a'	'r'	't'	's'	'\0'		
Suit[1]	'D'	'i'	'a'	'm'	'o'	'n'	'd'	's'	'\0'
Suit[2]	'C'	'i'	'u'	'b'	's'	'\'0'			
Suit[3]	'S'	'p'	'a'	'd'	'e'	's'	'\0'		

Example ...


```
#include <stdio.h>
int MAX = 4;
int main () {
   char *suit[] = {"Hearts", "Diamonds",  "Clubs",
     "Spades" };
   for (int i = 0; i < MAX; i++)
     printf("Value of names[%d] = %s\n", i, suit[i]);
   return 0;
}
```

Each pointer points to the first character of its corresponding string

Strings with different length



Pointers and Functions

Parameter Passing

Pass/Call by Reference

Passing an Array by Reference

Pointer Types of functions

Passing Parameters to a Function

69

- Call by Value (default)
  - When the argument is the name of a variable
  - A copy of the parameter is sent to the function to view/edit it locally.
  - The outcome of editing the copy does not affect the original.
- Call by Reference
  - When the argument is a pointer variable or an array
  - A copy of the 'memory address' or pointer that tells the function where to find the original value is sent to the function
  - **Efficient**: save both unnecessary memory allocations (on the stack) and also memory transfers of data (during marshalling of the function call).
  - **Dangerous:** you are telling the function where to find the original copy of data

Swap - Call by Value

70

```
void swap(int a,int b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
}
```

2 3
2 3

```
main()
{
 int x = 2, y = 3;

 printf("%d %d\n",x,y);
 swap(x,y);
 printf("%d %d\n",x,y);
}
```

Changes made in function swap are lost when the function execution is over

Swap - Call by Reference

71

```
void swap2(int *aptr,
           int *bptr)
{
  int temp;

  temp = *aptr;
  *aptr = *bptr;
  *bptr = temp;

}
```

2 3
3 2

```
main()
{
 int x = 2, y = 3;

 printf("%d %d\n",x,y);
 swap2(&x, &y);
 printf("%d %d\n",x,y);
}
```

Changes made in function swap are done on original x and y and.
So they do not get lost when the function execution is over

Cube – Call by Value

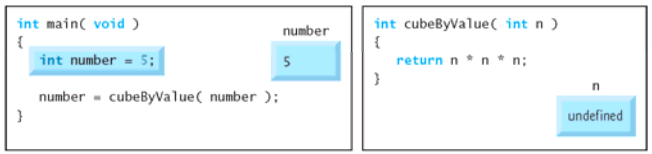
- passes the variable **number** by value to function cubeByValue

```
// Fig. 7.6: fig07_06.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue( int n ); // prototype

int main( void )
{
   int number = 5; // initialize number

   printf( "The original value of number is %d", number );

   // pass number by value to cubeByValue
   number = cubeByValue( number );

   printf( "\nThe new value of number is %d\n", number );
} // end main

// calculate and return cube of integer argument
int cubeByValue( int n )
{
   return n * n * n; // cube local variable n and return result
} // end function cubeByValue
```

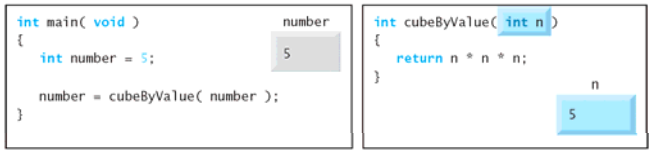
```
The original value of number is 5
The new value of number is 125
```

Analysis of cubeByValue

Step 1: Before `main` calls `cubeByValue`:

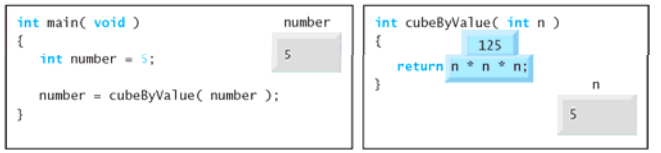


Step 2: After `cubeByValue` receives the call:

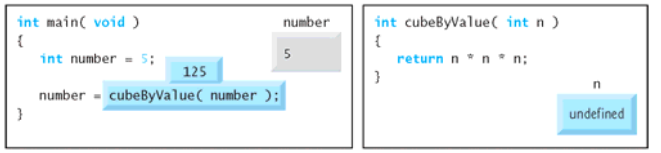


Analysis of cubeByValue ...

