ENG1008 Programming

- Pointers

Dr Kwee Hiong Lee

kweehiong.lee@singaporetech.edu.sg

Objectives

- Introduction to pointers and pointers operations
- To use pointers to pass arguments to functions by reference
- To use pointers to access arrays

Introduction

- In this chapter, we discuss one of the most powerful features of the C programming language, the pointer

- Pointers enable programs to simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees

- 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* (Fig. 7.2)

- *Referencing* a *value through* a *pointer* is called indirection

Pointer Variables

Declaring Pointers

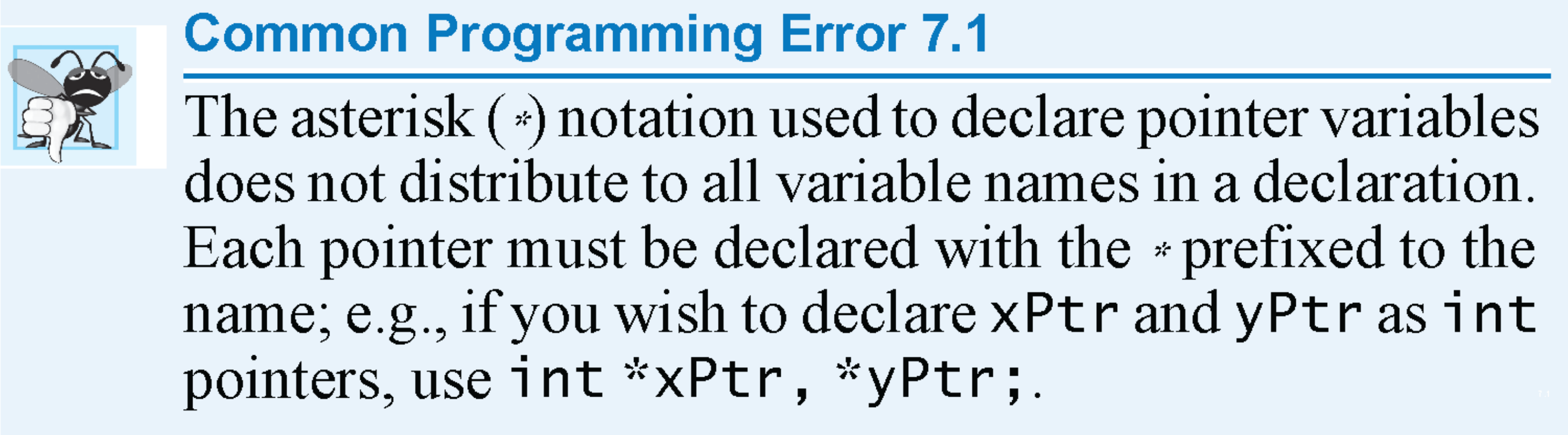
- Pointers, like all variables, must be defined before they can be used
- The definition

```
     int *countPtr, count;
```

specifies that variable countPtr is of type int * (i.e. a pointer to an integer) and is read (right to left), "*countPtr is a pointer to int*" or "countPtr points to an object of type int."

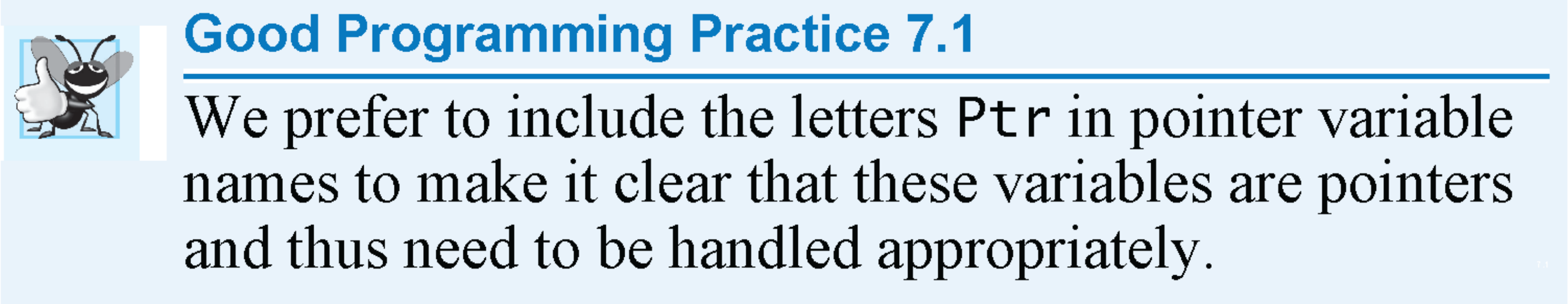
- Also, the variable *count* is defined to be an int, *not* a *pointer to an int*
- The * applies *only* to *countPtr* in the definition
- When * is used in this manner in a definition, it indicates that the *variable* being defined is a *pointer*
- *Pointers* can be *defined* to *point* to *objects* of *any type*

Pointer Variables



Common Programming Error 7.1

The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare `xPtr` and `yPtr` as `int` pointers, use `int *xPtr, *yPtr;`.



Good Programming Practice 7.1

We prefer to include the letters `Ptr` in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.

