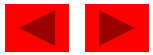
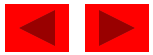
Chapter 7 - Pointers

Outline



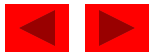
Objectives

- In this chapter, you will learn:
  - To be able to use pointers.
  - To be able to use pointers to pass arguments to functions using call by reference.
  - To understand the close relationships among pointers and arrays.
  - To understand the use of pointers to functions.



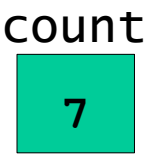
7.1 Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

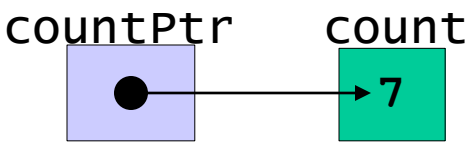


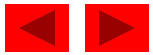
7.2 Pointer Variable Definitions and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value





7.2 Pointer Variable Definitions and Initialization

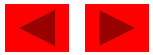
- Pointer definitions
  - `*` used with pointer variables

```
int *myPtr;
```

  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a * before each variable definition

```
int *myPtr1, *myPtr2;
```

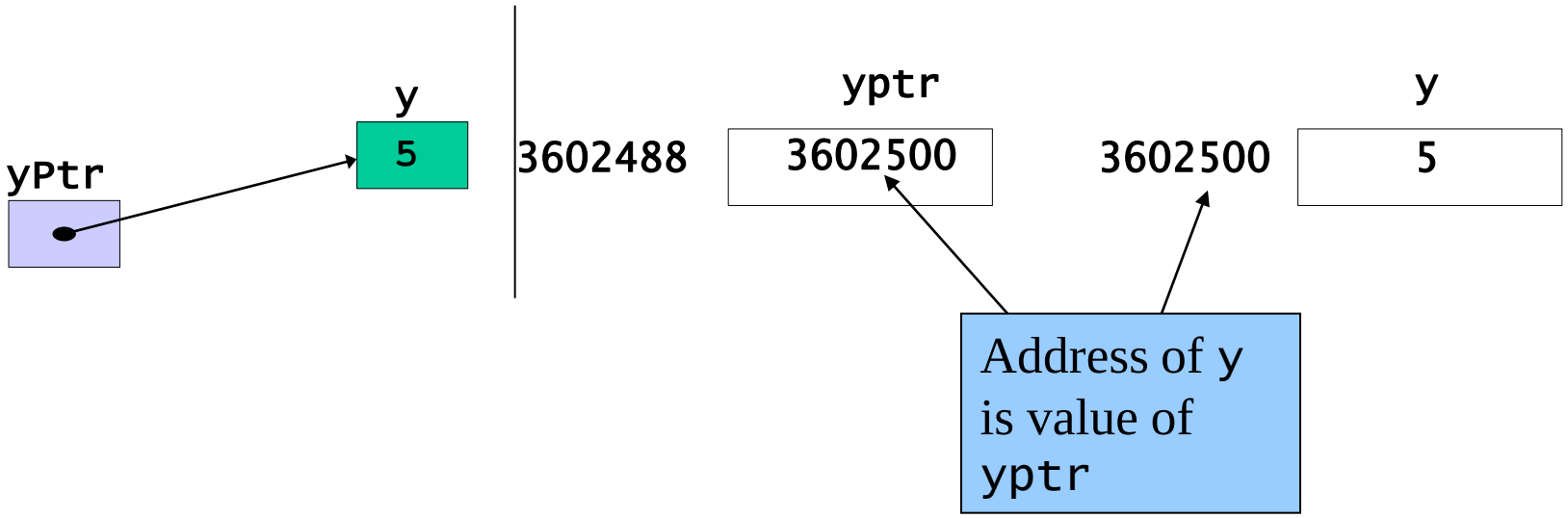
  - Can define pointers to any data type
  - Initialize pointers to 0, `NULL`, or an address
    - 0 or `NULL` – points to nothing (`NULL` preferred)

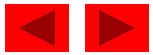


7.3 Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr "points to" y
```

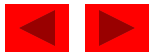




7.3 Pointer Operators

