Errata – Edition 201901

Example 7.2.4

Example 7.2.4 Inefficient if statement

```
if ( avg >= 90 )
    cout << "A" << endl;

if ( avg >= 80 && avg < 90 )
    cout << "B" << endl;
if ( avg >= 70 && avg < 80 )
    cout << "C" << endl;

if ( avg >= 60 && avg < 70 )
    cout << "D" << endl;

if ( avg < 60 )
    cout << "F" << endl;
```

For a value of 90 and keeping in mind short-circuit evaluation, all `if` statements need to be fully assessed resulting in a total of eight conditions evaluated. Now, how many conditions are evaluated in Example 7.2.5 if `avg` is 80?

Example 10.7.4

Example 10.7.4 Flushing the input buffer with the ignore function

```
char whole_name[81];

cout << "Please enter your whole name: ";

cin.ignore ( cin.rdbuf ( )->in_avail ( ) );// Flush garbage from buffer
cin.getline ( whole_name, 81 );
cin.clear ( );                              // Clear the error flags
cin.ignore ( cin.rdbuf ( )->in_avail ( ) );// Flush characters not read

cout << "Here is the whole name: " <<  whole_name << endl;
```

Notice in Example 10.7.4, we flush the buffer before and after our call to `.getline`. The first `.ignore` cleans the buffer prior to the `.getline` call. The second flush is used to clean any remaining characters in the buffer after the `.getline` function has executed. What exactly is that nasty looking line of code doing? Unfortunately, any detailed explanation at this point would probably be meaningless. Therefore, the simple explanation is the line of code flushes the input buffer. A little more detailed explanation is that the `.ignore` member function removes a certain number of characters from the keyboard buffer.

Everything in the parentheses for the `.ignore` function call allows us to find the number of characters currently in the keyboard buffer. Although not intuitive, `.getline` sets a flag if it reads the maximum number of characters. If this happens, the `.clear` function call is required to reset any flags that may have been set by the `.getline` function. Note: This method currently works with Windows and Visual Studio. Other platforms may not work as described.

Based upon the above discussion it may appear that `.getline` is more trouble than it's worth. In reality, however, `.getline` is *much* safer to read cStrings from the keyboard than `cin`. It has the benefit of being able to read spaces and also prevents buffer overrun errors.

Remember: Using cin allows us to read one word at a time. The .getline method allows us to read a sentence.

Example 12.15.5

Example 12.15.5 The qsort function

```
int main ( )
{
    const char * shrooms[10] =
    {
        "Matsutake", "Lobster", "Oyster", "King Boletus",
        "Shaggy Mane", "Morel", "Chanterelle", "Calf Brain",
        "Pig's Ear", "Chicken of the Woods"
    };

    int nums[10] = { 99, 43, 23, 100, 66, 12, 0, 125, 76, 2 };

    // The address of the array, number of elements the size of each
    // element, the function pointer to compare two of the elements
    qsort ( (void *)shrooms, 10, sizeof ( char * ), compare_strs );
    qsort ( (void *)nums, 10, sizeof ( int ), compare_ints );

    // Output sorted lists
    for ( int i = 0; i < 10; ++i )
        cout << shrooms[i] << endl;

    for ( int i = 0; i < 10; ++i )
        cout << nums[i] << endl;

    return 0;
}
int compare_ints ( const void * arg1, const void * arg2 )
{
    int return_value = 0;

    if ( *(int *)arg1 < *(int *)arg2 )
        return_value = -1;
    else if ( *(int *)arg1 > *(int *)arg2 )
        return_value = 1;

    return return_value;
}
int compare_strs ( const void * arg1, const void * arg2 )
{
    return ( _stricmp ( *( (char **) arg1 ), *( (char **) arg2 ) ) );
}
```