- To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown earlier, you should always declare only one variable per declaration

Pointer Variables

Initializing and Assigning Values to Pointers

- 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**
- A pointer with the value NULL points to *nothing*
- NULL is a *symbolic constant* defined in the <stddef.h> header (and several other headers, such as <stdio.h>)
- Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred

Pointer Operators

- The &, or address operator, is a unary operator that **returns** the address of its operand

- For example, assuming the definitions

```
int y = 5;
int *yPtr;
```

declare pointer

the statement

```
yPtr = &y;
```

assign pointer to the address

assigns the *address* of the *variable y* to *pointer variable* *`yPtr`*

- Variable `yPtr` is then said to "**point to**" `y`

- Pointers, like all variables, occupy space in *memory*

Pointer Operators

- **Figure 7.2** shows a schematic representation of memory after the preceding assignment is executed

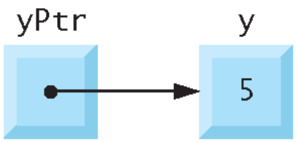


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

- **Figure 7.3** shows the representation of the pointer in memory, assuming that integer variable y is stored at location 600000, and pointer variable yPtr is stored at location 500000

- The *operand* of the address operator must be a variable (i.e. &y where the operand y is a variable); the address operator & *cannot* be applied to constants or expressions

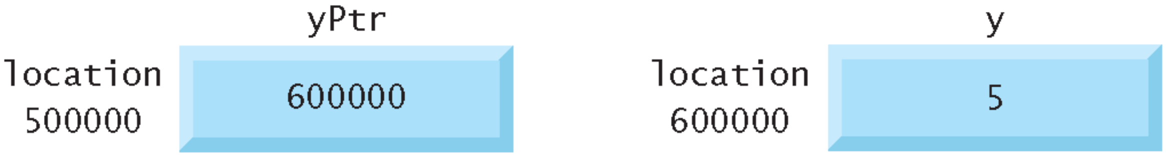


Fig. 7.3 | Representation of y and yPtr in memory.

Indirection (*) Operator

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the *value* of the object to which its operand (i.e., a pointer) *points* to

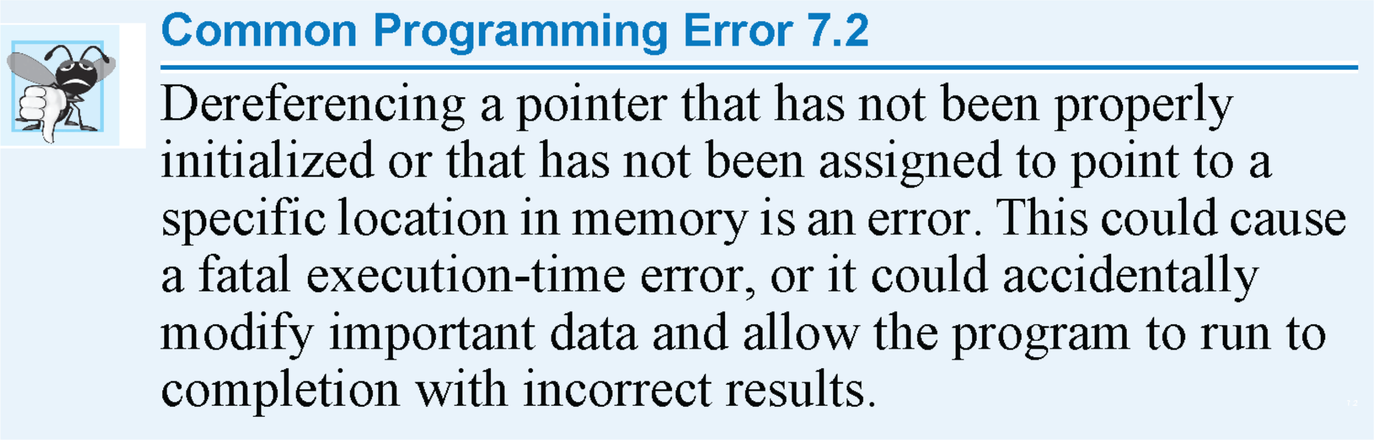
- For example, the statement

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

  dereferencing

  prints the value of variable y, namely 5

- Using * in this manner is called dereferencing a pointer



Common Programming Error 7.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

& and * operators

- **Figure 7.4** demonstrates the pointer operators & and *
- The *printf* conversion specifier **%p** outputs the *memory location* as a *hexadecimal* integer on most platforms
- Notice that the *address* of **a** and the *value* of **aPtr** are *identical* in the output, thus confirming that the *address of a is indeed assigned to the pointer variable aPtr* (line 11)
- The **&** and * operators are *complements of one another* - when they're both *applied consecutively* to aPtr in either order (line 21), the same result is printed

& and * operators

```c
// 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
```

Fig. 7.4 | Using the & and * pointer operators. (Part 2 of 2.)

- Passing arguments to functions

Passing Arguments to Functions

Pass-by-reference

- There are two ways to *pass arguments* to a *function* : pass-by-value and pass-by-reference
- All *arguments* in *C* are *passed by value*
- Many functions require the capability to **modify variables** in the *caller* or to pass a *pointer* to a *large data object* to avoid the *overhead* of *passing* the object by *value* (which incurs the time and memory overheads of making a copy of the object)
- In C, you use pointers and the indirection operator to **simulate** pass-by-reference
- When calling a function with arguments that should be modified, the addresses of the *arguments* are *passed*

Passing Arguments to Functions

Pass-by-reference

- This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified
- As we saw in Chapter 6, *arrays are not passed using operator &* because C automatically passes the *starting location* in memory of the array (the *name* of an array is equivalent to *&arrayName[0]*)
- When the *address* of a variable is passed to a function, the indirection operator (*) may be used in the function to *modify* the value at that location in the *caller's memory*

Passing Arguments to Functions

Pass-by-value and Pass-by-reference

- The programs in **Fig. 7.6** and **Fig. 7.7** present two versions of a function that cubes an integer—*cubeByValue* and *cubeByReference*

Pass-by-value

- **Figure 7.6** *passes* the variable number by **value** to function cubeByValue (line 14)
- The cubeByValue function cubes its argument and passes the new value back to main using a **return** statement
- The new value is assigned to number in main (line 14)

Passing Arguments to Functions

```c
// 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
```

Fig. 7.6 | Cube a variable using pass-by-value. (Part 2 of 2.)