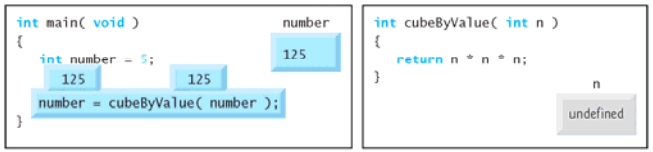
Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



Step 5: After `main` completes the assignment to `number`:



Cube – Call by Reference

- passes the variable `number` by reference (pointer) to function `cubeByReference`

```
// Cube a variable using pass-by-reference with a pointer argument.
#include <stdio.h>

void cubeByReference( int *nPtr ); // function prototype

int main( void )
{
   int number = 5; // initialize number

   printf( "The original value of number is %d", number );

   // pass address of number to cubeByReference
   cubeByReference( &number );

   printf( "\nThe new value of number is %d\n", number );
} // end main

// calculate cube of *nPtr; actually modifies number in main
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
} // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

`number` is passed by reference—the address of `number` is passed to function `cubeByReference`

Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr`

Cube – Call by Reference ...

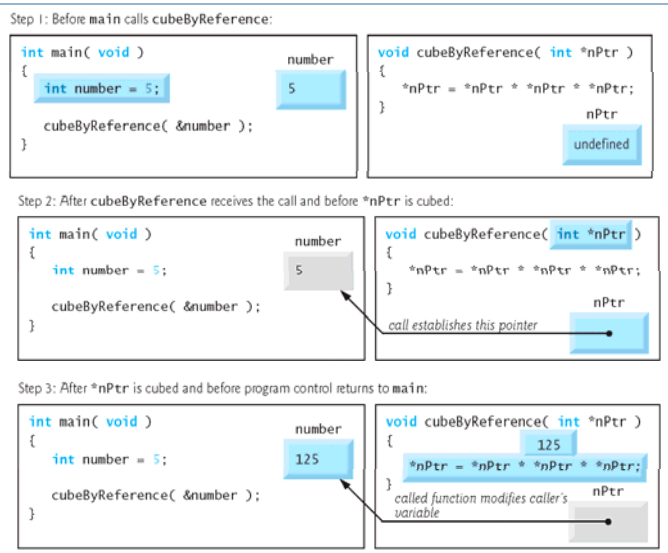
- Prototype: `void cubeByReference (int *nPtr);`
  - `cubeByReference` *receives* the *address* of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.
- Function Definition

```
void cubeByReference (int *nPtr)
{
      *ptr = * ptr * * ptr * * ptr;
}
```

no return, since the output is by modifying the parameter

- Function Call: `cubeByReference(&number);`

Analysis of cubeByReference



Pointers as Arguments

- Function definition:

```
void decompose(double x, long *int_part,
               double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

- Possible prototypes for decompose:

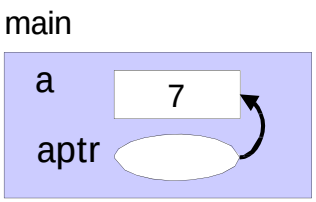
```
void decompose(double x, long *int_part,
               double *frac_part);
void decompose(double, long *, double *);
```

- A call of decompose:

```
decompose(3.14159, &i, &d);
```

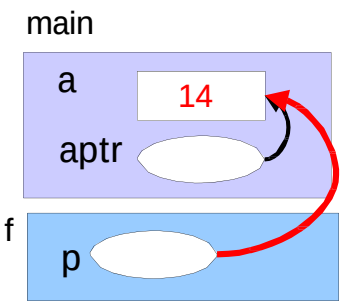
Example


```
void f(int *p) {
  *p = *p * 2;
}
int main() {
   int a = 7;
   int *aptr = &a;
   printf("%d",a); ->  7
   f(aptr);
   printf("%d",a);
   f(&a);
   printf("%d",a);
   return 0;
}
```



Example ...

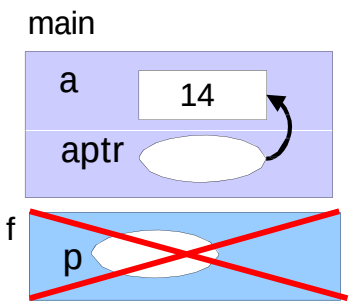

```
void f(int *p) {
  *p = *p * 2;
}
int main() {
   int a = 7;
   int *aptr = &a;
   printf("%d",a);
   f(aptr);
   printf("%d",a);
   f(&a);
   printf("%d",a);
   return 0;
}
```



Example ...

81

```
void f(int *p) {
  *p = *p * 2;
}
int main() {
    int a = 7;
   int *aptr = &a;
   printf("%d",a);
   f(aptr);
   printf("%d",a);
   f(&a);
   printf("%d",a);
   return 0;
}
```

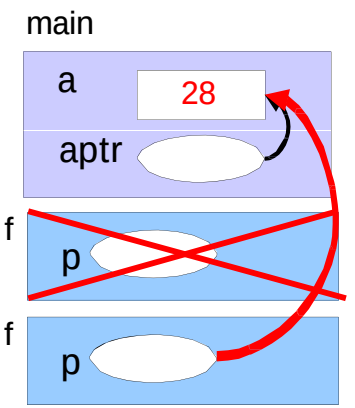


-> **14**

Example ...

