COSC 292 (Advanced Programming 2) Notes

# Introduction

Welcome to COSC 292 (Advanced Programming 2)! In this class, you will study a procedural programming language called C. For a description of the class and details about what will be covered, see “COSC292.25.docx”. For instructor information and the schedule for covering topics, see “COSC292 Supplemental Course Information Winter 2026.docx”.

The prerequisite for this class is COSC 183 (Intermediate Programming 2), so the assumption is that you have programming experience with languages like Java before you start this class.

## Programming environment

We will use Visual Studio 2022 as our programming environment in this class. After starting Visual Studio 2022, choose **Create a new project**:

Graphical user interface, application, email

Description automatically generated



From the ensuing dialog box, choose the language **C++** (Visual Studio doesn’t have a specific template for C files, so we will use the C++ template, since C++ is a *superset* of C – we will discuss this later) and choose an **Empty Project** and then select **Next**:

Graphical user interface, application, Teams

Description automatically generated



Enter a project name (such as **Intro**) and choose a location for the project, and then choose **Create**.

Note that in SaskPolytech labs, you MUST choose the location **C:\VisualStudioProjects**. This choice is required because C produces executable (.exe) files that would otherwise be blocked by SaskPolytech security settings. At home or elsewhere, you can choose any location that you like.

Note that **C:\VisualStudioProjects** is a local folder that is not backed up anywhere, so you should copy or move your projects to a backup location such as your OneDrive folder when you are done working on them and move them back if you need to change the project later.

Graphical user interface, text, application, email

Description automatically generated



You should see the Visual Studio environment with a **Solution Explorer** pane (on the right or the left – choose **View | Solution Explorer** if you don’t see this pane) and a **Standard** toolbar (right-click on the toolbar to get a list of available toolbars if you don’t see the **Standard** toolbar).

Graphical user interface, application

Description automatically generated

To create a C source file, right-click on **Source Files** and choose **Add > New Item…**. Visual Studio will suggest a **C++ File (.cpp)** – you can select that, but you need to change the **Name** of the file to something with a **.c** extension (such as **program.c**) to make it a C file. Choose **Add**.

It may show up in Compact View:

A screenshot of a computer

Description automatically generated

Or, if you select **Show All Templates**, it will show further options:

Graphical user interface, application

Description automatically generated



Let’s try writing a simple C program to verify that our setup works. In **program.c** (or whatever you named it), add the following code (you don’t have to enter the comments if you don’t want to):

#include <stdio.h> // add the standard input/output library

#include <stdlib.h> // add the standard library

// Create a function called main which is where the program will start

// The return type and arguments are standard, but could be replaced with "void"

int main(int argc, char\* argv[])

{

printf("Hello, world!\n"); // printf is a function to print to the console

return EXIT\_SUCCESS; // return a value of 0 (successful run)

}

Try to run the program by pressing **F5** (to start with debugging) or **Ctrl+F5** (to start without debugging) or using the buttons on the standard toolbar ( or ) or using the menu options (**Debug | Start Debugging** or **Debug | Start Without Debugging**). You should see a new window pop up with the output that looks like this:

Text

Description automatically generated

*If you can’t get your program to run, review these steps and then contact your instructor for help.*

## Procedural languages

The language that we will be studying in this class is C. C is a procedural programming language. A procedural language is one in which the tasks of a program are broken into procedures (also called subroutines or functions). To write a program, you decide which procedures and data structures you want to use and choose the best algorithm for each procedure. Support is based on functions communicating by passing arguments and returning values. Examples of procedural languages include C, Fortran, COBOL, BASIC, and Pascal.

This can be compared to object-oriented programming, where you decide what classes you need, provide a full set of operations for the class, and make commonality explicit using inheritance. You also separate what can be accessed from inside and outside the class using modifiers like private and public. Examples of object-oriented languages include Java and C#. Note that when you started writing programs in Java using only a single class and static methods, you were essentially doing procedural programming.

See also <https://en.wikipedia.org/wiki/Procedural_programming>

C is a popular language (among the top four in the TIOBE index - <https://en.wikipedia.org/wiki/TIOBE_index> - and along with C++, in the top 5 of the PYPL Popularity of Programming Language Index - <https://pypl.github.io/PYPL.html>). It is known for its low-level capabilities which allow it to create efficient programs that take advantage of the design of the machine that it is working on, making it well suited for things like operating systems and device drivers. Thus, part of what you will learn in this class is how the machine deals directly with data, seeing things that other languages may have hidden.

The name C came from its predecessors, BCPL and then B, with C being the next letter in line. C was developed in the early 1970s and was used to write the UNIX operating system. C did not have an original standard, although the book by Brian Kernighan and C’s creator Dennis Ritchie was commonly used, called K&R C from the initials of the authors. C was later standardized by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), with versions coming out in 1989/90 (C89 or C90), 1999 (C99), and later.

See also <https://en.wikipedia.org/wiki/C_(programming_language)#History>

C++ is an object-oriented *superset* of C, which means that it includes all the features of C but also includes extra object-oriented features. It was developed in the early 1980s. We will not cover C++ in this class, but you will have a basic background in it based on covering C.

See also <https://en.wikipedia.org/wiki/C%2B%2B>

# Basics

## Comments

In the original C language, all comments needed to be enclosed in /\* and \*/. As of C99, single-line comments beginning with // and extending to the end of the line were introduced.

## Pre-processor directives

C also includes pre-processor directives. Before the program is compiled, the pre-processor reads the source file looking for directives (starting with #). The pre-processor then applies the directives to the code.

#include is a pre-processor directive that says that there is some existing code that we want to use. The existing code is then substituted into the source code file. It can include definitions of functions that are in libraries that can be compiled into the final executable. Files referenced by #include can be included in <> (for files in the built-in standard libraries) or "" for local project files. #include is similar to “Using” statements in C# or “import” in Java.

Examples include:

#include <stdio.h> // add the standard input/output library

#include <stdlib.h> // add the standard library

#include "myfile.h" // add the local header file myfile.h

Another type of pre-processor directive is the define statement. The simplest version of the define statement is a substitution of a name (preferably meaningful) for a value, such as the following. Note that there is no semicolon at the end of the line – #define is not a programming language instruction.

#define PI 3.141519

## Data types

### Primitive types

C provides four primitive types: **char**, **int**, **float**, **double**.

