Chapter 7

Pointers

Chapter 7 - Pointers

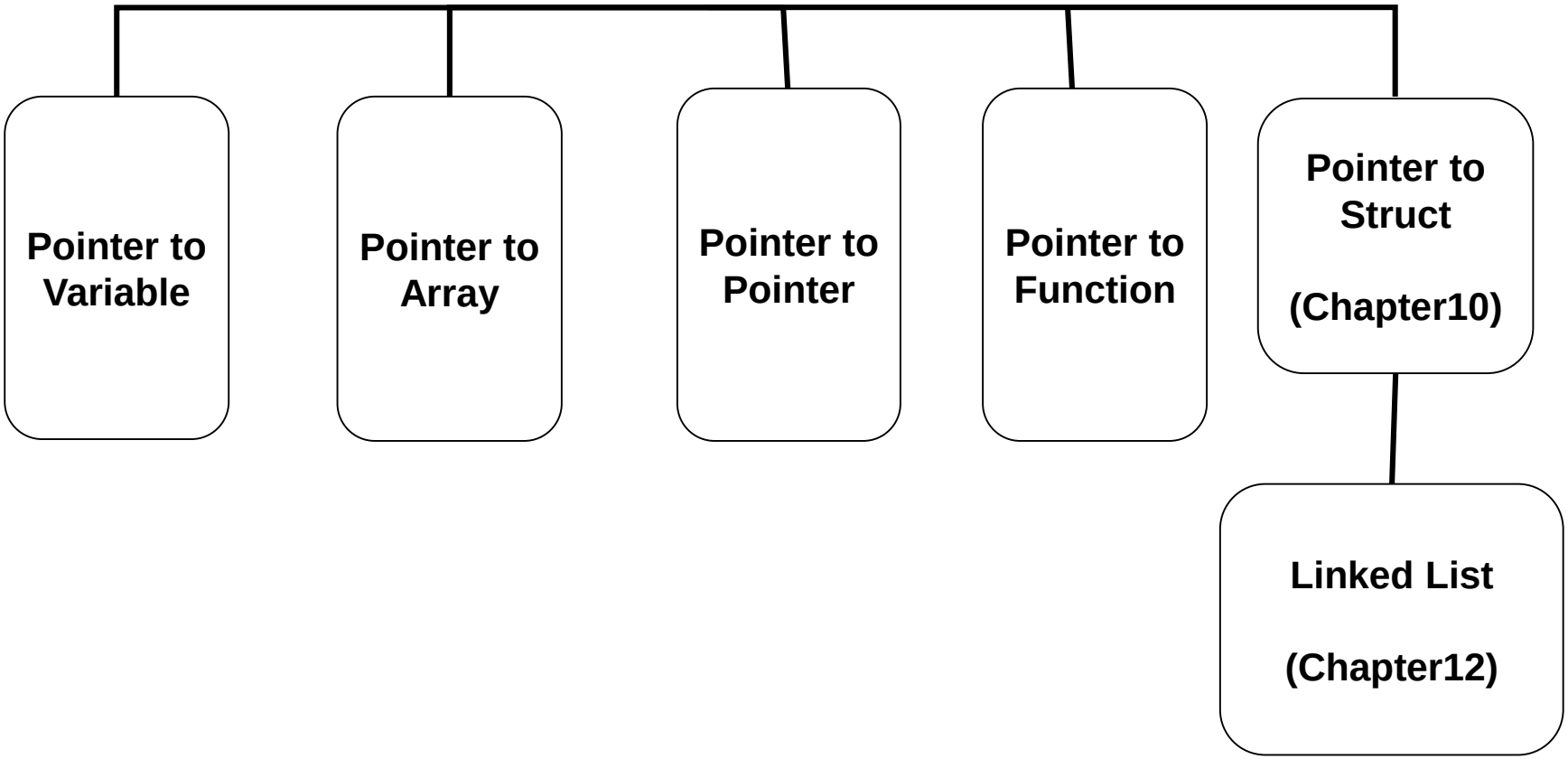
Outline

7.1 Introduction

- Pointers
  - A pointer is a variable that contains memory address of another variable (or function).
  - Powerful, but difficult to master.
  - Close relationship with arrays and strings.

  - *Mostly used for:*
    - *Dynamic memory allocation*
    - *Data Structures (Deitel Chapter 12)*

Categories of Pointers



Pointers to variables can be assigned with two methods:

1) Normal variable declaration
2) Dynamic memory allocation with *malloc()* function

Memory (RAM)

- In a 32-bit computer, memory locations are always 4 bytes.

  (1 byte = 8 bits,
  so 4 bytes = 32 bits).

- Memory addresses are also 4 bytes.

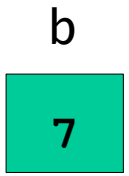
- Usually addresses are shown in hexadecimal notation.

Memory Address	Memory Content (4 bytes each)
00000000	?
00000004	?
00000008	?
0000000C	?
00000010	?
.....	.....
FFFFFFFF	?

7.2 Pointer Variable Definitions and Initialization

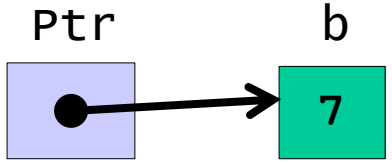
- **Normal variables** contain a specific value (**direct reference**)

```
int  b  = 7;
```



- **Pointer variables** contain address of a variable that has a specific value (**indirect reference**)
- Indirection means referencing a pointer value
- The content (value) of a pointer variable is actually just a long integer.

```
int * Ptr;

Ptr = &b;
```



Pointer Definitions

- 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)

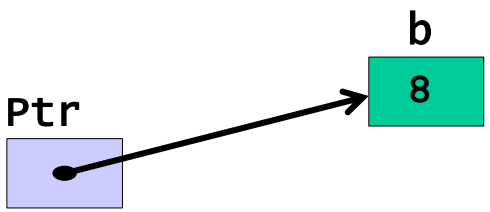
Pointer Operators

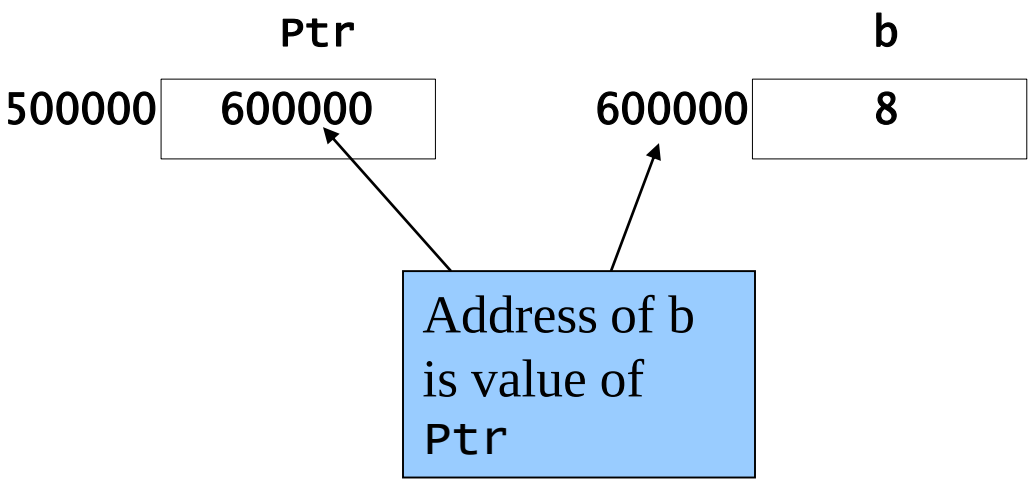
OPERATOR	MEANING	WHEN USED
&	Reference operator	• Used to get Memory Address of a variable.
*	Dereference operator	1) Used to get Memory Content of a pointed variable. 2) Also used to define a pointer variable.

