UEE1302
Introduction to Computers and Programming

C_Lecture 08:

Pointers

C: How to Program 7th ed.
Chapter 7 C Pointers

Agenda

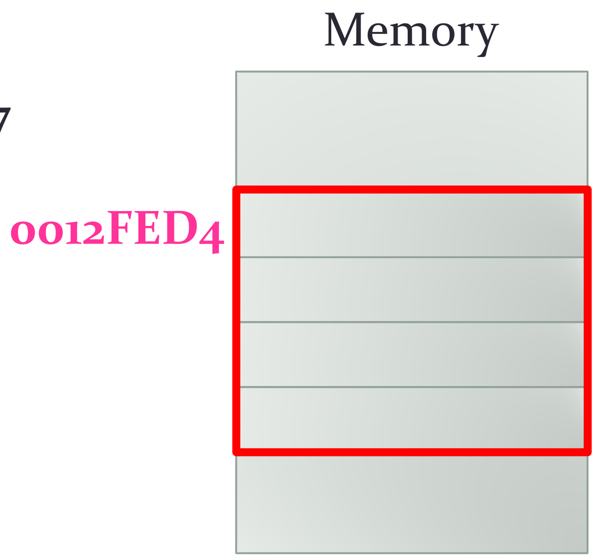
- Pointers
  - Data type and pointer variables
  - Declaration and manipulation
  - Passing arguments to functions by reference
- Using `const` with Pointer
- Selection Sort using Call-by-Reference
- Pointer Expressions and Pointer Arithmetic
- Array of Pointers
- Pointers to Functions

Variables

- Three major attributes of a variable:
  - Data type: declared in a declaration statement
  - Value: Stored in a variable by
    - Initialization when variable is declared
    - Assignment
    - User input
  - Address: For most applications, variable name is sufficient to locate the variable's contents
    - Translation of variable's name to a storage location is done by computer each time the variable is referenced

Memory Addresses

- Every byte in memory has an integer address
  - Ex: a computer with 256MB RAM
  - addresses start from 0 to 256×1024×1024
- An `int` variable uses 4 bytes
  - Ex: `int num = 10;`
  - each address stores one byte
  - store in address 0012FED4 to 0012FED7



Memory Addresses (cont.)

- Programmers are usually only concerned with a variable's value, not its address
- Address operator `&`: determines the address of a variable
  - `&` means "the address of"
  - When placed in front of variable `num`, it is translated as "the address of `num`"

Example of Address Operator

```c
#include <stdio.h>

int main ()
{
    int num = 10;

    printf("num = %d \n", num);
    printf("The address of num = %p\n", &num);

    return 0;
}
```

- screen output

```
num = 10
The address of num = 0012FED4
```

Storing Addresses

- Address can be stored in suitably declared variables
- Example: `numAddr = #`
  - Statement stores the address of `num` in the variable `numAddr`
  - `numAddr` is called a **pointer**

Introduction to Pointers

- [Definition] pointer:
  - is a memory address of a variable
- Syntax:

```
datatype *identifier;
```

  - Recall: memory divided
  - Numbered memory locations (index)
  - Addresses used as content of a pointer variable

Pointer Variables

- Pointers can be viewed as one datatype
  - can store pointers in variables
  - not **`int`**, **`double`**, **`char`** and etc.
  - instead, a pointer (or an arrow) to **`int`**, **`double`**, **`char`** and etc.
- Example: **`double *p;`**
  - `p` is declared a pointer-to-double variable
  - can hold pointers to variables of type `double`, but not other types like `char`

Declaring Pointer Variables

- Pointers declared like other types
  - add `*` before variable names
  - produce pointer to that type
- 3 forms are equivalent

```
int *ptr; //most suggested by textbook
int* ptr; //most convenient practically
int * ptr;
```

  - `'*'` must be located before each variable
- Example: `int *ptr1, var1, *ptr2, var2;`
  - `ptr1`, `ptr2` hold pointers to `int` variables
  - `var1`, `var2` are ordinary `int` variables

Initialize Pointer Variables

- C does not automatically initialize variables
- Initialize a pointer constant value 0 (a.k.a. null pointer

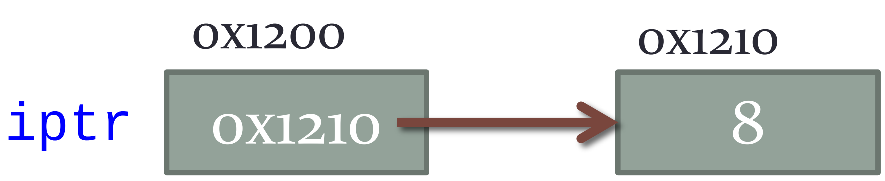
```
int *ptr = 0;
```

  - store the null pointer in `ptr`
  - `ptr` points to nothing
  - constant `NULL` is also equivalent

    ex: `int *prt = NULL;`
- Constant 0 is the only number that can be directly assigned to pointer variables.

Dereferencing Operator *

- Two roles of `'*'` in C:
  - binary multiplication operator, ex: `8*5`
  - a unary operator , ex: `int *iptr;`
- As `'*'` is used as a unary operator,
  - called dereferencing operator (or indirection operator)
  - refer to object to which its operand (that is, a pointer) points



Pointing To ...

