Intermediate Programming Methodologies in C++

CIS22B

Joe Bentley

DeAnzaCollege
Computer Information System
July 10, 2020

Contents

Review

CIS22A Basics

An Old CIS22A Midterm

1. Write a program that calculates the tax on an item purchased. You should prompt the user for cost of the item. Your program should make use of a named constant for the tax. This constant should represent a tax rate of 7%.

   Program execution:

   ```
   What is the cost of the item?  100.00    // User input is 100.00
   The tax is $7.00                        // Multiple the cost by the tax rate
   ```

2. Write a function definition for a function called getAge. The function prompts the user for their age and then returns it.

3. Write a function prototype for the function that you defined in problem 2.

4. Insert into the following main() an appropriate call to the function that you defined in problem 2.

   ```
   int main()
   {
      _________________  // function call

      return 0;
   }
   ```

5. Which of the following is NOT a valid identifier? Enter letter -> __________
   A. cout
   B. Cout
   C. _55
   D. 55_
   E. end

6. Assume the following declaration is made:

   double A = 9876.543;

   Write one cout statement that prints the number A two times with exactly the following format.

   ```
   ΔΔΔ9877ΔΔΔΔ9876.543
   ```
   // Note: each Δ represents 1 space

   ____

7. Assume the following declaration:

   int M = 12345;

   ```
   Write one statement that will print M three times like this:

   12345 45 12
   ```

   ____

8. What is the output from each cout statement?

   ```
   cout << (9 / 6 + 9 * 6);                             ______
   cout << static_cast<double>(2 / 5 + 5 * 2);          ______
   cout << 100.0/(2 + 8.0 * 6);                         ______
   cout << (6 / 9 ? 6 : 9)                              ______
   cout << rand() % 10 / 10 * 10 + 10 - 10;             ______
   ```

9. Assuming a = 3, b = 4, c = 5, and d = 6, what is the value of each cout statement?

   ```
   cout << d++ / --b;                                         ______
   cout << (a + c) % a - c;                                   ______
   cout << static_cast<char>(c * c * c - a * a * a);          ______
   cout << ++a % c++;                                         ______
   ```

10. Write a function that prints a day of the week. The function should take an int argument. It must use a switch.
    If the argument is 1, then the function should print Sunday.
    If the argument is 2, then the function should print Monday.
    If the argument is 3, then the function should print Tuesday.
    …
    If the argument is 7, then the function should print Saturday.
    If the argument is not between 1 and 7, it should print "error"

11. Write a program that prints multiples of 8 that are less than 9999. The output should look like this:

    ```
    8
    16
    24
    32
    …
    9992
    ```

Warning – stay away from these

C-style casts

```
float fraction;

fraction = (float) 1 / 2;                // C-style cast - BAD
fraction = static_cast<float>(1) / 2;    // C++ cast - GOOD
```

C standard header files

Do **not** use these header files

```
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdio.h>
```

Use these header files

```
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <cstring>
#include <iostream>
```

Variable length arrays

```
int size = 5;
int array[size];    // variable length array - BAD

const int size = 5;
int array[size];    // fixed length array - GOOD
```

File I/O

File streams

A file stream represents data to be written to a file (insertion) or read from a file (extraction). Interacting with a file stream is how file I/O is performed. Extraction, or file input, is the result of copying data from a file into computer memory. Insertion, or file output, is the result of copying data from computer memory into a file.

Input/Output classes

class	Purpose
ios	Provides functions and characteristics that are common to both input and output
ostream	Output to the screen (or monitor or console). cout is an ostream object. variable)
ofstream	Output to a file. The ofstream class inherits functions from the ostream class.
istream	Input from the keyboard. cin is an istream object.
ifstream	Input from a file. The ifstream class inherits functions from the istream class.
fstream	File input and output. Inherits functions from the ifstream and ofstream.

You can access the ostream class and istream class by including the <iostream> header file, and the ofstream class and ifstream class by including the <fstream> header file.

Opening a file

```
// Declare a stream object
ifstream fin;
ofstream fout;

// open a file associated with a file stream object
fin.open("inputfile.txt");
fout.open("outputfile.txt");
```

Opening a file using a constructor

```
// Declare a stream object and open a file at the same time
ifstream fin("input.txt");
ofstream fout("output.txt");
```

Checking for a successful open

There are several approaches for checking whether or not a file is successfully opened. Checking the file open is almost always a good idea. There are many reasons why a file many not be opened, such as:
The file name may be misspelled (and hence, does not exist).
The file name may reference a disk drive that does not exist.
The file name may reference a directory path that does not exist.
A referenced device may not exist.
The disk drive may be full.
The drive destination may be defective.

Functions

ios::good

returns true if the file is open and there are no errors in the file stream

ios::fail

returns true if the file is not open or there are errors in the file stream

ios::bad

returns true if there are errors in the file stream. Do **not** use the bad function to test a file open.

ios::operator!

returns true if the file is not open or there are errors in the file stream

ifstream::is_open / ofstream::is_open / fstream::is_open

returns true if the file is open

Example 1-1 – File open check

```
#include <iostream>
#include <fstream>
#include <cstdlib>  // for exit()
using namespace std;

int main()
{
    cout << boolalpha;
    ofstream fout("c:/temp/goodfile");
     cout << "fout.good()=" <<  fout.good() << endl;
     cout << "fout.fail()=" <<fout.fail() << endl;
     cout << "fout.bad()=" << fout.bad() << endl;
     cout << "fout.is_open()=" <<fout.is_open() << endl;
     cout << "!fout=" <<!fout << endl << endl;

     ifstream fin("c:/temp/fakefile");
     cout << "fin.good()=" <<  fin.good() << endl;
     cout << "fin.fail()=" <<fin.fail() << endl;
     cout << "fin.bad()=" << fin.bad() << endl;
     cout << "fin.is_open()=" <<fin.is_open() << endl;
     cout << "!fin=" <<!fin << endl << endl;

     // Check for file opens
     if (fout.is_open())
     {
         // ...
     }
     if (fout.fail())
     {
         cerr << "Unable to open file c:/temp/goodfile" << endl;
```

```
        exit(1);
    }
    if (!fin)
    {
        cerr << "Unable to open file c:/temp/fakefile" << endl;
        exit(2);
    }
}
```

****** Output ******

```
fout.good()=true
fout.fail()=false
fout.bad()=false
fout.is_open()=true
!fout=false

fin.good()=false
fin.fail()=true
fin.bad()=false
fin.is_open()=false
!fin=true

Unable to open file c:/temp/fakefile
```

Closing a file

Use the ifstream::close() / ofstream::close() / fstream::close() to close a file. Note, the stream *destructor* will automatically close a file. That is, when a stream object goes out of scope, the file is automatically closed.

Example

```
ifstream fin(“inputfile.txt”);
…
// read from the input file
…
fin.close();   // file is closed

…

void somefunction()
{
   ofstream fout(“outputfile.txt”);
   …
   // write to output file
   …
} ← output file is closed here
```

Reading from a file

istream::operator>>

Reads from a file into built-in (or primitive) types or into string objects. Whitespace[1] is a separator or delimiter for each value read.

istream::get()

Reads from a file into a null-terminated char array (c-string). The istream::get function is overloaded.

```
(1)   int get();
(2)   istream& get(char& c);
(3)   istream& get(char* s, streamsize n);
(4)   istream& get(char* s, streamsize n, char delim);
```

(1) Reads from the input file stream and returns the ascii code of the character (byte) read.
(2) Reads from the input file stream into a char
(3) Reads from the input file stream into a char array. A maximum of n-1 bytes are read. A newline (\n) acts as a delimiter and ends the read. The newline is not read and not stored in the char array. **The next read operator will begin at the newline character**.
(4) Reads from the input file stream into a char array. A maximum of n-1 bytes are read. The delim character acts as a delimiter and ends the read. The delimiter is **not** read and not stored in the char array. **The next read operator will begin at the delimiter.** If the delimiter character is not encountered in the specified number of bytes, n, then an error is introduced into the stream.

istream::getline()

Reads from a file into a null-terminated char array (c-string). The getline function is used to read an entire line from a file into a char array. The istream::getline function is overloaded.

```
(1)   istream& getline(char* s, streamsize n);
(2)   istream& getline(char* s, streamsize n, char delim);
```

(1) Reads from the input file stream into a char array. A maximum of n-1 bytes are read. A newline (\n) acts as a delimiter and ends the read. The newline is read and not stored in the char array. **The next read operator will begin at character (byte) after the newline character**.
(2) Reads from the input file stream into a char array. A maximum of n-1 bytes are read. The delim character acts as a delimiter and ends the read. The delimiter **is** read and not stored in

[1] Whitespace is a blank space, a newline, or a tab character

the char array. **The next read operator will begin at the character after the delimiter**. If the delimiter character is not encountered in the specified number of bytes, n, then an error is introduced into the stream.

std::getline(string)

Reads from a file into a string. The istream::getline function is overloaded.

```
(1)   istream& getline(istream& is, string& str, char delimiter);
(2)   istream& getline(istream& is, string& str);
```

(1) Reads from the input file stream into a string. The file contents are read until the delimiter character is found. The string, str, is automatically sized to hold the value read from the input file. The delimiter is read but not stored. The next read operator will begin at the character after the delimiter.
(2) Reads from the input file stream into a string. This syntax reads a line from an input file. It is probably the most common approach for reading string input from a file. The file contents are read until a newline character is found. The string, str, is automatically sized to hold the value read from the input file. The delimiter is read but not stored. The next read operator will begin at the character after the newline.