* **char** is used for small integer values. It is required to be at least 8 bits or 1 byte in length (but could be more on some systems – on our systems, it is 8 bits or 1 byte). Note that **char** variables can contain numbers or characters. When used as a character, it can store ASCII codes (8-bit) but not Unicode values that exceed its size.
* **int** is used for larger integer values. It is required to be at least 16 bits or 2 bytes in length (but could be more on some systems – on our systems, it is 32 bits or 4 bytes).
* **float** is used for single-precision floating-point values.
* **double** is used for double-precision floating-point values.

Let’s try writing a program in a new project in a source file named something like **program.c** to illustrate these types. Start with some include directives and one define directive:

#include <stdio.h>

#include <stdlib.h>

#define PI 3.141519f // f at the end of the literal denotes it as a float literal

Then create the **main** function. Note that we will use the **printf** function, which is included in <stdio.h>, for output. **printf** is similar to the Java method **System.out.printf()** – in fact, the Java method was based on the C function. **printf** displays a message to the standard output stream, which by default is the console.

For more information about any C function and in particular its Visual Studio implementation, google the function name and Microsoft. For instance, searching for **printf microsoft** will allow you to find <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/printf-printf-l-wprintf-wprintf-l?view=msvc-170>

The **main** function will look like this:

// Program starts with the main function

int main(int argc, char\* argv[])

{

// Use printf to output to the console

printf("Hello"); // does not automatically move to a new line

printf(" world\n\n"); // \n specifies a newline

primitiveTypes();

return EXIT\_SUCCESS; // defined in stdlib.h

}

Note the call to the **primitiveTypes** function – before we run this program, we need to create it. The C compiler needs to know what **primitiveTypes** looks like before it calls it in main, so put it above the **main** function; otherwise, you will get an error that **primitiveTypes** has been redefined (in other words, the call in **main** assumes that **primitiveTypes** is different from how the function is coded). So above **main**, write:

void primitiveTypes()

{

char myCharacter = 'A'; // character literals are in single quotes

int myValue = 4276803;

float myDigits = PI; // using PI from the #define above

double myLongerDigits = 12345.6579;

// Print the value of the variables using format specifiers that start with %

printf("The value of c is %c\n", myCharacter);

printf("The value of c as an int is %d\n", myCharacter);

printf("The value of x is %d\n", myValue);

printf("The value of x as a character is %c\n", myValue);

printf("The value of x in hex is %x\n", myValue);

// Can use the format specifier to modify output

// For instance, to display 10 characters with 2 decimals

printf("The value of d is %10.2f\n", myLongerDigits);

}

Note that character literals are expressed in single quotes like 'A', so we can assign that to a **char**. We then use **printf** to print the values in different formats. We can print the **char** variable as a character using the **%c** format specifier or as a number using **%d**. We can also print the **int** value as a number, as a character (using only the low-order 8 bits), or in hexadecimal using **%x**. We can print the double in various fashions, including modifying the number of digits it prints (try printing it without the **10.2** in the format specifier to see the difference). More about **printf** format specifiers in Visual Studio can be found at <https://learn.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=msvc-170>

How big are these primitive types and thus what sizes of values can they store? In C, there are certain guarantees – for instance, a **char** must be at least 8 bits (1 byte), and an **int** must be at least 16 bits (2 bytes). But the exact sizes depend on the environment and operating system used. There is an operator called **sizeof** in C that will tell you the size of a variable or data type on the system you are using – see <https://learn.microsoft.com/en-us/cpp/c-language/sizeof-operator-c?view=msvc-170>. So let’s add the following code to **primitiveTypes** – note the use of the format specifier **%zu** for the results of **sizeof**:

// What are the sizes of each data type on our system? (may vary with system)

// The sizeof operator gives the size of a data type, struct, array, etc.

// Measured in bytes

printf("The size of a char is %zu\n", sizeof myCharacter);

printf("The size of an int is %zu\n", sizeof myValue);

printf("The size of a float is %zu\n", sizeof (float));

printf("The size of a double is %zu\n", sizeof (double));

You will find that on our systems that **char** is 1 byte (8 bits) in size; **int** (which is only guaranteed to be 16 bits or 2 bytes) is 4 bytes in size; **float** is 4 bytes; and **double** is 8 bytes.

### Modifiers

C data types can also be altered by using modifiers in front of the type. The modifiers are **short**, **long**, **unsigned**, and **signed**. They are used as follows:

* **short** can be used in front of **int** (or as a shortcut, by itself as just **short**). A short int is guaranteed to be at least 16 bits or 2 bytes but could be more.
* **long** can be used in front of **int** or **double**. The shortcut **long** by itself stands for **long int**. A long is guaranteed to be at least 32 bits or 4 bytes. Note that this differs from Java, where a **long** is defined as always being 64 bits or 8 bytes.
* Since C99, you can also use a **long long int** (or just **long long**) which requires at least 64 bits (8 bytes).
* **unsigned** and **signed** can be used in front of **char** and any variation of **int**. Signed variables can contain both positive and negative values, while unsigned variables can only contain nonnegative values (and thus have twice the range of positive numbers). For instance, unsigned types are used in the definition of bitmap image files, since things like the number of colors and file size can only be nonnegative. The variations of int are all **signed** by default unless the **unsigned** modifier is used. It is system dependent whether **char** by itself is **signed** or **unsigned**, so use the modifier in any cases where you need to be sure. For instance, **signed char** can contain values from -128 to +127, while **unsigned char** values range from 0 to 255.

Let’s write some code to try using modifiers. In **main**, add a call to the function **modifiers**:

modifiers();

Then add the function **modifiers** as follows, again above main:

void modifiers()

{

// Note that C defaults to signed values (can be + or -)

// except for char, which is system-dependent

unsigned char u = 255; // unsigned char can store values from 0 to 255

signed char s = 127; // signed char can store values from -128 to 127

// "short" and "long"

short int x; // will be 2 bytes on this platform

short y; // short form for writing a short int

long int z; // no effect on this platform

// Note that the only guarantees are that short <= int <= long

// and that short must be at least 16 bits and long must be at least 32 bits

long long big; // since C99; 8 bytes on this platform

long float f; // nonstandard - doubles the size to 8 bytes (same as double)

long double d; // no effect on this platform (same as double)

// Try printing e and c without the next two lines and then with them

// Add 1 to e and c (to see the effect on unsigned vs. signed)

u = u + 1;

s += 1;

printf("The value of u is %d\n", u);

printf("The value of s is %d\n", s);

printf("Size of short: %zu %zu\n", sizeof(x), sizeof(y));

printf("Size of long: %zu\n", sizeof(z));

printf("Size of long long: %zu\n", sizeof(big));

printf("Size of long float: %zu\n", sizeof(f));

printf("Size of long double: %zu\n", sizeof(d));

}

See also <https://en.wikipedia.org/wiki/C_data_types>

### Casting

C does implicit casting whenever it can. Unlike Java, it will implicitly cast a larger data type to a smaller data type without error (Visual Studio will show a warning), even if it results in loss of precision. You can also explicitly cast values by using the data type in parentheses (as in Java).