- Terminology, view
  - talk of pointing, not addresses
  - pointer variable points to ordinary variable
  - leave address talk out
- Example:

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

  - set pointer variable `p1` to point to **`int`** variable `v1`
  - or `p1` is assigned the address of `v1`

Pointing To … (cont.)

- Example:

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- Two ways to refer to `v1` now:
  - variable `v1` itself:

    ```
    printf("%d", v1);
    ```

  - via pointer `p1`:

    ```
    printf("%d", *p1);
    ```

- Dereference operator, `*`
  - pointer variable *dereferenced*
  - mean: retrieve data that `p1` points to

Pointer Operators & and *

```
// 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);
```

Pointer Operators & and * (cont.)

```
    printf("\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr);
    printf("\nShowing that * and & are inverses of "
           "each other.\n&*aPtr = %p“
           “\n*&aPtr = %p\n", &*aPtr, *&aPtr);

    return 0;
}
```

- screen output

```
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 inverses of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Assignment of Pointers

- Pointer variables can be assigned:

```
int *p1, *p2;
p1 = p2; //ex: address of p2 is 5678
```

  - assign one pointer to another
  - make `p1` point to where `p2` points

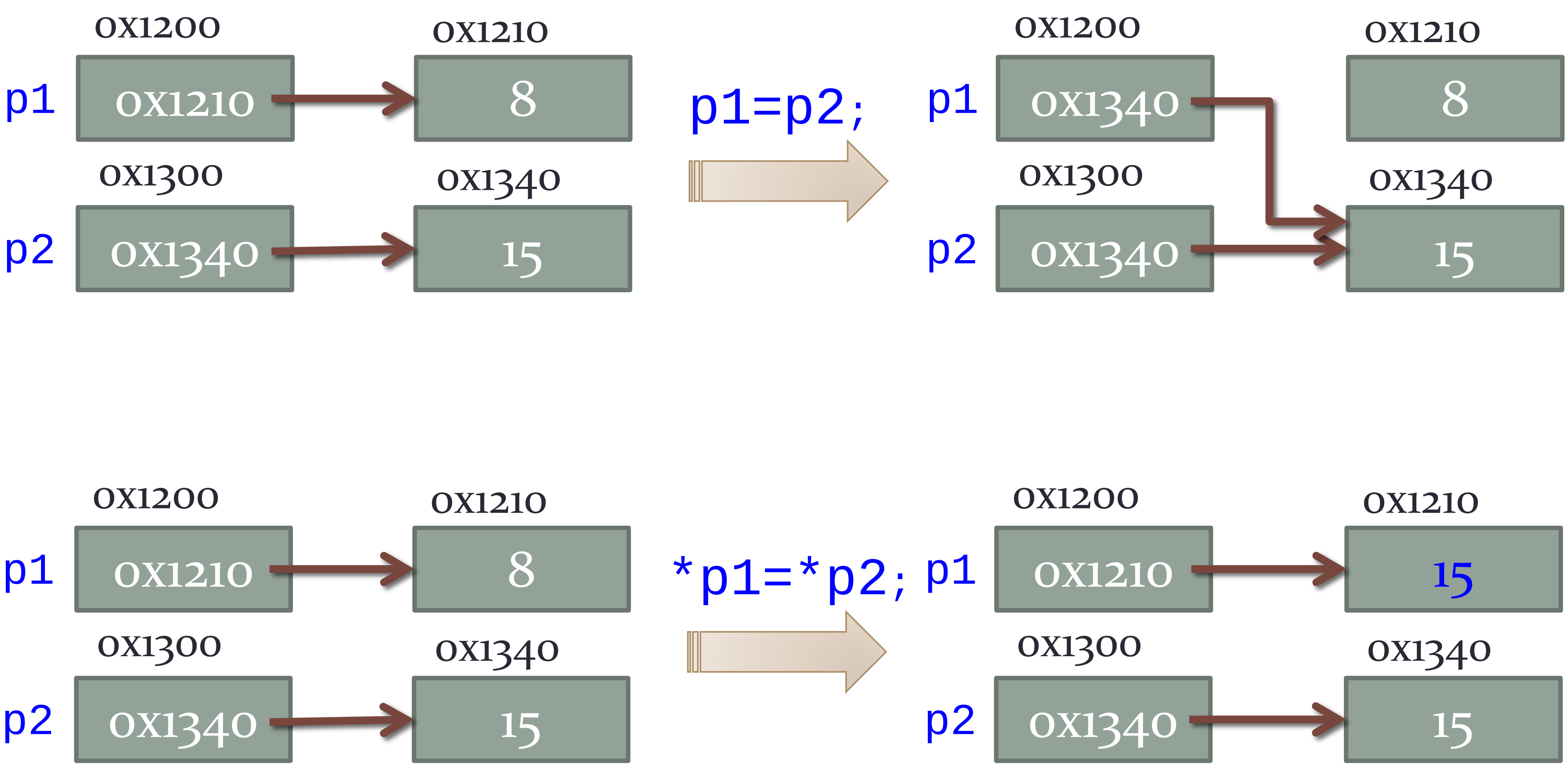
  => p1 is assigned the same address as p2

- How about this one?