Example 1-2 – Reading from a file

veryniceday.txt

```
Have a
very nice
day
```

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // for exit()
using namespace std;

int main()
{
    const string filename = "veryniceday.txt";
    string buffer;
    char cstring[80];
    char ch;
    ifstream fin(filename);
    if (!fin)
    {
        cerr << "Unable to open file " << filename << endl;
        exit(1);
    }

    while (fin >> buffer)
        cout << '/' << buffer;
    cout << endl << "-------------------------------\n";

    fin.clear();               // reset EOF state
    fin.seekg(0);              // reposition to the top of the file

    while (getline(fin, buffer))
        cout << buffer << endl;
    cout << endl << "-------------------------------\n";

    // Differentiate get and getline
    fin.clear();               // reset EOF state
    fin.seekg(0);              // reposition to the top of the file
    fin.get(cstring,sizeof(buffer),'e');
    cout << "cstring=" << cstring << endl;
    fin.get(ch);
    cout << "ch=" << ch << endl;
    cout << "fin.get()=" << fin.get() << endl;
    cout << endl << "-------------------------------\n";

    fin.seekg(0);                // reposition to the top of the file
    fin.getline(cstring,sizeof(buffer),'e');
    cout << "cstring=" << cstring << endl;
    fin.get(ch);
    cout << "ch=" << ch << endl;
    cout << "fin.get()=" << fin.get() << endl;
}
```

****** Output ******

```
/Have/a/very/nice/day
-------------------------------
Have a
very nice
day

-------------------------------
cstring=Hav
ch=e
fin.get()=32

-------------------------------
cstring=Hav
ch=
fin.get()=97
```

Writing to a file

<< operator

Writes built-in (or primitive) types or string objects into a file.

Manipulators

Manipulators are special functions designed to be used with the << and >> operators for input and output. Manipulators have the same effect on file streams that they have on input and output streams (cin and cout).

The iomanip header file

The **iomanip** header file is required for parameterized manipulators. A parameterized manipulator is one that uses are argument, such as setw or setfill.

Standard manipulators

Manipulator	I/O	Purpose

Independent Flags		**Turns Setting On**
boolalpha	I/O	sets boolalpha flag
showbase	O	sets showbase flag
showpoint	O	sets showpoint flag
showpos	O	sets showpos flag
skipws	I	sets skipws flag
unitbuf	O	sets unitbuf flag
uppercase	O	sets uppercase flag
Independent Flags		**Turns Setting Off**
noboolalpha	I/O	clears boolalpha flag
noshowbase	O	clears showbase flag
noshowpoint	O	clears showpoint flag
noshowpos	O	clears showpos flag
noskipws	I	clears skipws flag
nounitbuf	O	clears unitbuf flag
nouppercase	O	clears uppercase flag
Numeric Base Flags		
dec	I/O	sets dec flag for i/o of integers, clears oct,hex
hex	I/O	sets hex flag for i/o of integers, clears dec,oct
oct	I/O	sets oct flag for i/o of integers, clears dec,hex
hexfloat (C++11)	I/O	sets hexadecimal floating point formatting
defaultfloat (C++11)	I/O	clears the float field formats
Floating Point Flags		
fixed	O	sets fixed flag
scientific	O	sets scientific flag
Adjustment Flags		
internal	O	sets internal flag
left	O	sets left flag
right	O	sets right flag
Input Only		
ws	I	extracts whitespace
Output Only		
endl	O	inserts a newline **and flushes output stream**
ends	O	inserts a null
flush	O	flushes stream
Parameterized Manipulators(these require the ***iomanip*** header file)		
resetiosflags(ios_base::fmtflags mask)	I/O	clears format flags specified by mask
setbase(int base)	I/O	sets integer base (8, 10, or 16)
setfill(char_type ch)	O	sets the fill character to ch
setiosflags(ios::base::fmtflags mask)	I/O	sets format flags to mask value
setprecision(int p)	O	sets precision of floating point numbers
setw(int w)	O	sets output field width to w
get_money (C++11)	I	parses a monetary value
put_money (C++11)	O	formats and outputs a monetary value

get_time (C++11)	I	parses a date/time value
put_time (C++11)	O	formats and outputs a date/time value
quoted (C++14)	I/O	Allows input/output of quoted text

ostream::put()

Writes a single char into a file.

```
ostream& put (char c);
```

Example 1-3 – Writing to a file

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // for exit()
#include <iomanip>  // for setw, setfill, setprecision
using namespace std;

int main()
{
     const string filename = "output.txt";
     string haveaniceday = "have a nice day";
     string have = "have";
     string a = "a";
     string nice = "nice";
     string day = "day";

     ofstream fout(filename);
     if (!fout)
     {
         cerr << "Unable to open output file " << filename << endl;
         exit(1);
     }

     fout << haveaniceday << endl;
     fout << have << a << nice << day << endl;

     // setw
     fout << setw(3) << have << a << nice << day << endl;
     fout << setw(6) << have << a << nice << day << endl;
     fout << setw(6) << have << setw(6) << a << setw(6) << nice
         << setw(6) << day << endl;

     // left and setw
     fout << left << setw(6) << have << setw(6) << a << setw(6)
          << nice << setw(6) << day << endl << endl;

     // setprecision
     double pi = 3.141592654;

     auto saveprec = fout.precision();
     fout << "default precision for fout = " << saveprec << endl;

     fout << pi << ' ' << setprecision(4) << pi << endl;

     // setprecision and fixed
     fout << setprecision(saveprecision);  // reset precision
     fout << pi << ' ' << setprecision(4) << fixed << pi << endl;

     // setfill and setw
50      int number = 123;
51      fout << right;
52      fout << endl << number << setw(5) << number << endl;
53      fout << setfill('0');
54      fout << number << setw(5) << number << endl;
55      fout << setw(16) << pi << endl;
56      fout << left << setw(16) << pi << endl;
57      fout << setfill(' ');
58      fout << setw(16) << pi << endl;
59      fout << right << setw(16) << pi << endl;
60  }
```

Output file: output.txt

```
have a nice day
haveaniceday
haveaniceday
  haveaniceday
  have     a  nice   day
have  a     nice  day

default precision for cout = 6
3.14159 3.142
3.14159 3.1416

123  123
12300123
00000000003.1416
3.14160000000000
3.1416
          3.1416
```

Explanation

Line 25: 4 string values are written to the output file with no spaces separating each value.
Line 28: The setw(3) only applies to the first string written. Since the first string value is 4 characters long, the 3 argument is of no consequence. **Note, the setw manipulator is not used to insert spaces in between values**.
Line 29: The setw(6) only applies to the first string written. The first string value, "`have`", is written with a width of 6 and is right justified in that *field*. The remaining 3 strings are written using the minimum width necessary.
Lines 30 & 31: There is setw manipulator in front of each string value, so each string is written into the file using a width of 6. Note, each value is right justified in the field of width 6.
Line 34: A left manipulator is inserted in front of the first string, so all 4 string values are written left justified in a field of width 6.
Line 40: The default precision setting is saved into a variable, saveprec. This is so that it can be used again later to reset the output precision to the original default setting. The keyword, ***auto***, is

used here to declare the type for saveprec. This allows the compiler to determine the type and the user doesn't have to the know the type returned by the precision function. The auto feature was added in C++11.
Line 41: The default precision setting value is displayed.
Line 43: The value is pi is displayed twice, first using the default precision, then using precision 4. Note, precision 4 means *4 significant digits*.
Line 46: The default precision setting is set back to its original value.
Line 47: pi is again displayed twice, but the fixed manipulator is inserted in front of the second value to be output. Now, there is a different interpretation of setprecision(4). This now means display with 4 decimal places.
Line 51: The right manipulator is inserted into the file output stream, so values inserted into the stream will be right justified in the *field*.
Line 52: The int, number, is written twice into the output file. The first time with a default width and the second time with a width of 5. Note, the 2 extra spaces in front of the second value, caused by the setw(5).
Line 53: The fill character is set to '`0`'.
Line 54: The int is again displayed twice. This time, the 2 extra spaces resulting from the setw(5) are filled with '`0`'.
Line 55: The floating point value, pi, is written into the file using a width of 16. Notice, the right justification and the presence of the fill character.
Line 56: Again, pi is displayed in a width of 16 with the fill character, but this time left justified.
Line 57: The fill character is reset to its default value, a blank space.
Lines 58 and 59: pi is again printed using a setw(16), but the fill character is now a blank space.

What is newline?

The newline character, or endline is used to represent the end of a line of text. By now, you should have experience with this. You may have used the ***endl*** manipulator or ***n*** to represent a newline. Your purpose was to simply end a line of output. However, in reading a file, the newline character can present unexpected problems. Depending on how the input file was created, the newline character may occupy either one or two bytes. The newline character may be represented as only the line feed control character (ascii code 10) or as the carriage return and line feed control characters (ascii codes 13 and 10). You'll see evidence of this in the following example. Because, you are not always the creator of your input file and it may come from a source that you did not expect, you'll need to learn to write code to adapt to either situation.

Example 1-4 – What is a newline?

This example demonstrates the newline character by first writing a text file containing two newlines. That output file is then reopened as an input file, first as a text file and then as a binary file. You should note the different storage and the different processing of the same code with 4 different compilers.

```
#include <iostream>
#include <fstream>
```

```
#include <string>
#include <cstdlib>  // for exit()
#include <iomanip>  // for setw
using namespace std;

int main()
{
     const string filename = "output.txt";
     streampos loc;

     ofstream fout(filename);
     if (!fout)
     {
         cerr << "Unable to open output file " << filename << endl;
         exit(1);
     }

     fout << "ABC\n";
     fout << "DEF\n";
     fout.close();

     ifstream fin(filename);
     if (!fin)
     {
         cerr << "Unable to open input file " << filename << endl;
         exit(2);
     }

     // determine file size
     fin.seekg(0, ios::end);
     loc = fin.tellg();
     cout << "File size = " << loc << endl;

     fin.seekg(0);

     // display byte location and each character in the file
     while (!fin.eof())
     {
         loc = fin.tellg();
         cout << loc << "   " << fin.get() << endl;
     }
     cout << "----------------------" << endl;

     // close the file & reopen in binary mode
     fin.clear();
     fin.close();
     fin.open(filename, ios::binary);
     if (!fin)
     {
         cerr << "Unable to open binary file " << filename << endl;
         exit(3);
```

```
    }

    // display byte location and each character in the file
    while (!fin.eof())
    {
        loc = fin.tellg();
        cout << loc << "   " << fin.get() << endl;
    }
}
```

****** Code::Blocks 17.12 on Windows ******

```
File size = 10
0   65
3   66
4   67
5   10
6   68
7   69
8   70
9   10
10   -1
---------------------
0   65
1   66
2   67
3   13
4   10
5   68
6   69
7   70
8   13
9   10
10   -1
```

****** Microsoft Visual Studio 2017 ******

```
File size = 10
0   65
1   66
2   67
3   10
5   68
6   69
7   70
8   10
10   -1
---------------------
0   65
1   66
```

```
2   67
3   13
4   10
5   68
6   69
7   70
8   13
9   10
10   -1
```

****** Linux gnu compiler g++ 8.1.0 / Xcode 9.4 on Mac ******

```
File size = 8
0   65
1   66
2   67
3   10
4   68
5   69
6   70
7   10
8   -1
---------------------
0   65
1   66
2   67
3   10
4   68
5   69
6   70
7   10
8   -1
```

Explanation

Line 11: streampos is an integer type that is used to represent a byte location in a file.
Lines 20 and 21: two lines are written into the output file.
Line 22: close the output file
Line 24: open the output file as an input file
Line 32: position to the end of the file (the last byte in the file)
Line 33: save the byte position in the file into the variable, loc.
Line 34: display the byte position – that’s the file size.
Line 36: reposition in the file to the beginning of the file
Lines 39-43: in a loop, display the file position (byte) and the character (ascii code) at that position
Line 47: clear the input file stream state – it’s EOF state
Line 48: close the file
Line 49: reopen the file in binary mode

Lines 57-61: in a loop, display the file position (byte) and the character (ascii code) at that position. Note the extra carriage return character (ascii code 13) in the PC compilers.