Passing Arguments to Functions

Pass-by-reference

- **Figure 7.7** *passes* the variable number by **reference** (line 15) - the *address* of number is passed to function cubeByReference
- Function cubeByReference takes as a *parameter* a ***pointer to an int*** called nPtr (line 21)
- The function ***dereferences*** the pointer and cubes the value to which nPtr points (line 23), then assigns the result to ***nPtr** (which is *really number* in main), thus *changing* the *value* of *number* in main

Passing Arguments to Functions

```c
// Fig. 7.7: fig07_07.c
// 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
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 2 of 2.)

Passing Arguments to Functions

Pass-by-reference

- A *function receiving* an **address** as an *argument* must define a **pointer parameter** to *receive* the *address*
- For example, in **Fig. 7.7** the header for function cubeByReference (line 21) is:

```
void cubeByReference( int *nPtr )
```

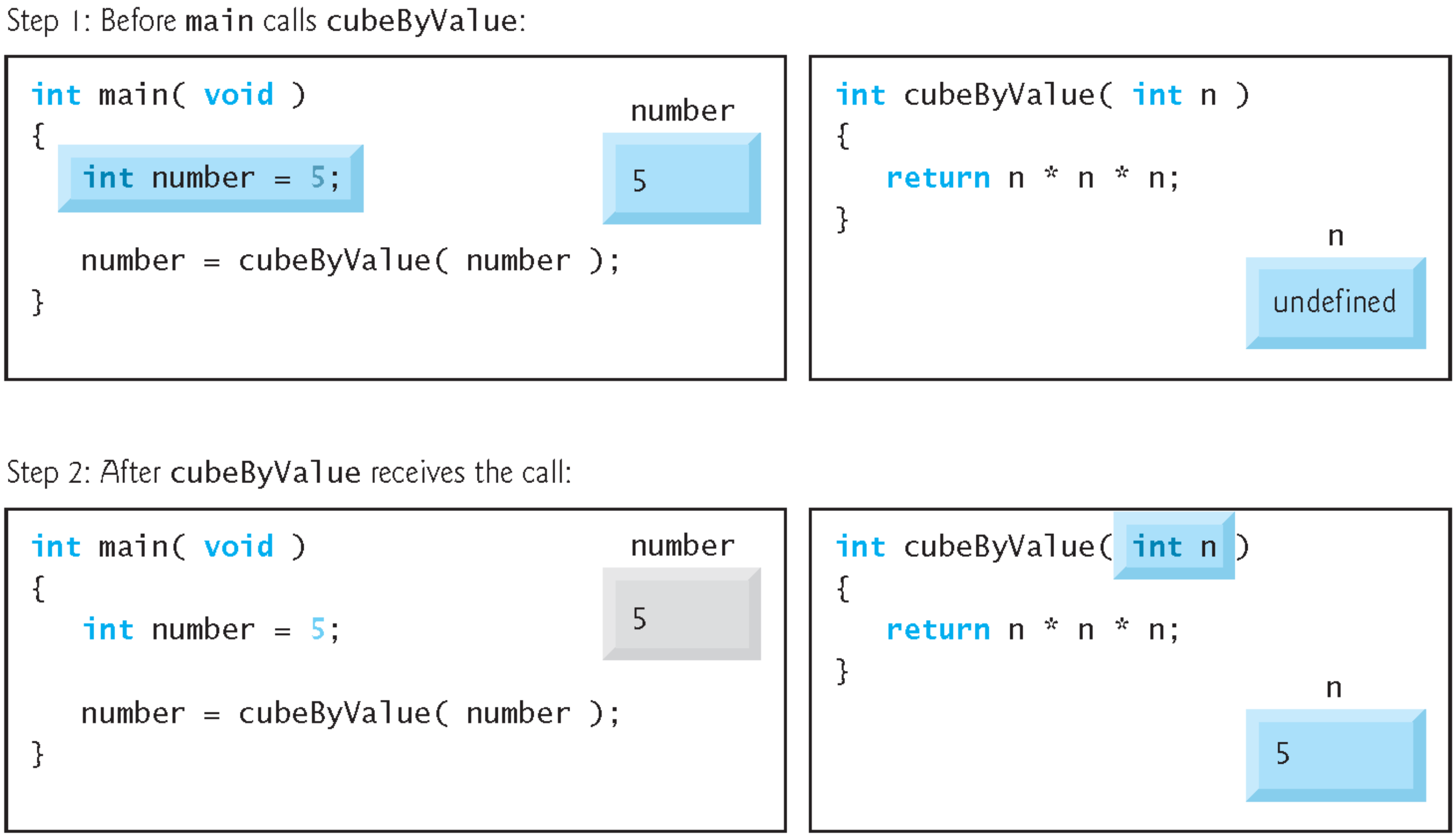
- The header specifies that cubeByReference *receives* the *address* of an integer variable as an *argument*, *stores the address locally* in *nPtr* and *does not return a value*
- The function prototype for cubeByReference (line 6) contains int * in parentheses
- *Names* included for documentation purposes are *ignored* by the *C compiler*

Passing Arguments to Functions

Pass-by-reference

- For a function that expects a **single-subscripted array** as an **argument**, the *function's prototype and header* can use the **pointer** *notation* shown in the *parameter list* of function cubeByReference (line 21)
- 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
- When the *compiler encounters a function parameter* for a **single-subscripted array** of the form int b[]**,** the *compiler* **converts** the **parameter** to the **pointer** *notation* int *b
- The two forms are **interchangeable**

Passing Arguments to Functions



Step 1: Before main calls cubeByValue:

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

   number = cubeByValue( number );
}
```