```
*p1 = *p2;
```

  - assign the value pointed to by `p1` to the value pointed to by `p2`
  - copy the content that `p2` points to the content that `p1` points

Assignment of Pointers (cont.)



Passing Arguments to Functions

```
// 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);

    return 0;
}
```

Passing Arguments to Functions (cont.)

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

- screen output

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

Analysis of a Typical Call-by-Value

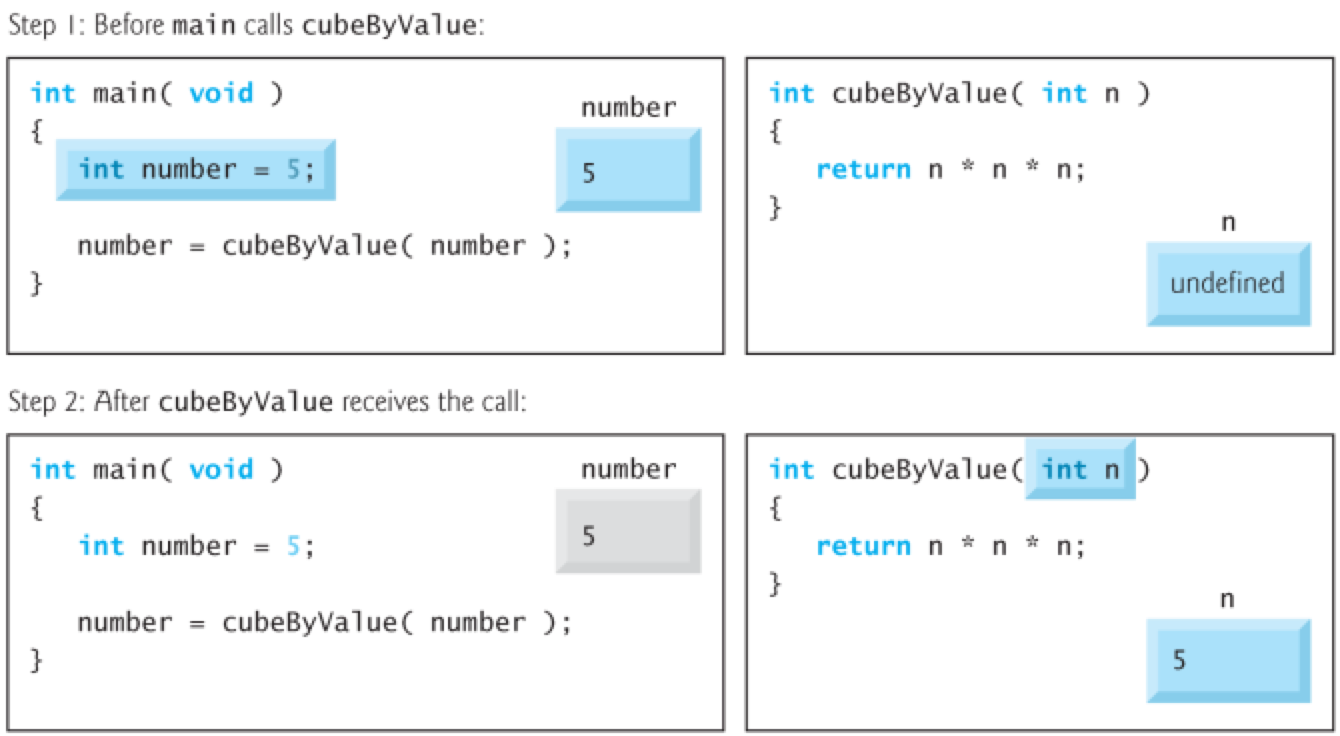


Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

Analysis of a Typical Call-by-Value (cont.)

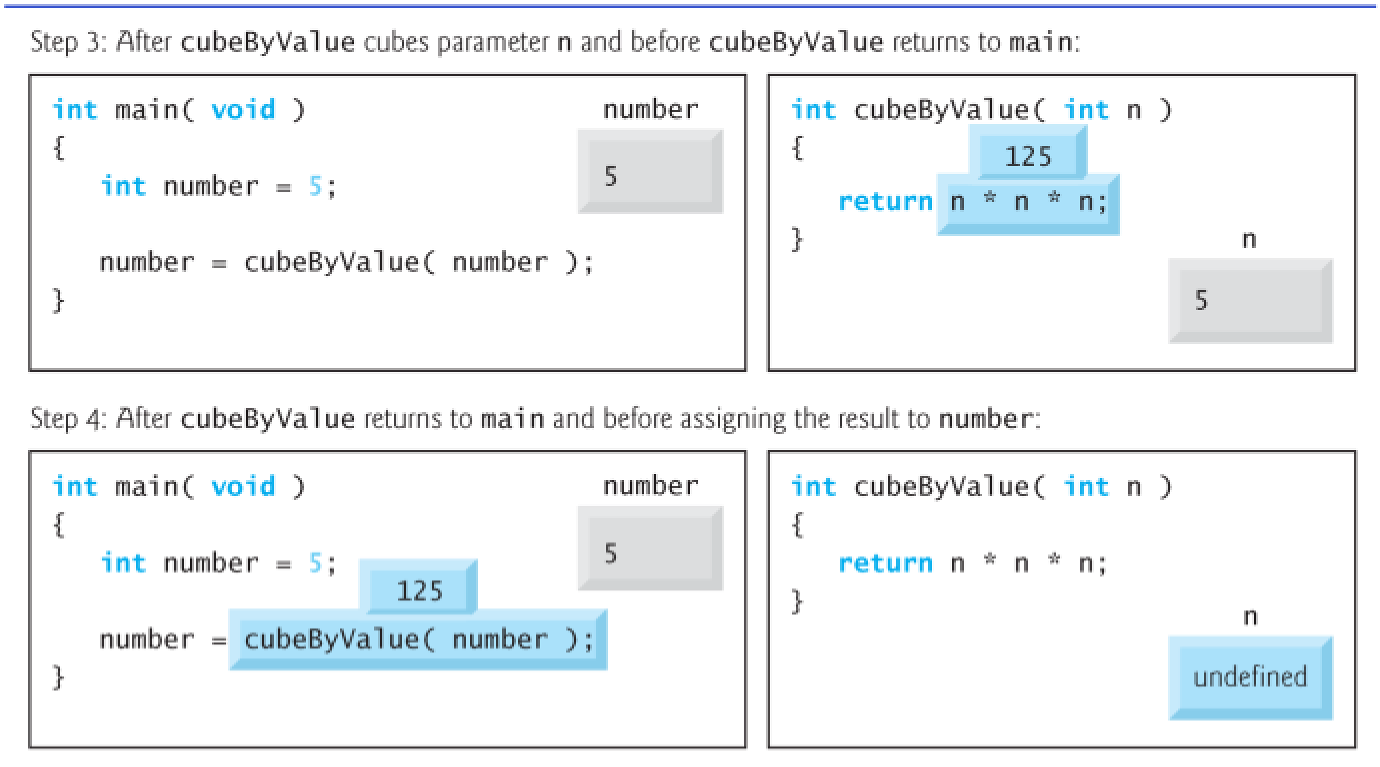


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Analysis of a Typical Call-by-Value (cont.)

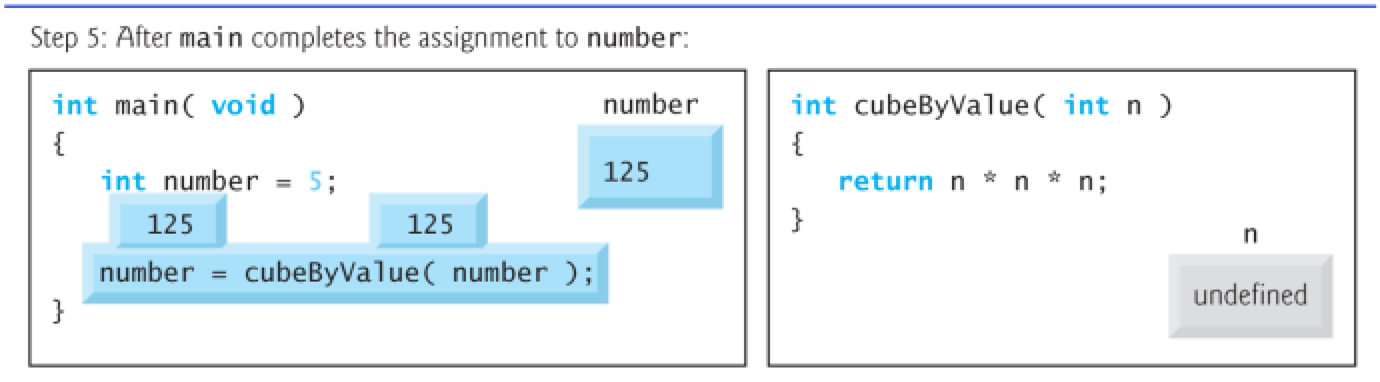


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Passing Arguments to Functions