Try adding the following code in **main**:

castDemo();

Then above **main**, add the following **castDemo** function:

void castDemo()

{

int myInt;

float myFloat = 3.2; // 3.2 is a double literal, implicitly cast to float

float fahr = 115; // 115 is an integer literal, implicitly cast to float

float celc; // Note that uninitialized values may have random numbers

// Assign myFloat to myInt

myInt = myFloat; // implicitly cast

printf("The value of myInt is %d\n", myInt);

myInt = (int)myFloat; // explicitly cast

printf("The value of myInt is %d\n", myInt);

// Temperature conversion

printf("The temperature in Fahrenheit is %f\n", fahr);

celc = ((float)5 / 9) \* (fahr - 32); // cast to avoid integer division

// could also use 5.0 or 5.0f, 9.0 or 9.0f, etc.

printf("The temperature in Celsius is %f\n", celc);

}

### Constants

C can also have constants. Use the keyword **const** in front of the variable declaration and set the variable equal to its value on the same line as the declaration. A constant cannot be assigned a new value.

Try adding the following code in **main**:

constantsDemo();

Then above **main**, add the following **constantsDemo** function:

void constantsDemo()

{

// Note keyword const

const float pi = 3.141519f;

//pi = pi + 1; // cannot change the value – see error if uncommented

printf("The value of pi is %f\n", pi);

}

### Boolean values

C originally had no Boolean data type. Boolean comparisons were done with any value (including **char** and **float** and **double**) where 0 was false and any non-zero value was true. Such comparisons can still be used.

In C99, a **\_Bool** type was introduced. This type still stores integers, but only 0 and 1. Assigning any non-zero value to a **\_Bool** type is allowed, but it will be stored/displayed as 1.

There is also a system header file called **<stdbool.h>** which contains a #define directive that redefine **bool** as **\_Bool** (so you can use **bool** as a data type if you include this header – it will be the same as **\_Bool**). It also contains #define directives to redefine **true** as 1 and **false** as 0, so you can use **true** or **false** in place of 1 and 0 (but the values are still the integers 1 and 0 in the underlying code).

If you want to see what **stdbool.h** looks like (or any other system header file), go to **C:\Program Files\Microsoft Visual Studio\2022** and search for the filename. For instance, on my system, **stdbool.h** is at **C:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.34.31933\include** and contains the following code:

//

// stdbool.h

//

// Copyright (c) Microsoft Corporation. All rights reserved.

//

// The C Standard Library <stdbool.h> header.

//

#ifndef \_STDBOOL

#define \_STDBOOL

#define \_\_bool\_true\_false\_are\_defined 1

#ifndef \_\_cplusplus

#define bool \_Bool

#define false 0

#define true 1

#endif /\* \_\_cplusplus \*/

#endif /\* \_STDBOOL \*/

Note that all **stdbool.h** is doing is redefining **bool**, **true**, and **false** so that they will be replaced with their corresponding values when the pre-processor finds them in the code. (We will learn more about the structure of header files later.)

Let’s try out Boolean values, both as integers and using the **\_Bool** type. First, go to the top of the file and add the include directive for **stdbool.h**:

#include <stdbool.h>

Then try adding the following code in **main**:

booleanDemo();

Then above **main**, add the following **booleanDemo** function:

void booleanDemo()

{

int myValue = 3;

// boolean values are integers - 0 for false, 1 for true

printf("False comparison prints as: %d\n", (myValue > 3));

printf("True comparison prints as: %d\n", (myValue <= 3));

if (myValue) // any non-zero integer is true

{

printf("3 is non-zero, so it is considered true\n");

}

// In C99 and up, there is a Boolean type called \_Bool.

// It still stores integer values (either 0 or 1).

\_Bool myBool = 3; // any non-zero integer is still true, prints as 1

printf("Boolean myBool: %d\n", myBool);

// We can use the <stdbool.h> header file to define

// bool (which is just \_Bool redefined),

// true (which is really 1), and false (which is really 0)

bool myRedefinedBool = false;

printf("Boolean myRedefinedBool: %d\n", myRedefinedBool);

}

### Strings

Wait a minute! C doesn’t have objects or a string primitive type, so how does it handle strings?

C can store strings in an array of **char** values. You can set a variable as an array by using square brackets after the variable name (not after the type!), so **char myName[] = "Chris";** will create an array that is automatically sized to store the string **"Chris"**. What size is that? Take a guess and then let’s try the code to determine the size using the **sizeof** operator!

Try adding the following code in **main**:

stringDemo();

Then above **main**, add the following **stringDemo** function:

void stringDemo()

{

char myInitial = 'c'; // single quotes for a character literal

char myName[] = "Chris"; // Can initialize a string literal to a char []

printf("The value of myInitial is %c\n", myInitial);

printf("The value of myName is %s\n", myName); // %s for string

// can use sizeof on an array defined in the current function

printf("The size of myName is %zu\n", sizeof(myName));

}

Run the program and you will see the size of **myName** is 6 bytes. Did you guess that, or did you guess 5 for **C h r i s**? Because strings aren’t objects, there is no length property, so C has to determine the length of strings in a different way. It adds one more character at the end storing the *null terminator* – the character with the ASCII value 0 (not the digit zero!), written as **'\0'**. So the array **myName** contains **C h r i s \0**.