Example 1-5 – Writing to a file

Input File

```
This is line 1
This is line 2
This is line 3
```

← This file was created as a **Windows text file**

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // for exit()
using namespace std;

void readFile(const string& filename);

int main()
{
    readFile("windows_input.txt");
}

void readFile(const string& filename)
{
    ifstream fin(filename);
    if (!fin)
    {
        cerr << "Unable to open input file " << filename << endl;
        exit(1);
    }
    string buffer;
    cout << "Reading file: " << filename << endl;
    while (!fin.eof())
    {
        getline(fin,buffer);
        cout << buffer;
    }
}
```

****** Output: Windows: Code::Blocks ******

```
Reading file: windows_input.txt
This is line 1This is line 2This is line 3
```

****** Output: Linux gnu compiler ******

```
Reading file: windows_input.txt
```

****** Output: Mac Xcode / Mac Eclipse ******

```
Reading file: windows_input.txt
This is line 1
This is line 2
This is line 3
```

****** Output: Mac Linux gnu compiler ******

```
Reading file: windows_input.txt
```

Comments

This example illustrates how the *type* of a newline can affect program processing. It also demonstrates that the getline function may work differently on different compilers. Since the input file was created as a Windows text file, its newline consists of the two-byte carriage return and line feed (\r and \n). The getline function on Windows compilers reads up to the newline character(s) and *consumes* them. Rather they are not stored in the getline string buffer. The getline function on Mac and Linux compilers reads up to the \n character (and ignores it), and stores the \r character in the getline string buffer. The \r character is the carriage return character. It erases any data on a line and starts the line over again. That is why the output is missing on the gnu compilers. The Windows (Code::Blocks) compiler shows the 3 file lines concatenated together in the output, because the \r and \n are removed. For the Mac Xcode and Eclipse compilers the file output appears on 3 lines because the \r (line feed) is left in the buffer.

So, what is the point? The source of an input file may seriously impact how your file is processed. Later, you'll see techniques to write code to handle either type of newline in a file. Your goal should be to write code that can handle either situation.

Detecting EOF

The presence or absence of a newline character at the end of a file may affect the file processing.

Example 1-6 – Detecting EOF(1)

Input Files

colors1.txt

```
red
white
blue      ← there is a newline after the last line in the file
```

colors2.txt

```
red
white
blue      ← there is no newline after the last line in the file
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

void readFile(const string& filename);

int main()
{
    readFile("colors1.txt");
    readFile("colors2.txt");
}

void readFile(const string& filename)
{
    ifstream fin(filename.c_str());
    string buffer;
    if (!fin)
    {
        cerr << "Can't open " << filename << endl;
        exit(1);
    }
    while (!fin.eof())
    {
        fin >> buffer;
        cout << buffer << endl;
    }
    cout << "End of file" << endl;
    fin.close();
}
```

****** Output ******

```
red
white
blue
blue
End of file
red
white
blue
End of file
```

Explanation

Notice the extra occurrence of blue in the output for the first input file. Why does that happen? Look at the while loop (lines 24-28) – that is where the file processing occurs. The while loop condition specifies that the loop processing continues until EOF is read. With the first input file the third record, “blue”, is successfully read and displayed, so the loop continues. An attempt to read the 4th record is made (line 26). That read fails (EOF error), but we aren’t checking for EOF yet, so processing continues to line 27. Even though the read operation failed, the previous “blue” is still in the input buffer. Hence, it displays the extra “blue” (line 27). Next, we are back at the top of the while loop. This time, with the attempted read of the fourth record, EOF is detected and the while loop ends.

Note, that the second input file, without the newline at the end, does not exhibit this problem.

So, what is the point? An input text file may or may not contain a newline as the last character in the file. You might not be the one that created the input file, so the location of EOF may be beyond your control. You will want to learn to write code to handle either situation.

Example 1-7 – Detecting EOF(2)

Input Files

numbers1.txt

```
  1   2   3
  4   5   6
  7   8   9      ← there is a newline after the last line in the file
```

numbers2.txt

```
  1   2   3
  4   5   6
  7   8   9      ← there is **no** newline after the last line in the file
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int sumOf2ndColumnInFile(const string& filename);

int main()
 {
     cout << sumOf2ndColumnInFile("numbers1.txt") << endl;
     cout << sumOf2ndColumnInFile("numbers2.txt") << endl;
 }
```

```

int sumOf2ndColumnInFile(const string& filename)
{
    ifstream fin(filename.c_str());
    int sum = 0, number, dummy;
    if (!fin)
    {
        cerr << "Can't open " << filename << endl;
        exit(1);
    }
    while (!fin.eof())
    {
        fin >> dummy >> number >> dummy;
        sum += number;
    }
    return sum;
}
```

****** Output ******

```
23
15
```

Explanation

This example is supposed to add the second column of numbers in the file. The two input files differ only by the presence or absence of the newline at the end of the file. The correct answer, of course, is 15. Why does the first input file yield an answer of 23? This is really the same problem that occurs in the last example. The EOF is not checked until after an attempt is made to read a 4th line in the file.

Functions

Terminology

- **Function definition**
- **Function declaration**
- **Function call**
- **Function arguments**
  - **Pass by value**
  - **Pass by reference**
    - Reference to const

```
// function declarations or prototypes
void funk1();
void funk2(int arg);
void funk3(int& arg);
void funk4(const int& arg);

int main()
{
    int q = 19;
    // function calls
    funk1();
    funk2(7);
    funk3(q);
    funk4(q);
}

// function definitions
void funk1()  // function heading
{
    …  // body of function
}

void funk2(int a)  // pass by value
{
    …
}

void funk3(int& ref)   // pass by reference
{
    …
}

void funk4(const int& cref)  // pass reference to const
{
    // cref = 76;   // ERROR
    …
}
```

Pass by value vs. pass by reference

Use pass by reference when you want the function to effect a change in the argument value. Use pass by value when the function does not need to *permanently* change the argument value.

```
// function declarations or prototypes

void funk1(int pbv);
void funk2(int& pbr);

int main()
{
    int q = 19;
    funk1(q);
    cout << q << endl;  // prints 19
    funk2(q);
    cout << q << endl;  // prints 6
}

void funk1(int arg)   // pass by value
{
   arg = 6;
}

void funk2(int& arg)   // pass by reference
{
   arg = 6;
}
```

Local variables

The term, local variable, refers to a variable that is declared inside a function. A local variable may only be accessed inside the function in which they are declared.

```
void funk(void);

int main()
{
    funk();
    // cout << z << endl;  // ERROR
}

void funk()
{
   int z = 7;  // z is a local variable
}
```

Function arguments, when passed by value, are also considered *local to a function*. A local variable means that the variable's ***scope*** is the function.

Local static variable

When a *non-static* variable is local to a function, then it is defined each time the function is called. A local static variable is a local variable that is defined only one time during the execution of a program. The scope of a local static variable, like any local variable, is within the function.

Example 1-8 – Local static variable

```
#include <iostream>
using namespace std;

void funk1();
void funk2();

int main()
{
    funk1();
    funk1();
    funk1();
    cout << endl;
    funk2();
    funk2();
    funk2();
}

void funk1()
{
    cout << "this is funk1: ";
    int x = 1;
    ++x;
    cout << x << endl;
}

void funk2()
{
    cout << "this is funk2: ";
    static int x = 1;
    ++x;
    cout << x << endl;
}
```

****** Output ******

```
this is funk1: 2
this is funk1: 2
this is funk1: 2

this is funk2: 2
this is funk2: 3
this is funk2: 4
```

Return type vs. return value

```
// prototypes

void funk1();                       // return type is void
int funk2(float);                   // return type is int
double funk3(float, float);         // return type is double

Whatever funk4(string s)            // return type is Whatever
{
…
}
```

The return type of a function is declared in the function prototype or in the function heading. For example,

The return value is that value following a return inside the body of a function. That value may be converted to match the type specified in the function heading. For example,

```
void funk1()
{
   cout << "have a nice day" << endl;
}

void funk2()
{
   cout << "have a nice day" << endl;
   return;
}

int funk3()
{
   cout << "have a nice day" << endl;
   return 6;
}

int funk4()
{
   cout << "have a nice day" << endl;
   return 6.5;
}

int funk5()
{
   cout << "have a nice day" << endl;
}
```

Explanation

funk1 has a void return, or you would say "it has no return value".
funk2 has a void return, or you would say "it has no return value".
funk3 has an int return, its return value is 6.
funk4 has an int return, its return value is 6. The double, 6.5, would be converted to an int upon return.
funk5 has an error. The function is supposed to return an int, but no value is returned.

Default arguments

A default argument is a value that is automatically passed as function argument is no argument value is provided in the function call.

- Default arguments should be placed in the function prototype. If a prototype is not provided., then the default arguments must be placed in the function heading.
- In the function argument list, mandatory arguments must precede default arguments.
- Default arguments may not be specified in both the function prototype and the heading of the function definition.

- All of a function's arguments may have default values.
- A default value may not be applied to a reference argument.

Why is this an error?