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

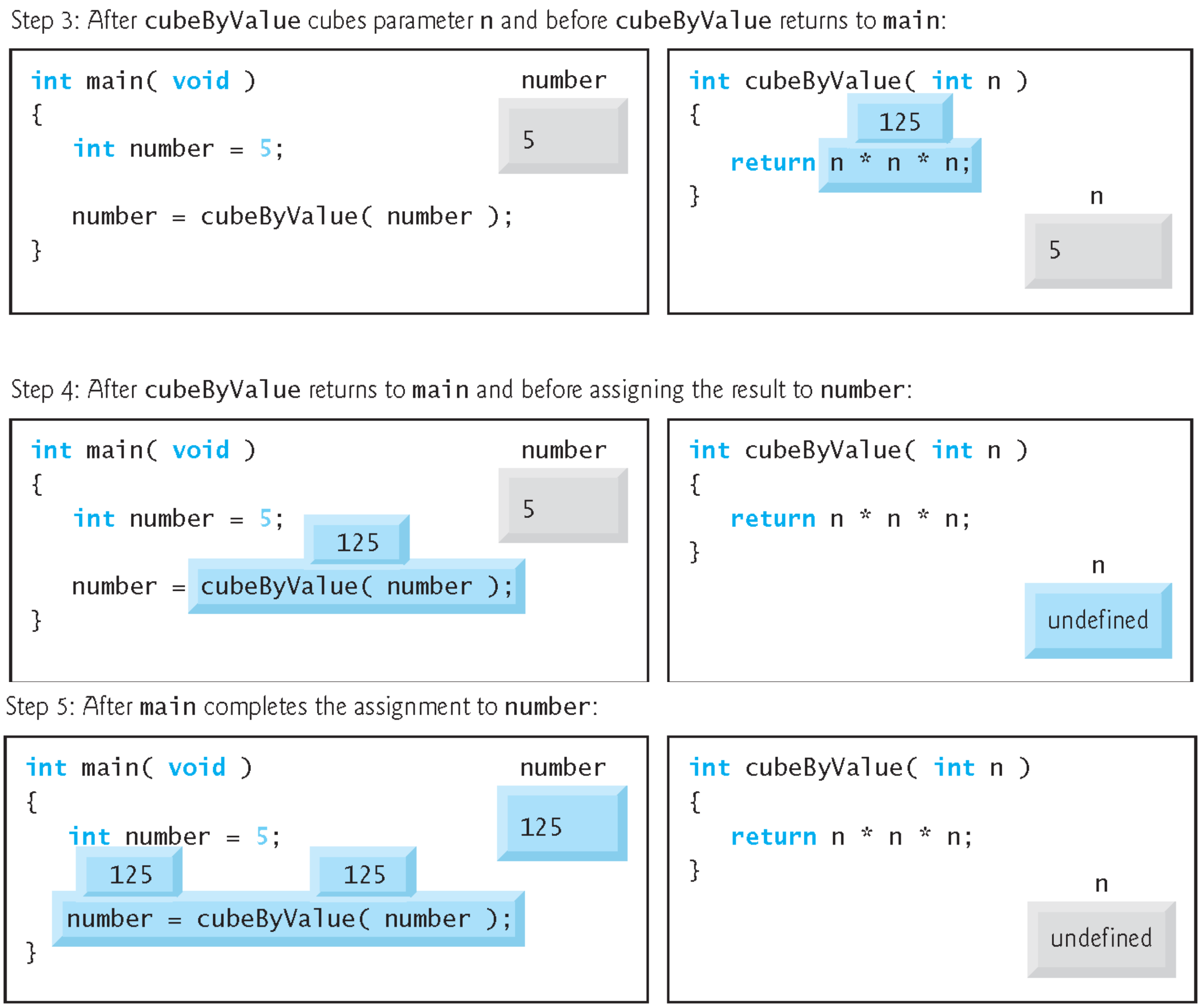
Step 2: After cubeByValue receives the call:

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

   number = cubeByValue( number );
}
```

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

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

Passing Arguments to Functions



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

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

   number = cubeByValue( number );
}
```

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

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

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

   number = cubeByValue( number );
}
```

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

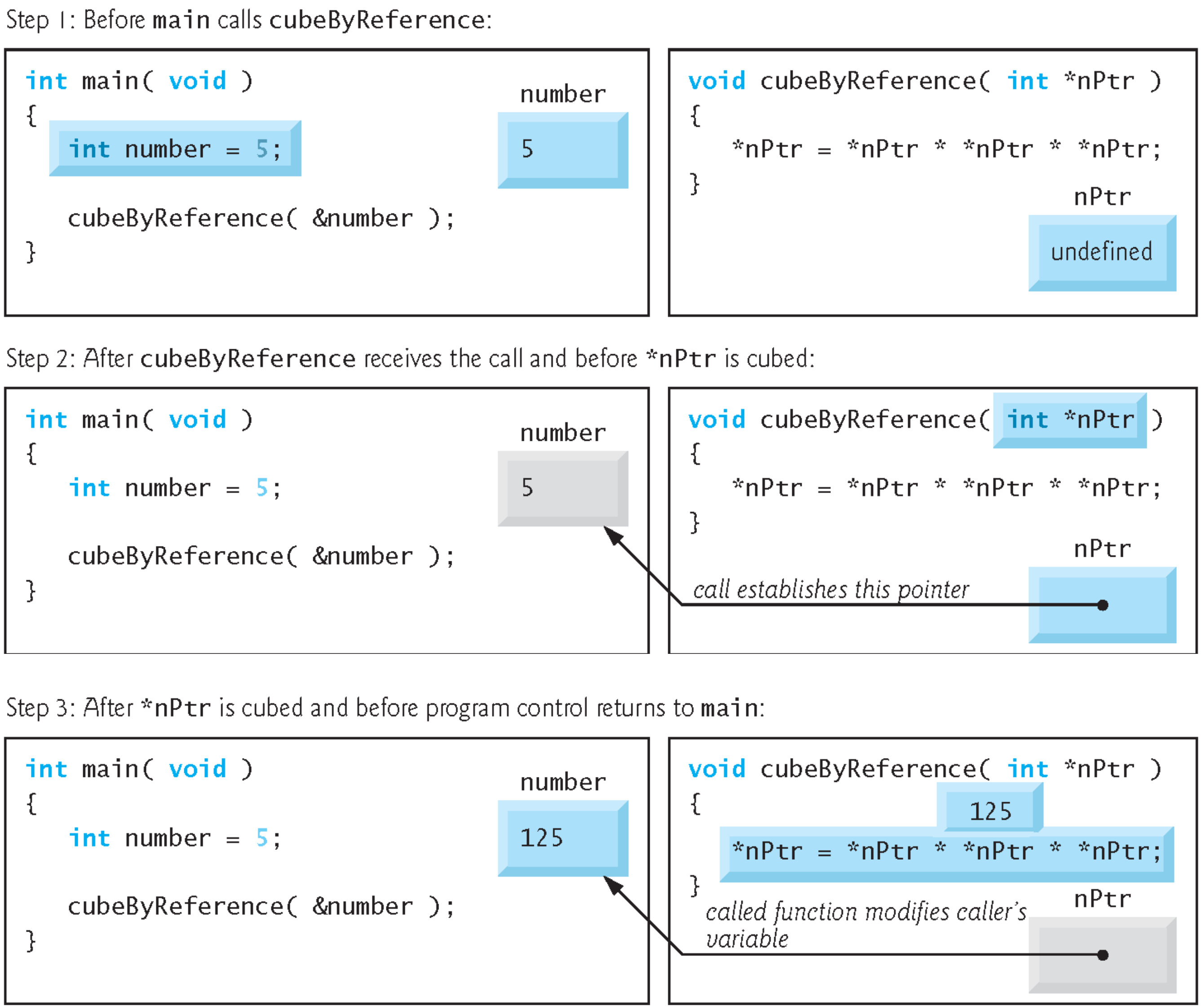
Step 5: After main completes the assignment to number:

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

   number = cubeByValue( number );
}
```

