## Writing to and reading from files

`printf()` and `scanf()` are actually short-hand versions of more comprehensive functions, `fprintf()` and `fscanf()`.

The difference is that `fprintf()` includes a *file pointer* in its arguments to tell the program where the output should be printed. Similarly, `fscanf()` has a file pointer that tells where where the input should be obtained from.

The file pointer is a special data type used for reading and writing files. It is defined as part of `<stdio.h>`. The syntax for declaring a file pointer is

```
FILE* dataFile;      ( or FILE *dataFile; )
```

where `dataFile` is the pointer to the external information.

## stdout and stdin

`<stdio.h>` has definitions for two special file pointers, `stdout` and `stdin`. `stdout` points to the standard display device, usually your computer screen. `stdin` points to the standard input device, usually the keyboard.

These special pointers can be used with `fprintf()` and `fscanf()` to write to the screen and read from the keyboard. For example:

```
fprintf( stdout, “%lf”, 16.732);
```

will print 16.732 to the screen. Obviously, this behaves exactly like

```
printf(“%lf”, 16.732);
```

Similarly

```
fscanf( stdin, “%d”, &anInteger);
```

will read an integer typed from the keyboard and store in the integer variable `anInteger`. Clearly, this behaves identically to

```
scanf(“%d”, &anInteger);
```

There is probably not much need to use `fprintf` instead of `printf` and `fscanf` instead of `scanf`, but it is good to know about these variations.

## External files

Writing to and reading from external files is another matter though.

First, we need to realize that the common format for external data files is simple text. The data stored in the files is simply a one long string of characters.  It should be readable from any text editor.  The `fprintf` function will take care of converting an integer or a double (or any other type of variable) into a corresponding string of characters to be written into the file.  Likewise `fscanf` will take a string of characters from a file and convert them into the appropriate type of data to be stored within a variable within the program.

We will need a file pointer to point to the file that you are writing to or reading from.

To associate a file pointer to a particular file, we use the `fopen` function. This function connects the file pointer to a specific file in the computer's file system and sets the operation to either read or write.

When the program is finished reading or writing to the file, the connection should be shut down using the `fclose` function. (Note: any open file pointers will be closed automatically when the program terminates with a return value of 0. However, good programming protocol suggests that you care care of this explicitly within your program.) Here is the syntax for writing:

```
FILE* myWriteFile;

myWriteFile = fopen(“fileNameOnDisk.dat”, “w”);

   //Use fprintf() to write some stuff to the file.

fclose( mWriteFile );
```

And for reading a file:

```
FILE* myReadFile;

myReadFile = fopen(“fileNameOnDisk.dat”, “r”);

   //Use fscanf() to read some stuff from the file.

fclose( mReadFile );
```

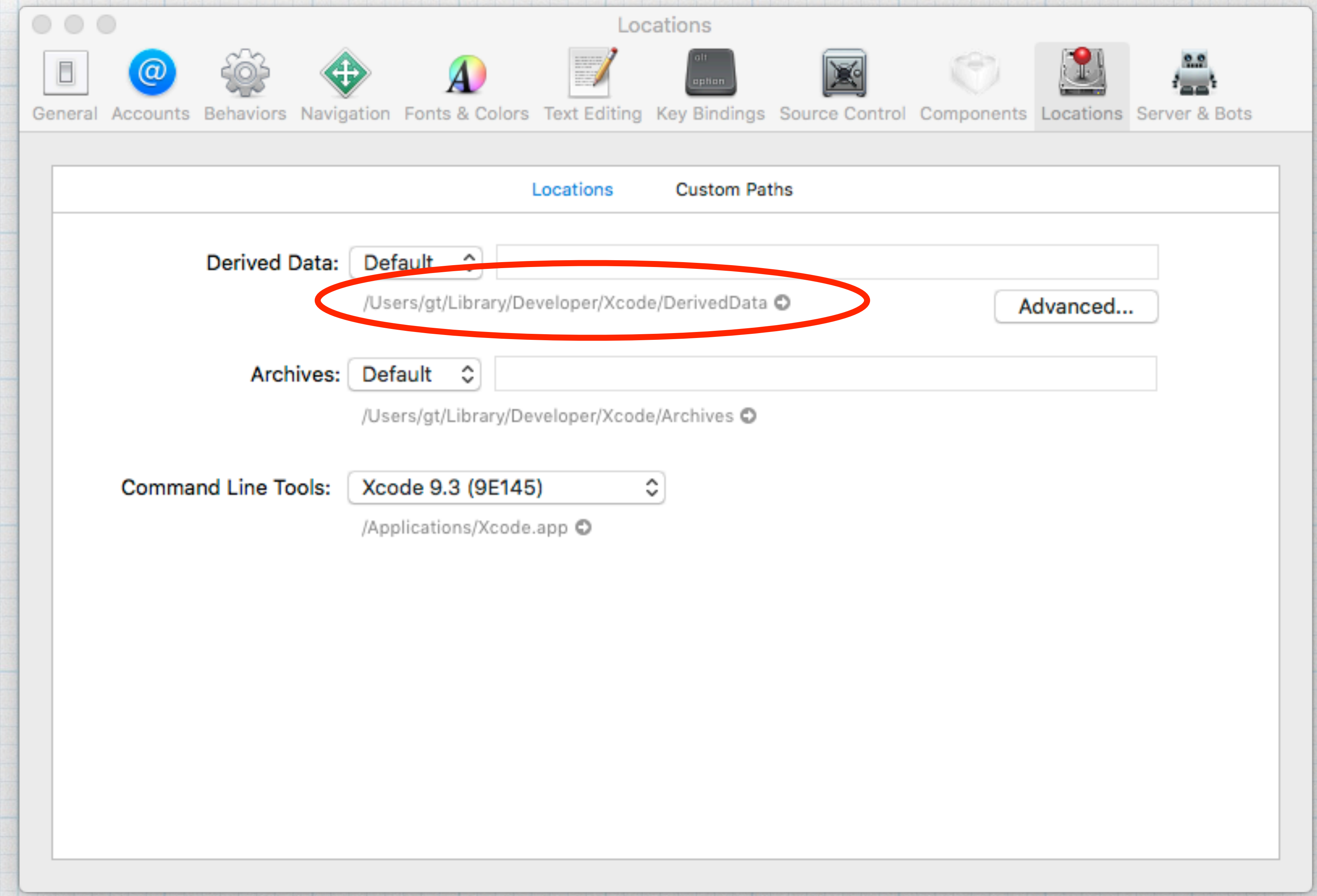
## Comments

- The file extension (like `.dat`) is optional, but it is probably a good idea.
- A commonly used file type is *comma separated variables* (`.csv`). These files can be read by many other programs, like Excel. Of course, when writing to something like a `.csv` file, it is your job to make certain that the data is arranged in a manner that is expected for that file type.
- If `fopen` fails to make the connection with a specific file, the pointer is set to `NULL`. It is always a good idea to check this before trying to do any reading or writing.
- When writing to a file, if the the file does not exist, the operating system will create a new file and make the connection to the file pointer in the program. (Finding it on the HD may be a challenge.)
- When reading a file, the file must already exist. If it doesn't, the `fopen` function reports the problem by returning a `NULL`.
- In reading from a file, you will need to know how the data is organized. You may need some mechanism to determine how much data is in the file and to know when all of the data has been read.