```
void funk(int arg1 = 7, int arg2);
```

A function that is prototyped like this,

```
void funk(int x = 2, int y = 4, int z = 6);
```

may be called in 4 different ways:

```
funk(1,2,3);     // first argument = 1, second = 2, third = 3

funk(1,2);       // first argument = 1, second = 2, third = 6

funk(1);         // first argument = 1, second = 4, third = 6

funk();          // first argument = 2, second = 4, third = 6
```

Arrays

An array is a multi-part variable that is stored in *contiguous* memory. Arrays are used to represent multiple occurrences of data of the same type. Use of an array allows you to avoid having to declare multiple variables of the same type. Arrays also facilitate passing multiple values to a function and performing similar operations on different values.

The *parts* of an array variable are called elements. Elements are accessed using the index operator.

Declaration

Some examples

```
int data[5];            // declare a 5 element array of int
double d[300];          // declare a 300 element array of double
char ch[2];             // declare a 2 element array of char
long double ld[25];     // declare a 25 element array of long double
Dog myPets[5];          // declare a 5 element array of Dog
```

How big is an array?

An array's size in memory is the size of the type multiplied by the number of elements in the array. For example, given

```
int data[5];
```

data's size would be 20 bytes, since the size of an int is 4 bytes. You can use the *size of* operator to determine the size of an array. So,

```
cout << sizeof(data);
```

would display 20.

Warning: you can only use the sizeof operator on a variable that is *in scope*.

The number of elements in an array is called the dimension of the array.

Initialization

The follow examples demonstrate different syntax for initializing elements of an array.

```
int arr[5] = { 2,3,5,7,11};    // assigns 2 3 5 7 11 to the array
int arr[5] = { 2,3};           // assigns 2 3 0 0 0 to the array
int arr[] = { 2,3,5,7,11};     // assigns 2 3 5 7 11 to the array
int arr[5] = {0};              // assigns 0 0 0 0 0 to the array
int arr[5] = {7};              // assigns 7 0 0 0 0 to the array
```

An array's address

An array's address in memory is its name. For example, if you declare an array as

```
double z[5000];
```

Then z represents the address of z's location in memory.

This is an important concept that will be used to pass an array to a function.

How is an array stored in memory?

Because an array is stored in contiguous memory, you already know the address of each element of the array. The first element is stored at the address that is the array's name. For example, if you have an int array,

int x[10];

Then, you can assume that the first element is stored at the address referenced by x. The second element has an address that is 4 bytes larger than x, since int variable are stored in 4 bytes of memory. The following example illustrates this idea.

Example 1-9 – How is an array stored in memory?

```
#include <iostream>
using namespace std;

int main()
{
    int array[5];

    // What is the address of the array?
    cout << array << endl;      // using array name
     cout << &array << endl;     // using the address of operator
     cout << &array[0] << endl;  // using address of first element

     // What is the address of the array?
     for (int i = 0; i < 5; i++)
         cout << &array[i] << "   ";
     cout << endl;

    // Display addresses as a decimal number
    cout << reinterpret_cast<long long>(array) << endl;
    for (int i = 0; i < 5; i++)
        cout << reinterpret_cast<long long>(&array[i]) << "   ";
    cout << endl;
}
```

****** Output - Code::Blocks ******

```
0x6dfee4
0x6dfee4
0x6dfee4
0x6dfee4   0x6dfee8   0x6dfeec   0x6dfef0   0x6dfef4
7208676
7208676   7208680   7208684   7208688   7208692
```

****** Output - MS Visual Studio 2017 ******

```
010FFD4C
010FFD4C
010FFD4C
010FFD4C   010FFD50   010FFD54   010FFD58   010FFD5C
17825100
17825100   17825104   17825108   17825112   17825116
```

****** Output - Linux g++ ******

```
0x7fff388f9530
0x7fff388f9530
0x7fff388f9530
0x7fff388f9530   0x7fff388f9534   0x7fff388f9538   0x7fff388f953c
0x7fff388f9540
140734142321968
140734142321968   140734142321972   140734142321976   140734142321980
140734142321984
```

Indexing

The *index operator*, **[]**, is used to access individual elements of an array. For example, with

```
int data[5];
```

you can access the first element using

```
data[0]
```

the second element as

```
data[1]
```

and the last element in the array as

```
data[4]
```

The first element of an array is always element 0. The last element is always one less than the number of elements in the array.

The bounds of an array are permissible integers that may be used to index an array. For example, with the data array above, the bounds are 0 through 4. **Indexing an array outside the bounds of the array should be avoided**. It may not result in a compilation error, but it may result in unpredictable behavior. This problem is referred to as a segmentation fault (or a segmentation violation).

The index operator may be used to access an r-value or an l-value. These terms are used to identify how a value is to be used. An r-value is a value that is read from memory and an l-value on one that is (or can be) written into memory. For example,

```
int a[3];        // declare a 3 element array of int

a[0] = 7;        // assign 7 to the first element of the array

cout << a[0];    // print the value in the first element of the array
```

In the statement,

```
a[0] = 7;
```

the index operator returns an l-value.

In the statement,

```
cout << a[0];
```

the index operator returns an r-value.

Traversing an array

Traversing an array are methods used to access array elements in index order. This is usually done using a for loop where the index variable of the for loop is used to index array elements. For example,

```
// print each element of an array
for(int i = 0; i < sizeofArray; i++)
{
   cout << array[i] << endl;
}
```

Passing an array to a function

Because it is common to use a function to process an array, it is necessary to pass the array to a function. To accomplish this, the pass is made by using the name of the array in the calling function. That is because an array's name is its address. The called function refers to the array by using an address. For example,

```
…
int array[50];

// function call, pass the array to the function
populate_array(array);
…


// function assigns random int to each element of the array
void populate_array(int a[])
{
   for (int i = 0; i < 50; i++)
   {
      a[i] = rand()%100+1;
   }
}
```

Notice in the code above the called function references the array as **`int a[]`**. This syntax identifies the argument as an int address. Because only the array's address is passed to the function, the function does not know how many elements are contained in the array. In this example the number of elements is *hard-coded* as 50 in the for loop. Instead of *hard-coding* the number of elements, it is common to pass that number as a function argument.

It is also inappropriate to use **`int a[50]`** as a function argument. Placing 50 inside the parentheses is meaningless, since the compiler only sees the argument as an address and not the actual size of the array. It is also common to use *pointer notation* in the function heading to refer to the array address, like this:

```
void populate_array(int* a)
…
```

`int* a` and `int a[]` mean the same thing.

The size of an array passed to another function

When an array is passed to another function, it is passed as an address (pointer). The receiving function does not see the number of elements contained in the passed array. For that reason, it is common to pass the array size as another function argument along with the array.

```
…
int main()
{
    int array[50];
    cout << sizeof(array) << endl;   ➔ 200
    funk(array);
}

void funk(int* a)
{
    cout << sizeof(a) << endl;    ➔ 8 (prints 4 on a 32 bit compiler)
}
```

Passing an array to a function as a const

It is common to use a function to only read values from an array and not change them. To accomplish this, pass the array as a const.

```
…
int main()
{
    int array[50];
    funk1(array);
    funk2(array);
    funk3(array);
}

void funk1(int* a)
{
    a[0] = 9;    ➔ OK
}

void funk2(const int* a)
{
    a[0] = 9;    ➔ compile error
}

void funk3(const int* a)
{
    cout <<a[0];    ➔ OK
}
```

Passing an array element to a function

```
…
int main()
{
    int array[50];
    funk1(array[0]);   // pass the first element to funk1
    funk2(array[0]);   // pass the first element to funk2
}

void funk1(int a)   // argument passed by value
{
    a = 9;
}

void funk2(int& a)   // argument passed by reference
{
    a = 9;
}
```

Swapping array elements

Two swap two array elements with a function, pass the two elements by reference.

```
…
int main()
{
    int array[50];
…
    swap(array[0],array[49]);  // swap first and last element of the array
}

void swap(int& a, int& b)    // arguments are passed by reference
{
    int temp = a;
    a = b;
    b = temp;
}
```

Sorting and Searching an Array

Bubble Sort

The bubble sort is a commonly used algorithm, easy to write, but not usually the most efficient. The algorithm works by repeatedly stepping through the array, comparing adjacent items and swapping them if they are out of order. The name of this algorithm comes from the logic that forces the smallest values to “bubble” to the top, or to the bottom if you want a descending sort.

The basic logic looks like this:

```
void sort (int a[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size-1; j++)
        {
            if (a[j] > a[j+1])    // use < for a descending sort
            {
                swap(a[j],a[j+1]);
            }
        }
    }
}
```

This logic is not very efficient. The following example contains some efficiency improvements.

Example 1-10 – Bubble Sort

```
#include <iostream>
using namespace std;

void print(int a[], int size);
void bubble_sort (int a[], int size);
void swap (int& a, int& b);

int main()
{
     int array[] = { 7,9,6,2,5,3 };
     int size = sizeof(array) / sizeof(int);
     bubble_sort(array, size);
     print(array, size);
     return 0;
 }


 void print (int a[], int size)
```

```
{
    int i;
    for (i = 0; i < size; i++)
    {
        cout << a[i] << '\t';
    }
    cout << endl;
}


void bubble_sort (int a[], int size)
{
    bool swapOccurred;

    do
    {
        swapOccurred = false;
        for (int i = 0; i < size-1; i++)
        {
            if (a[i] > a[i+1])
            {
                swap(a[i],a[i+1]);
                swapOccurred = true;
            }
        }
    }
    while (swapOccurred);
}


void swap (int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

****** Output ******

```
2       3       5       6       7       9
```

Explanation

Efficiency is added to this code by decreasing amount the looping and exiting when no swap occurs in a loop.

The following output shows what the array looks like after each pass through the *do-while* loop:

```
7   6   2   5   3   9
```

6	2	5	3	7	9
2	5	3	6	7	9
2	3	5	6	7	9
2	3	5	6	7	9

Further efficiency can be achieved by decreasing the length of the inner for loop, like this:

```
for (int i = 0; i < size-i-1; i++)
{
    if (a[i] > a[i+1])
    {
        swap(a[i],a[i+1]);
        swapOccurred = true;
    }
}
```

With this change, the inner loop executes only 12 times, instead of 25 times.

Selection Sort

The selection sort algorithm, is also easy to implement. The algorithm works by dividing the list into two parts, unsorted and sorted. The unsorted part is searched for the minimum value and that value is then moved to the sorted part. So, the sorted part grows by one during each iteration of the loop and the unsorted shrinks by one.