7.3 Pointer Operators

- & (reference operator)
  - Returns address of operand

```
int b = 8;
int *Ptr;    // Ptr is defined as a pointer to int
Ptr = &b;    // Ptr gets address of variable b
                (Now Ptr "points to" b )
```





7.3 Pointer Operators

- * (dereference operator)
  - Returns a synonym/alias of what its operand points to
  - `*Ptr` returns `b` (because `Ptr` points to `b`)
  - * can be used for assignment (returns alias to an object)

```
*Ptr = 5;  // changes b to 5
5 = *Ptr;  // not valid
```

- * and & are inverses
  - They cancel each other out

Example: Pointer operators

```c
/* Fig. 7.4: fig07_04.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 );

} // 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

Program Output

```
The address of a is 0022FF44
The value of aPtr is 0022FF44


The value of a is 5
The value of *aPtr is 5


Showing that * and & are complements of each other.

&*aPtr = 0022FF44
*&aPtr = 0022FF44
```

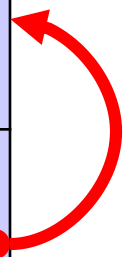
```
int a;
int * aPtr;
```

Variable Name	Memory Address	Memory Content
a	0022FF44	?
aPtr	0022FF48	?
	0022FF4C	?
	0022FF50	?

a = 5;

aPtr = &a;

Variable Name	Memory Address	Memory Content
a	0022FF44	5
aPtr	0022FF48	0022FF44
	0022FF4C	?
	0022FF50	?



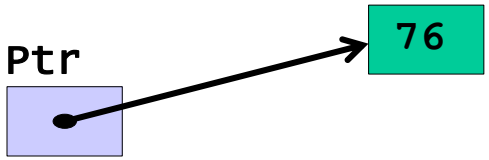
Operator Precedences

Operators						Associativity	Type
()	[]					left to right	highest
++	--	!	* (dereference)	& (reference)	(type)	right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical and
||						left to right	logical or
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma

Dynamic Memory Allocation

- Use the *malloc()* and *sizeof()* functions which are defined in <stdlib.h>

```
int  *Ptr;
Ptr = malloc ( sizeof (int) );  // Allocate 4 bytes and get the address
*Ptr = 76;
```



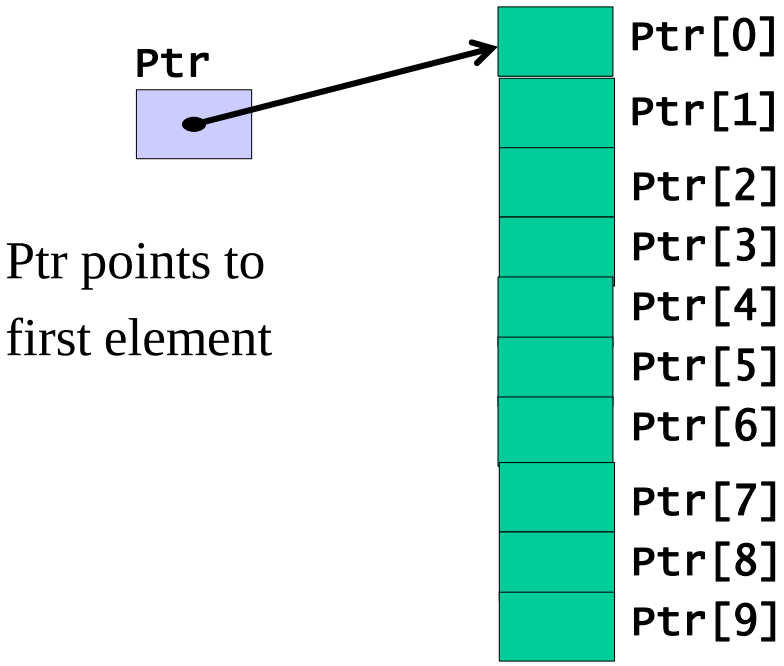
- Note that the allocated place is unnamed, which can be accessible thru the Ptr only.

```
Ptr = malloc ( 4 ); // Same as above
```

Dynamic Array Allocation

- Use the malloc() function with number of elements

```
int  *Ptr;
Ptr = malloc ( 10 * sizeof (int) ); // Allocate 40 bytes and get the address
```




```
Ptr = malloc ( 40 ); // Same as above
```

Example: Pointer to Dynamically Allocated Variable

```c
#include <stdio.h>

int main()  {
 int  *  Ptr;  // Ptr is null at this moment.

 Ptr = malloc ( sizeof (int) ) ;
 // Dynamically allocate 4 bytes in memory and assign its address to Ptr

 *Ptr = 6;  // Assign 6 the place which is pointed by Ptr

 printf("Initial Content:\n");
 printf("*Ptr = %d \n", *Ptr); // Display content of pointed place
 printf("Ptr = %p  \n", Ptr);  // Display address of pointed place
 printf("&Ptr = %p  \n", &Ptr);  // Display address of pointed place

 *Ptr  =  *Ptr  +  3; // Increment content of pointed place

 printf("Content after increment:\n");
 printf("*Ptr = %d \n", *Ptr);
 printf("Ptr = %p  \n", Ptr);
 printf("&Ptr = %p  \n", &Ptr);  // Display address of pointed place

} // end main
```

Program Output

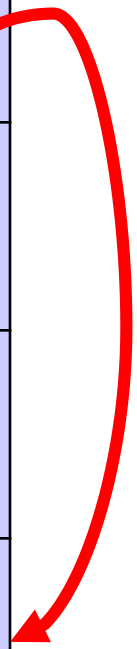
```
Initial Content:
*Ptr = 6
Ptr = 002D0F58
&Ptr = 0022FF44


Content after increment:
*Ptr = 9
Ptr = 002D0F58
&Ptr = 0022FF44
```

int * Ptr ;

Variable Name	Memory Address	Memory Content
Ptr	0022FF44	?
	0022FF48	?
	0022FF4C	?
	0022FF50	?

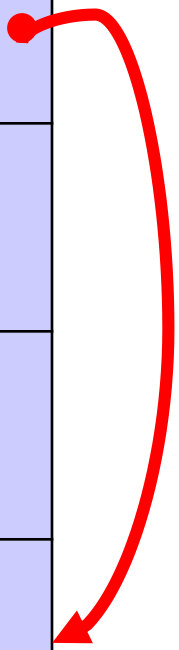
Ptr = malloc (sizeof (int)) ;

Variable Name	Memory Address	Memory Content
Ptr	0022FF44	002D0F58
	.....	.....
	.....	.....
	002D0F58	?



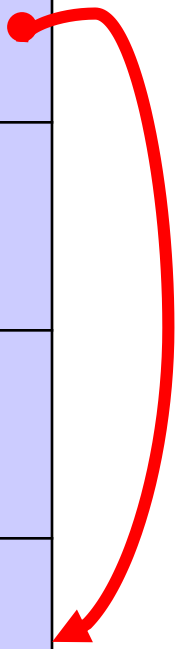
*Ptr = 6 ;

Variable Name	Memory Address	Memory Content
Ptr	0022FF44	002D0F58
	.....	.....
	.....	.....
	002D0F58	6



*Ptr = *Ptr + 3 ;

Variable Name	Memory Address	Memory Content
Ptr	0022FF44	002D0F58
	.....	.....
	.....	.....
	002D0F58	9



Example: Strings

```c
#include <stdio.h>

int main() {
  char * ogrenci1 = "Ahmet";
  char ogrenci2[ ] = "Mehmet";

  printf("ogrenci1 = %s \n", ogrenci1);
  printf("&ogrenci1 = %p \n", &ogrenci1);

  printf("ogrenci2 = %s \n", ogrenci2);
  printf("&ogrenci2 = %p \n", &ogrenci2);

} // end main
```

Program Output

```
ogrenci1 = Ahmet
&ogrenci1 = 0022FF3C


ogrenci2 = Mehmet
&ogrenci2 = 0022FF20
```

sizeof() Function

- `sizeof`
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals 4 bytes, then

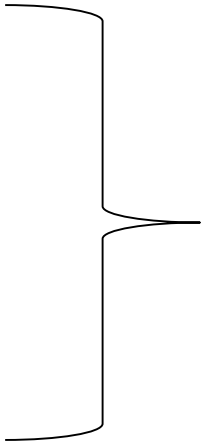
```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - will print 40