All strings in C must have the null terminator. For instance, when **printf** uses the **%s** format specifier for a string, it knows it has reached the end of the string to print when it encounters the null terminator. So let’s try creating a string one character at a time by creating an array for it. Add the following code to **stringDemo**:

char name2[5];

//name2 = "John"; // can't do assignment

name2[0] = 'J'; // arrays start at 0 are referenced with []

name2[1] = 'e';

name2[2] = 'a';

name2[3] = 'n';

// You must remember to null terminate your strings

name2[4] = '\0';

printf("The value of name2 is %s\n", name2);

printf("The size of name2 is %zu\n", sizeof(name2));

Note that if you create a character array first (like char name2[5];), you can’t just assign a string to it – you must assign one character at a time. Like in Java, arrays start at 0, and array elements are referenced with square brackets.

What happens if you comment out name2[4] = '\0';? You will see that a bunch of garbage characters are printed out after “Jean” because C prints until it encounters a byte with the value 0 to end the string!

What happens if you change the assignment of name2[2] = 'a'; to name2[2] = '\0';? That’s right, the string will just print as “Je”, because the null terminator is encountered, so even though there are other array elements containing characters following it, they are ignored. The size of the array is unchanged since you declared it as being size 5, and the **sizeof** operator gets the size in bytes.

### Pointers

C also has another class of data types, which are pointers. A *pointer* is a variable that stores a memory address. Because other data is stored at a memory address, a pointer “points” to that other data.

A pointer is declared with another data type followed by an asterisk, like the following:

char\* cPtr; // a pointer to a char

int\* iPtr; // a pointer to an int

We generally initialize pointers to either the address of data or NULL, a special value for the address 0 that is defined in **stdio.h** or **stdlib.h**. Any attempt to access the memory location NULL will cause an error (crashing the program), so setting pointers to NULL means that you won’t accidentally access memory that you shouldn’t access.

How big are pointers? It depends on the platform that you are working on. Try adding the following code in **main**:

pointerDemo();

Then above **main**, add the following **pointerDemo** function:

void pointerDemo()

{

char cArray[] = "This is a char array with a null terminator";

char\* cPtr = cArray; // a pointer to a char, set to the address of cArray

int\* iPtr = NULL; // a pointer to an int, set to NULL

int iValue = 42424242;

printf("Size of cArray: \t%zu\n", sizeof(cArray));

printf("Size of cPtr: \t%zu\n", sizeof(cPtr));

printf("Size of iPtr: \t%zu\n", sizeof(iPtr));

}

Run the program with the **x64** platform selected from the Standard toolbar and you will find that the size of **cArray** is 44 bytes (43 characters plus the null terminator), and the size of both pointers is 8 bytes (or 64 bits, since **x64** is a 64-bit platform and has 64-bit addresses). Remember that the pointer stores an address, so whether it is the address of a **char** or the address of an **int**, it will be the same size!

Now try with the **x86** platform selected. You will find that the size of the pointers is 4 bytes (or 32 bits, since **x86** is a 32-bit platform).

To print out a pointer, use the **"%p"** specifier. Try adding the following in **pointerDemo**:

printf("Value of cArray: \t%p\n", cArray);

// This works because cArray is the address of the start of the array

printf("Value of cPtr: \t%p\n", cPtr); // recall cPtr = cArray;

printf("Value of iPtr: \t%p\n", iPtr); // recall iPtr = NULL;

You will see that pointers print out in hexadecimal with 8 digits (in the **x86** platform) or 16 digits (in the **x64** platform). You will also see that the addresses (like **cArray**, which is the address of the array of characters) vary every time that you run the program. That is because by default, the project is set to use a randomized base address for security.

Pointers can be set using the “address of” operator **&**. For instance, we can set **iPtr = &iValue;** so that **iPtr** contains the address of the variable **iPtr**. Try adding that to **pointerDemo**:

iPtr = &iValue;

printf("New value of iPtr: \t%p\n", iPtr);

You will notice that **iPtr** now contains an address that is near the address of **cArray**. Note that we didn’t need to use the “address of” operator when initializing **cPtr = cArray;** because the name of an array is the address of the array, although you could also do **cPtr = &cArray;** without a problem.

Pointers can be “dereferenced” using the indirection operator **\*** (which is also used for multiplication, of course). Dereferencing a pointer gives back a value of the pointer’s data type. Thus, dereferencing a **char\*** will give back a **char** (one byte, which can be printed as a character or a decimal value or a hexadecimal value), while dereferencing an **int\*** will give back an **int**. Try adding the following in **pointerDemo**:

printf("cPtr dereferences as %c or %d or %x\n", \*cPtr, \*cPtr, \*cPtr);

printf("iPtr dereferences as %d or %x\n", \*iPtr, \*iPtr);

You will see that **cPtr** dereferences as **'T'** (from **cArray**’s **"This…"**) or **84** (the decimal ASCII value of the character **'T'**) or **54** (the hexadecimal ASCII value of the character **'T'**). You can verify the ASCII character values by googling “ASCII table” for a site like <https://www.asciitable.com/> or <https://www.rapidtables.com/code/text/ascii-table.html>. Note that some students think that the computer stores either a **'T'** or **84** or **0x54** (the “**0x**” prefix just means hexadecimal), but the computer really just stores an 8-bit pattern which we can interpret as either a character or a decimal value or a hexadecimal value.

Similarly, **iPtr** dereferences as **42424242** decimal or **28757b2** in hexadecimal. You can verify that these are the same number by using the Windows Calculator. Go to the three bars in the top-left corner and from the menu, choose the **Programmer** calculator. Type in the decimal value **42,424,242** and note that the **HEX** value displays as **287 57B2**. You can also click on **HEX** and enter a hexadecimal value and see the corresponding decimal conversion. Get familiar with the **Programmer** Calculator – you will find that it is useful throughout the class!

#### Pointer math

You can also add values to pointers. Adding a value to a pointer changes the address by *the size of the pointer’s data type*! So for instance, **cPtr++** would add 1 to the address in **cPtr** because the **char** data type is 1 byte in size. But **iPtr++** would add 4 to the address in **iPtr** on our systems because the **int** data type is 4 bytes on our system! Try it out with the following code in **pointerDemo**:

iPtr = (int\*)cPtr; // assign the address in cPtr to iPtr, treated as int\*

printf("cArray is: \t%p\n", cArray);

printf("cPtr + 2 is: \t%p\n", cPtr + 2);

printf("iPtr + 2 is: \t%p\n", iPtr + 2);

You will see that **cPtr + 2** is two more than **cArray** (remember to do the addition in hexadecimal!) while **iPtr + 2** is eight more than **cArray**.