# Xcode - where do files go?

Go the Xcode preferences. (Under the Xcode menu.) (Or use the (shift-comma key combination.)

If you are using default settings, there will be a file path listed that tells you where the written files will go.



Derived Data: Default
/Users/gt/Library/Developer/Xcode/DerivedData

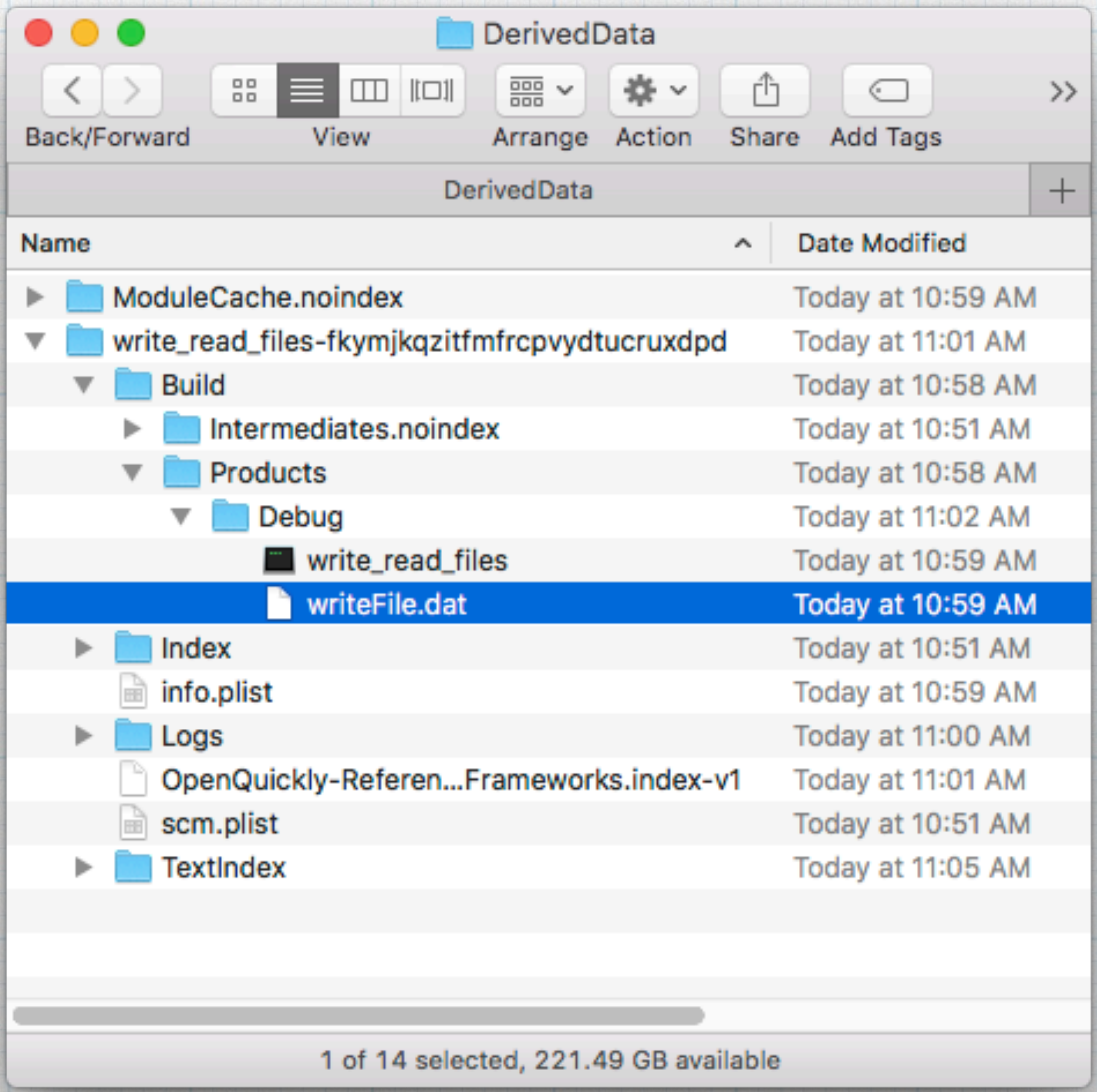
Archives: Default
/Users/gt/Library/Developer/Xcode/Archives

Command Line Tools: Xcode 9.3 (9E145)
/Applications/Xcode.app

For example, if I have project titled "write_read_files", and have run a program in that project, inside the "Derived Date Folder" another folder will be created with a title something like "write_read_files-fkymjkqzitfmfrcpvydtucruxdpd". The extra "gibberish" is appended so that Xcode can keep track of different runs of the same program.

Inside the particular project folder will be a folder called "Build". Inside that is one called "Products" and inside that is another folder "Debug", which holds two things in our case: an executable file called "write_read_files" and the file that our program created — "writeFile.dat". This is the default location to external files are written to — and where files should be read from.

The entire file path is: "Users/*account*/Library/Developer/Xcode/DerivedData/Build/Products/Debug/". Of course, "account" will be the specific account name set up on your Mac.