- `sizeof` can be used with
  - Variable names
  - Type name
  - Constant values

Sizeof Values for Data Types (in a 32-bit computer)

```
sizeof (char)          = 1
sizeof (int)           = 4
sizeof (float)         = 4
sizeof (double)        = 8
sizeof (long double)   = 12

sizeof (char *)        = 4
sizeof (int *)         = 4
sizeof (float *)       = 4
sizeof (double *)      = 4
```



All kind of pointers are 4 bytes

Example: Sizeof for an array and a pointer

```c
/* Fig. 7.16: fig07_16.c
   Sizeof operator when used on an array name
   returns the number of bytes in the array. */
#include <stdio.h>

size_t  getSize( float *ptr ); // prototype

int main()
{
   float array[ 20 ]; // create array
   printf( "The number of bytes in the array is %d"
           "\nThe number of bytes returned by getSize is %d\n",
           sizeof( array ), getSize( array ) );

} // end main


// return size of ptr
size_t   getSize( float *ptr ) {
   return sizeof( ptr );

} // end function getSize
```

Program Output

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

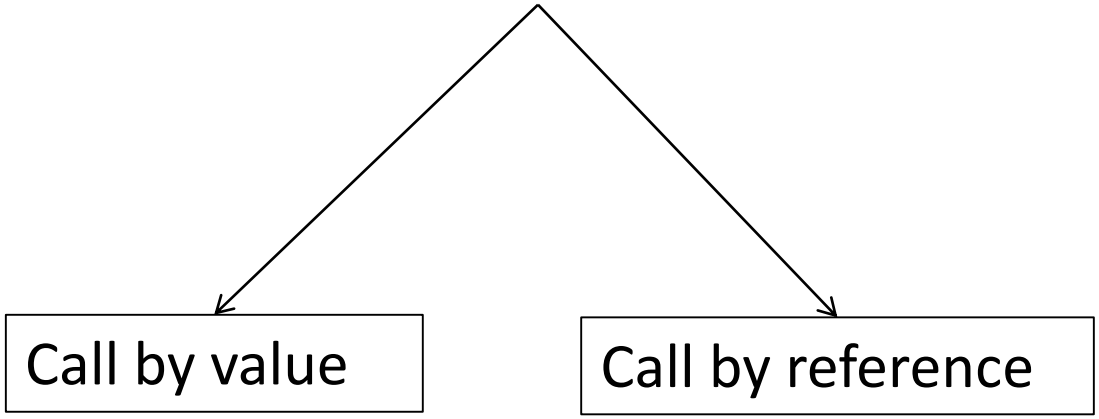
Example: Sizeof for data types

```c
/* Fig. 7.17: fig07_17.c
   Demonstrating the sizeof operator */
#include <stdio.h>
int main() {
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ];  // create array of 20 int elements
   int *ptr = array; // create pointer to array

   printf( "     sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n    sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
          sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
          sizeof( int ), sizeof l, sizeof( long ), sizeof f,
          sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
          sizeof( long double ), sizeof array, sizeof ptr );
} // end main
```

Program Output

```
     sizeof c = 1       sizeof(char)        = 1
     sizeof s = 2       sizeof(short)       = 2
     sizeof i = 4       sizeof(int)         = 4
     sizeof l = 4       sizeof(long)        = 4
     sizeof f = 4       sizeof(float)       = 4
     sizeof d = 8       sizeof(double)      = 8
    sizeof ld = 12      sizeof(long double) = 12

sizeof array = 80
  sizeof ptr = 4
```

Methods of Parameter Passing to a Function



- When an argument is passed call-by-value, a *copy* of the argument's value is made and passed to the called function.

- In call-by-reference, the addresses of the parameters are passed to the function. Thus, the function is given direct access to the parameters.

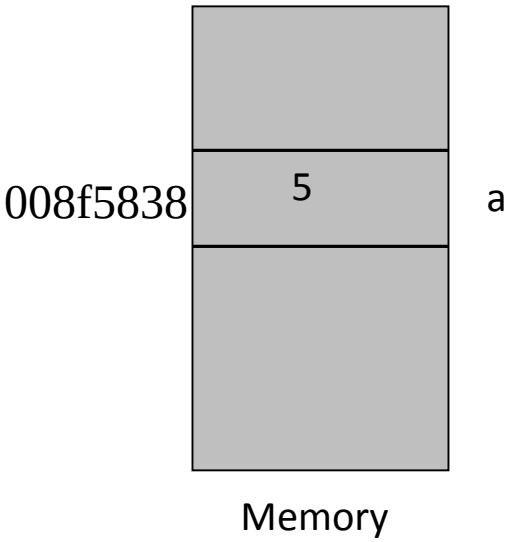
Call-by-Value

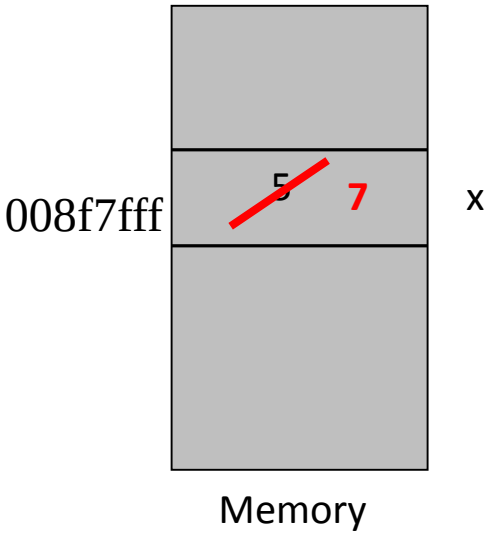
Main program

```
void main()
int a=5;
func(a);
cout << a;
}
```

Function

```
void func(int x)
{
   x=7;
}
```





When x=7 is executed in the function,
the value of x changes but the value of a does not.

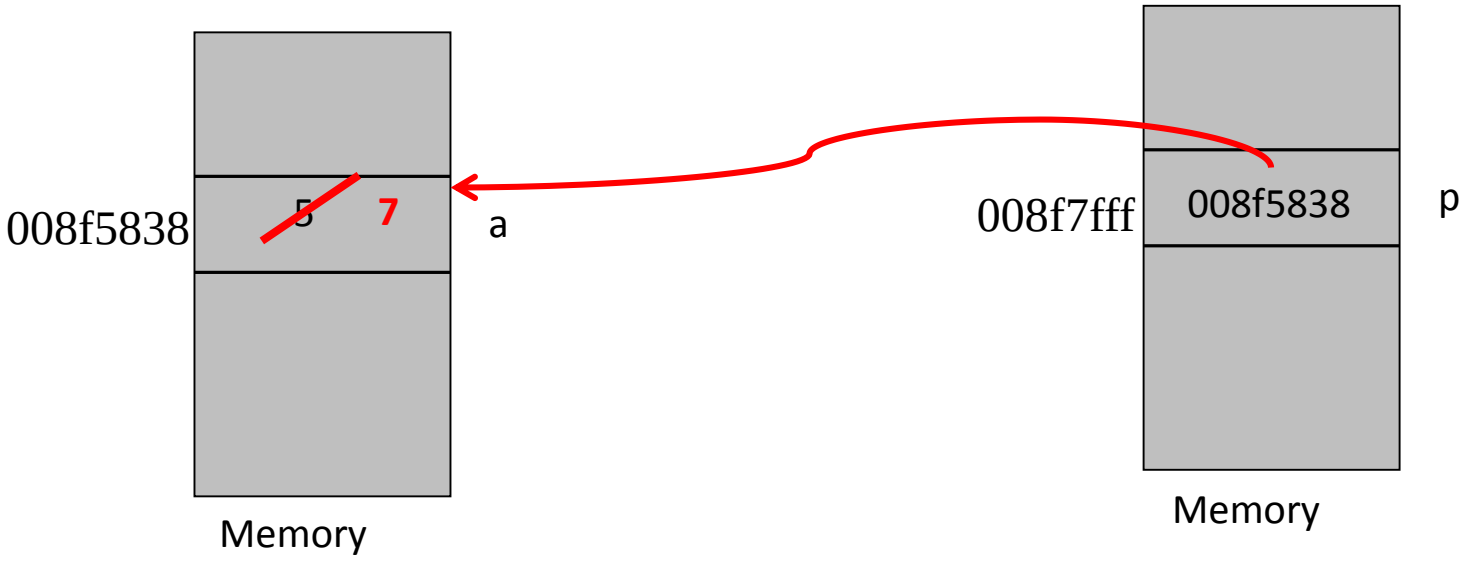
Call-by-Reference

Main program

```
void main()
int a=5;
func(&a);
cout << a;
}
```

Function

```
void func(int * p)
{
   *p=7;
}
```



When *p=7 is executed in the function, the value p points to (that is, a) changes.