The logic looks like this:

```
void sort(int a[], int size)
{
    // minIndex is the position in the unsorted part of the minimum
    int minIndex;

    for (int i = 0; i < size - 1; i++)
    {
        // find the position of the minimum in the unsorted part
        minIndex = i;

        for (int j = i+1; j < size; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j;
            }
        }

        // swap the value of the minimum in the unsorted part with
        // the next value in the sorted part
        if(minIndex != i)   // don't swap if not necessary
        {
            swap(a[i],a[minIndex]);
        }
    }
}
```

Example 1-11 – Selection Sort

```
// Example 1-10 Selection sort

#include <iostream>
using namespace std;

void print(int a[], int size);
void selection_sort(int a[], int size);
void swap(int& a, int& b);

```

```cpp
int main()
{
    int array[] = { 7,9,6,2,5,3 };
    int size = sizeof(array) / sizeof(int);
    selection_sort(array, size);
    print(array, size);
    return 0;
}


void print (int a[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        cout << a[i] << '\t';
    }
    cout << endl;
}


void selection_sort(int a[], int size)
{
    // minIndex is the position in the unsorted part of the minimum
    int minIndex;
    for (int i = 0; i < size - 1; i++)
    {
        // find the position of the minimum in the unsorted part
        minIndex = i;
        for (int j = i+1; j < size; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j;
            }
        }

        // swap the value of the minimum in the unsorted part with
        // the next value in the sorted part

        if(minIndex != i)    // don't swap if not necessary
        {
            swap(a[i], a[minIndex]);
        }
    }
}


void swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

****** Output ******

```
2       3       5       6       7       9
```

Explanation

The following shows the array after each iteration through the outer for loop. Notice that the sorted part of the array (the left-hand side), grows by one element during each pass of the array.

```
2       9       6       7       5       3
2       3       6       7       5       9
2       3       5       7       6       9
2       3       5       6       7       9
2       3       5       6       7       9
```

In terms of efficiency, the selection sort is usually more efficient that the bubble sort. What makes this sort more efficient the that the swapping usually occurs less. This is especially true when the array size is larger and the array is mostly unsorted.

Insertion Sort

The insertion sort algorithm, is also fairly easy to implement. The algorithm works by inserting each element of the array into a sort list. This algorithm is efficient for *relatively small* arrays.

Example 1-12 – Insertion Sort

```
// Example 1-11 Insertion sort

#include <iostream>
using namespace std;

//    Function prototypes
void insertion_sort(int* list, int size);
void print(int* a, int size);

int main()
{
    int array[] = {7,9,6,2,5,3};
    int size = sizeof (array) / sizeof (int);
    insertion_sort(array, size);
    print(array, size);
    return 0;
}
```

```

void insertion_sort(int* list, int size)
{
    int i, j;       // for loop indexes
    int valueToBeInserted;
    bool found;     // flag to indicate insert position found

    for (i = 1; i < size; i++)
    {
        // The ith element will be inserted into the "sorted" list in the correct location
        found = false;
        // valueToBeInserted will hold the ith element in the "unsorted list
        valueToBeInserted = list[i];

        // find the position in sorted list for insertion
        for (j = i - 1; j >= 0 && !found;)
        {
            // if valueToBeInserted < the jth element, shift unsorted elements (to the right)
            if (valueToBeInserted < list[j])
            {
                list[j + 1] = list[j];
                j--;
            }
            // otherwise the insertion position is found
            else
            {
                found = true;
            }
        }
        // insert valueToBeInserted into its correct position in the sorted list
        list [j + 1] = valueToBeInserted;
    }
}

void print(int* a, int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        cout << a[i] << '\t';
    }
    cout << endl;
}
```

****** Output ******

```
2       3       5       6       7       9
```

Explanation

The insertion sort array is divided into two parts, sorted and unsorted. The logic begins assuming the first element of the array, list, is sorted. The loop, lines 25-49, processes each remaining element to be inserted into the sorted part of the array. The inner for loop, lines 33-46, finds the position in which to insert the target element value. While the position is not yet located, line 36, the unsorted part of the array shifts to a higher (greater index) position to fill the void of the target element position. Once the insertion position is found (line 42), the found flag is set and the inner for loop terminates. The value to be inserted in then inserted into the sorted part.

The following shows the appearance of the array after each pass through the output for loop. The bold-faced elements are in the sorted part of the list.

7	**9**	6	2	5	3
6	**7**	**9**	2	5	3
2	**6**	**7**	**9**	5	3
2	**5**	**6**	**7**	**9**	3
2	**3**	**5**	**6**	**7**	**9**

Sequential Search

A sequential search of an array involves traversing an array and checking for the existence of a target value. The logic looks like this:

```
for (int i = 0; i < size; i++)
{
    if (a[i] == target_value)
    {
        return true;
    }
}
```

The search criteria may test for equality or some another relationship to array element values or expressions involving the element values. A sequential search may return a *Boolean* value indicating success or failure of the search, the element value itself, or the index position of where the relevant value was located in the array.

Binary Search

A binary search is a search in which the array data is repeatedly split in half until the search key is found or it is determined that the search value is not present. The data must be sorted on the search key.

Example 1-13 – Binary Search

```
// Example 1-12 Binary search

#include <iostream>
#include <cstdlib>
using namespace std;

void fillArrayWithRandomNumbers(int a[], int size);
void print(int a[], int size);
void sort(int a[], int size);
void swap(int& a, int& b);
int search(int searchValue, int a[], int size);

int main()
{
    int array[100];
    int number;
    int size = sizeof(array) / sizeof(int);
    fillArrayWithRandomNumbers(array, size);
    sort(array, size);
    print(array, size);

    while (1)
    {
        cout << "What number do you want to search for (0 to exit)? ";
        cin >> number;
        if (number == 0)
            break;
        cout << search(number, array, size) << endl;
    }

    return 0;
}

void print (int a[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        cout << a[i] << '\t';
    }
    cout << endl;
}

// selection sort
void sort (int a[], int size)
{
    int minIndex;
    for (int i = 0; i < size - 1; i++)
    {
        minIndex = i;
        for (int j = i+1; j < size; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j;
            }
        }
        if(minIndex != i)
        {
            swap(a[i], a[minIndex]);
        }
    }
}

void swap (int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}


void fillArrayWithRandomNumbers(int a[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        a[i] = rand()%1000;
    }
}

// Returns array index position of searchValue
// Returns -1 if searchValue is not found
int search(int searchValue, int a[], int size)
{
    int low, high, middle;
    low = 0;
    high = size-1;

    while (low <= high)
    {
        middle = (low + high) / 2;
        if (searchValue < a[middle])
        {
            high = middle - 1;
        }
        else if (searchValue > a[middle])
99          {
100              low = middle + 1;
101          }
102          else
103          {
104              return middle;
105          }
106      }
107
108      // Return -1 if searchValue not found
109      return -1;
110  }
```

****** Sample Run ******

```
7       11      30      49      93      104     108     119     119     126
134     164     169     169     184     190     190     195     223     236
245     256     256     256     262     301     306     310     327     330
330     334     337     344     351     352     356     363     366     393
397     405     425     427     431     484     490     491     497     508
509     509     529     549     551     572     592     596     613     620
624     629     635     649     656     665     675     690     695     700
705     710     711     719     727     738     743     743     749     752
754     760     783     801     824     843     847     862     868     876
917     920     929     932     932     933     944     963     972     980
What number do you want to search for (0 to exit)? 490
46
What number do you want to search for (0 to exit)? 491
47
What number do you want to search for (0 to exit)? 7
0
What number do you want to search for (0 to exit)? 980
99
What number do you want to search for (0 to exit)? 981
-1
What number do you want to search for (0 to exit)? 0
```

Explanation

The binary search function uses 3 variables to repeatedly split the array in half. The **low** index holds the minimum index position of a *half* and the **high** holds the maximum index position of a *half*. The **middle** holds that index position that is examined during each iteration of the while loop (lines 91-106). The looping continues as long as the **searchValue** is not found (lines 94 & 98). If the **searchValue** is found (line 102), the index position, **middle**, is returned. Otherwise, if the **low** will eventually match or exceed the **high** index position and the non-exist value, **-1**, is returned.

Multidimensional Arrays

A wise man once said, "*There is no such thing as a two-dimensional (or three-dimensional) array. There are only one-dimensional arrays.*" If that is the case, then you can think of a two-dimension array as a just a one-dimension array in which each element is a one-dimension array.

Two-Dimensional Arrays

Declaration and Initialization

```
int a[3][4];     // declare a 2D array with 3 rows and 4 columns

// declare and initialize a 2D array
// first row gets 1  2  3.  Second row gets 4  5  6
int b[2][3] = {{1,2,3},{4,5,6}};
```

How is a 2D array stored in memory?

Example 2-1 – How is a 2D array stored in memory?

```
// Example 2-1 How is a 2D array stored in memory?

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // declare and initialize a two dimensional array
     int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

     // print the array elements
     for (int row = 0; row < 3; row++)
     {
         for (int col = 0; col < 4; col ++)
         {
             cout << setw(10) << a[row][col];
         }
         cout << endl;
     }
     cout << endl;

     // print the addresses of the array elements
     for (int row = 0; row < 3; row++)
     {
         for (int col = 0; col < 4; col ++)
         {
             cout << setw(10) << &(a[row][col]);
         }
```

```
        cout << endl;
    }
    cout << endl;

    // print the addresses of the array elements as decimal numbers
    for (int row = 0; row < 3; row++)
    {
        for (int col = 0; col < 4; col ++)
        {
            cout << setw(10)
                 << reinterpret_cast<long>(&(a[row][col]));
        }
        cout << endl;
    }
    cout << endl;

    // print the address of the array and of each row
    cout << "Address of a = " << &a << endl;
    cout << "Address of the first row = " << &a[0] << endl;
    cout << "Address of the second row = " << &a[1] << endl;
    cout << "Address of the third row = " << &a[2] << endl;
}
```

****** Output – Code::Blocks ******

```
         1         2         3         4
         5         6         7         8
         9        10        11        12

  0x6dfeb8  0x6dfebc  0x6dfec0  0x6dfec4
  0x6dfec8  0x6dfecc  0x6dfed0  0x6dfed4
  0x6dfed8  0x6dfedc  0x6dfee0  0x6dfee4

   7208632   7208636   7208640   7208644
   7208648   7208652   7208656   7208660
   7208664   7208668   7208672   7208676

Address of a = 0x6dfeb8
Address of the first row = 0x6dfeb8
Address of the second row = 0x6dfec8
Address of the third row = 0x6dfed8
```

2D Arrays and Functions

How do you pass a 2D array to a function?