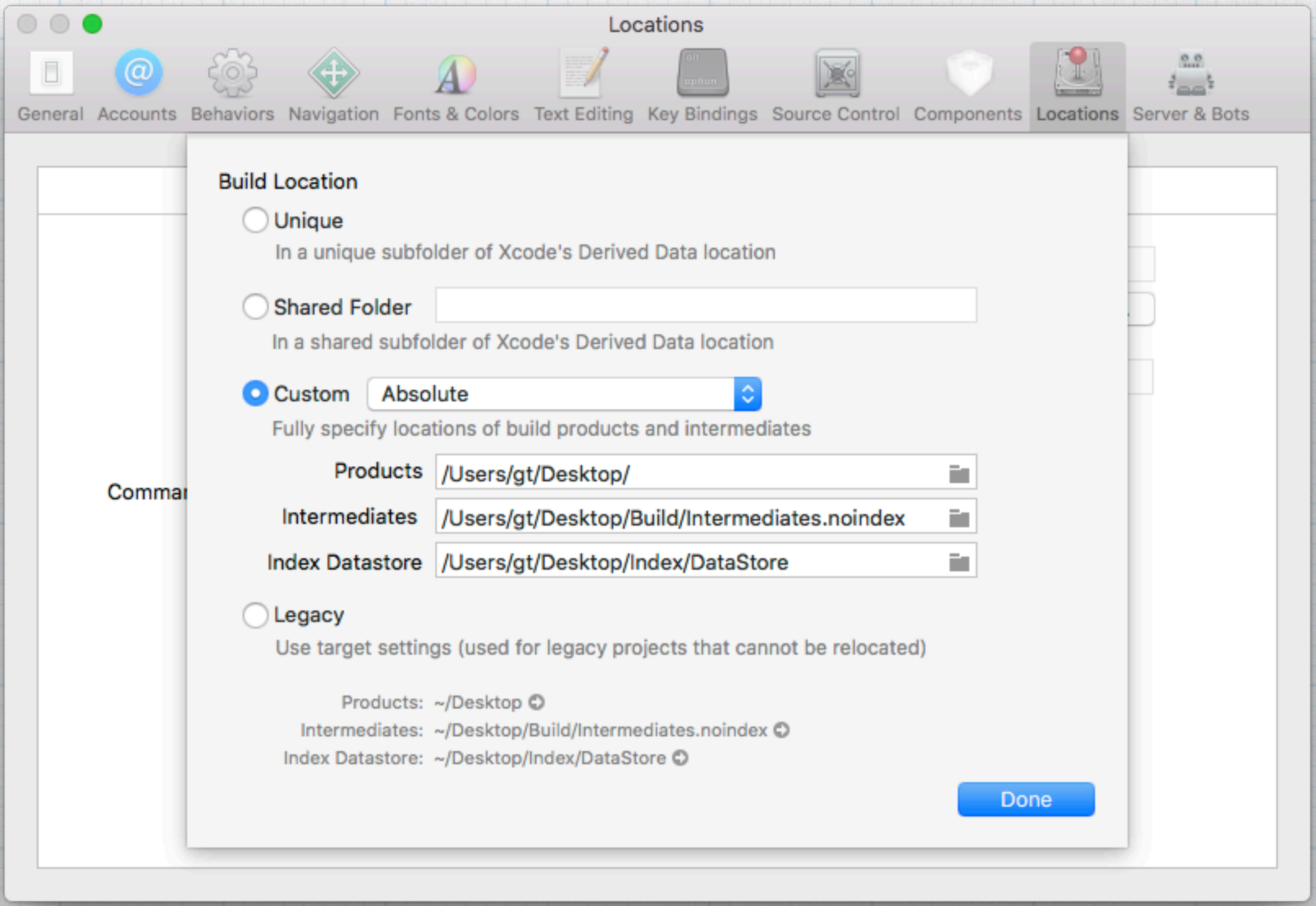


| Name | Date Modified |
|---|---|
| ▶ ModuleCache.noindex | Today at 10:59 AM |
| ▼ write_read_files-fkymjkqzitfmfrcpvydtucruxdpd | Today at 11:01 AM |
| ▼ Build | Today at 10:58 AM |
| ▶ Intermediates.noindex | Today at 10:51 AM |
| ▼ Products | Today at 10:58 AM |
| ▼ Debug | Today at 11:02 AM |
| write_read_files | Today at 10:59 AM |
| writeFile.dat | Today at 10:59 AM |
| ▶ Index | Today at 10:51 AM |
| info.plist | Today at 10:59 AM |
| ▶ Logs | Today at 11:00 AM |
| OpenQuickly-Referen...Frameworks.index-v1 | Today at 11:01 AM |
| scm.plist | Today at 10:51 AM |
| ▶ TextIndex | Today at 11:05 AM |

1 of 14 selected, 221.49 GB available

Using the defaults is OK, but rather unwieldy. You can change the location of where the files go when compiling and running a program. To change, go the Xcode preferences and choose the Locations tab at the top. Click on the "Advanced…" button.

Select the "Custom" option and choose "Absolute" from the drop down. In the "Products" line, choose a different location to write the files. It can be anywhere. Usually, when I'm working hard on programs, I have the output go to the desktop.



Build Location

Unique
In a unique subfolder of Xcode's Derived Data location

Shared Folder
In a shared subfolder of Xcode's Derived Data location

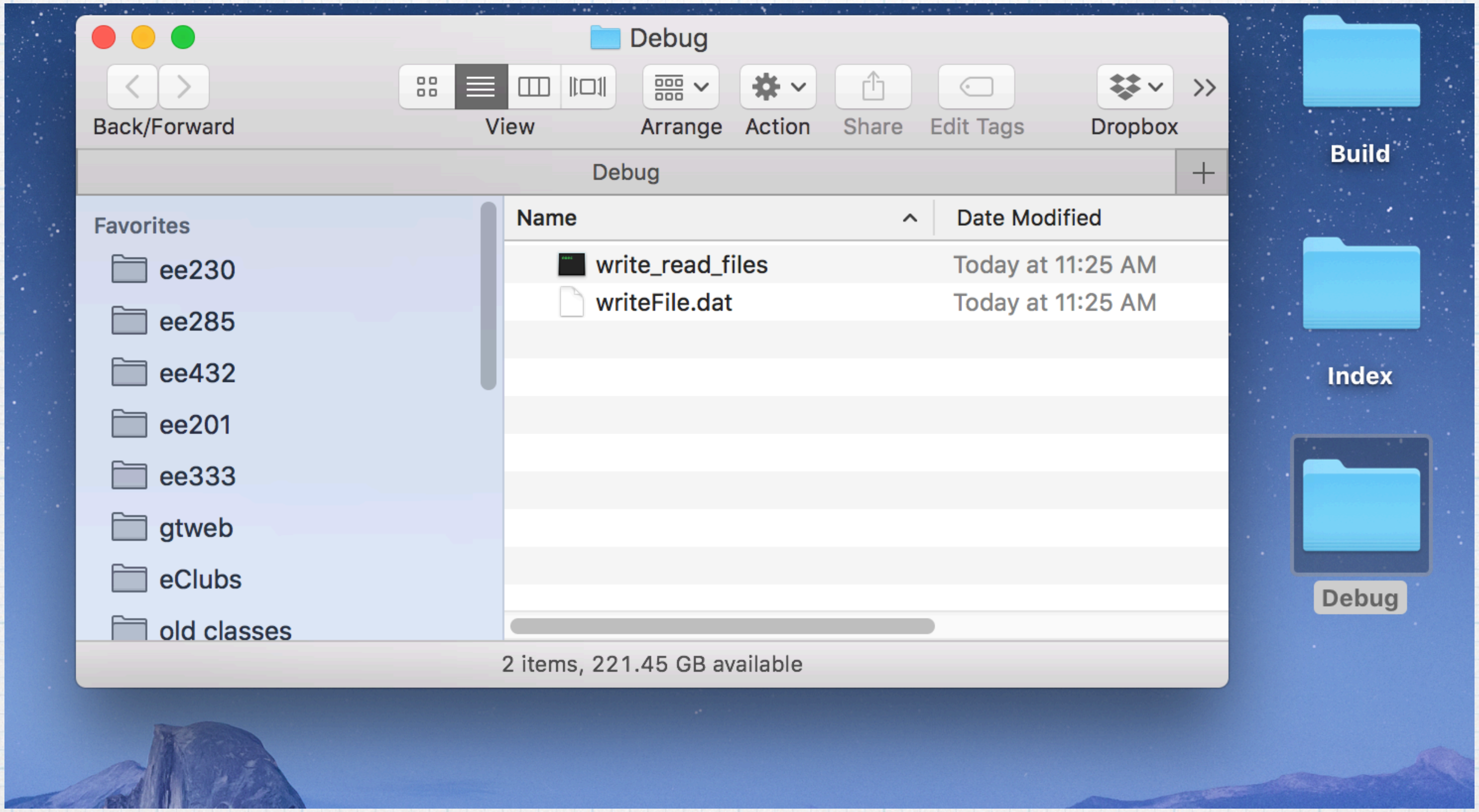
Custom: Absolute
Fully specify locations of build products and intermediates

Products: /Users/gt/Desktop/
Intermediates: /Users/gt/Desktop/Build/Intermediates.noindex
Index Datastore: /Users/gt/Desktop/Index/DataStore

Legacy
Use target settings (used for legacy projects that cannot be relocated)

Products: ~/Desktop
Intermediates: ~/Desktop/Build/Intermediates.noindex
Index Datastore: ~/Desktop/Index/DataStore

Then folders start showing up on your desktop. The Debug folder holds the files that are most immediately relevant. As you re-compile and re-run the program, the new versions are re-written into Debug.