7.4 Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change content of actual location in memory
  - Arrays are not passed with & because the array name is already a pointer

- * (dereference operator)
  - Used as alias/nickname for variable inside of function

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

  - *number used as nickname for the variable passed

Example: Cube Call-by-value

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

int cubeByValue( int n ); // prototype

int main() {
   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
```

Program Output

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

Example: Cube Call-by-reference


```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument
*/

#include <stdio.h>

void cubeByReference( int *nPtr ); // prototype

int main() {
   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; modifies variable number in main
void cubeByReference( int *nPtr ) {
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube *nPtr
} // 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).

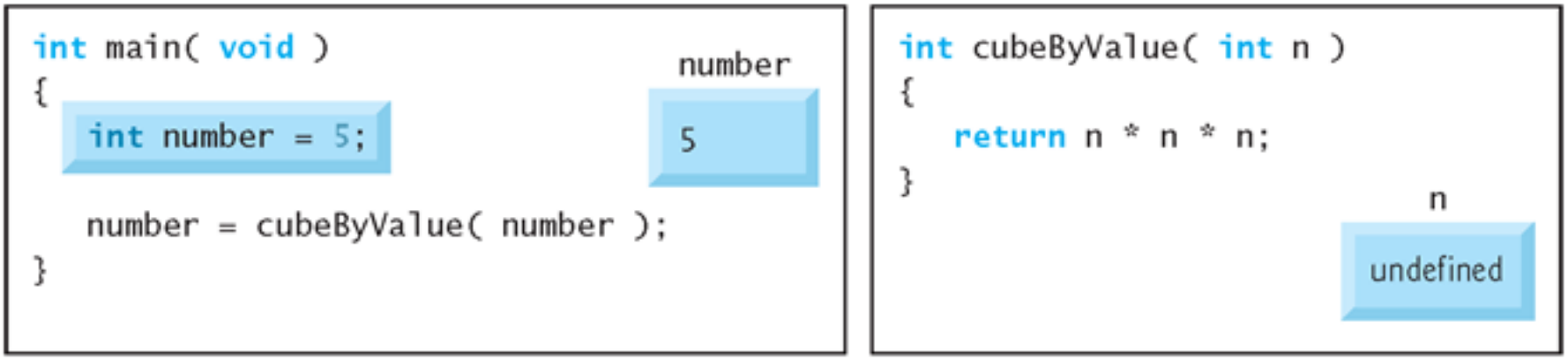
Inside `cubeByReference`, `*nPtr` is used (`*nPtr` is `number`).

Program Output

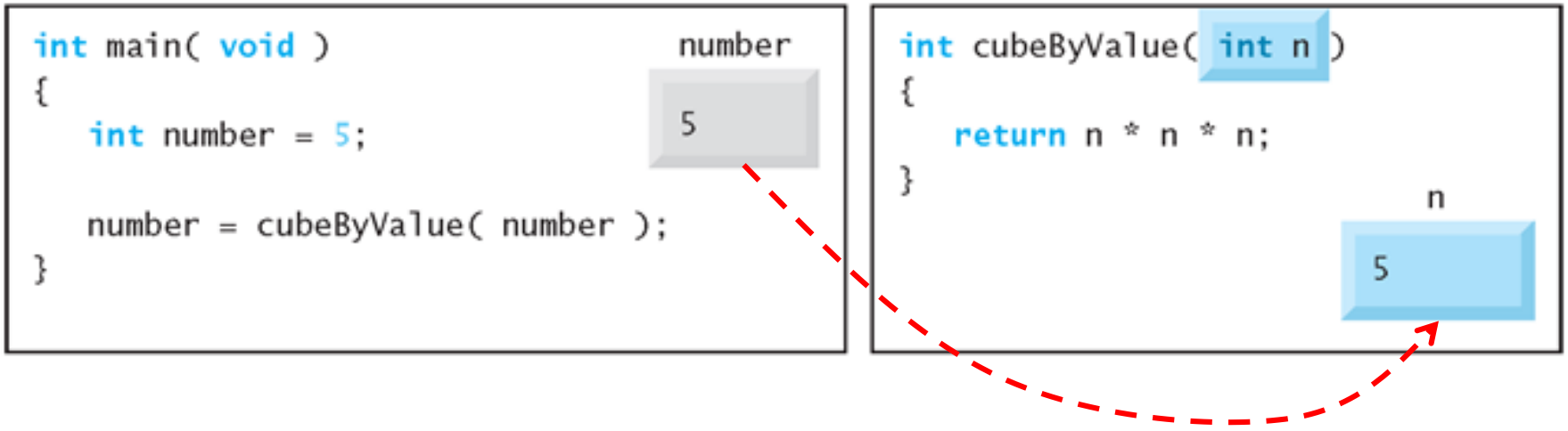
```
The original value of number is 5
The new value of number is 125
```

Analysis of a typical call-by-value. (Part 1 of 3)

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

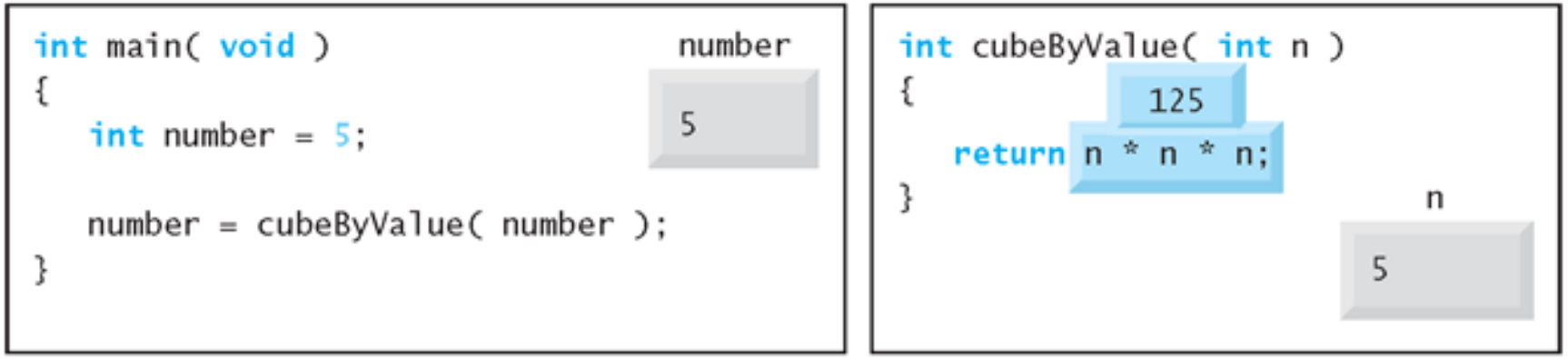


Step 2: After `cubeByValue` receives the call:

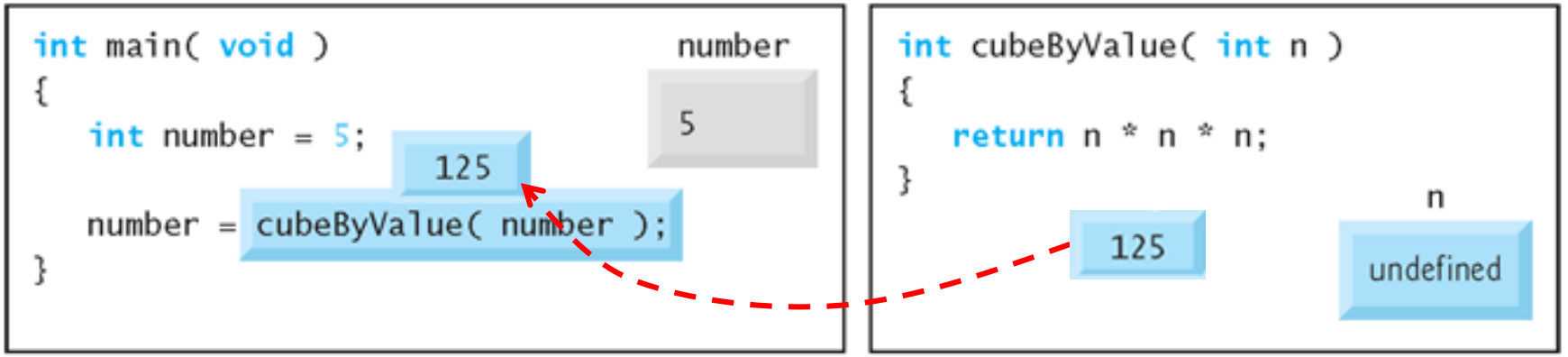


Analysis of a typical call-by-value. (Part 2 of 3)

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

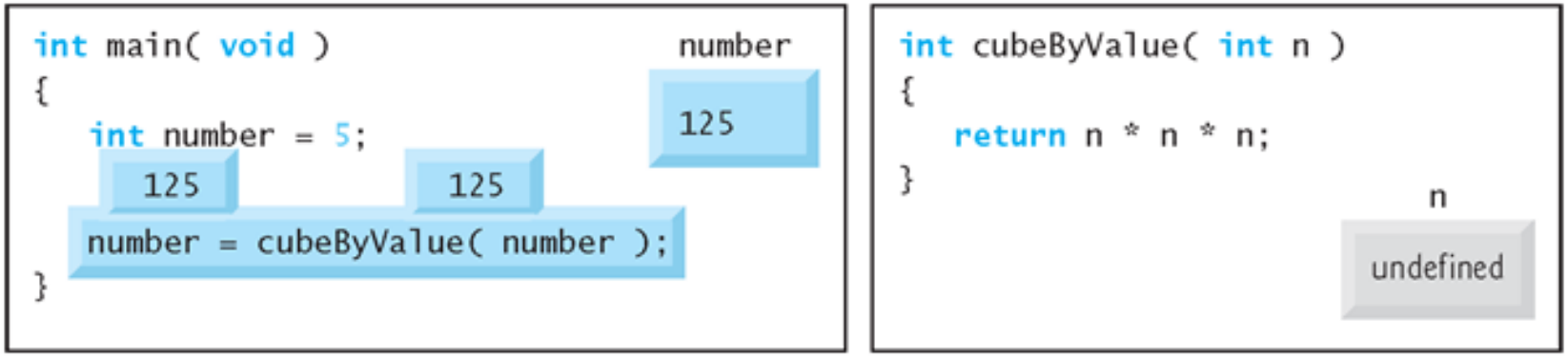


Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



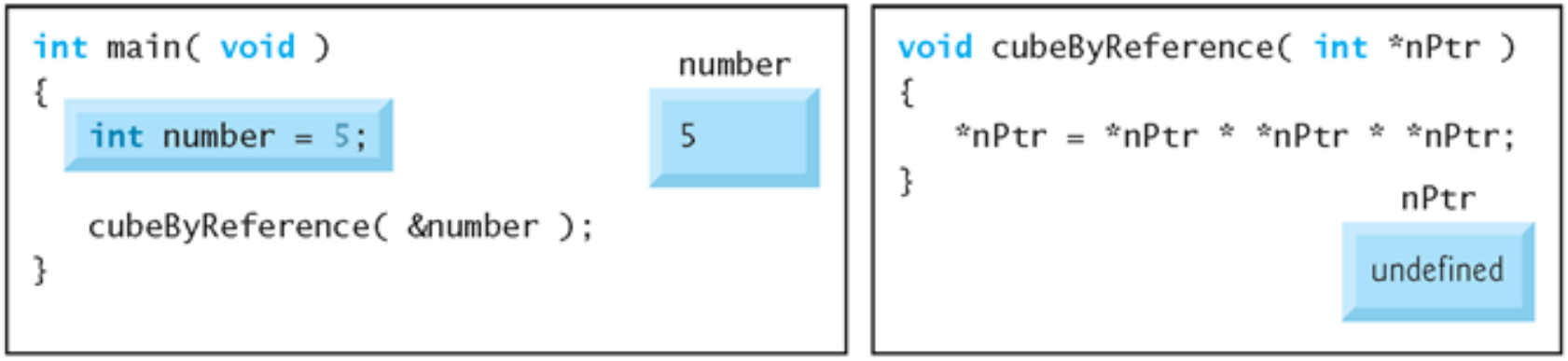
Analysis of a typical call-by-value. (Part 3 of 3)

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

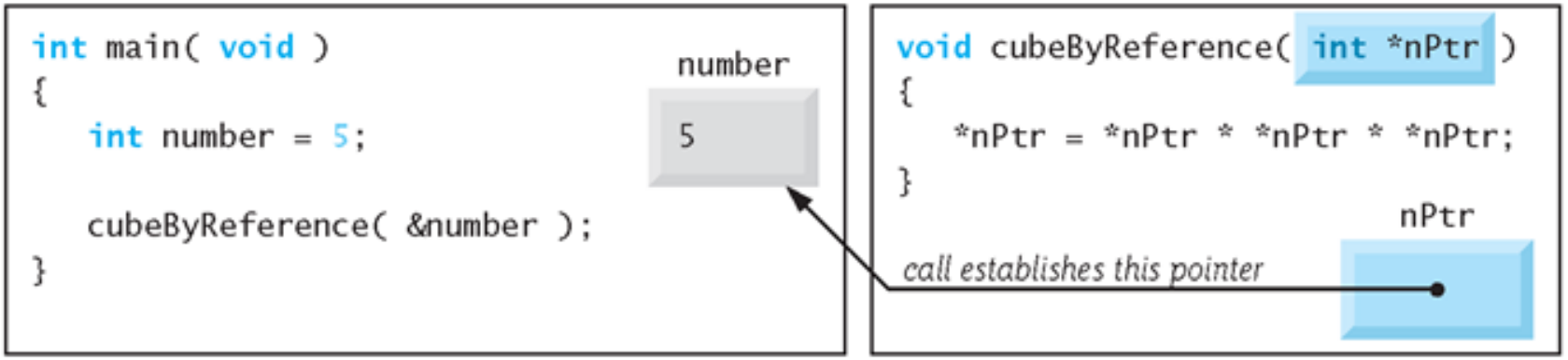


Analysis of a typical call-by-reference with a pointer argument. (Part 1 of 2)

Step 1: Before `main` calls `cubeByReference`:

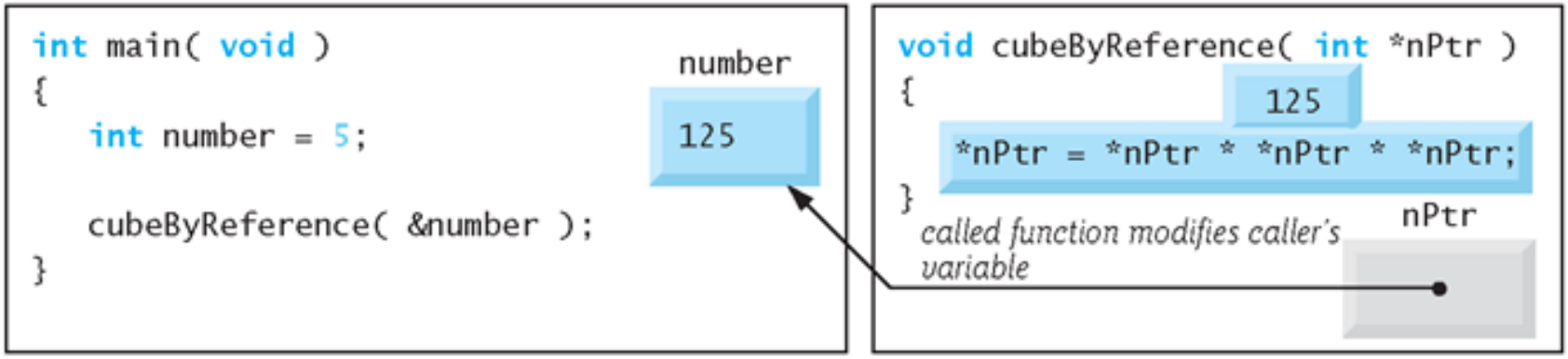


Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Analysis of a typical call-by-reference with a pointer argument. (Part 2 of 2)