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

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

Passing Arguments to Functions



Step 1: Before main calls cubeByReference:

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

   cubeByReference( &number );
}
```

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

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

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

   cubeByReference( &number );
}
```

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

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

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

   cubeByReference( &number );
}
```

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

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

Using the const Qualifier with Pointers

Const

- The const qualifier enables you to inform the compiler that the *value* of a particular variable *should not be modified*
- Always award a function enough access to the data in its parameters to accomplish its specified task, but absolutely no more
- If an attempt is made to *modify a value that's declared const*, the *compiler* catches it and issues either a *warning* or an *error*, depending on the particular compiler

Using the const Qualifier with Pointers

```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 );

int main( void )
{
   // initialize char array
   char string[] = "print characters of a string";

   puts( "The string is:" );
   printCharacters( string );
   puts( "" );
} // end main

// 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
```

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

Fig. 7.11 | Printing a string one character at a time using a non-constant pointer to constant data. (Part 2 of 2.)

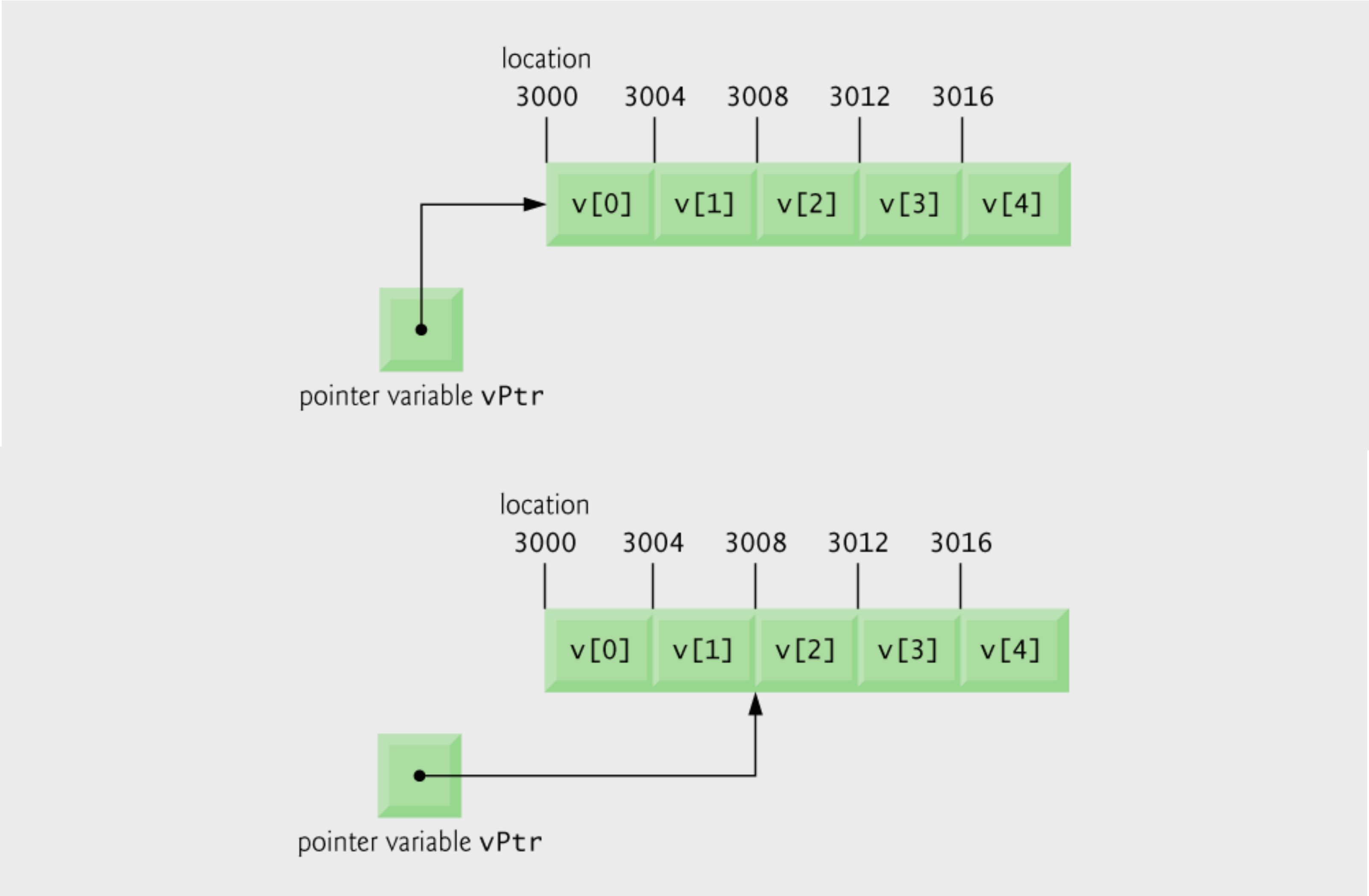
- Pointers and Arrays

Pointer Arithmetic

Pointer Arithmetic

- *Arithmetic operations* can be performed on *pointers*
  - **Increment/decrement** pointer (++ or --)
  - **Add** an **integer** to a pointer(+ or += , - or -=)
  - Pointers may be **subtracted** from each other
  - **Operation** is **meaningless unless** it is **performed** on an array
- For example, 5 element *int array* on machine with *4-byte* `integers`
  - **vPtr** points to first element **v[0]** at memory location 3000
    - sets **vPtr** to 3000
  - **vPtr += 2** (or **vPtr = vPtr + 2**)
    - sets **vPtr** to 3008
    - vPtr points to v[2] (incremented by 2), but the machine has 4-byte ints, so it points to address 3008

Pointer Arithmetic



Pointers and Arrays

- Arrays and pointers are closely related
  - *Array name* is like a *constant pointer*
  - Pointers can do array subscripting operations
- For example, define an array **b[5]** and a pointer **bPtr**
  - To set them equal to one another

    ```
    bPtr = b;
    ```

  - The array name **b** is actually the address of first element of the array b

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

  - Explicitly assigns **bPtr** to the address of the first element of **b**

Pointers and Arrays

- For element **b[3]**, it can be accessed as
  - *pointer/offset* notation

    ***(bPtr + 3)** where 3 is the offset
  - *pointer/subscript* notation

    bPtr[3] where 3 is the subscript/index
  - performing *pointer arithmetic* on the array itself

    ***(b + 3)**
  - **bPtr[3]** is the same as **b[3]** and
    ***(bptr + 3)** is the same as ***(b + 3)**

Pointers and Arrays

- The name of an array on its own is a pointer to the first element of the array:

  ```
  arr == &arr[0]
  ```

- To print out each element of an array:

```
int arr[SIZE], *p, i;

p = arr;

for (i = 0; i < SIZE; i++)
{
  printf (“%d”, *p++);
}
```

- We could also use `printf (“%d”, *(p+i));`

ENG1008 Programming

- Files I/O

Dr Kwee Hiong Lee
kweehiong.lee@singaporetech.edu.sg

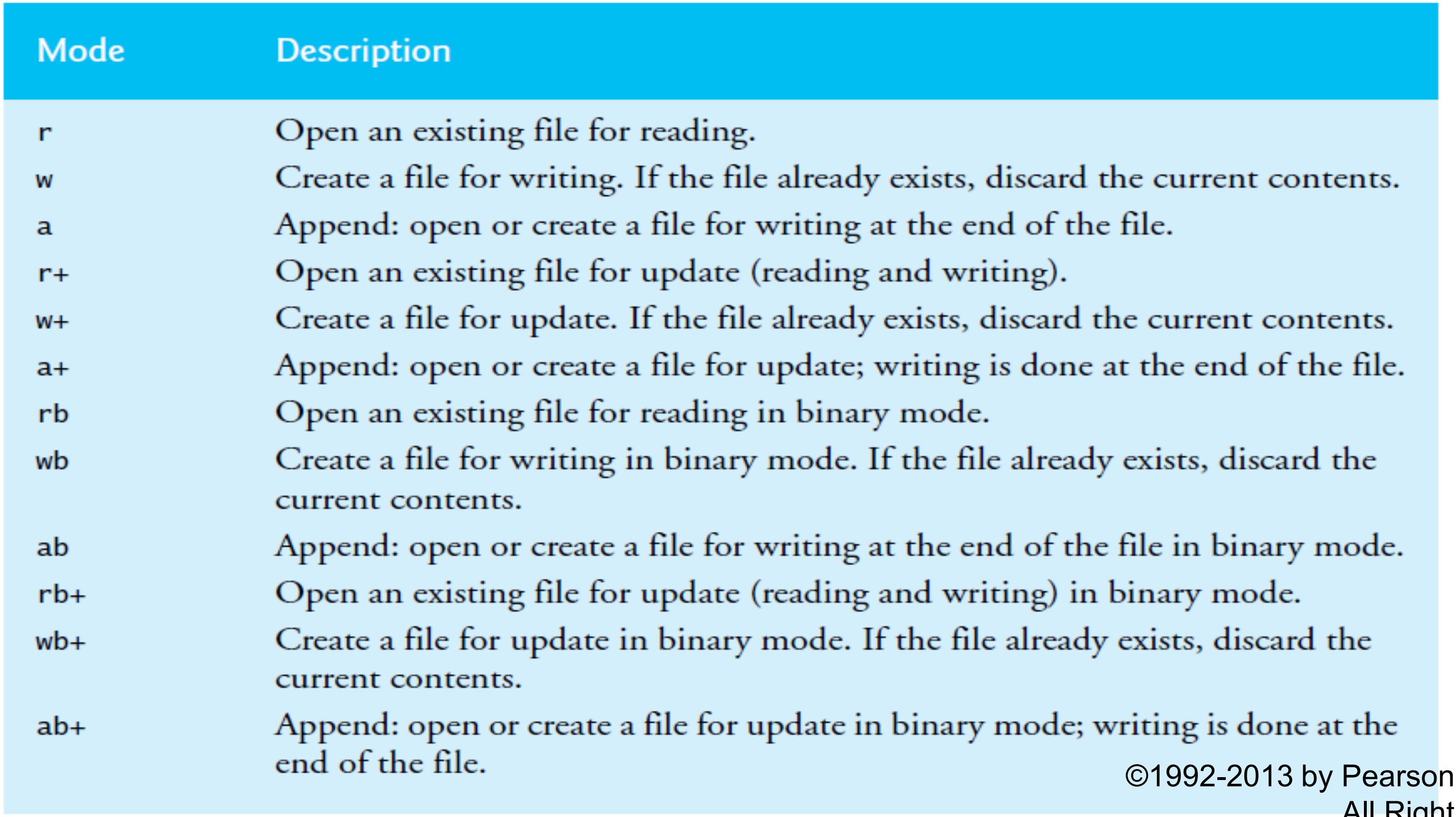
Files Input/Output

- Use for permanent retention of data
- **Stream** of bytes ending with **end-of-file marker** or at specific byte number recorded in administrative data structure
- When program runs, three files and their streams, which provide communication channels between files and programs, are opened
  - **Standard input** (stdin) *stream reads data* from keyboard
    - fgetc, similar to getchar, reads one character from a file
    - fgets reads a line from a file
  - **Standard output** (stdout) *stream prints data* on screen
    - fputc, similar to putchar, writes one character to a file
    - fputs write a line to a file
  - **Standard error** (stderr) *stream writes error messages* to the screen

Sequential-Access File I/O

- C program administers each file with a separate FILE structure
- A pointer of type FILE for each open file is required
- fopen takes two arguments (filename and file open mode) and returns pointer to FILE structure for file opened

File opening modes



Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append: open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append: open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Sequential-Access File I/O

- A pointer of type FILE; fopen using “r” - **Read** data stored in files
- fscanf receives **file pointer** (e.g. cfPtr) for file from which data will be read, retrieving data from file