| Name | Date Modified |
|---|---|
| write_read_files | Today at 11:25 AM |
| writeFile.dat | Today at 11:25 AM |

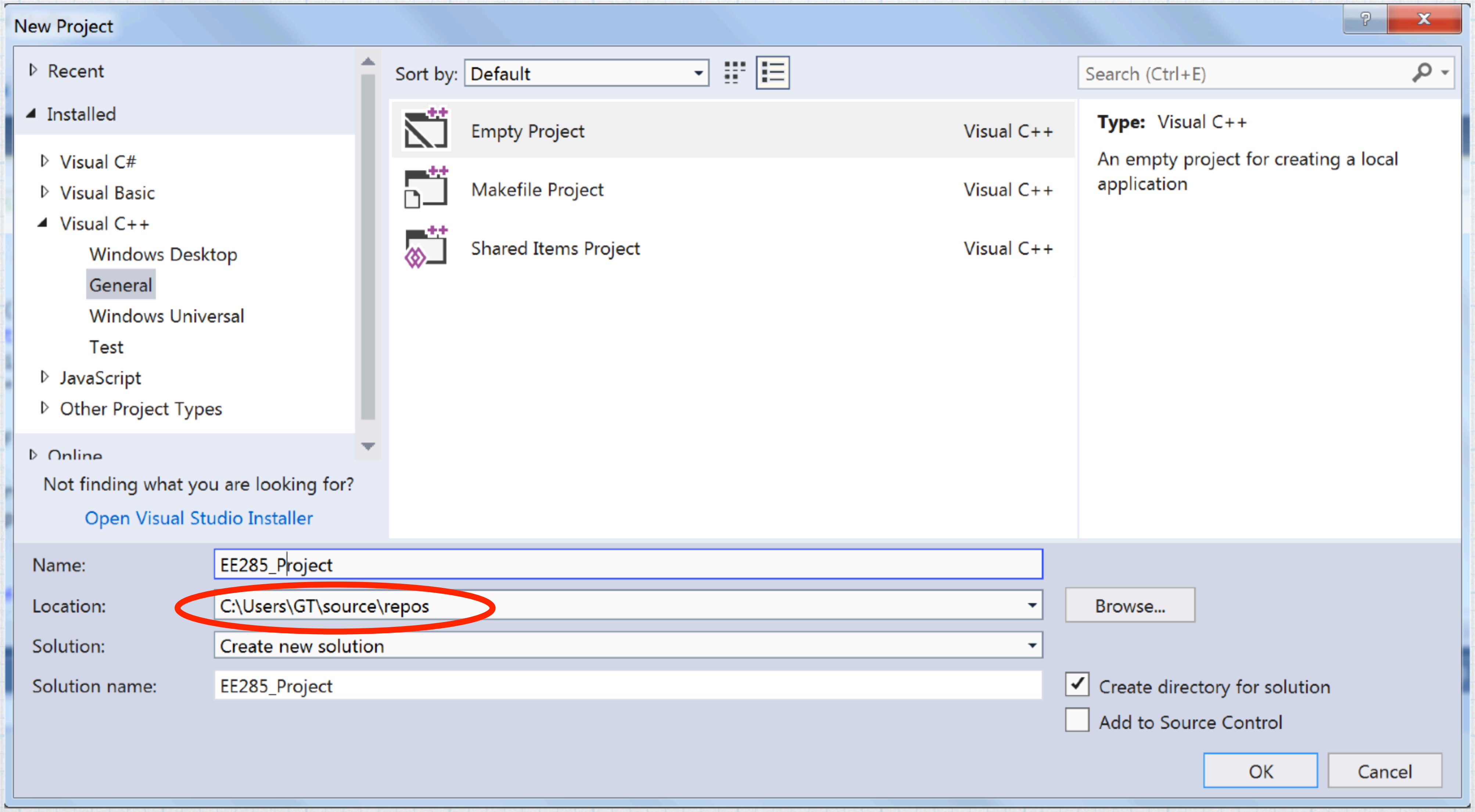
2 items, 221.45 GB available

When you are all done working, it is probably a good idea to clean up the desktop and change all settings back to the defaults.

## Visual Studio - where do files go?

VS is a little less obtuse that Xcode about file locations.

When you create a project, you are given the location of where the project files will be stored. The default location is a new folder — …\source\repos\ — in your user space on the hard drive. This is where your code files (main.c, etc) are stored and this is where files generated by your programs are written to and where files used by your programs are read from. It's pretty easy.



| | |
|---|---|
| Empty Project | Visual C++ |
| Makefile Project | Visual C++ |
| Shared Items Project | Visual C++ |