Step 3: After *nPtr is cubed and before program control returns to main:



Example: String conversion (lower to upper)

Part 1 of 2

```c
/* Fig. 7.10: fig07_10.c
   Converting lowercase letters to uppercase letters
   using a non-constant pointer to non-constant data */

#include <stdio.h>
#include <ctype.h>

void convertToUppercase( char *sPtr ); // prototype

int main()
{
   char string[] = "characters and $32.98"; // initialize char array

   printf( "The string before conversion is: %s", string );
   convertToUppercase( string );
   printf( "\nThe string after conversion is: %s\n", string );


} // end main
```

Part 2 of 2

```
// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) // current character is not '\0'
   {
      if ( islower( *sPtr ) ) {    // if character is lowercase,
         *sPtr = toupper( *sPtr ); // convert to uppercase
      } // end if

      ++sPtr;  // move sPtr to the next character
   } // end while

} // end function convertToUppercase
```

Program Output

```
The string before conversion is: Expenses are $32.98

The string after conversion is: EXPENSES ARE $32.98
```

Example: String printing one by one

Part 1 of 2

```c
/* Fig. 7.11: fig07_11.c
   Printing a string one character at a time using
   a non-constant pointer to constant data */

#include <stdio.h>

void printCharacters( const char *sPtr ); // prototype

int main()
{
   // initialize char array
   char string[] = "Apple and Orange";

   printf( "The string is:\n" );
   printCharacters( string );
   printf( "\n" );


} // end main
```

Part 2 of 2

```
/* sPtr cannot modify the character to which it points,
   i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr )
{
   // loop through entire string
   for ( ; *sPtr != '\0'; sPtr++ ) {   // no initialization
      printf( "%c", *sPtr );
   } // end for

} // end function printCharacters
```

Program Output

```
The string is:
Apple and Orange
```

7.6 Bubble Sort Using Call-by-reference

- In Deitel Chapter-6, Bubble Sort was implemented in the main().
- Here, it will be implemented in a function.
- Implement the 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

- Psuedocode

  Initialize array

  Print data in original order

  Call function bubblesort

  Print sorted array

  Define the bubblesort function

Example: Bubble sort using Call-by-reference

Part 1 of 3

```c
/* Fig. 7.15: fig07_15.c
   This program puts values into an array, sorts the values into
   ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int * const array, const int size ); // prototype

int main()
{
   // initialize array a
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   int i; // counter

   printf( "Data items in original order\n" );

   // loop through array a
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } // end for

   bubbleSort( a, SIZE ); // sort the array
```

`Bubblesort` gets passed the address of array elements (pointers). The name of an array is a pointer.

Part 2 of 3

```
 printf( "\nData items in ascending order\n" );

   /* loop through array a */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */

} // end main
```

```
/* sort an array of integers using bubble sort algorithm */
void bubbleSort( int * const array, const int size ) {
   void swap( int *element1Ptr, int *element2Ptr ); // prototype
   int pass; // pass counter
   int j;    // comparison counter

   // loop to control passes
   for ( pass = 0; pass < size - 1; pass++ ) {

      // loop to control comparisons during each pass
      for ( j = 0; j < size - 1; j++ ) {

         // swap adjacent elements if they are out of order
         if ( array[ j ] > array[ j + 1 ] ) {
            swap( &array[ j ], &array[ j + 1 ] );
         } // end if

      } // end inner for

   } // end outer for

} // end function bubbleSort
```

Part 3 of 3

```
/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

Program Output

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37

Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

7.7 Pointer Expressions and Pointer Arithmetic

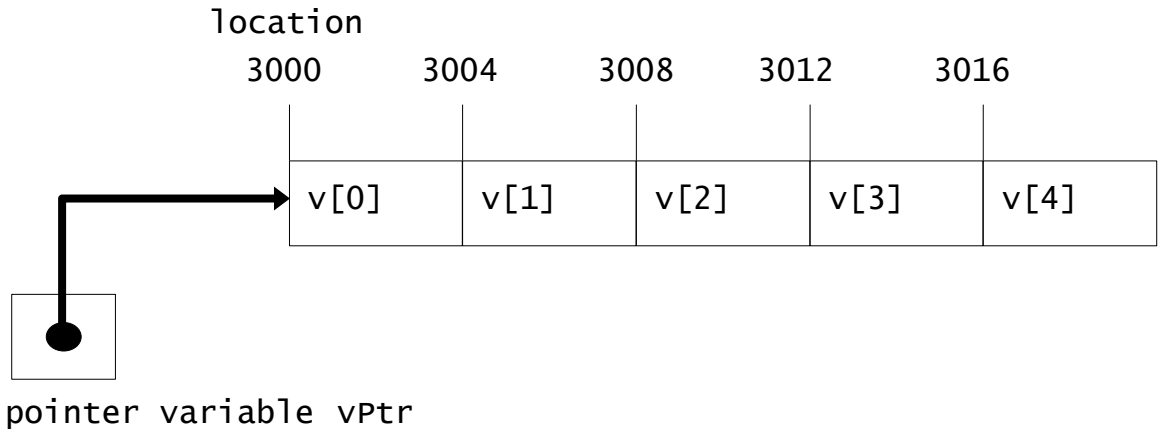
- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add/subtract an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array
  - Pointer multiplication and division are not used.

- In pointer arithmetic, adding/subtracting 1 really means adding/subtracting sizeof(data type).
  - For example *int pointer adds 4, *double pointer adds 8, and so on.

Adding to a Pointer