```
// Fig 7.7: fig07_07.c
// Cube a variable using call-by-reference.
#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 cubeByRerence
    cubeByReference(&number);
    printf("\nThe new value of number is %d\n", number);

    return 0;
}
```

Passing Arguments to Functions (cont.)

```
void cubeByRference( int *nPtr )
{
    *nPtr * *nPtr * *nPtr;
}
```

- screen output

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

Analysis of a Typical Call-by-Reference

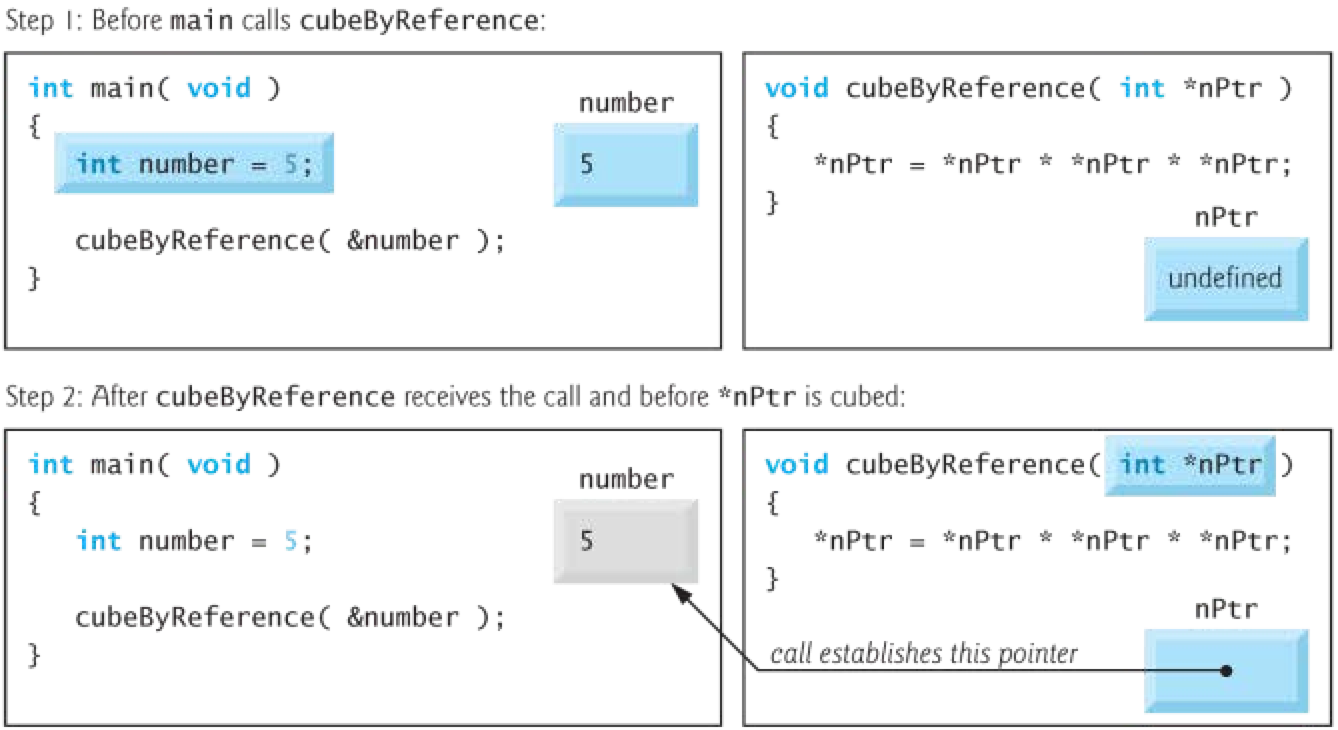


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Analysis of a Typical Call-by-Reference (cont.)

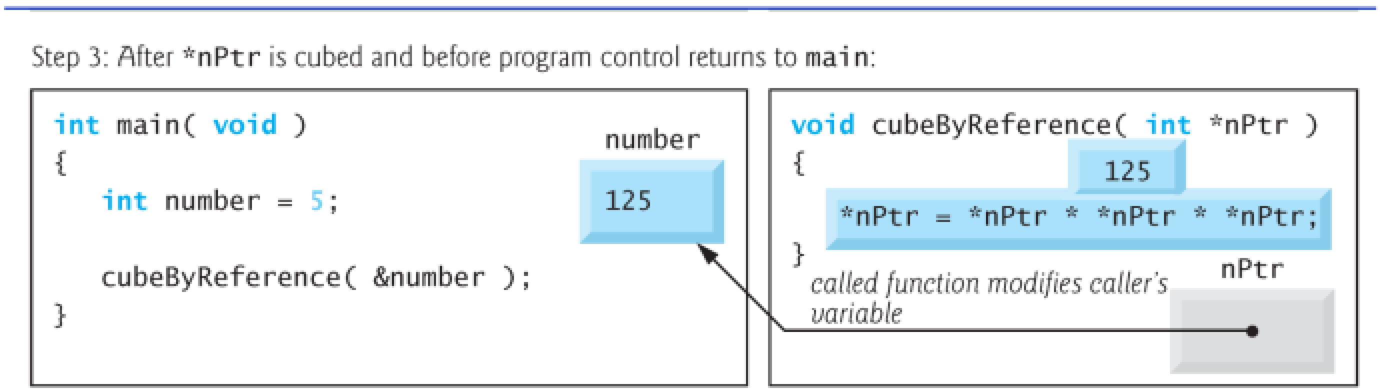


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Using `const` with Pointers

- Many possibilities exist for using (or not using) `const` with function parameters.
  - If a value does not (should not) change in the body of a function to which it's passed, the parameter should be declared `const`.

- Principle of least privilege
  - Always give a function enough access to the data in its parameters to accomplish its specified task, but no more.

Using `const` with Pointers (cont.)

- There are four ways to pass a pointer to a function
  - a non-constant pointer to non-constant data (Fig. 7.10)
  - a non-constant pointer to constant data (Fig. 7.11, Fig. 7.12)
  - a constant pointer to non-constant data (Fig. 7.13)
  - a constant pointer to constant data (Fig. 7.14)
- Each combination provides a different level of access privilege.

1. Non-constant Pointer to Non-constant Data

- The highest access is granted by a non-constant pointer to non-constant data

- The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data.
  - Such a pointer's declaration (e.g., `int *countPtr`) does not include `const`.

1. Non-constant Pointer to Non-constant Data (cont.)