Note that unlike **cPtr++**, **cPtr + 2** does not change **cPtr** – you would need to assign the new value to **cPtr** as well using **cPtr = cPtr + 2;** or **cPtr += 2**; (the same as with any other variable).

## Application: counting the length of a string

Recall that in C, strings are just null-terminated arrays of characters. They do not have a length property. If we want to find out the length of a string, we can count the number of characters until we reach the null terminator. Recall also that the null terminator is the value 0 (or **'\0'** as a character), and that zero is treated as false in C while any non-zero value is treated as true.

So a simple approach to counting the length of a string is to go through each character, counting as we go along, while we haven’t reached the null terminator. Why would we need this when we have the **sizeof** operator? The **sizeof** operator works on arrays declared in the same function, but arrays are passed to other functions by passing only the address of the array, so **sizeof** will not work in those cases (it would find the size of the address or in other words, the size of a pointer). For our simple application here, we will use arrays in the same function. Add the following code to **main:**

countLengthDemo();

Then above **main**, add the following **countLengthDemo** function:

void countLengthDemo()

{

char cArray[] = "This is a char array with a null terminator";

char\* cPtr = cArray; // a pointer to a char, set to the address of cArray

int iLength;

printf("The size of cArray is \t%zu\n", sizeof cArray);

printf("The size of cPtr is \t%zu\n", sizeof cPtr);

}

Note that the size of **cArray** is 44 (43 characters plus the null terminator), but the size of the pointer to **cArray** is only 4 or 8 bytes (depending on your solution platform).

Let’s try counting the characters by using a **while** loop:

iLength = 0;

while (cArray[iLength] != '\0')

{

iLength++;

}

printf("The length of the string is \t%d\n", iLength);

Note that the length calculated here is 43, since it does *not* include the null terminator (so it is the length we would expect in Java or C#).

As we pointed out, the null terminator is 0, and 0 is false in C. So we could shorten our loop to:

while (cArray[iLength])

{

iLength++;

}

Instead of using the array and array notation, we could “walk” the pointer through the string. Add the following code:

for (iLength = 0; \*cPtr; iLength++) // Recall \*cPtr gives us the character

{

cPtr++; // Walk the pointer to the next character

}

printf("Walking the string gives \t%d\n", iLength);

Because the **++** operator has precedence over the **\*** indirection operator (we will look at operator precedence in more detail later), we could even shorten the loop to the following code:

for (iLength = 0; \*cPtr++; iLength++); // semicolon at end for no loop body

That’s somewhat confusing, so let’s go back to the previous version and add a bit of code to show what is happening:

for (iLength = 0; \*cPtr; iLength++)

{

printf("The current value pointed to by cPtr is '%c'\n", \*cPtr);

printf("The value of cPtr is \t%p\n", cPtr);

cPtr++; // makes cPtr point to the next char

// (in other words, add 1 to the address in cPtr)

}

We will see later that the **<string.h>** library provides a function called **strlen** that calculates the string length.

# Header files

So far, we have written all our code in one file. Often, we want to separate the code into multiple files – for instance, one file for the **main** function and other test functions, and one file as a library for related functions.

In addition, so far, we have always added our functions above **main**. Why did we need to do that, and what could we do to add functions later?

Let’s create a new project with a source file called **program.c**:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

printf("Welcome to main!\n");

return EXIT\_SUCCESS;

}

We will use **void** as the parameters for **main** since we are not using the command-line arguments anyway.

Now try adding the following to **main**:

demo();

And add the following **demo** function *below* **main**:

void demo()

{

printf("Welcome to demo!\n");

}

Try compiling/running the program. You will that there are build errors. In particular, the warnings and errors will be listed as something similar to the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Severity** | **Code** | **Description** | **Project** | **Line** | **Suppression State** |
| Warning | C4013 | 'demo' undefined; assuming extern returning int | HeaderFiles | 8 |  |
| Error | C2371 | 'demo': redefinition; different basic types | HeaderFiles | 14 |  |

What is happening? The compiler reads the code from top to bottom, so it looks at **main** first and sees the call to **demo()**. It hasn’t seen **demo** yet, so it assumes that it is a function that returns an **int**. Then the compiler looks at **demo** and sees that it is a **void** function. That creates a redefinition of **demo**, so the compiler doesn’t know which definition to use, which causes an error.

One way to solve that is to create a *function prototype* before the function is created. A function prototype simply lists the return type and parameter types for a function so that the compiler knows how to call that function. So after the #include directives but before **main**, add a function prototype for **demo** as follows:

void demo();

Now you will see that the program compiles and runs successfully.

What if we put code into a separate source file? Create another source file named **pingpong.c** with the following code:

#include <stdio.h>

void ping()

{

printf("Ping\n");

}

And add the following line to the end of **demo** in **program.c**:

ping();

You will see that the program compiles and runs, but there is a warning that “'ping' is undefined; assuming extern returning int”. In some environments, that would cause the program not to link correctly (each file would compile successfully on its own but not link together to form the executable).

Further, let’s try adding a **pong** function in **pingpong.c**, and call it from **ping**, and then call **ping** from **pong**. We’ll also add a static variable in **pong** to prevent the recursion from going on forever (a static variable is one that exists only once in memory and retains its value from call to call of the function). The code in **pingpong.c** should now look like:

#include <stdio.h>

void ping()

{

printf("Ping\n");

pong(); // pong() is not defined yet - assumed returning int

}

void pong() // actual definition of pong() is here - void

{

static int iCount = 0; // only 1 copy in memory, only accessible in function

iCount++;

printf("pong\n");

if (iCount < 10)

{

ping();

}

}

If you try compiling and running, you will see it doesn’t work because **pong()** is called in **ping** and then redefined afterwards. But if we reverse the order of the functions, it won’t work either, since **ping()** is called in **pong** and then redefined afterwards.

To avoid the problems in **program.c** and in **pingpong.c**, we would need to create function prototypes for **ping** and **pong**, but then we might be adding the function prototypes in two places. To overcome these problems and create a single place where we create function prototypes (and often #define directives), we can create a *header file* and then add that to both source files.