**Type:** Visual C++

An empty project for creating a local application

Name: EE285_Project

Location: C:\Users\GT\source\repos

Solution: Create new solution

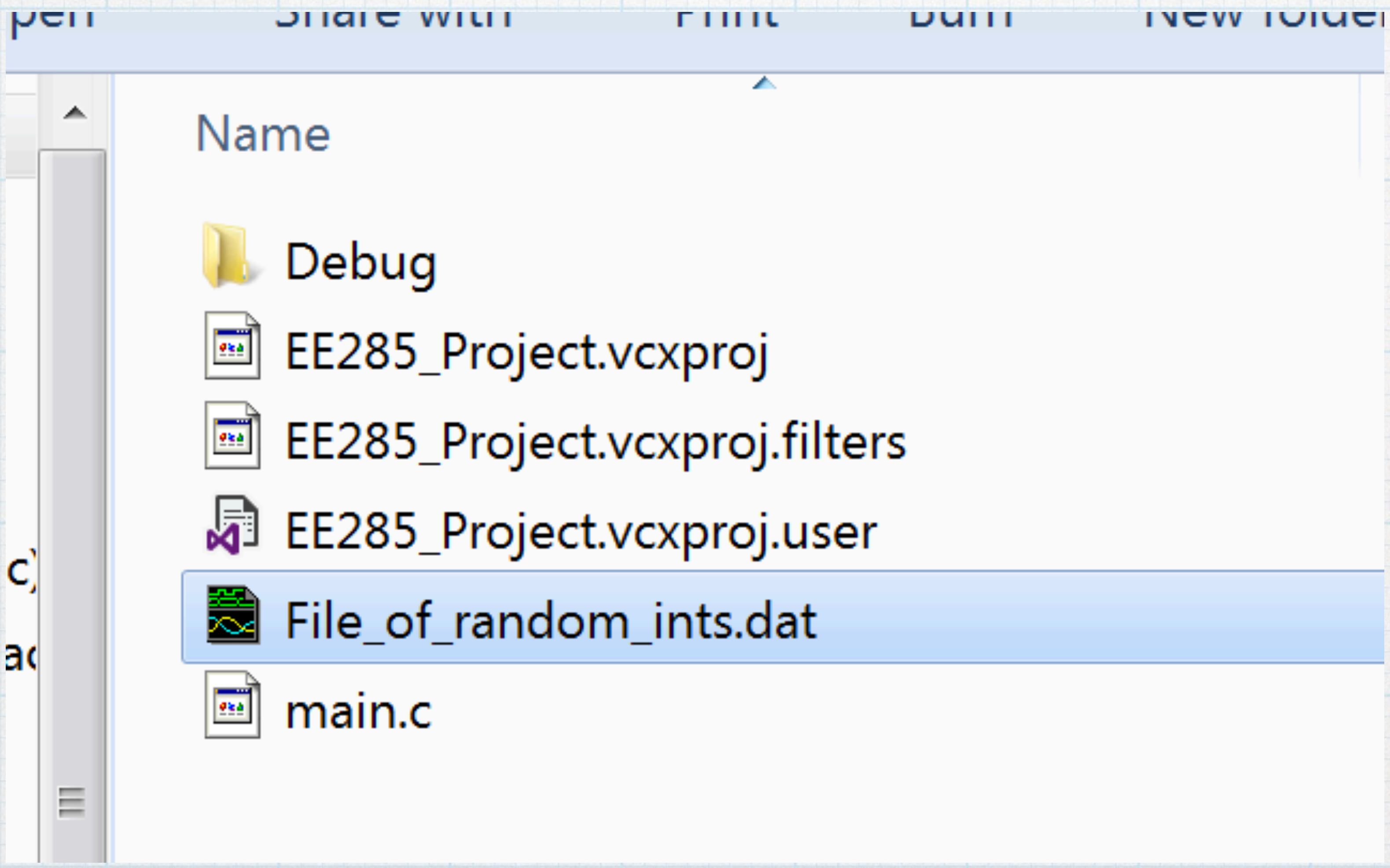
Solution name: EE285_Project

[✓] Create directory for solution

[ ] Add to Source Control

For a specific example, I have a project EE285_project (the generic project that I reuse for most EE 285 examples). In running one of the example programs from these notes — you will see it a few slides from now — a file is created and stored in this default location. The complete path is

C:\Users\GT\source\repos\EE285_Project\EE285_Project\File_of_random_ints.dat

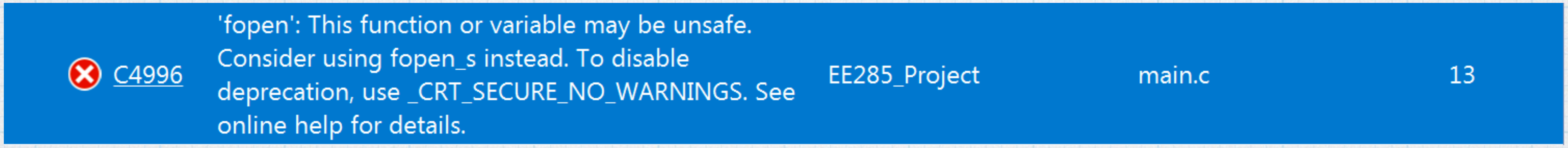


Debug
EE285_Project.vcxproj
EE285_Project.vcxproj.filters
EE285_Project.vcxproj.user
File_of_random_ints.dat
main.c

Of course, you can change the file location for a project at the time a project is started.

## Visual Studio - turning off certain "errors"

In C, writing and reading files is one of the places where errors that can be exploited by hackers can occur. MSVS "helpfully" warns you about this possibility when opening a file and provides an MS-only alternative that is supposed to be "safer": `fopen_s()`. If you use the standard `fopen()`, it is flagged as a fatal error when building the project. Of course, this is quite annoying. You have two options: use their special `fopen_s()` function or turn off the checking that flags `fopen()` as an error.



```
C4996   'fopen': This function or variable may be unsafe.   EE285_Project   main.c   13
        Consider using fopen_s instead. To disable
        deprecation, use _CRT_SECURE_NO_WARNINGS. See
        online help for details.
```

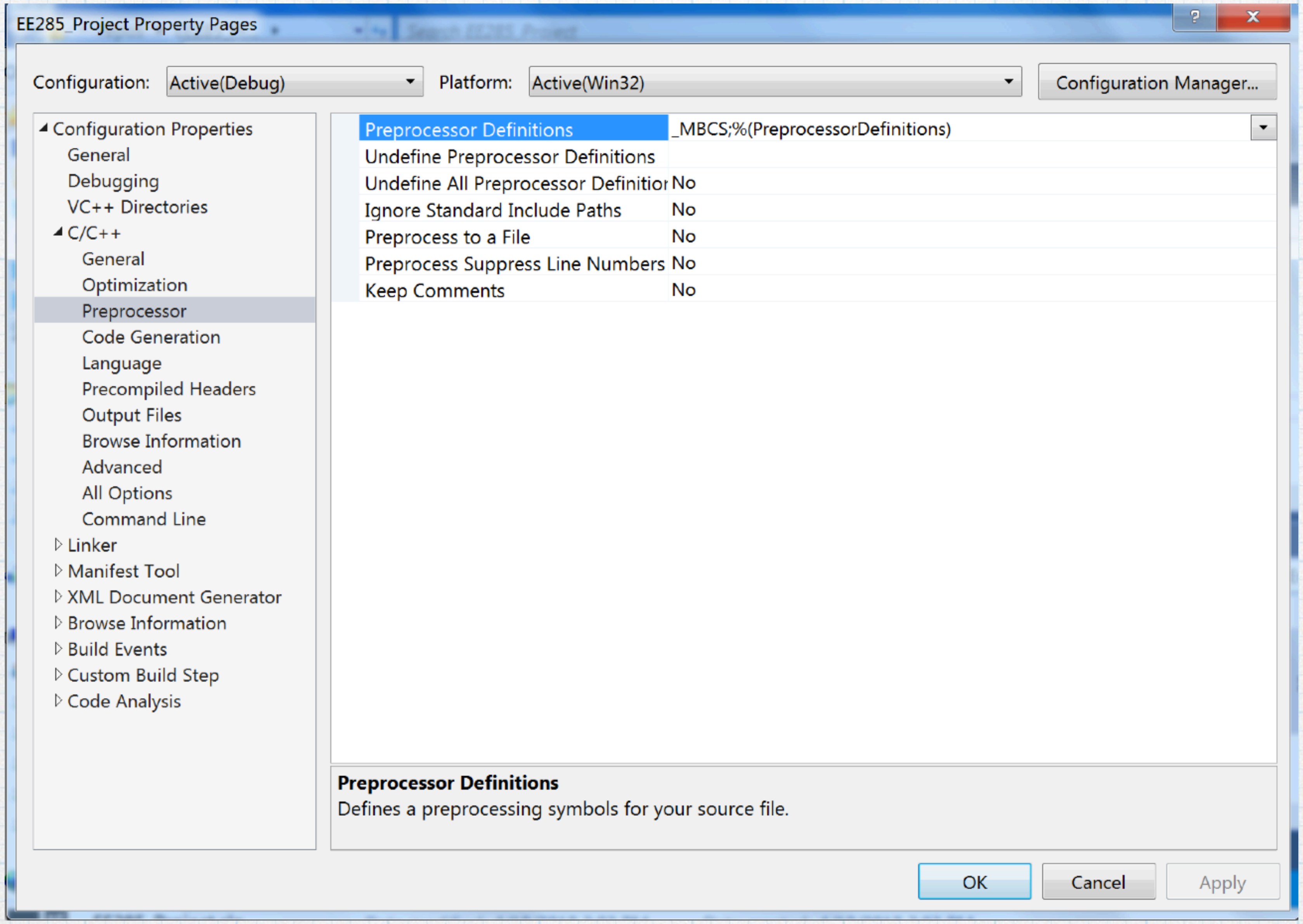
To turn off the error checking:

1. Open the project properties window (last item under the project menu or right-click on the project title on the "explorer" pane on the left.
2. Click on the C/C++ expander triangle and select the Preprocessor option.
3. In the first line — Preprocessor Definitions — change whatever is there to: _CRT_SECURE_NO_WARNINGS.
4. This will tell the compiler to stop flagging legitimate commands as errors.