```c
// Fig. 11.6: fig11_06.c
// Reading and printing a sequential file
#include <stdio.h>

int main( void )
{
   unsigned int account; // account number
   char name[ 30 ]; // account name
   double balance; // account balance

   FILE *cfPtr; // cfPtr = clients.dat file pointer

   // fopen opens file; exits program if file cannot be opened
   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
      puts( "File could not be opened" );
   } // end if
   else { // read account, name and balance from file
      printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
      fscanf( cfPtr, "%d%29s%lf", &account, name, &balance );

      // while not end of file
      while ( !feof( cfPtr ) ) {
         printf( "%-10d%-13s%7.2f\n", account, name, balance );
         fscanf( cfPtr, "%d%29s%lf", &account, name, &balance );
      } // end while

      fclose( cfPtr ); // fclose closes the file
   } // end else
} // end main
```

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
```

cfPtr is a pointer to FILE structure

Open existing file for reading

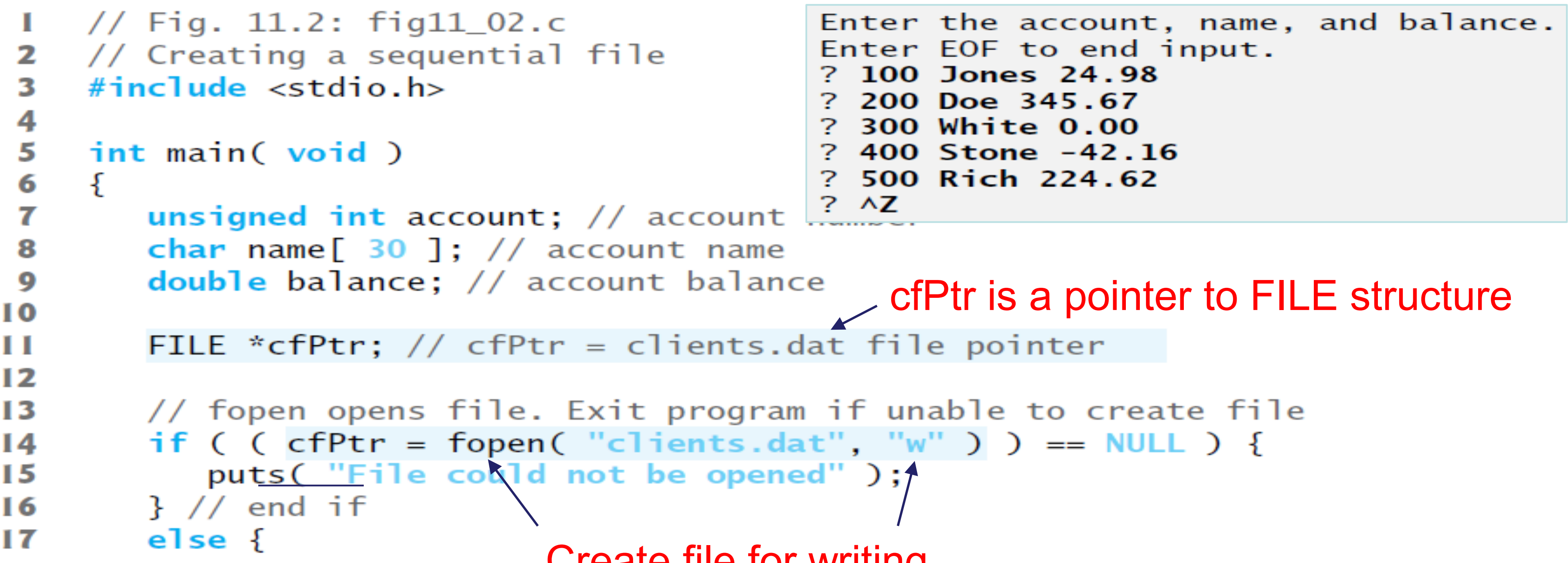
fscanf reads data from the opened file

fclose closes the opened file

Sequential-Access File I/O

- fclose receives *file pointer* for file to be closed, hence closing open file
- If fclose is not called explicitly, operating system closes file when program execution terminates