```
#include "stdafx.h"
void main()
{
int y = 5;
int *yPtr;
yPtr = &y;
printf("\ny=%d", y);
printf("\n*yPtr=%d", *yPtr);

printf("\n\n&y=%d", &y);
printf("\nyPtr=%d", yPtr);
printf("\n\n&yPtr=%d", &yPtr);
}
```

```
Program Output

y=5
*yPtr=5

&y=3602500
yPtr=3602500

&yPtr=3602488
```

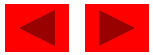
7.3 Pointer Operators

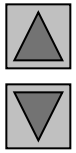
- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yptr` returns `y` (because `yptr` points to `y`)
  - * can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - They cancel each other out

```
printf("&*yPtr = %d", &*yPtr); ⟶ 3602500
```



Outline

fig07_04.c

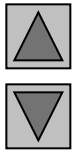
```c
/*
   Using the & and * operators */
#include <stdio.h>

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

   a = 7;
   aPtr = &a;    /* aPtr set to 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 );

   return 0; /* indicates successful termination */

} /* end main */
```

The address of a is the value of aPtr.

The * operator returns an alias to what its operand points to. aPtr points to a, so *aPtr returns a.

Notice how * and & are inverses

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

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

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```



Outline

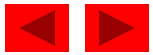
Program Output

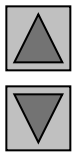
7.4 Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer
- * operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - `*number` used as nickname for the variable passed

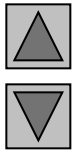




Outline

fig07_06.c

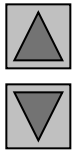
```
1  /*
2     Cube a variable using call-by-value */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* prototype */
6
7  int main()
8  {
9     int number = 5; /* initialize number */
10
11    printf( "The original value of number is %d", number );
12
13    /* pass number by value to cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "\nThe new value of number is %d\n", number );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
22 /* calculate and return cube of integer argument */
23 int cubeByValue( int n )
24 {
25    return n * n * n;   /* cube local variable n and return result */
26
27 } /* end function cubeByValue */
```



Outline

Program Output

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



Outline

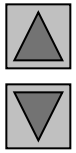
fig07_07.c

```
1  /*
2     Cube a variable using call-by-reference with a pointer argument */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* prototype */
7
8  int main()
9  {
10    int number = 5; /* initialize number */
11
12    printf( "The original value of number is %d", number );
13
14    /* pass address of number to cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
23 /* calculate cube of *nPtr; modifies variable number in main */
24 void cubeByReference( int *nPtr )
25 {
26    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
27 } /* end function cubeByReference */
```

Notice that the function prototype takes a pointer to an integer.

Notice how the address of `number` is given - `cubeByReference` expects a pointer (an address of a variable).

Inside `cubeByReference`, `*nPtr` is used (`*nPtr` is `number`).



Outline

Program Output

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

Before `main` calls `cubeByValue` :

```
int main()
{
   int number = 5;

   number=cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

After `cubeByValue` receives the call:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