```
{
   int array[3][4];
   …
   somefunction(array);
   …
}
…
void somefunction(int a[][4])
{
   …
}
```

Explanation

A 2D array is passed to a function using the array's name. Recall, **an array's name is its address** (in memory). The function "sees" the array argument as the address of a 1D array. In this case, the argument appears as the address of an array of 4 ints. This is important. In passing a 2D array, it is passed as the address of a 1D array. Similarly, when you want to pass a 1D array to a function, it is passed as the address of one element (type) of the array.

One more point of this code snippet, the function argument may be expressed using pointer notation as

```
void somefunction(int (*a)[4])
{
   …
}
```

The argument can be declared as a *pointer* to an array of 4 ints.

How do you pass an element of a 2D array to a function?

```
{
   int array[3][4];
   …
   // pass element from second row, third column
   somefunction(array[1][2]);
   …
}
…
void somefunction(int element)
{
   …
}
```

Explanation

Each element of the array is, in this case, an int. To access an element, you need to *index* the array twice. So, you need a function that takes an int argument. The function can also accept the int argument passed by reference (or reference to const).

How do you pass a row of a 2D array to a function?

```
{
   int array[3][4];
   …
   // pass the second row to the function
   somefunction(array[1]);
   …
}
…
void somefunction(int row[])
{
   …
}
```

Explanation

To access one row of a 2D array you need to *index* the array once. The called function sees the argument as a 1D array. From the function's perspective, it does not know that the source of its argument is a 2D array. You can also use pointer notation in the function argument, like this:

```
void somefunction(int* row)
{
   …
}
```

How do you pass a column of a 2D array to a function?

```
{
   int array[3][4];
   …
   somefunction(array,1);
   …
}
…
void somefunction(int a[][4], int col)
{
   …
}
```

Explanation

It turns out – you can't. In order to pass a column of an array, you must pass the entire array, along with the column that you are interest in. Then the called function must operate on only the targeted column.

The following example illustrates the approaches to passing arrays and parts of an array to a function.

Example 2-2 – 2D Arrays and Functions

```
// Example 2-2 - 2D Arrays and Functions

#include <iostream>
#include <iomanip>
using namespace std;

const int Cols = 4;

void print2dArray(int [][Cols], int rows);
void print1ElementOf2dArray(int element);
void print1RowOf2dArray(int row[]);
void print1ColumnOf2dArray(int [][Cols], int rows, int col);

int main()
{
    const int Rows = 3;
    int a[Rows][Cols] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

    // print the array
    print2dArray(a,Rows);

    // Print the second row
    cout << "The second row" << endl;
    print1RowOf2dArray(a[1]);

    // Print the second element in the third row
    cout << "The second element in the third row" << endl;
    print1ElementOf2dArray(a[2][1]);

    // Print the third column
    cout << "The third column" << endl;
    print1ColumnOf2dArray(a, Rows, 2);

    return 0;
}

void print2dArray(int A[][Cols], int rows)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < Cols; j++)
            cout << setw(3) << A[i][j];
        cout << endl;
    }
    cout << endl;
}


void print1ElementOf2dArray(int element)
{
    cout << setw(3) << element << endl << endl;
}


void print1RowOf2dArray(int A[])
{
    for (int i = 0; i < Cols; i++)
        cout << setw(3) << A[i];
    cout << endl;
}


void print1ColumnOf2dArray(int A[][Cols], int rows, int col)
{
    for (int i = 0; i < rows; i++)
    {
        cout << setw(3) << A[i][col] << endl;
    }
    cout << endl;
}
```

****** Output ******

```
  1  2  3  4
  5  6  7  8
  9 10 11 12

The second row
  5  6  7  8
The second element in the third row
 10

The third column
  3
  7
 11
```

Is there another way?

It turns out that you can pass an entire array to another function using a reference to the array. It looks like this:

Example 2-3 – Passing an array to a function by reference

```
// Example 2-3 - Passing an array to a function by reference

#include <iostream>
#include <iomanip>
using namespace std;

const int Rows = 3;
const int Cols = 4;

 void print2dArray(int(&)[Rows][Cols]);

 int main()
 {
     int a[Rows][Cols] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

     // print the array
     print2dArray(a);
 }

 void print2dArray(int(&A)[Rows][Cols])
 {
     for (int i = 0; i < Rows; i++)
     {
         for (int j = 0; j < Cols; j++)
             cout << setw(3) << A[i][j];
         cout << endl;
     }
     cout << endl;
 }
```

****** Output ******

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

Explanation

This example illustrates passing the entire array by reference. First of all, notice the syntax representing the function argument in both the prototype and the function heading. This syntax is a little funny – the (&A) means that A is a reference to an array with 3 rows and 4 columns. Contrast this approach with the print2dArray function in the previous example. There, the array is passed as the address of a 1D array. In this example the print2dArray function sees the argument A as a 2D array.

Is this approach better than the previous example. No, not really. And in terms of efficiency, it's probably a tie. But, for two reasons you should probably use the prior approach.

1. The previous example permits more flexibility. Your array argument can have any number of rows. In the second approach the number of rows is not flexible.
2. It is important that you learn to work with addresses or pointers and that you learn to interpret the contents of the pointer.

Read data from a file into a 2D array

Example 2-4 - Read data from a file into a 2D array

The following example illustrates the process of reading data from a file into a 2D array. In this example the file shown below is read. The data from this file will be stored into a 2D **int** array and then printed back out to the console in the same format as originally stored in the file. The code addresses a potential issue - what if the file contains more or less than 50 records?

Input File: appledata.txt

```
Date    Open    High    Low    Close*    Adj Close**    Volume
Oct 03, 2018    230.05   233.47   229.78   232.07   232.07   28,563,300
Oct 02, 2018    227.25   230.00   226.63   229.28   229.28   24,788,200
Oct 01, 2018    227.95   229.42   226.35   227.26   227.26   23,600,800
Sep 28, 2018    224.79   225.84   224.02   225.74   225.74   22,929,400
Sep 27, 2018    223.82   226.44   223.54   224.95   224.95   30,181,200
Sep 26, 2018    221.00   223.75   219.76   220.42   220.42   23,984,700
Sep 25, 2018    219.75   222.82   219.70   222.19   222.19   24,554,400
Sep 24, 2018    216.82   221.26   216.63   220.79   220.79   27,693,400
Sep 21, 2018    220.78   221.36   217.29   217.66   217.66   96,246,700
Sep 20, 2018    220.24   222.28   219.15   220.03   220.03   26,608,800
…
```

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <cstdlib>
#include <cmath>
using namespace std;

const int NumRows = 50;

int getDataFromFile(int [][7], const string& filename);
void printData(int [][7], int NumRows);
int getMonthNumberFromString(string month);
string getMonthFromNumber(int num);

int main()
{
    int data[NumRows][7];
    int numNumRowsFromFile;
    string filename("appledata.txt");
    numNumRowsFromFile = getDataFromFile(data,filename);
    printData(data,numNumRowsFromFile);
}

// returns number of records read into array
int getDataFromFile(int data[][7], const string& filename)
{
    ifstream fin(filename.c_str());
    int month, day, year;
    int recordCount = 0;    // the number of records in input file
    float floatTemp;
    int intTemp;
    char commaBuffer[5];

    if (!fin)
    {
        cerr << "Unable to open file " << filename << endl;
        exit(1);
    }

    // discard headings
    string buffer;
    getline(fin,buffer);

    while (!fin.eof())
    {
        // get month
        fin >> buffer;
        if (fin.eof() || recordCount == NumRows)
            break;
        month = getMonthNumberFromString(buffer);

        // get the day of the month
        fin.get(commaBuffer, sizeof(commaBuffer),',');
        fin.get();
        day = atoi(commaBuffer);

        // get the year
        fin >> year;

        data[recordCount][0] = 10000 * year + 100 * month + day;

        // Get next 5 float columns
        for (int i = 1; i <= 5; i++)
        {
            fin >> floatTemp;
            data[recordCount][i] =
               static_cast<int>(round(floatTemp*100));
        }

        // Get volume, remove commas
        fin.getline(commaBuffer, sizeof(commaBuffer),',');
        data[recordCount][6] = 1000000 * (atoi(commaBuffer)% 100);
        fin.getline(commaBuffer, sizeof(commaBuffer),',');
        data[recordCount][6] += 1000 * atoi(commaBuffer);
        fin >> intTemp;
        data[recordCount][6] += intTemp;
        recordCount++;
    }
    return recordCount;
}

// Returns the month number for month string
int getMonthNumberFromString(string month)
{
    if (month == "Jan")
        return 1;
    if (month == "Feb")
        return 2;
    if (month == "Mar")
        return 3;
    if (month == "Apr")
        return 4;
    if (month == "May")
        return 5;
    if (month == "Jun")
        return 6;
    if (month == "Jul")
        return 7;
    if (month == "Aug")
        return 8;
    if (month == "Sep")
        return 9;
    if (month == "Oct")
        return 10;
    if (month == "Nov")
        return 11;
    if (month == "Dec")
        return 12;
    return 0;
}

// Returns the month string for the month number
string getMonthFromNumber(int num)
{
    const string month[12] =
       {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep",
        "Oct","Nov","Dec"};
    return month[num-1];
}

// Prints the array values in the original format
void printData(int data[][7], int NumRows)
{
    int monthNum;
    cout << setprecision(2) << fixed << setfill('0');
    for (int i = 0; i< NumRows; i++)
    {
        // Print date
        monthNum =   data[i][0] / 100 % 100;
        cout << getMonthFromNumber(monthNum) << ' '
             << setw(2) << data[i][0]%100
             << ", " << data[i][0] / 10000 << '\t';

        // Print 5 $ floating point columns
        for (int col = 1; col <= 5 ; col++)
            cout << data[i][col]/100.f << '\t';

        // Print volume data with commas
        cout << data[i][6]/1000000 << ',' << setw(3)
             << data[i][6]/1000 % 1000 << ','
             << setw(3) << data[i][6] % 1000 << endl;
    }
}
```

****** Output ******

```
Oct 03, 2018    230.05  233.47  229.78  232.07  232.07  28,563,300
Oct 02, 2018    227.25  230.00  226.63  229.28  229.28  24,788,200
Oct 01, 2018    227.95  229.42  226.35  227.26  227.26  23,600,800
Sep 28, 2018    224.79  225.84  224.02  225.74  225.74  22,929,400
Sep 27, 2018    223.82  226.44  223.54  224.95  224.95  30,181,200
Sep 26, 2018    221.00  223.75  219.76  220.42  220.42  23,984,700
Sep 25, 2018    219.75  222.82  219.70  222.19  222.19  24,554,400
Sep 24, 2018    216.82  221.26  216.63  220.79  220.79  27,693,400
Sep 21, 2018    220.78  221.36  217.29  217.66  217.66  96,246,700
Sep 20, 2018    220.24  222.28  219.15  220.03  220.03  26,608,800
…
```

Explanation

The file data will be stored in the int array, data (line 18). The function, getDataFromFile, reads the data from the input file, appledata.txt. (line 21 and lines 26-81).