Chapter 12 Debug Exercise

Chapter 12 Debugging exercise

```
/*****************************************************************
* File: Chapter 12 Debug.cpp
*
* General Instructions: Complete each step before proceeding to the
* next.
*
* Debugging Exercise 1
*
* 1) Create Breakpoint 1 where indicated in the code.
* 2) Run the program using the Start Without Debugging option. Ignore
*    any warnings that your compiler may identify.
* 3) Your program should produce a runtime error.
* 4) Click on the Abort button of the runtime error notification
*    dialog box.
* 5) Run to Breakpoint 1.
* 6) Step over the delete statement to verify that this is the line
*    causing the problem. What caused the runtime error?
* 7) Close the runtime error dialog box.
* 8) Initialize int_ptr to nullptr instead of &int_var.
* 9) Run the program using the Start Without Debugging option.
* 10) Notice the runtime error no longer appears. What does this
*     tell you about deleting nullptr?
* 11) Disable Breakpoint 1.
*
* Debugging Exercise 2
*
* 1) Run the program using the Start Debugging option.
* 2) When the program is finished executing, inspect the Output
*    window (not the console window) in the compiler.
* 3) Notice that there are memory leaks identified in the Output
*    window.
* 4) Inspect the information displayed in the Output window. Notice
*    the number of bytes left allocated to the program. Also notice
*    the hexadecimal display of the data.
* 5) Plug the leaks! Uncomment the delete statements located just
*    above the return statement.
* 6) Rerun the program using the Start Debugging option.
* 7) You broke it! Create Breakpoint 2 as indicated in the code.
* 8) Run to Breakpoint 2.
* 9) Put watches on both double pointers. Notice that they have the
*    same address. Deleting the first pointer releases the memory.
*    Deleting the second pointer most of the time will produce an
*    error because the memory has already been released.
* 10) Step over until your program once again crashes.
* 11) Notice that there are 40 bytes of memory still
*     allocated when the program crashes. This is because dbl_ptr1
*     now has the address of dbl_ptr2, not the original memory
*     allocated when dbl_ptr1 was declared. The memory originally
*     allocated will not be able to be recovered because there
*     isn't a pointer to that address any more.
* 12) Comment out the following line of code in the program:
*         dbl_ptr1 = dbl_ptr2;
* 13) Run to Breakpoint 2.
* 14) Step over the delete statement.
* 15) You broke it again! Now it is broken on the delete statement
```

```
*     for dbl_ptr1.
* 16) Look at the for loop. Notice that it is going out of bounds of
*     the dynamic array. Remember, when dynamic memory is
*     deallocated, the heap is inspected in the general area of the
*     deallocated memory. If it has been disturbed, the program
*     will crash.
* 17) Fix the condition so that it stays within the bounds of the
*     array.
* 18) Rerun the program to Breakpoint 2 and step over the delete
*     statement.
* 19) Notice there is still an error on the delete statement.
*
* Debugging Exercise 3
*
* 1) Create Breakpoint 3 as indicated in the code.
* 2) Run to Breakpoint 3.
* 3) Write down the address currently stored in dbl_ptr1.
* 4) Run to Breakpoint 2.
* 5) What address is currently stored in dbl_ptr1? Notice the
*    difference? Remember, you must delete using the address
*    returned by new.
* 6) Stop debugging.
* 7) Uncomment the following lines of code in the program:
*         double * dbl_ptr3 = dbl_ptr1;
*         dbl_ptr1 = dbl_ptr3;
* 8) Disable all breakpoints.
* 9) Run your program using the Start Debugging option.
* 10) Verify that there are no memory leaks and that the program
*     doesn't crash.
**********************************************************************/

#define _CRTDBG_MAP_ALLOC

#include <crtdbg.h>

int main ( )
{
    _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );

    int int_var = 99;
    int * int_ptr = &int_var;
    char * ch_ptr = new char ( );
    double * dbl_ptr1 = new double[5];
    double * dbl_ptr2 = new double[10];
    //double * dbl_ptr3 = dbl_ptr1;

    // Breakpoint 3
    // Put a breakpoint on the following line
    for ( int i = 0; i <= 5; i++, dbl_ptr1++ )
        *dbl_ptr1 = i;

    dbl_ptr1 = dbl_ptr2;

    //dbl_ptr1 = dbl_ptr3;

    // Breakpoint 1
    // Put a breakpoint on the following line
    delete int_ptr;
```

```
    // delete ch_ptr;

    // Breakpoint 2
    // Put a breakpoint on the following line
    // delete [] dbl_ptr1;
    // delete [] dbl_ptr2;

    return 0;
}
```

Table 14.6.1

Table 14.6.1 Bitwise operators

Operator	Description
~	NOT. Reverse the bit. A 1 results in a 0 and a 0 results in a 1.
>>	SHIFT RIGHT. Shift all bits to the right by the specified number of bits.
<<	SHIFT LEFT. Shift all bits to the left by the specified number of bits.
&	AND. The two bits must be 1s to result in a 1.
^	XOR. The two bits must be different to result in a 1.
|	OR. The two bits must be 0s to result in a 0.

Example 14.6.8

Example 14.6.8 Determining permissions

```
#define STUDENT 1
#define FACULTY 2
#define ADMIN 4

int main ( )
{
    unsigned char permissions = FACULTY | ADMIN;
    DisplayPermissions ( permissions );

    return 0;
}
void DisplayPermissions ( char permissions )
{
    cout << "You have the following permissions:\n";
    if ( permissions & STUDENT )
        cout << "\tStudent" << endl;
    if ( permissions & FACULTY )
        cout << "\tFaculty" << endl;
    if ( permissions & ADMIN )
        cout << "\tAdministrator" << endl;
}
```

15.9 Answers to Chapter Exercises

Section 15.5

1. `fin.seekg( 10, ios::cur );`