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

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

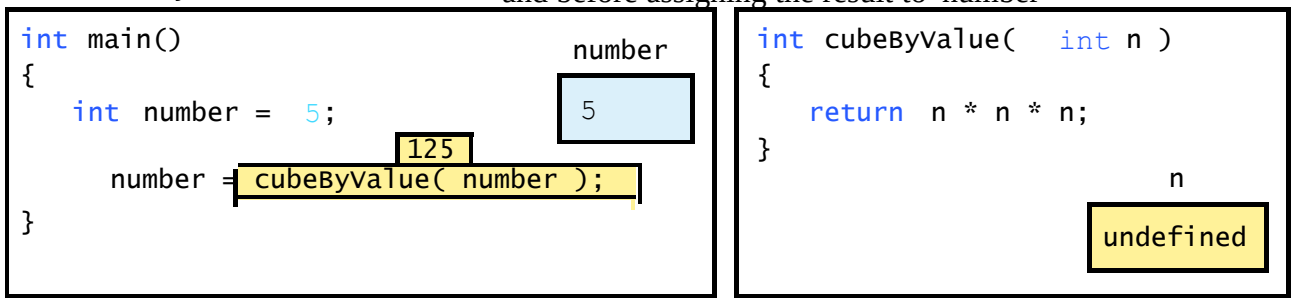
```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

Fig. 7.8 Analysis of a typical call-by-value. (Part 1 of 2.)

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



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

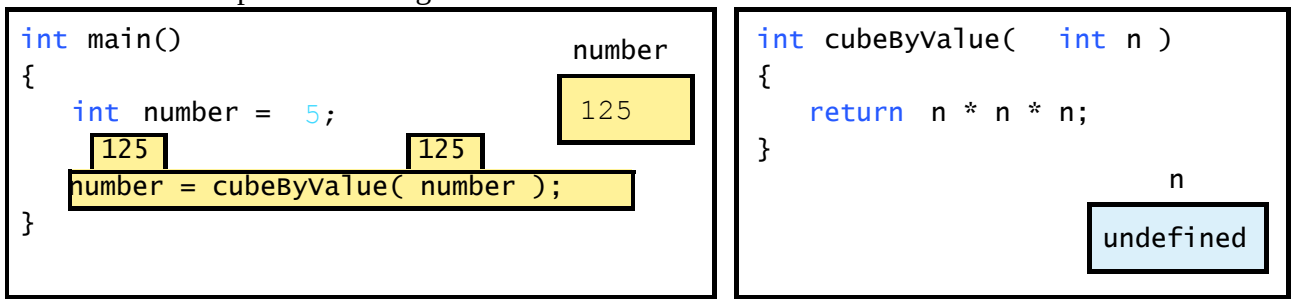
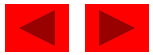


Fig. 7.8 Analysis of a typical call-by-value. (Part 2 of 2.)



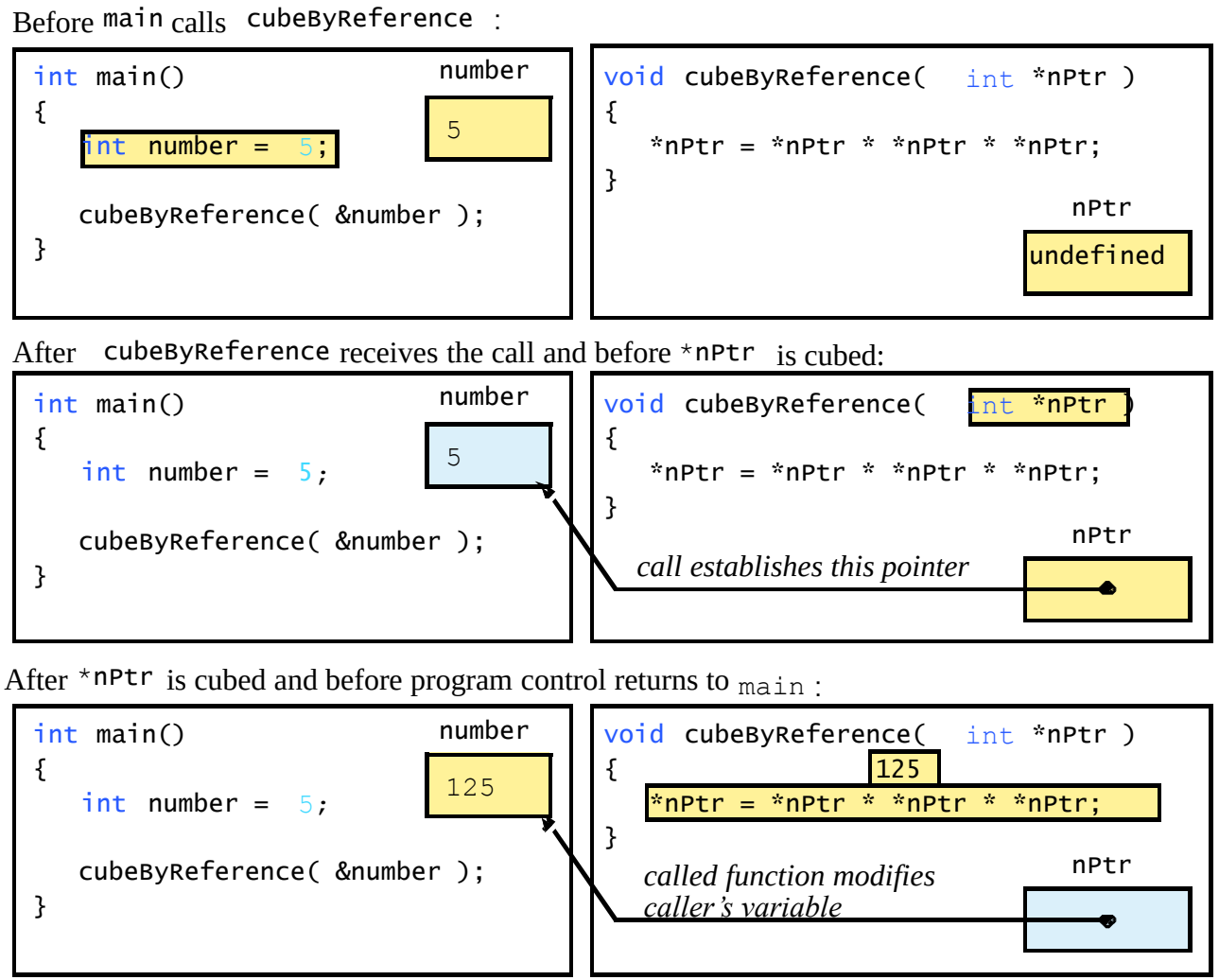
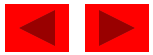


Fig. 7.9 Analysis of a typical call-by-reference with a pointer argument.

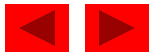
7.5 Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - Swap two elements
  - `swap` function must receive address (using &) of array elements
    - Array elements have call-by-value default
  - Using pointers and the * operator, `swap` can switch array elements