```
int v[5];
int * vPtr = & v;
```

- `vPtr` points to first element `v[ 0 ]`
- Assume array starts at location 3000 (`vPtr = 3000`)

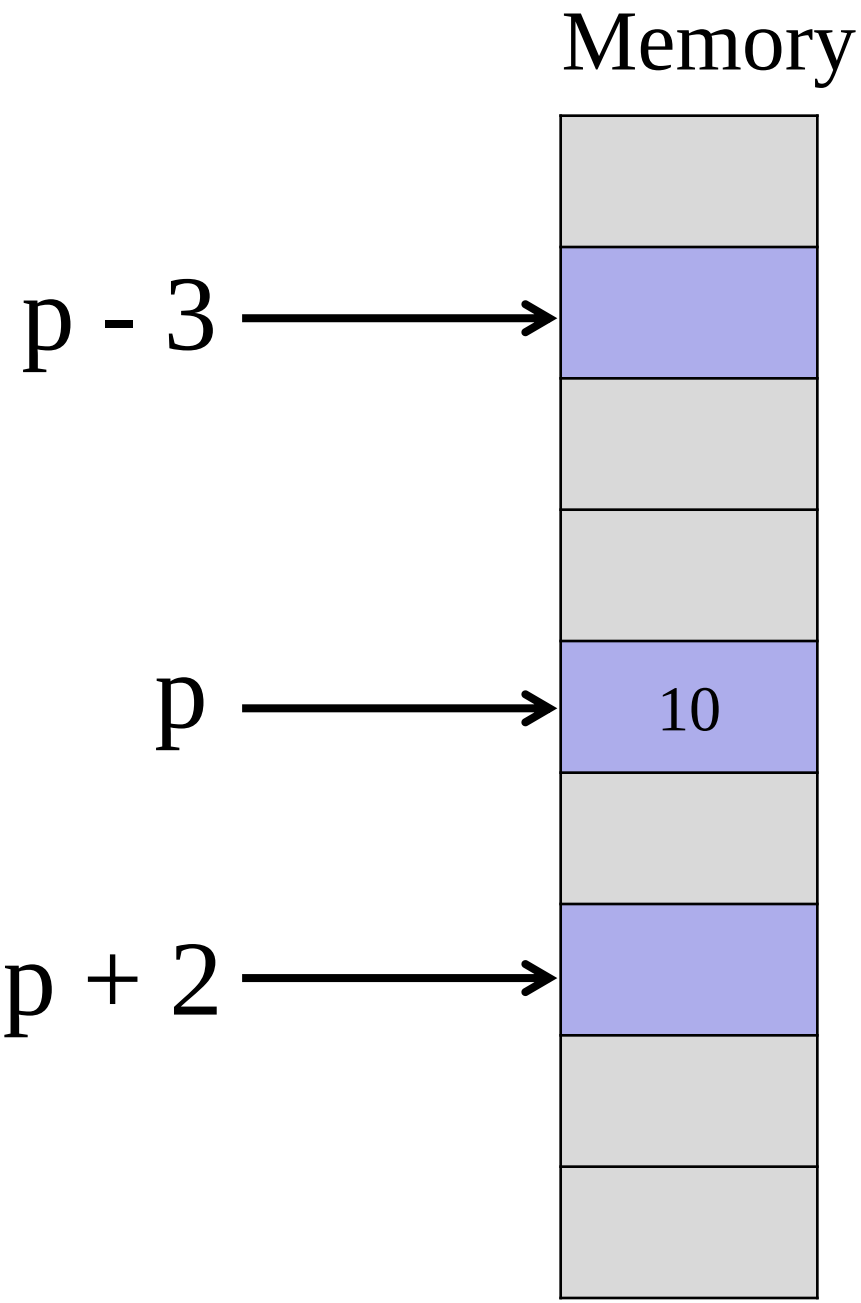



```
vPtr  += 2;
```
//sets vPtr to 3008

- `vPtr` now points to `v[ 2 ]` (incremented by 2), since the machine has 4 byte `ints`, so it points to address 3008

EXAMPLE:

```
int a = 10;
int *p = &a;
```



Subtracting two Pointers

- Subtracting pointers
  - Returns number of elements from one to the other.

```
int v[5] = {10,20,30,40,50};
int *vPtr1, *vPtr2;
int fark;

vPtr1 = &v[ 0 ];
vPtr2 = &v[ 3 ];
fark = vPtr2 - vPtr1;     // Result is 3
fark = *vPtr2 - *vPtr1;   // Result is 30
fark = v[3] - v[0];       // Result is 30
```

- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0 (NULL)`

Assignment between Pointers

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used

```
int a = 5;

int  *Ptr1 , *Ptr2;

Ptr1 = &a;  // Get address of a into Ptr1

Ptr2 = Ptr1;   // Copy address of a into Ptr2

printf("*Ptr1 = %d\n", *Ptr1); // Display 5

printf("*Ptr2 = %d\n", *Ptr2); // Display 5
```

Pointer to void

- Pointer to `void` (type `void *`)
  - Generic pointer, represents any type
  - No casting needed to convert a pointer to `void` pointer
  - `void` pointers cannot be dereferenced without typecasting

```
void  * Ptr;

int a = 5;
char isim[20] = "Mehmet Uslu";

Ptr  = &a;
printf("Ptr = %p \n", Ptr);  // Display address of a

Ptr  = &isim;
printf("Ptr = %p \n", Ptr); // Display address of isim
```

Example: Pointer to void

```
#include <stdio.h>
int main() {
  void  *  Ptr;

  int a = 5;
  char isim[20] = "Mehmet Uslu";

  Ptr  = &a;
  printf("Ptr = %p Data = %d \n", Ptr, * (int*)(Ptr) );

  Ptr  = &isim;
  printf("Ptr = %p Data = %s \n", Ptr, (char*)(Ptr) );

} // end main
```

Program Output

```
Ptr = 0023FEA8  Data = 5
Ptr = 0023FE94  Data = Mehmet Uslu
```

7.8 The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - In C, an array name has a special status: Array name is considered 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:

```
int b[5],
int *bPtr;
bPtr = b;
```

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

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

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

7.8 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 )`

Example: Subscripting and pointer for array

Part 1 of 2

```
/* Fig. 7.20: fig07_20.cpp
   Using subscripting and pointer notations with arrays */

#include <stdio.h>
int main() {
   int b[] = { 10, 20, 30, 40 }; // initialize array b
   int *bPtr = b;                // set bPtr to point to array b
   int i;                        // counter
   int offset;                   // counter

   // output array b using array subscript notation
   printf( "Array b printed with:\nArray subscript notation\n" );

   // loop through array b
   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   }

   // output array b using array name and pointer/offset notation
   printf( "\nPointer/offset notation where\n"
           "the pointer is the array name\n" );

   // loop through array b
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
   }
```

Part 2 of 2

```
   // output array b using bPtr and array subscript notation
   printf( "\nPointer subscript notation\n" );

   // loop through array b
   for ( i = 0; i < 4; i++ ) {
      printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
   }

   // output array b using bPtr and pointer/offset notation
   printf( "\nPointer/offset notation\n" );

   // loop through array b
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
   }

} // end main
```

Program Output

```
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
```

7.9 Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

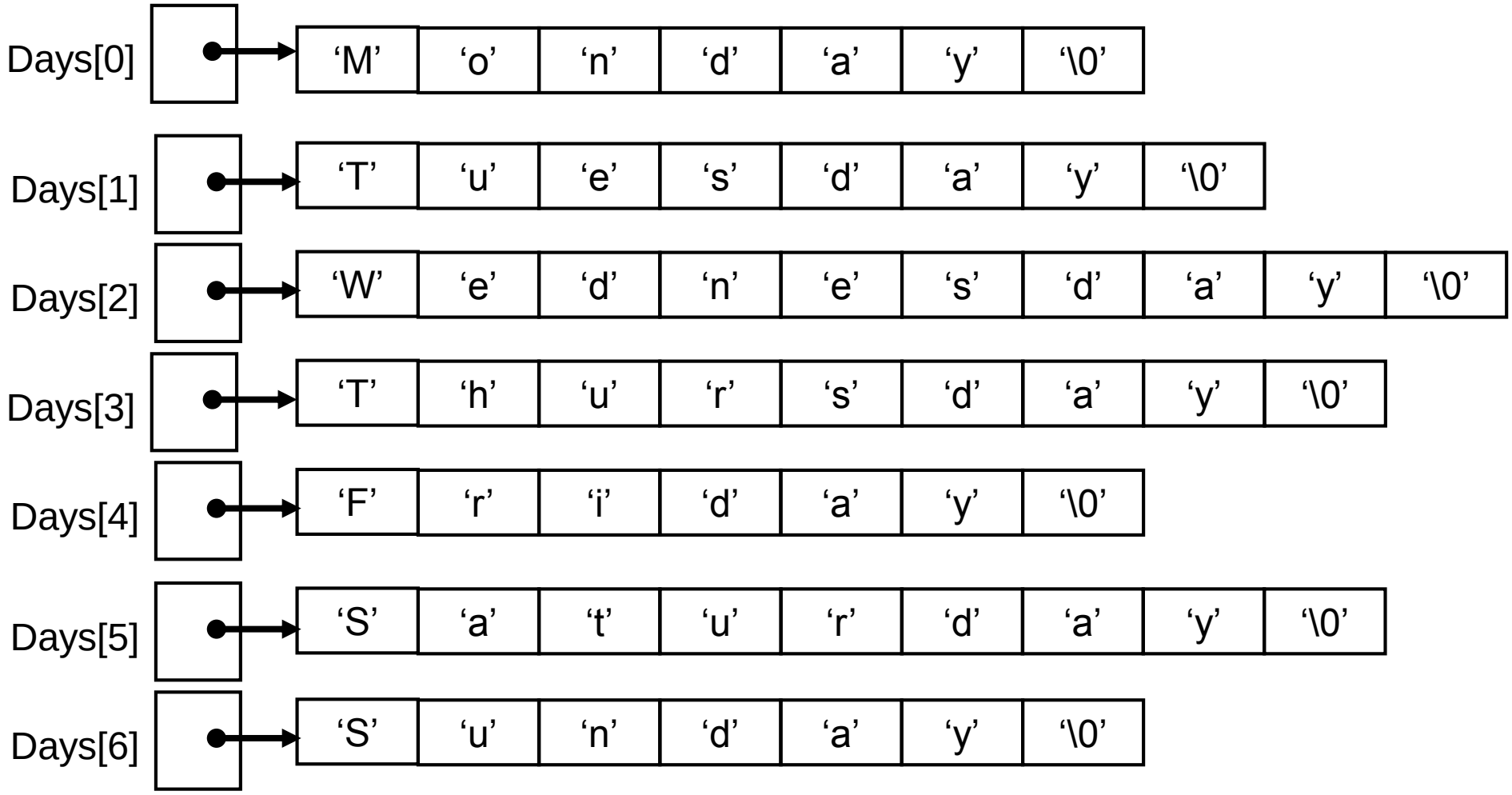
```
char * Days[7] = { "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday",
  "Saturday", "Sunday" };
```

Or