A header file is just a file that contains function prototypes, definitions, and sometimes other common information. We have already used header files built into the system like:

* **<stdio.h>**, which tells the compiler what common input/ouput (I/O) functions look like
* **<stdlib.h>**, which has some common definitions like EXIT\_SUCCESS
* **<stdbool.h>**, which defines **bool** as a type and **true** and **false** as values

As we have seen, system header files are contained within angle brackets **<>**.

We can also create our own header files. These can be added to a source file using #include, but with the filename in quotation marks **""**. All that #include does is substitute those files into our current file, so it is like putting our prototypes at the top of the file. We can include the header file at the top of multiple source files, so we can add a header file to both **program.c** and **pingpong.c** in our example.

To create a header file, right-click on **Header Files** in the **Solution Explorer** pane and choose **Add > New Item…**. Choose a **Header File (.h)** from the list and add a filename with **.h** in the **Name:** field. For this example, let’s call our header file **pingpong.h**. You will see that Visual Studio creates the file with the default text #pragma once at the top. We will see what #pragma once means later, along with alternatives to #pragma once. For now, let’s add the function prototypes for the functions in **pingpong.c** to the header file as follows:

#define MAX\_PING\_PONGS 10 // Maximum number of ping-pong calls

// Must write function prototypes to indicate that these functions

// are coded somewhere in the project

void ping();

void pong(); // semicolons after function prototypes

*Note:* What if you clicked on **Source Files** instead of **Header Files**? Or selected **C++ File (.cpp)** instead of **Header File (.h)**? Don’t worry. If your filename has the **.h** extension, it will be treated as a header file. You may need to add #pragma once (or its alternatives), and you may want to drag the file to the **Header Files** section of the **Solution Explorer**.

Now add the #include directive at the top of both **program.c** and **pingpong.c** as follows. Also change the line in **pong** that reads if (iCount < 10) to if (iCount < MAX\_PING\_PONGS).

#include "pingpong.h"

// include local files with "" so function prototypes are known ahead of use

You should now see that the program compiles, links, and runs successfully with no warnings or errors.

What of the #pragma once directive? In big projects, one header file may be included inside another header file. To prevent repeated code, #pragma once says that the header file should be included only once by the compiler. See <https://en.wikipedia.org/wiki/Pragma_once> for details. As that article notes (as of retrieval on January 31, 2023), “#pragma once is not necessarily available in all compilers and its implementation is tricky and might not always be reliable”. On Visual Studio 2022, you can be assured that it is implemented, so it is fine to use in this course. But we should also look at alternatives to “#pragma once; in particular, *header guards* or *include guards* (see <https://en.wikipedia.org/wiki/Include_guard>).

Another (perhaps more reliable) method of ensuring the header is not included twice is to use a “header guard”. A header file with a header guard consists of the following:

* A #ifndef directive to check if some identifier is already defined
* A #define directive to define that identifier if it is not (so it will be defined the next time that it is checked by #ifndef)
* The prototypes and definitions
* A #endif directive to end the #ifndef directive

To try out a header guard, modify **pingpong.h** so it reads as follows:

// Use Header Guards to prevent these prototypes from being added

// more than once in a file

//#pragma once // will ensure that the file is only loaded once

#ifndef PINGPONG\_H // If Not Defined (will be true the first time it is encountered)

#define PINGPONG\_H // Define (makes the ifndef false every other time)

#define MAX\_PING\_PONGS 10 // Maximum number of ping-pong calls

// Must write function prototypes to indicate that these functions

// are coded somewhere in the project

void ping();

void pong(); // semicolons after function prototypes

#endif // End the ifndef

You will not see any changes in how the program runs or compiles, but it is good practice to always include header guards or #pragma once in your header files. You will see both methods used in system header files (for instance, if you look back at the code for **<stdbool.h>** that we found earlier, you will see that it uses a header guard).

# Memory

Because C works a little more directly with the machine, we should look at how it stores values in memory. Memory is byte-addressable, so we can access the value of each individual byte. Each byte has its own address. Addresses are generally written in hexadecimal. Because a byte is 8 bits, we often write byte values in hexadecimal as well, since each hexadecimal digit corresponds to 4 bits, so 2 hexadecimal digits fully represent a byte. A quick review of hexadecimal and binary conversions:

|  |  |  |
| --- | --- | --- |
| **Decimal** | **Hexadecimal** | **Binary** |
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

Memory can be visualized as an array with each memory location taking up one spot. For instance, suppose we are working with a 32-bit platform (like the **x86** platform in Visual Studio 2022). We might see a chunk of memory with addresses like the following (note that we will write the addresses going *up* from the bottom – a reason for this will be explained later):

|  |  |
| --- | --- |
| **Byte’s Address** | **Byte’s Value** |
| 0x0019FEDF | ? |
| 0x0019FEDE | ? |
| 0x0019FEDD | ? |
| 0x0019FEDC | ? |
| 0x0019FEDB | ? |
| 0x0019FEDA | ? |
| 0x0019FED9 | ? |
| 0x0019FED8 | ? |

Let’s suppose that we have the following bit of code:

char cTest = 'A'; // ASCII value 65 or 0x41 in hex

int iTest = 0x12569ADE; // 0x for hex value - 307,665,630 in decimal

The variables **cTest** and **iTest** would both have locations in memory, so we could add their aliases and values to our table of values. Note that these are just examples of where the variables could be stored in memory – where variables would be stored exactly depends on multiple factors.

|  |  |  |
| --- | --- | --- |
| **Byte’s Address** | **Alias** | **Byte’s Value** |
| 0x0019FEDF | **cTest** | 0x41 = 'A' |
| 0x0019FEDE |  | ? |
| 0x0019FEDD |  | ? |
| 0x0019FEDC |  | ? |
| 0x0019FEDB | **iTest** | 0x12569ADE |
| 0x0019FEDA |
| 0x0019FED9 |
| 0x0019FED8 |

Since **iTest** is an **int**, it takes up 4 bytes. Since we wrote the value for **iTest** in hexadecimal, we know those bytes contain the values 0x12, 0x56, 0x9A, and 0xDE. (Note that the 0x is not stored by the machine – it just indicates the number that we are writing is in hexadecimal.) But which bytes correspond to which addresses? To learn about that, we need to learn about *endianness*.

## Endianness

There are two common schemes for arranging the bytes in a variable: big-endian and little-endian.