```c
// Fig 7.10: fig07_10.c
// Converting a string to uppercase 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";
    printf("The string before conversion is: %s", string);
    convertToUppercase( string );
    printf("\nThe string after conversion is: %s", string);

    return 0;
}
```

1. Non-constant Pointer to Non-constant Data (cont.)

```
void convertToUppercase( char *sPtr )
{
    while ( *sPtr != '\0')
    {
        if ( islower(*sPtr) )
            *sPtr = toupper(*sPtr);
        ++sPtr;
    }
}
```

- screen output

```
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS and $32.98
```

2. Non-constant Pointer to Constant Data

- A non-constant pointer to constant data
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer.
- Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data.
- Any attempt to modify the data in the function results in a compilation error.
- Sample declaration:

```
const int *countPtr;
```

  - Read from right to left as “countPtr is a pointer to an integer constant.”

2. Non-constant Pointer to Constant Data (cont.)

```
// 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 ()
{
    char string[]= "print characters of a string";
    printf("The string is:\n");
    printCharacters( string );
    printf("\n");

    return 0;
}
```

2. Non-constant Pointer to Constant Data (cont.)

```
void printCharacters ( const char *sPtr )
{
    for ( ; *sPtr != '\0'; sPtr++)
        printf("%c", *sPtr);
}
```

- screen output

```
The string is:
print characters of a string
```

2. Non-constant Pointer to Constant Data (cont.)

```
// Fig 7.12: fig07_12.c
// Attempt to modify data through a
// non-constant pointer to constant data
#include <stdio.h>

void f( const int *xPtr); // prototype

int main ()
{
    int y;  // define y
    f(&y);   // f attempts illegal modification

    return 0;
}
```

2. Non-constant Pointer to Constant Data (cont.)

```
void f( const int *xPtr)
{
    *xPtr = 100; // error: cannot modify a const object
}
```

- screen output

```
fig07_12.c: In function ‘f’:
fig07_12.c:18: error: assignment of read-only location
```

2. Non-constant Pointer to Constant Data

(cont.)

```
int x = 4, y = 7;
const int *pt = &x; //a pointer to a const.
printf("%d", *pt);  //print 4 on screen
pt = &y;
printf("%d", *pt);  //print 7 on screen
*pt = 11;           //ERROR! cannot modify
```

- The pointer `pt` to a constant used in this example can store different addresses
  - can pointer to different variable, `x` or `y`
- However, cannot change the dereferenced value that `pt` points to

3. Constant Pointer to Non-constant Data

- A constant pointer to non-constant data is a pointer that always points to the same memory location; the data at that location can be modified through the pointer.
- An example of such a pointer is an array name, which is a constant pointer to the beginning of the array.
- All data in the array can be accessed and changed by using the array name and array subscripting.

3. Constant Pointer to Non-constant Data (cont.)

- A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation.
- Pointers that are declared `const` must be initialized when they're declared.

```
int var1 = 15, var2 = 8;
//a constant pointer to a declared variable
int * const cpt = &varl;
*cpt = 34;//change the value cpt points to
printf("%d", var1); //print 34 on screen
//ERROR! a const. pointer cannot be changed
cpt = &var2;
```

4. Constant Pointer to Constant Data

- The minimum access privilege is granted by a constant pointer to constant data.
  - Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer.
- This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array.
- Ex: `const int * const ptr = &x.`
  - This declaration is read from right to left as "`ptr` is a constant pointer to an integer constant."

4. Constant Pointer to Constant Data (cont.)

```c
// Fig 7.14: fig07_14.c
// Attempt to modify a constant pointer to constant data
#include <stdio.h>

int main ()
{
    int x = 5, y;
    const int *const ptr = &x;

    printf("%d\n", *ptr);

    *ptr = 7;  // error: *ptr is const
    ptr = &y;  // error: ptr is const

    return 0;
}
```

Selection Sort Using Pass-by-Reference

- Selection sort
  - Easy-to-program, but inefficient, sorting algorithm.
  - The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element.
  - The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element.
  - The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.
  - After the *i*th iteration, the smallest *i* items of the array will be sorted into increasing order in the first *i* elements of the array.

Selection Sort

```
// Selection sort with pass-by-reference.
#include <stdio.h>
#define SIZE 10
void selectionSort(int * const array, const int size);
void swap(int *, int *);

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

    printf("Data items in original order\n");
    int i;
    for ( i = 0; i < SIZE; i++)
        printf("%4d", a[i]);
```

Selection Sort (cont.)

```
    selectionSort(a, SIZE);
    printf("\nData items in ascending order\n");
    for ( i = 0; i < SIZR; i++)
        printf("%4d", a[i]);
    printf("\n");

    return 0;
}

void swap(int *element1Ptr, int *elemen2Ptr)
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}
```

Selection Sort (cont.)