- fopen using “w” - Create a sequential-access file



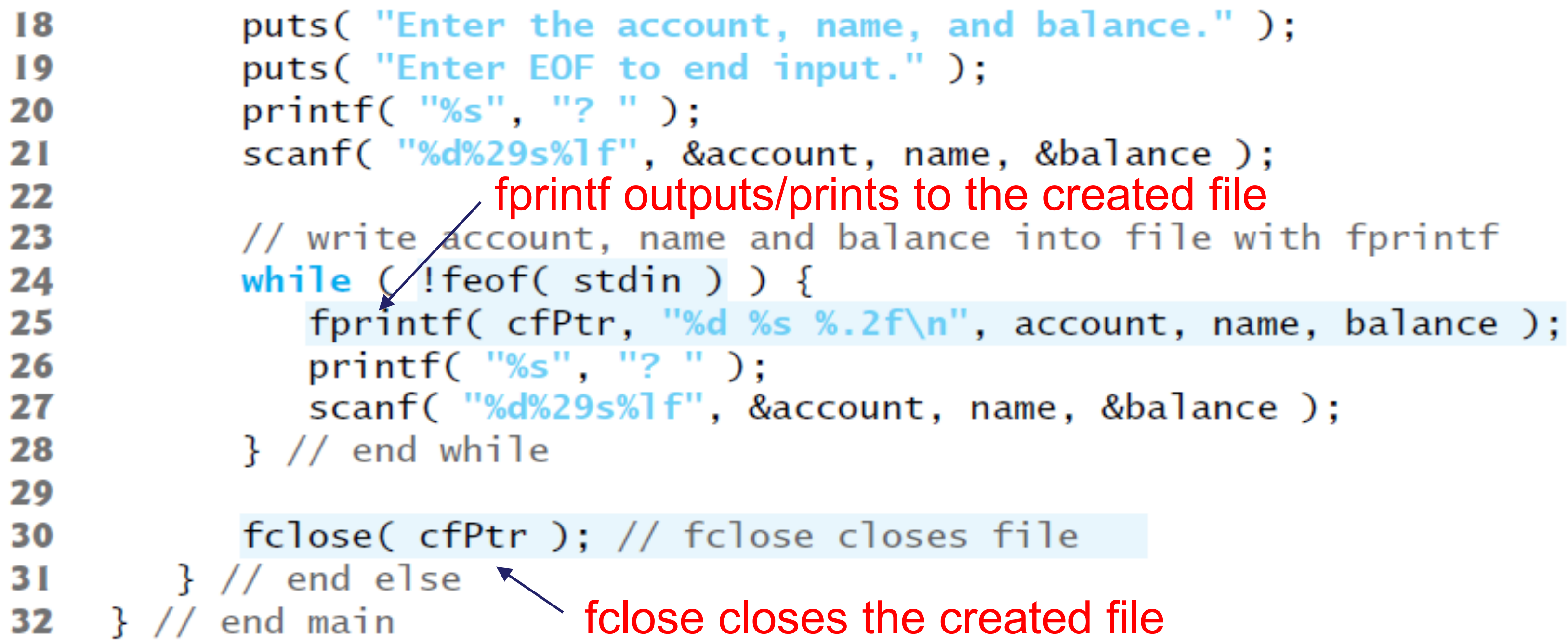
```c
// Fig. 11.2: fig11_02.c
// Creating a sequential file
#include <stdio.h>

int main( void )
{
   unsigned int account; // account number
   char name[ 30 ]; // account name
   double balance; // account balance

   FILE *cfPtr; // cfPtr = clients.dat file pointer

   // fopen opens file. Exit program if unable to create file
   if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
      puts( "File could not be opened" );
   } // end if
   else {
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

cfPtr is a pointer to FILE structure

Create file for writing

Sequential-Access File I/O

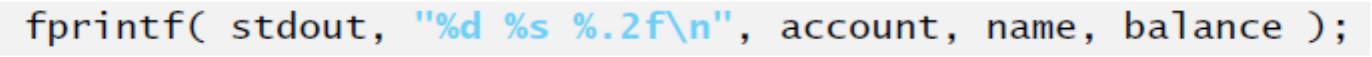


```
      puts( "Enter the account, name, and balance." );
      puts( "Enter EOF to end input." );
      printf( "%s", "? " );
      scanf( "%d%29s%lf", &account, name, &balance );

      // write account, name and balance into file with fprintf
      while ( !feof( stdin ) ) {
         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
         printf( "%s", "? " );
         scanf( "%d%29s%lf", &account, name, &balance );
      } // end while

      fclose( cfPtr ); // fclose closes file
   } // end else
} // end main
```

fprintf outputs/prints to the created file

fclose closes the created file

- fprintf receives file pointer (e.g. cfPtr) for file to which data will be written, hence writing data to file
- Use **stdout** as file pointer to output data



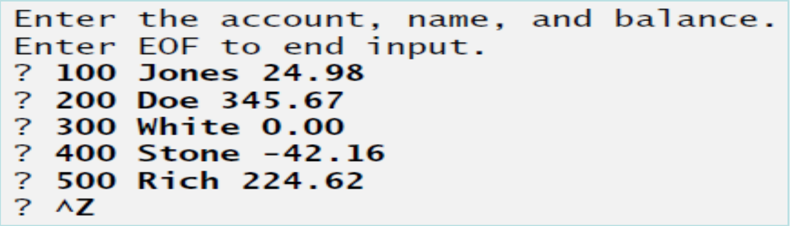
```
fprintf( stdout, "%d %s %.2f\n", account, name, balance );
```

Sequential-Access File I/O

```
      puts( "Enter the account, name, and balance." );
      puts( "Enter EOF to end input." );
      printf( "%s", "? " );
      scanf( "%d%29s%lf", &account, name, &balance );

      // write account, name and balance into file with fprintf
      while ( !feof( stdin ) ) {
         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
         printf( "%s", "? " );
         scanf( "%d%29s%lf", &account, name, &balance );
      } // end while

      fclose( cfPtr ); // fclose closes f
   } // end else
} // end main
```

feof checks for end-of-file indicator

Read 29 characters



```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

- feof checks if end-of-file indicator is set for file that stdin refers
  - Microsoft Windows system: Crtl+z
  - Linux/UNIX/Mac OS X systems: Crtl+d
  - Return nonzero (true) value when end-of-file indicator has been set; otherwise, return zero (false)