Reading data from the file into the array

Line 43: the headings line is read from the file and ignored.
Line 48: the 3-character month abbreviation is read into the string variable, buffer.
Lines 49-50: a check is made for EOF or the number of records in the file exceeding the array row size (of 50). This prevents reading more than 50 records into the array.
Line 51: the month abbreviation string is converted to an int by the function getMonthNumberFromString.
Line 54: the day of the month is read from the file into a c-string variable, commaBuffer. The comma is skipped over.
Line 55: the blank space following the comma in the input file is read, and ignored.
Line 56: the c-string, commaBuffer, is converted to an int using the *atoi* function and stored in the int variable, day.
Line 59: the year is read from the file.
Line 61: the date (month, day, and year) is translated into an int value and stored in the array (see array data below).
Lines 64-69: the next 5 columns (contains dollar amounts) are read and stored as int values after multiplying each float value by 100.
Lines 72-77: the volume data, containing commas is read and stored in the array. The commas are removed. This volume number is read in three pieces due to the presence of the two commas. The millions value is read first as a c-string (line 72). The comma is skipped over. That value is converted to an int and multiplied by 1000000, then stored in the 7th column (index 6) in the array (line 73). Next, thousands value is read as a c-string (line 74). The comma is skipped. That value is converted to an int and multiplied by 1000, then added to the value stored in the 7th column in the array (line 75). Finally, the rest of the volume number is read (line 76) as an int and added to the value in the 7th column in the array (line 77).
Line 78: the recordCount variable is incremented. This count is the actual number of records read from the file. It will be returned.

Writing data from the array back out to the console (lines 123-143)

Line 126: output is set to display 2 decimal places and a ‘0’ fill character is specified.
Line 130: the date int value in column 0 is parsed to get the 5th and 6th numeric digits that represent the month number.
Line 131: the month number is converted and displayed to the 3-character month abbreviation by the getMonthFromNumber function. One blank space is added to the output.
Line 132: the day of the month is extracted from (last two digits of) the date int value. It is displayed along with
Line 133: a comma and a space. Finally, the year value is extracted and displayed by dividing the date (int) value by 10000.
Lines 136-137: The 5 columns of floating-point dollar amounts (2nd-6th column) are displayed by dividing each value by 100.f (float value for 100).
Line 140-141: The volume data is display by parsing the int value and inserting commas for the millions and thousands.

2D array data

```
20181003  23005  23347  22978  23207  23207  28563300
20181002  22725  23000  22663  22928  22928  24788200
20181001  22795  22942  22635  22726  22726  23600800
20180928  22479  22584  22402  22574  22574  22929400
20180927  22382  22644  22354  22495  22495  30181200
20180926  22100  22375  21976  22042  22042  23984700
20180925  21975  22282  21970  22219  22219  24554400
20180924  21682  22126  21663  22079  22079  27693400
20180921  22078  22136  21729  21766  21766  96246700
20180920  22024  22228  21915  22003  22003  26608800
…
```

Sort a 2D array by a column

Example 2-5 – Sort a 2D array by a column

The following example illustrates the process of sorting a 2D array by a column.

```
#include <iostream>
#include <iomanip>  // for setw()
#include <cstdlib>   // for rand()
using namespace std;

// Constants
const int NumRows = 5;
const int NumCols = 6;

 // Prototypes
```

```
void initializeArray(int [][NumCols]);
void printArray(int [][NumCols]);
void sortArray(int [][NumCols], int col);
void swapRow(int a[], int b[]);
void swap(int& a, int& b);


int main()
{
    int A[NumRows][NumCols];
    initializeArray(A);
    printArray(A);

    // sort on first column
    sortArray(A,0);
    cout << "Sorted on first column\n";
    printArray(A);

    // sort on third column
    sortArray(A,2);
    cout << "Sorted on third column\n";
    printArray(A);

    // sort on sixth column
    sortArray(A,5);
    cout << "Sorted on sixth column\n";
    printArray(A);
}


void initializeArray(int data[][NumCols])
{
    for (int r = 0 ; r < NumRows; r++)
        for (int c = 0; c < NumCols; c++)
            data[r][c] = rand() % 100 + 1;    // random 1 - 100
}


void printArray(int data[][NumCols])
{
    for (int r = 0 ; r < NumRows; r++)
    {
        for (int c = 0; c < NumCols; c++)
        {
            cout << setw(5) << data[r][c];
        }
        cout << endl;
    }
    cout << endl;
}

```

```
void sortArray(int data[][NumCols], int col)
{
    int i, j;
    for (i = 0; i < NumRows - 1; i++)
    {
        for (j = i+1; j < NumRows; j++)
        {
            if (data[i][col] > data[j][col])
                swapRow(data[i],data[j]);
        }
    }
}

void    swapRow(int a[], int b[])
{
    for (int i = 0; i < NumCols; i++)
    {
        swap(a[i],b[i]);
    }
}

void swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

****** Output ******

```
   42   68   35    1   70   25
   79   59   63   65    6   46
   82   28   62   92   96   43
   28   37   92    5    3   54
   93   83   22   17   19   96

Sorted on first column
   28   37   92    5    3   54
   42   68   35    1   70   25
   79   59   63   65    6   46
   82   28   62   92   96   43
   93   83   22   17   19   96

Sorted on third column
   93   83   22   17   19   96
   42   68   35    1   70   25
   82   28   62   92   96   43
   79   59   63   65    6   46
   28   37   92    5    3   54
```

```
Sorted on sixth column
   42   68   35    1   70   25
   82   28   62   92   96   43
   79   59   63   65    6   46
   28   37   92    5    3   54
   93   83   22   17   19   96
```

Explanation

Lines 21 and 41-46: the 5x6 array is initialized with random ints between 1 and 100

The sort function (lines 62-73)

The sort function uses simple bubble sort logic.
Line 62: the function arguments represent the array to be sorted and the column to sort on.
Line 69: the comparison is of the data in the desired column in two different rows. If the comparison is true,
Line 70: the entire rows are swapped.
Lines 75-81: the swapRow function swaps each element in the row.

Three-Dimensional Arrays

You can think of a 3-dimensional array as an array of two-dimensional arrays, or an array of an array of one-dimensional arrays. The indexes or dimensions of a 3D array are usually called rows, columns, and sets or pages.

3D Arrays – declaration, initialization, and functions

Example 2-6 – An easy 3D example

The following example demonstrates declaration and initialization of a 3D array, passing it to a function, and indexing the array.

```
#include <iostream>
#include <iomanip>
using namespace std;

const int NumRows = 3, NumCols = 4;

void print(int data[][NumRows][NumCols], int sets);

int main()
 {
     const int numSets = 2;
     int array[numSets][NumRows][NumCols] =
        {{{2,3,5,7},{11,13,17,19},{13,29,31,37}},
        {{41,43,47,53},{59,61,67,71},{73,79,83,89}}};
     print(array,numSets);
 }

 void print(int data[][NumRows][NumCols], int numSets)
 {
     for (int set = 0; set < numSets; ++set)
     {
         for (int row = 0; row < NumRows; ++row)
         {
             for (int col = 0; col < NumCols; ++col)
             {
                 cout << setw(5) << data[set][row][col];
             }
             cout << endl;
         }
         cout << endl;  // extra blank line after each set
     }
 }
```

****** OUTPUT ******

```
    2    3    5    7
   11   13   17   19
   13   29   31   37

   41   43   47   53
   59   61   67   71
   73   79   83   89
```

Example 2-7 – A 3D example with some calculations

This example illustrates the use of a 3-D array in which calculations are performed on the array contents. The program calculates column totals, row totals, and "page" totals.

```
#include <iostream>
#include <iomanip>
using namespace std;

const int NumRows = 3;
const int NumCols = 4;

void printArray(int[][NumRows][NumCols], int);
int sumRow(int[]);
int sumCol(int[][NumCols], int whichCol);
int sumPage(int[][NumCols]);

int main()
{
    const int NumPages = 2;

    int A[NumPages][NumRows][NumCols] =
       {{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
       {{13,14,15,16},{17,18,19,20},{21,22,23,24}}};
    printArray(A,NumPages);
}


void printArray(int array[][NumRows][NumCols], int pages)
{
    for (int p = 0; p < pages; p++)
    {
        for (int r = 0; r < NumRows; r++)
        {
            for (int c = 0; c < NumCols; c++)
            {
                cout << setw(5) << array[p][r][c];
            }
            cout << setw(8) << sumRow(array[p][r]) << endl;
        }
        cout << endl;
        for (int c = 0; c < NumCols; c++)
            cout << setw(5) << sumCol(array[p],c);
        cout << setw(8 ) << sumPage(array[p]) << endl << endl;
    }
}

int sumRow(int array[])
{
    int sum = 0;
    for (int c = 0; c < NumCols; c++)
    {
        sum += array[c];
    }
    return sum;
}

int sumCol(int array[][NumCols], int whichCol)
{
    int sum = 0;
    for (int r = 0; r < NumRows; r++)
    {
        sum += array[r][whichCol];
    }
    return sum;
}

int sumPage(int array[][NumCols])
{
    int sum = 0;
    for (int r = 0; r < NumRows; r++)
```

```
67      {
68          for (int c = 0; c < NumCols; c++)
69              sum += array[r][c];
70      }
71      return sum;
72  }
```

****** OUTPUT ******

```
    1    2    3    4       10
    5    6    7    8       26
    9   10   11   12       42

   15   18   21   24       78

   13   14   15   16       58
   17   18   19   20       74
   21   22   23   24       90

   51   54   57   60      222
```

Explanation

Lines 24-41: the printArray function displays the output shown above. The triple for loop displays each element in the array (line 32). At the end of each row, the sum of the elements in the row is displayed using a call to the sumRow function (line 34). The function argument for that call is a row in the array. That row represents one row on one page. That row represents a 1D array.
At the end of each page, column totals are displayed, and at the end of that line, the total for all elements on that page. The column totals are calculated with the sumCol function (line 38). The arguments for that function is one page of the array and the targeted column index. Notice that the array page is passed as a 2D array.
Finally, the page total (78 and 222) is calculated with the sumPage function (line 39). That function call is made using a page argument. That page is passed as a 2D array.
Appreciate that the three functions sumRow, sumCol, and sumPage are independent of a 3D array. Their arguments contain only 1D or 2D arrays.

Pointers and Arrays

A pointer is a data type that is used to hold a memory address. Pointers are associated with a type of data. For example, an int pointer, also called a pointer to int can hold (or contain) the address of an int value. You might also say that the pointer points to an int.

Pointers are used as function arguments or return types, and pointers are used to iterate through an array.

Pointer Basics

Declare and initialize a pointer

The *star* character is used in the declaration of a pointer. For example,

```
int* p;
```

You would say, “p is an int pointer” or “p is a pointer to int” or “p can hold the address of an int”. But right now, p is uninitialized.

Let’s look at a few more examples.