```
#include "stdafx.h"
void swap(int *p)
{
   int temp,i,j;
   for (i = 1; i < 7; i++)
   {
     for (j = 0; j < 6; j++)
     {
      if (*(p+j) > *(p+j+1))
      {
        temp = *(p + j);
        *(p + j) = *(p + j + 1);
        *(p + j + 1) = temp;
       }
     }
   }
}

void main()
{
   int arr[7] = { 64, 34, 25, 12, 22, 11, 90 }, i;
   printf("\nOriginal Values\n");
   for (i = 0; i < 7; i++)
       printf("%4d", arr[i]);
   swap(arr);
   printf("\nSorted Values\n");
   for (i = 0; i < 7; i++)
      printf("%4d", arr[i]);
}
```

```
Original Values
  64   34   25   12   22   11   90
Sorted Values
  11   12   22   25   34   64   90
```

7.6 Pointer Expressions and Pointer Arithmetic

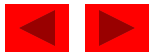
- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or −−)
  - Add an integer to a pointer(+ or += , - or −=)
  - Operations meaningless unless performed on an array

```
*p+=1;
++*p;
*p++;
```

all of them increments the value of the address in p by 1

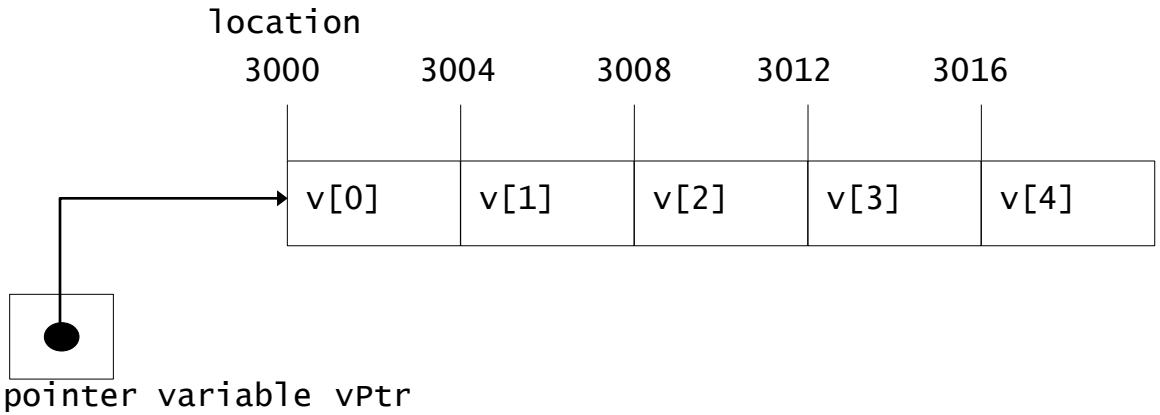
```
*(p++);
```

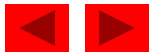
is indicating the value of the NEXT address which p has



7.6 Pointer Expressions and Pointer Arithmetic

- 5 element `int` array on machine with 4 byte `int`s
  - `vPtr` points to first element `v[ 0 ]`
    - at location `3000` (`vPtr = 3000`)
  - `vPtr += 2;` sets `vPtr` to `3008`
    - `vPtr` points to `v[ 2 ]` (incremented by 2), but the machine has 4 byte `int`s, so it points to address `3008`





7.7 The Relationship Between Pointers and Arrays

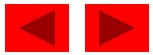
- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