82

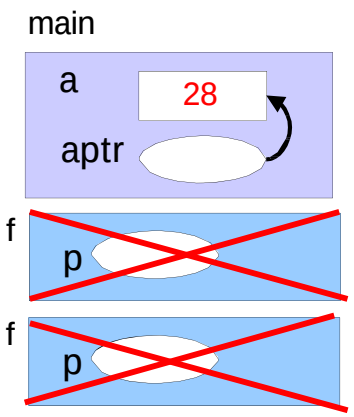
```
void f(int *p) {
  *p = *p * 2;
}
int main() {
    int a = 7;
   int *aptr = &a;
   printf("%d",a);
   f(aptr);
   printf("%d",a);
   f(&a);
   printf("%d",a);
   return 0;
}
```



Example ...

83

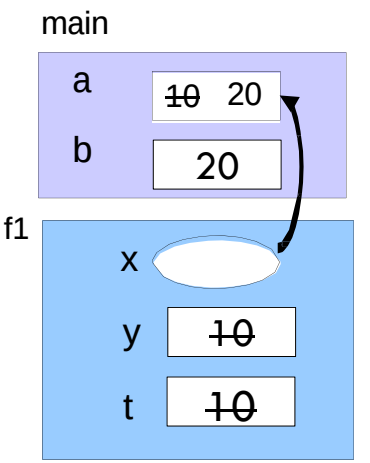
```
void f(int *p) {
  *p = *p * 2;
}
int main() {
   int a = 7;
   int *aptr = &a;
   printf("%d",a);
   f(aptr);
   printf("%d",a);
   f(&a);
   printf("%d",a);
   return 0;
}
```



-> **28**

Question 1

```
#include <stdio.h>
void f1(int *x, int y) {
  int t;
  t=*x;
  *x=y;
  y=t;
}
int main() {
   int a=10, b=20;
   f1(&a,b);
   printf("%d %d\n",a,b);
   return 0;
}
```



Question 2 – Trace a program

85

```
#include <stdio.h>
void max_min(int a, int b, int c, int *max, int *min){
  *max = a;
  *min = a;
  if (b > *max) *max = b;
  if (c > *max) *max = c;
  if (b < *min) *min = b;
  if (c < *min) *min = c;
  printf("F: %d  %d\n", max, *max);
}
int main() {
  int x, y;
  max_min(4, 3, 5, &x, &y);
  printf("First: x = %d  y = %d\n", x, y);
  max_min(x, y, 2, &x, &y);
  printf("Second: x = %d  y = %d\n", x, y);
  return 0;
}
```

name	Addr	Value
x	1	
y	2	
	3	
	4	
	5	
a	6	
b	7	
c	8	
max	9	1
min	10	2

Passing an Array by Reference

86

- For a function that expects a single-subscripted array as an argument, the function's prototype and header can use the pointer notation.
- The compiler does not differentiate between a function that receives a pointer and one that receives a single-subscripted array.
- This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference.

Example 1

87

```
  int find_largest(int a[], int n)
// int find_largest(int *a, int n)
    int i, max;
    max = a[0];
    for (i = 1; i < n; i++)
      if (a[i] > max)
        max = a[i];
    return max;
  }
```

A function Prototype:

```
int find_largest(int *, int );
```

A function call :

```
int b[] = {30,6,98, 45}, N = 4;
int largest = find_largest(b, N);
```

Example 2

88

```
int BubbleSort(int A[], int size, int order);
int BubbleSort(int *ptrA, int size, int order);
```

- Both prototypes accomplish a pass by reference.
- When passed to a function, an array name is treated as a pointer.
- Function call:

```
#define ASCENDING 1
#define DESCENDING -1
...
myArray[4] = {1, 5, 2, 65};
BubbleSort(myArray, 4, ASCENDING);
```

  - This call causes a pointer to the first element of `myArray` to be assigned to A; the array itself isn't copied.

Example 3

89

```
#include <stdio.h>
#define SIZE 10
void copyStr( char *s1, const char *s2 );
//void copyStr( char s1[], const char s2[] );
int main( void ) {
   char string1[SIZE]; // create array string1
   char *string2 = "Hello"; // create a pointer to a string
   copyStr( string1, string2 );
   printf( "string1 = %s\n", string1 );
}
void copyStr( char *s1, const char *s2 ) {
     while (*s2 != '\0') {
      *s1 = *s2;
      s1++; s2++;
     }
}
```

Array Arguments

90

- The fact that an array argument is treated as a pointer has some important consequences.
- The time required to pass an array to a function doesn't depend on the size of the array.
  - There's no penalty for passing a large array, since no copy of the array is made.
  - Declaring a variable to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Array Arguments ...