* Big-endian: most significant byte at the smallest memory address
* Little-endian: least significant byte at the smallest memory address

The *least significant byte* is the byte that changes the value the least, while the *most significant byte* changes the value the most. For instance, given int x = 999999999;, we can plug it into the programmer calculator to see that in hexadecimal, it is 3B9AC9FF. 3B is the most significant byte, while FF is the least significant byte. To see this, try changing each byte to 0 and see the effect in the programmer calculator:

* Changing the most significant byte to 009AC9FF results in a decimal value of 10,144,255 (over 980 million different from the original value)
* Changing the least significant byte to 3B9AC900 results in a decimal value of 999,999,744 (only 255 different from the original value)

So on a big-endian machine, the value might be represented as:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Memory address (going up)** | **Value (in hex)** |
|  | 0x0012FF83 | FF |
|  | 0x0012FF82 | C9 |
|  | 0x0012FF81 | 9A |
| **int x** | 0x0012FF80 | 3B |

While on a little-endian machine, the value would be represented as:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Memory address (going up)** | **Value (in hex)** |
|  | 0x0012FF83 | 3B |
|  | 0x0012FF82 | 9A |
|  | 0x0012FF81 | C9 |
| **int x** | 0x0012FF80 | FF |

The platforms that we are working on (both **x64** and **x86**) are little-endian machines. Therefore, any further descriptions of memory that we use will be little-endian.

See <https://en.wikipedia.org/wiki/Endianness> for more details.

So returning to our previous example, **iTest** would be written as the following in little-endian notation:

|  |  |  |
| --- | --- | --- |
| **Byte’s Address** | **Alias** | **Byte’s Value** |
| 0x0019FEDF | **cTest** | 0x41 = 'A' |
| 0x0019FEDE |  | ? |
| 0x0019FEDD |  | ? |
| 0x0019FEDC |  | ? |
| 0x0019FEDB |  | 0x12 |
| 0x0019FEDA |  | 0x56 |
| 0x0019FED9 |  | 0x9A |
| 0x0019FED8 | **iTest** | 0xDE |

Note that by writing the memory addresses going up (from the lowest address at the bottom to the highest address at the top), we can read the value of **iTest** going down: 0x12569ADE.

## Memory maps

When we write out the addresses, aliases, and values of the bytes in a computer’s memory, we are creating a *memory map*. Memory maps can be useful in understanding how values are stored, how we can manipulate those values, and how pointers work.

Let’s create a new project to demonstrate memory maps. Create separate files with one file named **program.c** for the **main** function and one file named **memory.c** for a function named **memoryMap** that takes in a **float** and returns an **int**. Create an appropriate header file named **memory.h** as well. The **main** function should simply create a **float** variable and call the **memoryMap** function. Your code should look something like the following:

**program.c**:

#include <stdlib.h>

#include "memory.h"

void main(void)

{

float passingGrade = 49.5f;

memoryMap(passingGrade);

return EXIT\_SUCCESS;

}

**memory.h**:

#ifndef MEMORY\_H

#define MEMORY\_H

// Prototype the function. Note that the formal parameter name is optional.

int memoryMap(float);

#endif // !MEMORY\_H

*Note that I used a header guard for* ***memory.h*** *– you could also have used the* ***#pragma once*** *directive.*

**memory.c**:

#include <stdio.h>

#include "memory.h"

int memoryMap(float x)

{

printf("The value of the argument is %f\n", x);

return 0;

}

Let’s add some variables to the top of **memoryMap**:

static int i = 0;

int a = 1235257896;

double d = 9.0;

int\* iPtr = &a;

char c = 'a';

short s = 10815;

char cArray[] = "Hello";

Now let’s print out the addresses of the variables and some of the values of the variables as well:

printf("The value of the argument is %f\n", x);

// Address of the parameter

printf("The address of float x is \t%p\n", &x);

// Print the address of each variable

// The "Address of" operator is the overloaded &

// Note that %p is used for printing addresses in hexadecimal

printf("The address of int i is \t%p\n", &i);

printf("The address of int a is \t%p\n", &a);

printf("The value of int a in hex is \t%x\n", a); // %x says print in hex

// We can print an int as a char - prints only least significant byte

printf("The value of a as a char is \t%c\n\n", a);

printf("The address of double d is \t%p\n", &d);

printf("The address of pointer iPtr is \t%p\n", &iPtr);

printf("The value stored in iPtr is \t%p\n", iPtr);

printf("The address of char c is \t%p\n", &c);

printf("The value of c in hex is \t%x\n\n", c);

printf("The address of short s is \t%p\n", &s);

printf("The value of s in hex is \t%x\n", s);

// We can print a short as a char - prints only least significant byte

printf("The value of s as a char is \t%c\n\n", s);

// Note that the name of an array IS its address. & is not required.

printf("The address of cArray is \t%p\n", cArray);

printf("The address of cArray is \t%p\n", &cArray); // & will still work

// Print the address of the function itself.

// Note that the name of the function stores its address.

printf("The address of memoryMap is \t%p\n", memoryMap);

Try running the program. You will see that the addresses change every time that the program runs. The changing addresses are because by default, Visual Studio 2022 randomizes the base address of the program for additional security. To change this default property and make it easier for us to create a memory map, change the following setting: **Project | Properties > Configuration Properties > Linker > All Options > Randomized Base Address: No (/DYNAMICBASE:NO)**. See the following screenshot:

Graphical user interface, text, application, email

Description automatically generated



Note that if you switch from the **Debug** to the **Release** configuration or from the **x64** to **x86** platform, you will have to set this property for each configuration. Let’s start in the **Debug** configuration for **x86**.

To create a memory map for this program, download “Memory Map Template.xlsx” from the **Notes** folder. Open the file and you will see that it has a block of addresses (going up from bottom to top and from right to left). Save the file as “Memory Map Debug.xlsx” (keeping the template clean for later use).

We can change the address of the highest variable so that the block of addresses is appropriate for our program. Go to cell D1 and choose the type of the highest memory address; this will ensure that there is enough space to store all the bytes for that type. Then go to cell D2 and type in the highest address and see that the worksheet changes to reflect the new highest address.