```
void selectionSort(int * const array, const int SIZE)
{
    int smallest; // index of smallest element;
    int i, j;
    for ( i = 0; i < SIZE - 1; i++)
    {
        smallest = i;
        for ( j = i + 1; j < SIZE; j++)
        {
            if (array[j] < array[smallest])
                smallest = j;
        }
        swap(&array[i], &array[smallest]);
    }
}
```

Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- pointer arithmetic—certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type

Array Names as Pointers

- If grade is a single-dimension array containing five integers, the fourth element is `grade[3]`
- In C compiler, the computation for the address of `grade[3]`: (assuming 4 bytes per integer)

  &grade[3] = &grade[0] + (3 * 4)

  - This statement reads as "the address of grade[3] equals the address of grade[0] plus 12"

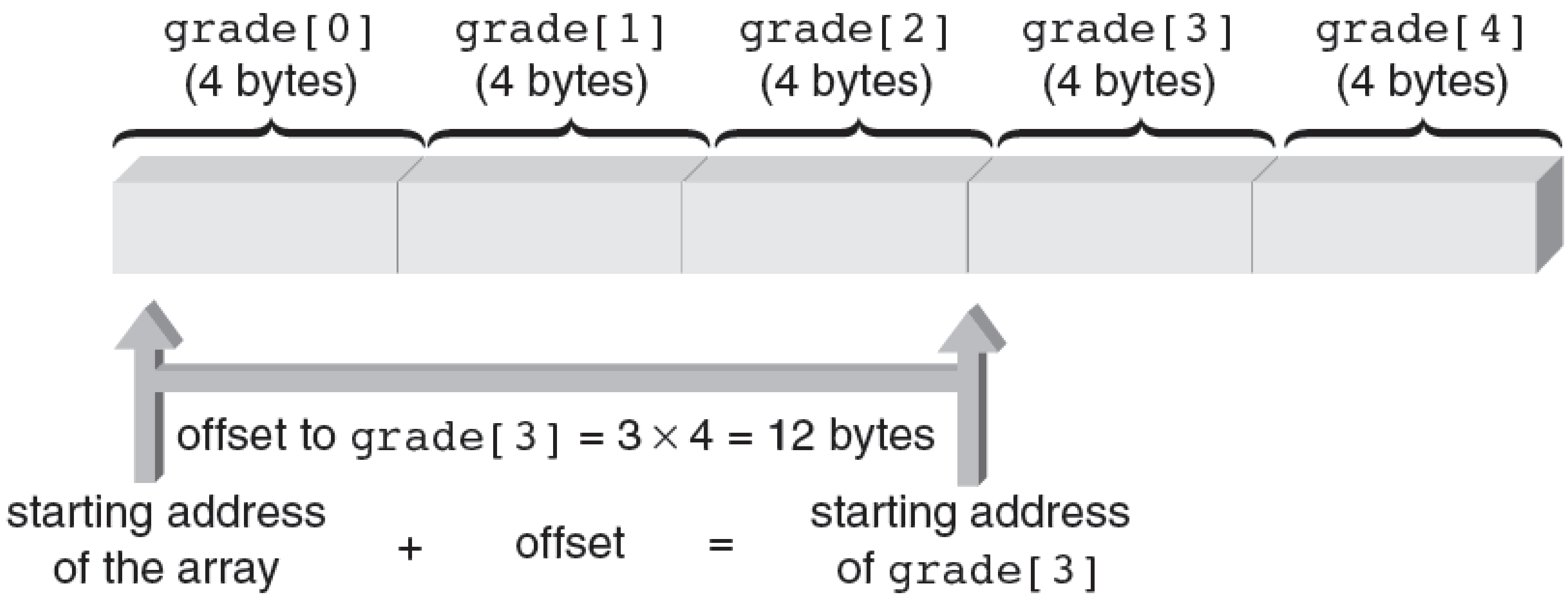
Access Array by Pointer

- An array name
  - returns the *starting address* of the array
  - the address of the *first element* of the array
  - can also be used as a *pointer to the array* ⇨ faster than indexing

```
int grade[5] = {90, 80, 70, 60, 50};
```

array indexing	pointer notation
`grade[0]`	`*grade`
`grade[1]`	`*(grade + 1)`
`grade[2]`	`*(grade + 2)`
`grade[3]`	`*(grade + 3)`
`grade[4]`	`*(grade + 4)`

Access Array by Pointer (cont.)



Example 1

```
//pointers & Arrays
#include <stdio.h>

int main ()
{
    int numbers[5];
    int *p;
    int n;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (n = 0; n < 5; n++)
        printf("%d ", numbers[n]);
    return 0;
}
```

Example 2

```
// Fig 7.21: fig07_21.c
// Copying a string using array and pointer notation.
#include <stdio.h>

void copy1 (char * const s1, const char * const s2);
void copy2 (char * s1, const char * s2);

int main ()
{
    char string1[10];
    char *string2 = "Hello";
    char string3[10];
    char string4[] = "Good Bye";