91

- An array parameter can be declared as a pointer if desired, But the same isn't true for a *variable*.
- The following declaration causes the compiler to set aside space for 10 integers:

  `int a[10];`
- The following declaration causes the compiler to allocate space for a pointer variable:

  `int *a;`

Array Arguments ...

92

- In (`int *a;`), `a` is not an array; attempting to use it as an array can have disastrous results.
- For example, the assignment

  `*a = 0;  /*** WRONG ***/`

  will store 0 where `a` is pointing.
- Since we don't know where `a` is pointing, the effect on the program is undefined.

Pointers as Return Values

- Pointers are just another data type, so functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A call of the max function:

```
int *p, i, j;
…
p = max(&i, &j);
```

After the call, p points to either i or j.

Example

- Be careful to differentiate between return values that are pointers (addresses in RAM) versus data (meaningful values).

```
double * Larger( double X, double Y ) {
  if( X > Y ) return &X ;
  else        return &Y ;
}
```

Assuming `double U=10.34, W=82.754, Z, *W;`

one can use this function to access the result

```
Z = *(Larger(U,W));
```

OR

```
W = Larger(U,W);
```

Const Qualifier

non-const pointer to non-const data
non-const pointer to const data
const pointer to non-const data
const pointer to const data

The **const** qualifier

- When an argument is a pointer to a variable X, we normally assume that X will be modified:

```
f(&x);
```

- It's possible, though, that f merely needs to examine the value of X, not change it.
- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.

The **const** qualifier …

97

- Using `const` provides *protection* (hence security) for data.
- We can use `const` to document that a function won't change an object whose address is passed to the function.
- `const` goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

Attempting to modify `*p` is an error that the compiler will detect.

The **const** qualifier …

98

- With pointers there are four kinds of patterns to consider.
  - non-const pointer to non-const data
  - non-const pointer to const data
  - const pointer to non-const data
  - const pointer to const data

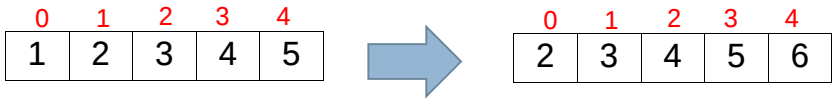
Non-const Pointer to Non-const Data

99

- When both the pointer, and the data to which it points, are intended to be modified
- Example:
  - Assume: `int N = 5, A[5] = {1,2,3,4,5} ;`
  - Function:

```
void Add1( int *B, int n ) {
    for( ; n>0 ; n--, B++ )
        (*B)++ ;
  }
```

  - Note that both the pointer B, and the array values (in A) are modified during the call: `Add1(A,N);`



Non-const Pointer to Const Data

100

- When the pointer to data is intended to be modified, but the data to which it points must not be modified
- Example
  - Assume: `int N;`

```
Const char A[]={'M','e','s','s','a','g','e','\0'} ;
```

  - Function:

```
int StrLen (const char *S) {
     int n = 0 ;
     for( ; *S != '\0' ; S++ ) n++ ;
     return n ;
 }
```

  - Note that the pointer S is modified, but the values in the string A are not modified in the call: `N = StrLen(A) ;`
    - Any attempt to modify data in A results in a compiler error.

Const Pointer to Non-const Data

- When the pointer to data must not be modified, but the data to which it points may be modified
- Example
  - Assume: `float Pi = 3.14159, *const ptrPi = &Pi ;`
  - Function:

```
void ChangePi ( float *const P ) {
     const float Q = 3.14 ;
     *P = 3.1415 ;
     P = &Q ;          // Compiler error !
 }
```

  - Note that the value of `Pi` may change, but the pointer P may not be modified to point at Q in the call: `ChangePi(ptrPi);`

Const Pointer to Const Data

- When neither the pointer to data, or the data to which it points, may be modified
- This technique is used to promote maximum protection over data and references to data
- Example:

```
int main ( ) {
    int N = 3, *ptrN = &N ;
    const int X = 5, *const ptrX = &X ;
    *ptrX = 3 ;    // Compiler error
    ptrX = &N ;  // Compiler error
    ptrX = ptrN ;  // Compiler error
}
```

- Always think about data protection when using const.

Lecture 4: Summary

- Pointer Concepts & Mechanisms
- Pointers Operations and Arithmetic
- Pointers and Arrays
- Pointes and Functions
- Assigned Reading:
  - Chapter 7 – Pointers
  - Chapter 8 – Characters and Strings in the C language
- Assignment
  - Deadline of second assignment is Feb. 12, 2017 at 23:59.
  - The third lab assignment has been posted on Blackboard.
- Midterm Exam #1, Mon. **Feb 6**, 2017 , **Education Bldg, Room #1101**.