See the screen shots on the next page.

Note that this change will only affect the current project. If you switch projects or start a new project, you will have to do this again. (This is one of the reasons why I tend to re-use the same project over and over and store my various program text files in a separate place.)

## This is the Preprocessor window.



EE285_Project Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

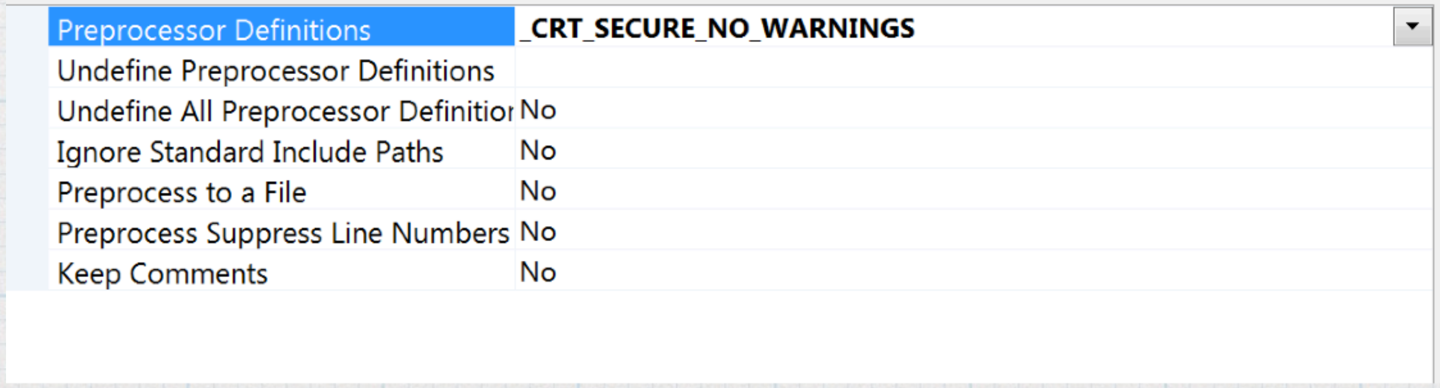
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

| | |
|---|---|
| Preprocessor Definitions | _MBCS;%(PreprocessorDefinitions) |
| Undefine Preprocessor Definitions | |
| Undefine All Preprocessor Definitior | No |
| Ignore Standard Include Paths | No |
| Preprocess to a File | No |
| Preprocess Suppress Line Numbers | No |
| Keep Comments | No |

**Preprocessor Definitions**
Defines a preprocessing symbols for your source file.

OK Cancel Apply

## Make this change…



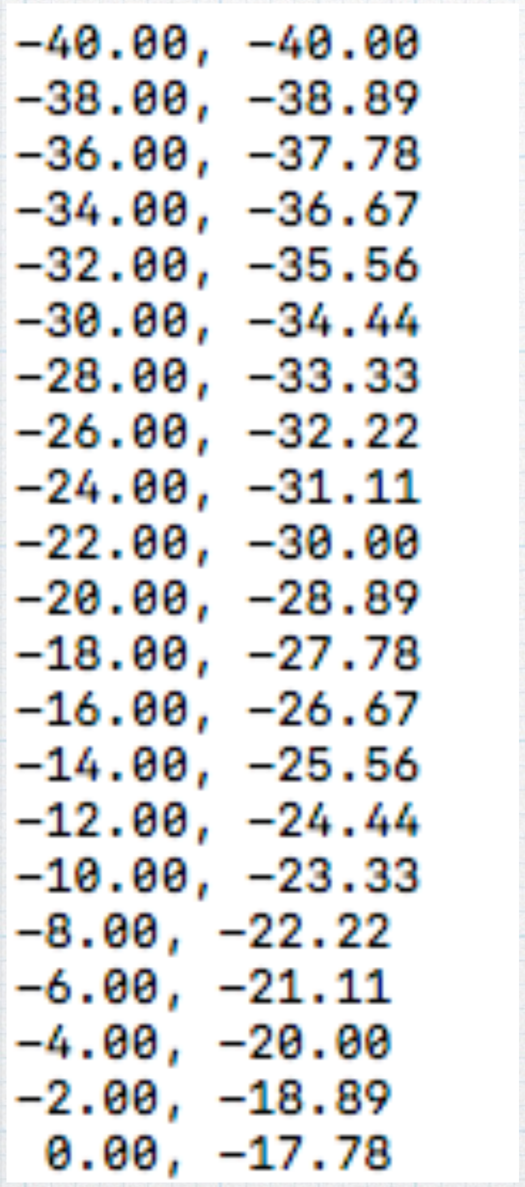
| | |
|---|---|
| Preprocessor Definitions | _CRT_SECURE_NO_WARNINGS |
| Undefine Preprocessor Definitions | |
| Undefine All Preprocessor Definitior | No |
| Ignore Standard Include Paths | No |
| Preprocess to a File | No |
| Preprocess Suppress Line Numbers | No |
| Keep Comments | No |

## Now some examples.

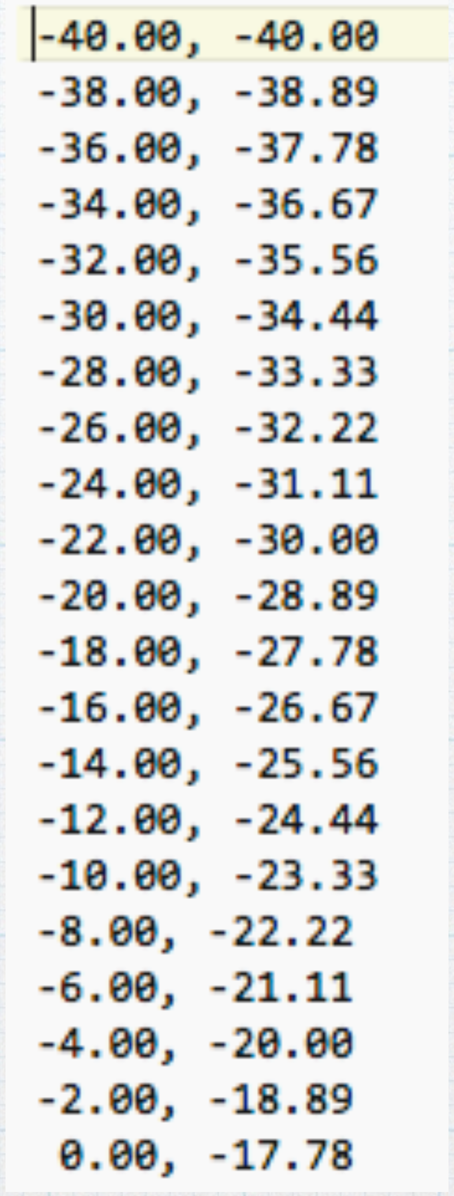
First us, let's use the old Fahrenheit to Celsius conversion program again, but this time write the results to a file as well as to the console.

```c
// EE 285 - writing files - example 1

#include <stdio.h>

int main( void ) {

   FILE* temperatureDataFile;
   double tempC, tempF;

   temperatureDataFile = fopen("temperature.csv", "w");

   if( temperatureDataFile == NULL){

      printf("Oops. Something went wrong in creating the file.");
   }
   else{

      for( tempF = -40; tempF <= 220; tempF = tempF + 2){

         tempC = 5.0 / 9.0 * (tempF - 32);
         fprintf( stdout, "%5.2lf, %5.2lf\n", tempF, tempC);
         fprintf( temperatureDataFile, "%5.2lf, %5.2lf\n", tempF, tempC);
      }

      fclose(temperatureDataFile);       //Close the file connection.
   }

   return 0;
}
```