    copy1( string1, string2 );
    printf("string1 = %s\n", string1);
```

Example 2 (cont.)

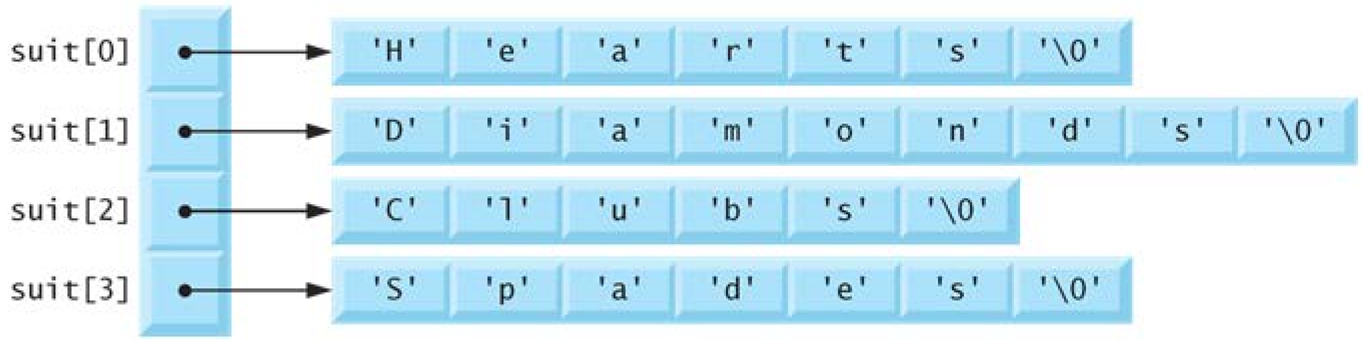
```
    copy1( string3, string4 );
    printf("string3 = %s\n", string3);
    return 0;
}

void copy1 (char * const s1, const char * const s2)
{
    int i;
    for ( i = 0; (s1[i] = s2[i]) != '\0'; i++ )
        ; // do nothing in body
}
void copy2 (char * s1, const char * s2)
{
    for ( ; (*s1 = *s2) != '\0'; s1++,s2++ )
        ; // do nothing in body
}
```

Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
  - In C, a string is essentially a pointer to its first character.
  - So each entry in an array of strings is actually a pointer to the first character of a string.

```
const char *suit[ 4 ] =
         { "Hearts", "Diamonds", "Clubs", "Spades" };
```



Pointers to Functions

- A pointer to a function contains the address of the function in memory.
  - We saw that an array name is really the address in memory of the first element of the array.
- Similarly, a function name is really the starting address in memory of the code that performs the function's task.
- Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

Bubble Sort

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

void bubble(int work[], const int size, int(*compare)(int
a, int b));
void swap(int *element1Ptr, int *element2Ptr);
int ascending(int a, int b);
int descending(int a, int b);

int main ()
{
    int order; // 1 for ascending, 2 for descending
    int i;
    int a[SIZE]={2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
```

Bubble Sort (cont.)

```
    prinft("Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: ");
    scanf("%d", &order);
    printf("Data items in original order\n");
    for ( i = 0; i < SIZE; i++)
        printf("%5d", a[i]);

    if ( order == 1)
    {
        bubble(a, SIZE, ascending);
        printf("\nData items in ascending order\n");
    }
    else {
        bubble(a, SIZE, descending);
        printf("\nData items in descending order\n");

    }
```

Bubble Sort (cont.)

```
    for ( i = 0; i < SIZR; i++)
        printf("%5d", a[i]);
    printf("\n");

    return 0;
}

void swap(int *element1Ptr, int *elemen2Ptr)
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}
```

Selection Sort (cont.)

```
void bubble(int work[], const int size, int(*compare)(int
a, int b))
{
    int pass, count;
    for ( pass = 1; pass < size; pass++)
        for ( count = 0; count < size - 1; count++)
            if ((*compare)(work[count], work[count+1])
                swap(&work[count],&work[count+1]);
}
int ascending(int a, int b)
{
    return b < a; // swap if b is less than a
}
int descending(int a, int b)
{
    return b > a; // swap if b is greater than a
}
```

Menu-Driven System

```c
// Fig 7.28: fig07_28.c
// Demonstrating an array of pointers to functions
#include <stdio.h>

void function1(int a);
void function2(int b);
void function3(int c);

int main ()
{
    // initialize array of 3 pointers to functions
    void (*f[3])(int)={function1, function2, function3};
    int choice;
    printf("Enter a number between 0 and 2, 3 to end: ");
    scanf("%d", &choice);
```

Menu-Driven System (cont.)

```
    while ( choice >= 0 && choice < 3)
    {
        (*f[choice])(choice);
         printf("Enter a number between 0 and 2, 3 to end:
");
         scanf("%d", &choice);
    }
    printf("\nProgram execution completed.\n");

    return 0;
}

void function1(int a)
{
    printf("You entered %d so function1 was called\n\n",a);
}
```

Menu-Driven System (cont.)

```
void function2(int b)
{
    printf("You entered %d so function2 was called\n\n",b);
}
void function3(int c)
{
    printf("You entered %d so function3 was called\n\n",c);
}
```

Common Programming Errors

- Using a pointer to access nonexistent array elements
- Incorrectly apply address and indirect operators

```
int *ptr1 = &45;
int *prt2 = &(miles+10);
```

  - Illegal to take the address of a value
- Taking addresses of pointer constants

```
int nums[25];
int * pt;
pt = &nums;
```

  - Correct form: `pt = nums;`

Common Programming Errors (cont.)

- Initializing pointer variables incorrectly

```
int *pt = 5;
```

  - `pt` is a pointer to an integer
  - must be a valid address of another integer variable or NULL

Summary

- Pointer is memory address
  - Provides indirect reference to variable
- C provides the capabilities for simulating call-by-reference using pointers and the indirection operator.
- An array name can be through of as a constant pointer.
- A pointer to a function contains the address of the function in memory.
  - A function name is really the starting address in memory of the code that performs the function's task.