```
int x;
int* px = &x;  // px is a pointer to int containing the address of x

int *ps2;  // the star can be next to the type or the variable
ps2 = &x;

double* p;   // p is a pointer to a double

whatever *pw;   // pw is a pointer to a whatever

whocares* ptr;   // ptr is a whocares pointer
```

Assign a value to a pointer

```
int x = 9, y = 10, z;

int* p1;          // p1 is declared
p1 = &z;          // p1 is assigned a value

int* p2 = &x;     // p2 is declared and initialized
p2 = &y;          // p2 is assigned, or you could say p2 is moved
                  // p2 now points to y

double d = 2.54;
double* pd = &d; // p2 is declared and initialized

string s(“Have a nice day”);
string* pS;       // pS is declared (a pointer to a string)
pS = &s;          // pS is assigned the address of s
```

Dereference a pointer

Ok, you’ve got the idea that a pointer holds the address of a variable. Just like an int holds an int value, an int pointer holds a value, but its value is an address. Keep in mind that pointers are types, just like int, float, double, char, string, and long double.

You assign a value to a primitive type using the assignment operator, like

```
int q;
q = 7;
```

But there is another way to do this. Consider,

```
int* pq;
pq = &q;
```

So, pq holds the address of q, and q contains 7. Suppose you want to change q to 19. Well, of course, that is

```
q = 19;
```

But wait, we can do it this way

```
*pq = 19;
```

That star is the (unary) dereferencing operator. Also, called the indirection operator. It means, take the value that is contained in the address pointed to. You might read that line as "*star pq gets 19*".
Why would you want to use that dereferencing operator? We'll see.

Here's something else to think about – what if you want to print out q. Well, of course,

```
cout << q;
```

Well, there's another way …

```
cout << (*pq);
```

*pq means take the **value** at an address.

If you think about the two examples above where we had the expression, *pq, that expression was used in two different ways. In both cases it meant take the value at the address of q. In the first case the dereference operator was used to assign a value to q. In the second case the dereference operator was used to retrieve the value contained in q. This is often described as the * operator can return an l-value (the first case) or an r-value (the second case). These terms come from the assignment operator. You can only put l-values on the left side of an assignment operator and r-values on the right side of the assignment operator.

A Pointer as a function argument

Pointers are often used as function arguments. It may be more efficient to pass a pointer to a function than passing a variable or an object. That is because the variable or object may occupy more memory than a pointer. And usually the pointer will be dereferenced in the called function to access the *value that the pointer points to.*

Example 3-1 – A pointer as a function argument

```
#include <iostream>
#include <iomanip>
using namespace std;

void add5(int*);

int main()
{
    int x = 19;
    cout << x << endl;
    add5(&x);
    cout << x << endl;
}

void add5(int* ptr)
```

```
{
    *ptr += 5;
}
```

****** Output ******

```
19
24
```

Pointers and Arrays

How are pointers and arrays similar? How are they different?

- An array's name is it address in memory. A pointer's value is an address in memory.

Example 3-2 - Memory addresses of arrays and pointers

```
#include <iostream>
using namespace std;

int main()
{
    int array[10];                      // declare an array
    int i = 19;                         // declare an int, i
    int* ptr = &i;                      // ptr holds address of i
    cout << "array=" << array << endl;  // the address of the array
     cout <<"&i=" << &i<< endl;          // the address of the int i
     cout << "ptr=" << ptr << endl;      // contents of the pointer
     ptr = array;                        // assign the pointer
     cout << "ptr=" << ptr << endl;      // contents of the pointer
}
```

****** Output - Code::Blocks ******

```
array=0x6dfed4
&i=0x6dfed0
ptr=0x6dfed0
ptr=0x6dfed4
```

- Both pointers and arrays can be passed to a function. That function argument represents an address, or pointer.

- Both pointers and arrays can be indexed. In the case of a pointer, that may be appropriate if the pointer contains the address of an array.

Example 3-3 - Passing arrays and pointers to a function

```
#include <iostream>
using namespace std;

void printAnArray(int* ptr);
void printAnInt(int* ptr);

int main()
{
    int a = 18;
     int array[5] = {1,2,3,4,5};
     int b = 19;
     int* p = &a;

     cout << "printAnInt(p): ";
     printAnInt(p);
     cout << "printAnInt(&b): ";
     printAnInt(&b);
     cout << "printAnInt(array): ";
     printAnInt(array);
     cout << endl;
     cout << "printAnArray(array): ";
     printAnArray(array);
     cout << "printAnArray(p): ";
     printAnArray(p);
     cout << "printAnArray(&b): ";
     printAnArray(&b);
 }

 void printAnArray(int* ptr)
 {
     for (int i = 0; i < 5; i++)
         cout << ptr[i] << "  ";
     cout << endl;
 }

 void printAnInt(int* ptr)
 {
     cout << *ptr << endl;
 }
```

****** Output - Code::Blocks ******

```
printAnInt(p): 18
printAnInt(&b): 19
printAnInt(array): 1

printAnArray(array): 1  2  3  4  5
printAnArray(p): 18  7208696  4354464  7208736  7208852
printAnArray(&b): 19  1  2  3  4
```

Explanation

Lines 14-19: Three calls to the printAnInt function. This function displays one int by dereferencing and int pointer. These three calls demonstrate three different ways to pass a pointer (or address) to a function – an explicit pointer, an address, and an array's name. Notice that in the third call (line 19) the array's name is passed. In the printAnInt function the pointer is dereferenced and yields only one int.
Lines 21-26: Three calls to the printAnArray function. This function displays 5 ints (of an array) by indexing a pointer address – treating it like the address of an array. The first call (line 22) is made by passing the address of an array. The result is as expected. The second call (line 24) made by passing a pointer (to a single int). Notice in the result that the first value of a assumed array is as expected – 18, but the next 4 values look "funny". That's because those 4 values come from uninitialized memory addresses, offset from the address of a. Finally, the third call to the printAnArray function, line 26 passes the address of a single int. The result is as expected for the first value, 19. But the next four values look like the first four values of array. Why? Because you can assume that the array data is stored in memory right next to (after) the int b.

- The address held by a pointer or the address of an array can be offset by adding or subtracting an integer value from the array or pointer.

Example 3-4 - Offsetting the address of a pointer or an array

```
#include <iostream>
using namespace std;

void printAddress(int* ptr);

int main()
{
    int a = 18;
    int array[5] = {1,2,3,4,5};
    cout << "&a: " << endl;
    printAddress(&a);
    cout << "array: " << endl;
    printAddress(array);

    // offset the address of the array
    cout << "array+1=" << array+1 << "  decimal: "
         << reinterpret_cast<long>(array+1) << endl;
}

void printAddress(int* ptr)
{
    cout << "ptr=" << ptr << "  decimal: "
         << reinterpret_cast<long>(ptr) << endl;
    cout << "ptr+1=" << ptr+1 << "  decimal: "
```

```
           << reinterpret_cast<long>(ptr+1) << endl;
       cout << "ptr+3=" << ptr+3 << "  decimal: "
           << reinterpret_cast<long>(ptr+3) << endl;
       cout << "ptr-1=" << ptr-1 << "  decimal: "
           << reinterpret_cast<long>(ptr-1) << endl;
       cout << endl;
   }
```

****** Output - Code::Blocks ******

```
&a:
ptr=0x6dfeec  decimal: 7208684
ptr+1=0x6dfef0  decimal: 7208688
ptr+3=0x6dfef8  decimal: 7208696
ptr-1=0x6dfee8  decimal: 7208680

array:
ptr=0x6dfed8  decimal: 7208664
ptr+1=0x6dfedc  decimal: 7208668
ptr+3=0x6dfee4  decimal: 7208676
ptr-1=0x6dfed4  decimal: 7208660

array+1=0x6dfedc  decimal: 7208668
```

- The address held by a pointer can be changed. The address of an array cannot.

Example 3-5 - Change the address held by a pointer

```
#include <iostream>
using namespace std;

int main()
{
    int i = 7, j = 8;
    int arr1[3] = {1,2,3};
    int arr2[3] = {4,5,6};
    int* ptr;
     ptr = &i;
     cout << *ptr << endl;
     ptr = &j;
     cout << *ptr << endl;
     ptr = arr1;
     cout << *ptr << endl;
     ptr = arr2;
     cout << *ptr << endl;
     // arr1 = arr2;  // ERROR: Invalid array assignment
 }
```

****** Output ******

```
7
8
1
4
```

Explanation

Line 10: the pointer, ptr, is assigned the address of i
Line 11: the value of i is displayed by dereferencing ptr
Line 12: ptr is *moved* to the address of j
Line 13: the value of j is displayed by dereferencing ptr
Line 14: ptr is *moved* to the address of the array, arr1
Line 15: the first element of arr1 is displayed by dereferencing ptr
Line 16: ptr is *moved* to the address of the array, arr2
Line 17: the first element of arr2 is displayed by dereferencing ptr
Line 18: this line is commented out – it would result in a compile error. You cannot *move* an array line you do a pointer.

Example 3-6 - Traversing an array with pointers

```
#include <iostream>
using namespace std;

int main()
{
   int a[] = {2,3,5,7,11};
   for (int* p = a; p < a + 5; ++p)
      cout << *p << "  ";
   cout << endl;
}
```

****** Output ******

```
2  3  5  7  11
```