Short chunks of the output. The left is from the console and the right is the written file as viewed using a text editor. (BBEdit on a Mac in this case.) They are identical, as we would expect.



```
-40.00, -40.00
-38.00, -38.89
-36.00, -37.78
-34.00, -36.67
-32.00, -35.56
-30.00, -34.44
-28.00, -33.33
-26.00, -32.22
-24.00, -31.11
-22.00, -30.00
-20.00, -28.89
-18.00, -27.78
-16.00, -26.67
-14.00, -25.56
-12.00, -24.44
-10.00, -23.33
-8.00, -22.22
-6.00, -21.11
-4.00, -20.00
-2.00, -18.89
 0.00, -17.78
```



```
-40.00, -40.00
-38.00, -38.89
-36.00, -37.78
-34.00, -36.67
-32.00, -35.56
-30.00, -34.44
-28.00, -33.33
-26.00, -32.22
-24.00, -31.11
-22.00, -30.00
-20.00, -28.89
-18.00, -27.78
-16.00, -26.67
-14.00, -25.56
-12.00, -24.44
-10.00, -23.33
-8.00, -22.22
-6.00, -21.11
-4.00, -20.00
-2.00, -18.89
 0.00, -17.78
```

```c
// EE 285 - writing files - example 2

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
   const int ROWS = 10, COLUMNS = 5;
   int i, j, rando;
   FILE* randomFile;

   srand( (int)time( 0 ));

   randomFile = fopen( "File_of_random_ints.dat", "w" );

   if( randomFile == NULL )
      printf("Oops. Something bad happened when creating the file.");
   else{

      for( i = 0; i < ROWS; i++){
      for( j = 0; j < COLUMNS; j++ ){
         rando = rand()%50 + 25;
         fprintf( randomFile, "%d ", rando );
      }
      fprintf( randomFile, "\n" );
      }

      fclose( randomFile );   //Close the file connection.
   }
}
```

Below are the contents of the file "File_of_random_ints.dat" that was created on the disk.  View it with a text editor program.

```
32 74 48 33 55
47 69 53 48 34
65 40 67 67 62
28 52 54 65 37
28 44 34 32 35
58 74 53 41 60
72 51 37 42 35
58 54 74 54 46
42 47 68 61 60
70 53 66 69 32
```

## Reading files – `fscanf()`

- `fscanf()` uses a formatting string just like we seen before: `%d, %5.3lf, %c, %s`, etc.
- fscanf() reads one character at a time, and then goes to the next character in the line.
- However, it reads in characters for each item *until it hits a white space or end-of-line character.* For example, if you used `fscanf( fileptr, "%s", name );` to read the string `"Ferd Baloneyhead"`, only the `"Ferd"` part would be put into `name`. However, if the string was `Ferd_Baloneyhead"`, the whole string would be read into `name`.
- You can read multiple items with one `fscanf()`. For example, to read in the first and last names above, you could use `fscanf( fileptr, "%s %s", first, last );` Then, in reading in `"Ferd Baloneyhead"`, `"Ferd"` would be put into the string `first` and `"Baloneyhead"` into the string `last`.

```
// EE 285 - reading files - example 1

#include <stdio.h>

int main(){
   const int ROWS = 10, COLUMNS = 5;
   int i, j, rando, sum = 0;
   FILE* readFile;

   readFile = fopen( "File_of_random_ints.dat", "r" );       //Open the file.

   if( readFile == NULL )
      printf("Oops. Something bad happened when creating the file.");
   else{

      for( i = 0; i < ROWS; i++){
         for( j = 0; j < COLUMNS; j++ ){
            fscanf( readFile, "%d ", &rando );
            printf( "%d ", rando );
            sum = sum + rando;
         }
         printf( "\n" );
      }
      fclose( readFile );           //Close the file connection.

      printf( "\nThe total is %d and the average is %5.2lf.\n\n", sum, sum/50.0 );
   }
}
```

Below is output seen on the screen.

```
32 74 48 33 55
47 69 53 48 34
65 40 67 67 62
28 52 54 65 37
28 44 34 32 35
58 74 53 41 60
72 51 37 42 35
58 54 74 54 46
42 47 68 61 60
70 53 66 69 32

The total is 2580 and the average is 51.60.

Program ended with exit code: 0
```

Note that the read program had to know exactly how the data was arranged in the file. Maybe that is known and will always be a constant. But what if it's not? There are couple of simple ways to "tell" the reading program what to expect. One is to put information at the top of the file. In this case, we might put the number of rows and columns at the top of the file. These can be used by the reading program to know how big the "array" is.

As an example, consider another version of the "file-writing" program. This one also creates a bunch of random numbers and writes them to a file in the form of a rectangular grid. However to really randomize things, the program will make a randomly-sized grid of random numbers — random number of column and random number of rows. (Whew! That is a lot of randomness.) The number of rows is between 1 and 20, the number of columns is between 5 and 25, and the values in the array items are between 25 and 75. The array is created and then printed to the console and to a file — random_randomness.dat. The written file also includes the dimensions of the array, which are stored at the “top” of the file. Example output is shown below.

```c
// EE 285 - writing files - example 2, version 2
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
   int rows, columns;
   int i, j, rando;
   FILE* writeFile;

   srand( (int)time( 0 ));
   rows = rand()%20 + 1;
   columns = rand()%21 + 5;

   writeFile = fopen( "more_random_ints.dat", "w" );

   if( writeFile == NULL )
      printf("Oops. Something bad happened when creating the file.");

   else{
      fprintf( writeFile, "%d %d\n", rows, columns );

      for( i = 0; i < rows; i++){
         for( j = 0; j < columns; j++ ){
            rando = rand()%50 + 25;
            fprintf( writeFile, "%d ", rando );
            printf( "%d ", rando );
         }
         fprintf( writeFile, "\n" );
         printf( "\n" );
      }

      fclose( writeFile );   //Close the file connection.
   }
}
```

This is what is printed to the console.

```
34 60 58 46 67 72 43 60 41 51 74 55 55 25 40 29 41 43 73 33 54 67 56
73 26 25 26 26 60 64 51 37 59 47 37 38 36 74 39 52 54 55 52 59 58 40
67 29 26 50 46 64 71 70 63 48 30 45 52 25 34 61 35 74 37 65 42 70 72
43 50 60 47 31 65 61 44 40 30 61 47 27 62 29 50 71 34 56 73 53 39 65
33 55 58 45 36 33 37 46 69 68 51 25 31 68 29 30 50 47 35 34 33 37 61
58 65 35 53 61 32 29 71 51 58 28 46 25 71 53 30 34 64 36 35 59 61 47
65 69 53 33 45 59 61 47 57 50 27 66 40 64 48 51 48 50 56 72 43 35 25
Program ended with exit code: 0
```