```
bPtr = b;
```

    - The array name (`b`) is actually the address of first element of the array `b[ 5 ]`

```
bPtr = &b[ 0 ]
```

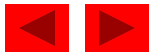
    - Explicitly assigns `bPtr` to address of first element of `b`



7.7 The Relationship Between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself

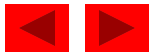
    ```
    *( b + 3 )
    ```

7.7 The Relationship Between Pointers and Arrays

Given the declarations on the page 22

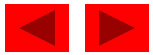
Elements / Values	
ArrayNotaion	Pointer Notation
b[0]	*bPtr
b[1]	*(bPtr + 1)
b[2]	*(bPtr + 2)
b[3]	*(bPtr + 3)
b[4]	*(bPtr + 4)

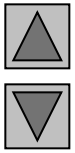


7.7 The Relationship Between Pointers and Arrays

Given the declarations on the page 22

Addresses	
Array Notaion	Pointer Notation
&b[0]	bPtr
&b[1]	(bPtr + 1)
&b[2]	(bPtr + 2)
&b[3]	(bPtr + 3)
&b[4]	(bPtr + 4)



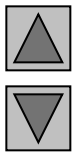


Outline

fig07_20.c (Part 1 of 2)

```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
```

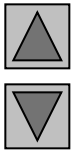
```
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```



Outline

fig07_20.c (Part 2 of 2)

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```



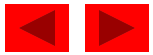
Outline

Program Output

7.9 Exercises

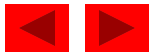
```
#include "stdafx.h"
void main()
{
 int x = 1, y = 2;
 int *ip;
 ip = &x;
 y = *ip;
 *ip = 3;
 printf("x=%d\ny=%d\n*ip=%d",x,y,*ip);
}
```

Answer: x=3 y=1 *p=3



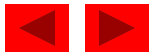
```
#include "stdafx.h"
void main()
{
	int x=1,*p;
	p=&x;
	*p=*p+10;
	printf("x=%d  *p=%d\n",x,*p);
}
```

Answer: x=11 *p=11



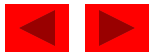
```
#include "stdafx.h"
void main()
{
        int a=3,b,*pa,*pb;
        pa=&a;
        b=*pa;
        pb=&b;
     printf("a=%d\nb=%d\n*pa=%d\n*pb=%d\n",a,b,*pa,*pb);

}
```

Answer: a=3 b=3 *pa=3 *pb=3



```
#include "stdafx.h"
void main()
{
	int a,b,c=3,*p;
	p=&c;
	a=2*(c+*p);
	b=*p+4;
	printf("a=%d b=%d\n",a,b);
}
```

Answer: a=12 b=7



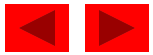
```
#include "stdafx.h"
void main()
{
        int a=1,b=2,*p;
        printf("a=%d b=%d\n",a,b);
        p=&a;
        *p=6;
        printf("a=%d b=%d\n",a,b);
        p=&b;
        *p=0;
        printf("a=%d b=%d\n",a,b);
}
```

Answer:

a=1 b=2

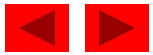
a=6 b=2

a=6 b=0



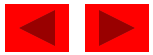
```c
#include "stdafx.h"
void main()
{
    int myarray[4] = {1,2,3,0};
    int *ptr = myarray;
    printf("*ptr=%d\n", *ptr);
}
```