```
char Days[7][10] = { "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday",
  "Saturday", "Sunday" };
```

  - Strings are pointers to the first character
  - `char *` – each element of `Days` is a pointer to a `char`
  - The strings are not actually stored in the array `Days`, only pointers to the strings are stored
- In the first example, Days array has a fixed size, but strings can be of any size

7.9 Arrays of Pointers



Example: Array of String Pointers

```
#include <stdio.h>

int main()
{
     char * Days[7] = { "Monday", "Tuesday", "Wednesday",
                        "Thursday","Friday", "Saturday",
                        "Sunday"};
     int i;

     for (i=0; i < 7; i++)
          printf("%s \n", Days[i]);

} // end main
```

Example: Pointer to Pointer

```c
#include <stdio.h>

int main()
{
int a = 20;
int *x;    //Pointer
int **y;   //Pointer to pointer

x = &a; // Get address of a
y = &x; // Get address of x

printf(" a = %d    &a = %p \n", a,  &a);
printf("*x = %d   **y = %d \n", *x, **y);
printf(" x = %p    *y = %p \n", x,  *y);
printf("&x = %p     y = %p \n", &x, y);
printf("&y = %p \n", &y);

} // end main
```

Program Output

```
a    = 20
&a = 0022FF74


 *x = 20
**y = 20


 x = 0022FF74
*y = 0022FF74


&x = 0022FF70
  y = 0022FF70


&y = 0022FF6C
```

Pointers to Functions

7.10 Pointers to Functions

- Pointer to function
  - This is an advanced feature in C, but not frequently used
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function

- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

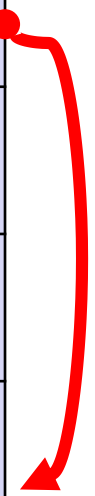
Example: Pointer to Function

```c
#include <stdio.h>

void kare(int x)
{
   printf("x= %d , Kare= %d \n", x, x*x);
}


int main()
{
  void (*ptr)(); // Define ptr as pointer to function

  kare(2);  // Normal function call

  ptr = &kare; // Get address of function kare
  (*ptr)(3);  // Call the pointed function

} // end main
```

Program Output

```
x = 2  Kare = 4

x = 3  Kare = 9
```

Variable Name	Memory Address	Memory Content
ptr	0022FF44	00401290
	………	………
	………	………
	00401290	
	00401294	
	00401298	
	………	………



7.10 Pointers to Functions

- Example: bubblesort
  - There are two helper functions: *ascending*, *and* *descending*
  - Function `bubble` takes a function pointer **compare*
    - `bubble` calls the helper function (without knowing which)
    - this determines ascending or descending sorting

  - The argument in `bubblesort` for the function pointer:

    ```
    int ( *compare )( int a, int b )
    ```

    tells `bubblesort` to expect a pointer to a function that takes two `ints` and returns an `int`

  - Actually there is no function named ***compare***.
  - *compare will point either to *ascending* function or to *descending* function

7.10 Pointers to Functions

– **CAUTION:** If the parentheses were left out:

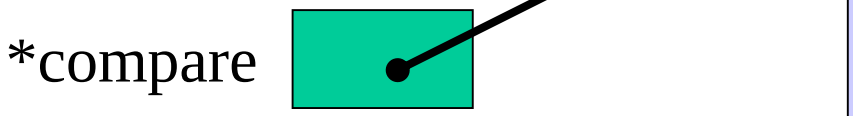
```
int *compare( int a, int b )
```

defines a function that receives two integers and returns a pointer to a `int`

Pointer to Functions

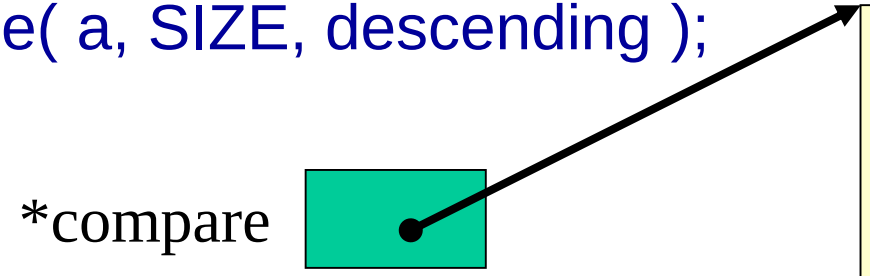
```
void bubble( int a[], const int size, int (*compare)( int a, int b ) );
```

```
bubble( a, SIZE, ascending );
```




```
int ascending( int a, int b )
{


}
```

```
bubble( a, SIZE, descending );
```




```
int descending( int a, int b )
{


}
```

Example: Pointer to functions

Part 1 of 4

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

// prototypes
void bubble( int work[], const int size,
             int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main()
{
   int order;   // 1 for ascending order or 2 for descending order
   int counter;

   // initialize array a
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );
```

Part 2 of 4

```
   printf( "\nData items in original order\n" );
   // output original array
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   }

   /* sort array in ascending order; pass function ascending as an
     argument to specify ascending sorting order */
   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nData items in ascending order\n" );
   } // end if
   else { /* pass function descending */
      bubble( a, SIZE, descending );
      printf( "\nData items in descending order\n" );
   } // end else

   // output sorted array
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   }

} // end main
```

Part 3 of 4

```
/* multipurpose bubble sort; parameter compare is a pointer to
   the comparison function that determines sorting order */
void bubble( int work[], const int size,
             int (*compare)( int a, int b ) )
{
   int pass;  // pass counter
   int count; // comparison counter

   void swap( int *element1Ptr, int *element2ptr ); // prototype

   // loop to control passes
   for ( pass = 1; pass < size; pass++ ) {

      // loop to control number of comparisons per pass
      for ( count = 0; count < size - 1; count++ ) {

         // if adjacent elements are out of order, swap them
         if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
            swap( &work[ count ], &work[ count + 1 ] );
         } // end if

      } // end inner for

   } // end outer for

} // end function bubble
```

Part 4 of 4

```
/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold; // temporary holding variable

   hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

```
/* determine whether elements are out of order for an ascending
   order sort */
int ascending( int a, int b )
{
   return b < a;   // swap if b is less than a

} // end function ascending
```

```
/* determine whether elements are out of order for a descending
   order sort */
int descending( int a, int b )
{
   return b > a;   // swap if b is greater than a

} // end function descending
```

Program Output 1

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1


Data items in original order
    2    6    4    8   10   12   89   68   45   37


Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

Program Output 2

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2


Data items in original order
    2    6    4    8   10   12   89   68   45   37


Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```