This is what is contained in the "more_random_ints.dat" that was created on the disk. Note the row and column info at the top.

```
7 23
34 60 58 46 67 72 43 60 41 51 74 55 55 25 40 29 41 43 73 33 54 67 56
73 26 25 26 26 60 64 51 37 59 47 37 38 36 74 39 52 54 55 52 59 58 40
67 29 26 50 46 64 71 70 63 48 30 45 52 25 34 61 35 74 37 65 42 70 72
43 50 60 47 31 65 61 44 40 30 61 47 27 62 29 50 71 34 56 73 53 39 65
33 55 58 45 36 33 37 46 69 68 51 25 31 68 29 30 50 47 35 34 33 37 61
58 65 35 53 61 32 29 71 51 58 28 46 25 71 53 30 34 64 36 35 59 61 47
65 69 53 33 45 59 61 47 57 50 27 66 40 64 48 51 48 50 56 72 43 35 25
```

Then we need to modify the "file reading program" to use the row/column provided within the file.

```c
// EE 285 - reading files - example 1, version 2
#include <stdio.h>

int main(){
   int columns, rows;
   int i, j, rando, sum = 0;
   double average;
   FILE* readFile;

   readFile = fopen( "more_random_ints.dat", "r" );        //Open the file.

   if( readFile == NULL )
      printf("Oops. Something bad happened when creating the file.");
   else{

      fscanf( readFile, "%d ", &rows );          //Read the number of rows.
      fscanf( readFile, "%d ", &columns );       //And columns.

      for( i = 0; i < rows; i++){
         for( j = 0; j < columns; j++ ){
            fscanf( readFile, "%d ", &rando );
            printf( "%d ", rando );
            sum = sum + rando;
         }
         printf( "\n" );
      }
      fclose( readFile );            //Close the file connection.

      average = sum/(double)(rows*columns );

      printf( "\nThe total is %d and the average is %5.2lf.\n\n", sum, average );
   }
}
```

This is what is printed to the console by the reading program.

```
34 60 58 46 67 72 43 60 41 51 74 55 55 25 40 29 41 43 73 33 54 67 56
73 26 25 26 26 60 64 51 37 59 47 37 38 36 74 39 52 54 55 52 59 58 40
67 29 26 50 46 64 71 70 63 48 30 45 52 25 34 61 35 74 37 65 42 70 72
43 50 60 47 31 65 61 44 40 30 61 47 27 62 29 50 71 34 56 73 53 39 65
33 55 58 45 36 33 37 46 69 68 51 25 31 68 29 30 50 47 35 34 33 37 61
58 65 35 53 61 32 29 71 51 58 28 46 25 71 53 30 34 64 36 35 59 61 47
65 69 53 33 45 59 61 47 57 50 27 66 40 64 48 51 48 50 56 72 43 35 25

The total is 7856 and the average is 48.80.

Program ended with exit code: 0
```

One other approach is to simply append an "end-of-file" value at the end of the "good" data. This works if we don't really care that the data is some sort "row/column" arrangement. The reading program would then use a while loop to read in data from the file until it hits the "marker". The marker would need to be something very distinct from the valid data in the file, so that there would no chance of mis-interpreting its meaning.

On the following page is a third version of the file-writing program. It adds the number -100 at the end of the file as and end-of-file marker.

```c
// EE 285 - writing files - example 2, version 3
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
   int rows, columns;
   int i, j, rando;
   FILE* writeFile;

   srand( (int)time( 0 ));
   rows = rand()%20 + 1;
   columns = rand()%21 + 5;

   writeFile = fopen( "even_more_random_ints.dat", "w" );

   if( writeFile == NULL )
      printf("Oops. Something bad happened when creating the file.");
   else{
      for( i = 0; i < rows; i++){
         for( j = 0; j < columns; j++ ){
            rando = rand()%50 + 25;
            fprintf( writeFile, "%d ", rando );
            printf( "%d ", rando );
         }
         fprintf( writeFile, "\n" );
         printf( "\n" );
      }

      fprintf( writeFile, "%d\n", -100);      //Write the "end-of-file" marker
      fclose( writeFile );                    //Close the file connection.
   }
}
```

This is what is printed to the console.

```
44 32 69 66 28 63 52 63 32 74 42 29 45 48 36 33 29 44 52 52
45 54 52 50 42 53 53 66 26 73 65 72 67 52 32 53 26 51 46 29
67 57 60 66 26 25 25 67 69 38 36 25 52 38 39 51 31 38 28 33
66 49 60 54 65 71 41 56 35 41 74 64 44 30 41 65 70 28 63 29
72 66 54 53 70 37 39 39 61 28 48 29 41 60 73 58 70 63 27 48
42 45 55 47 69 74 74 49 54 52 61 52 72 47 65 39 32 51 60 73
Program ended with exit code: 0
```

This is what is contained in the "more_random_ints.dat" that was created on the disk. Note the "-100" at the end. Again, view this with a text editor.

```
44 32 69 66 28 63 52 63 32 74 42 29 45 48 36 33 29 44 52 52
45 54 52 50 42 53 53 66 26 73 65 72 67 52 32 53 26 51 46 29
67 57 60 66 26 25 25 67 69 38 36 25 52 38 39 51 31 38 28 33
66 49 60 54 65 71 41 56 35 41 74 64 44 30 41 65 70 28 63 29
72 66 54 53 70 37 39 39 61 28 48 29 41 60 73 58 70 63 27 48
42 45 55 47 69 74 74 49 54 52 61 52 72 47 65 39 32 51 60 73
-100
```

The reading program simply reads in a long string of numbers, while watching for the -100 value — it stops reading when it gets to that point. In the program, I printed the numbers to the console. Since there the file gives no indication of the original row/column arrangement, I can print the numbers however I want. I chose to print them in rows of 10.

```c
// EE 285 - reading files - example 2, version 3
#include <stdio.h>

int main(){
   int i = -1, rando, sum = 0, readCount = 0;;
   double average;
   FILE* readFile;

   readFile = fopen( "even_more_random_ints.dat", "r" );        //Open the file.

   if( readFile == NULL )
      printf("Oops. Something bad happened when creating the file.");
   else{

      fscanf( readFile, "%d ", &rando );

      while( rando != -100 ){          //Watch for end of file marker.
         sum = sum + rando;
         readCount++;                  //Also, need to keep track of how many.
         if( i++ < 9 )                 //Print the numbers, 10 to a row.
            printf( "%d ", rando );
         else{
            printf( "\n%d ", rando );  //After 9 items, print the 10th with \n.
            i = 0;
         }
         fscanf( readFile, "%d ", &rando );
      }
      fclose( readFile );            //Close the file connection.

      average = sum/(double)readCount;

      printf( "\n\nThe total is %d and the average is %5.2lf.\n\n", sum, average
);
   }
}
```

This is what the reading program printed to the console. Again, in this case, the printing format is arbitrary.

```
44 32 69 66 28 63 52 63 32 74
42 29 45 48 36 33 29 44 52 52
45 54 52 50 42 53 53 66 26 73
65 72 67 52 32 53 26 51 46 29
67 57 60 66 26 25 25 67 69 38
36 25 52 38 39 51 31 38 28 33
66 49 60 54 65 71 41 56 35 41
74 64 44 30 41 65 70 28 63 29
72 66 54 53 70 37 39 39 61 28
48 29 41 60 73 58 70 63 27 48
42 45 55 47 69 74 74 49 54 52
61 52 72 47 65 39 32 51 60 73

The total is 6006 and the average is 50.05.

Program ended with exit code: 0
```