Answer: *ptr=1



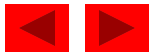
```
void main()
{
        int a[]={2,4,3,1,6,7},i,*p;
        p=a;

        for(i=0;i<6;i++)
        printf("%4d",a[i]);
        printf("\n\n");

        p=p+2;
        *p=*a+3;
        for(i=0;i<6;i++)
        printf("%4d",*(a+i));
        printf("\n\n");

        *(p+2)=*(a+1) * 2;
        for(i=0;i<6;i++)
        printf("%4d",*(a+i));
    printf("\n\n");
}
```

```
   2   4   3   1   6   7

   2   4   5   1   6   7

   2   4   5   1   8   7
```

```
include "stdafx.h"
void main()
{
  int myarray[4]= {1,2,3,0},*p,*x;
  p = myarray;
  x = p;
  p = p + 1;
  printf("*p = %d, *x = %d\n", *p, *x);
}
```

Answer: *p = 2, *x = 1



CALL BY VALUE

```
#include "stdafx.h"
void func(int p)
{
	p=p+2;
}
void main()
{

	int x=5;
	printf("before x=%d\n",x);
	func(x);
	printf("after x=%d\n",x);
}
```

CALL BY REFERENCE

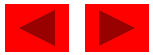
```
#include "stdafx.h"
void func(int *p)
{
	*p=*p+2;
}
void main()
{

	int x=5;
	printf("before x=%d\n",x);
	func(&x);
	printf("after x=%d\n",x);
}
```

```
before x=5
after x=5
```

```
before x=5
after x=7
```

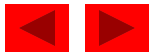

```
#include "stdafx.h"
void func(int *p)
{
        for(int i=0;i<5;i++)
                *(p+i)=*(p+i)*2;
}
void main()
{

        int x[]={5,6,7,8,9};
        printf("before \n");
        for(int i=0;i<5;i++)
                printf("%4d", *(x+i));
        func(x);
        printf("\nafter \n");
        for(int i=0;i<5;i++)
                printf("%4d", *(x+i));
}
```

before
5 6 7 8 9
after
10 12 14 16 18



What will be the output?

```
#include "stdafx.h"
void test(int *px, int *py, int pz)
{
  *px+=5;
  printf("%3d + %3d + %3d = %3d\n",*px,*py,pz, *px+*py+pz);
  *py+=(*px)++;
  printf("%3d + %3d + %3d = %3d\n",*px,*py,pz, *px+*py+pz);
  pz=((*py)+=2)*3;
  printf("%3d + %3d + %3d = %3d\n",*px,*py,pz, *px+*py+pz);

}

void main()
{
 int x=-3,y=6,z=2;
 printf("%3s%6s%6s%9s\n","x","y","z","x+y+z");
 printf("---------------   ------\n");
 printf("%3d + %3d + %3d = %3d\n",x,y,z,x+y+z);
 test(&x,&y,z);
 printf("%3d + %3d + %3d = %3d\n",x,y,z,x+y+z);
 getch();
}
```

What will be the output?

```
#include "stdafx.h"
void main()
{
 int u1,u2,i;
 int list[5]={10,15,20,25,30};
 int *ip;
 ip=list+2;
 printf("The Adress of ip : %d  \t ",ip);
 printf("Value of ip : %d  \n ",*ip);
  for (int i=0;i<5;i++)
               printf("LIST [%d]:%d  \t ",i,list[i]);
 ip=ip-1;
 printf("\n\nThe Adress of ip : %d  \t ",ip);
 printf("Value of ip : %d  \n ",*ip);

 *(ip+2)=*ip;
 printf("\n\nThe Adress of ip : %d  \t ",ip);
 printf("Value of ip : %d  \n ",*ip);

  for (i=0;i<5;i++)
      printf("LIST [%d] :%d  \t ",i,list[i]);
   *(list+4)=*(ip-1)+2;
  printf("\n \n The Adress of ip : %d  \t ",ip);
  printf("Value of ip : %d  \n ",*ip);

  for (i=0;i<5;i++)
     printf("LIST [%d] :%d  \t ",i,list[i]);
}
```