My highest address is the address for the parameter **float x**, so I will change the data type to **float** and the highest address to 0019FF28. Note that your addresses may be different than mine.

|  |  |
| --- | --- |
| Data Type of Highest Addressed: | float |
| Address of Highest: | 0019FE08 |

To make it easier to see the addresses, run your program; resize the output window so that it takes up less space; press **Alt+Print Screen** to copy a picture of the output window to memory; and paste that copy into cell A31 in the Memory Map. My output window looks like the following:

A screenshot of a computer

Description automatically generated with medium confidence

We can now add the variable names in the **Alias** column. Don’t add the aliases for **i** (the **static int**) and the function **memoryMap** itself. Those aliases are in different areas of memory – we will discuss the areas of memory later.

Be careful to add the alias for **iPtr** at the address for **iPtr**, not its value (which is the same as the address of **a**).

We also know the sizes of each of these variables on our system, so use different background colors to highlight the range of values associated with each variable. Note that the addresses go up from the bottom, so each variable will go up from the address associated with its alias.

Recall that a **float** is 4 bytes; an **int** is 4 bytes; a **double** is 8 bytes; a pointer for the **x86** platform is **4** bytes (for **x64** systems, both the pointer size and the address size would be 8 bytes); a **char** is 1 byte; and a **short** is 2 bytes.

What about the size of **cArray**? Since cArray[] = "Hello", there are 6 bytes (including the byte for the null terminator).

Finally, add the values (in hex) for **int a**, **short s**, **char c**, **cArray** (with the characters going up from **'H'** at the bottom to the null terminator at the top), and **iPtr**. Remember to write all the values in little-endian notation! Don’t worry about writing the values for **float x** or **double d** – that would require us to get into the floating-point notation used internally, which we don’t want to worry about.

Your results should look something like this:



Note that to write the value “00” as the most significant byte in **iPtr**, I used a “Text” format so that it wouldn’t display the value simply as “0”.

In **Debug** mode, the compiler puts extra information in memory, which explains why the variables are further apart than necessary. Note also that each variable starts at a 4-byte boundary (except **char c**). This is because we are compiling on an **x86** platform (32-bit). This makes fetching quicker, but may require some wasted space between variables that is essentially wasted (for instance, the two bytes above **short s**).

Try creating a separate file named “Memory Map Release.xlsx” running in **Release** mode for the **x86** platform. You should see that the arrangement of the variables changes, and there is no more wasted space between variables (except the extra space around the **char c** and **short s**, and some extra space after the parameter **float x**). My results for the output window:

Text

Description automatically generated

And for the memory map in the worksheet:



Note the difference between the address of **iPtr** and the value of **iPtr**. The address of **iPtr** is the address where the pointer is stored. The value of **iPtr** is also an address – in this case, the address of **a**. Dereferencing **iPtr** with **\*iPtr** would then give us the value that **iPtr** is pointing to – in other words, the value of **a**.

## Pointers, pointer math, and memory maps

Memory maps can be particularly useful to understand pointers and pointer math. For instance, suppose we have the following (fictional) memory map for the character array (string) given by   
char cExample[] = "Example";**:**

|  |  |  |
| --- | --- | --- |
| **Byte’s Address** | **Alias** | **Byte’s Value** |
| 0x0019FEDF |  | 0x00 = '\0' |
| 0x0019FEDE |  | 0x65 = 'e' |
| 0x0019FEDD |  | 0x6C = 'l' |
| 0x0019FEDC |  | 0x70 = 'p' |
| 0x0019FEDB |  | 0x6D = 'm' |
| 0x0019FEDA |  | 0x61 = 'a' |
| 0x0019FED9 |  | 0x78 = 'x' |
| 0x0019FED8 | **cExample** | 0x45 = 'E' |

Suppose we also have three pointers:

char\* cPtr = cExample;

short\* sPtr = (short\*)cExample;

int\* iPtr = (int\*)cExample;

If we print out **cPtr**, **sPtr**, or **iPtr**, each will print out **0x0019FED8**, since they are all storing the address of **cExample**.

If we dereference each of the pointers, we will get different values. A dereferenced pointer will result in the data type it points to. Suppose, for example, that we print the dereferenced pointers using the specifier **"%x"**. Then we would get the following:

* **\*cPtr** would result in 45, since it references one byte (45 is the hexadecimal code for 'E' – recall that we can choose how to handle values, so we can print them out as characters, as decimal values, or as hexadecimal values).
* **\*sPtr** would result in 7845, since it references a **short int** which is two bytes on our system.
* **\*iPtr** would result in 6D617845, since an **int** is four bytes on our system.

If we add an integer to a pointer, it will change the address that the pointer stores by the size of the pointer’s data type. For example:

* **cPtr + 1** would equal 0x0019FED9 since a **char** is one byte so we increase the address by 1, and **\*(cPtr + 1)** would be 78 if printed out in hexadecimal and 'x' if printed as a character
* **cPtr + 5** would equal 0x0019FEDD, and **\*(cPtr + 5)** would be 6C if printed out in hexadecimal and 'l' if printed as a character
* **sPtr + 1** would equal 0x0019FEDA since a **short** is two bytes so we increase the address by 2, and **\*(sPtr + 1)** would be 6D61 if printed out in hexadecimal
* **sPtr + 3** would equal 0x0019FEDE, and **\*(sPtr + 3)** would be 0065 if printed out in hexadecimal
* **iPtr + 1** would equal 0x0019FEDC since we increase the address by four bytes (the size of an **int**), and **\*(iPtr + 1)** would be 00656C70 in hexadecimal
* **iPtr + 2** would equal 0x0019FEE0, which is beyond our memory map fragment

Note that an expression like **cPtr + 1** does not change the value that **cPtr** stores. If we want to change the value that **cPtr** stores, we must assign the result back to **cPtr** using code such as cPtr = cPtr + 1; or cPtr += 1; or by using the increment or decrement operators **++** or **--**.

Also note that **\*cPtr + 1** is not the same as **\*(cPtr + 1)**. Because of the order of operations, dereferencing would be done first in **\*cPtr + 1**, so the result would be 45 + 1 = 46 or 'F'.

Note also that we could print any of these results as a character using the **"%c"** format specifier. This specifier will always print only the one character in the least significant byte, so printing **\*(iPtr + 1)** as a character would display 'p'.

Pointers can also be subtracted from. For instance, consider the following code:

cPtr += 7; // moves pointer to the null terminator at the end of "Example"

printf("%c\n", \*(cPtr - 4));

It would print 'm'.

Finally, we can get the address where the pointer is stored, since pointers are just variables. We could thus print out **&cPtr**, **&sPtr**, and **&iPtr**. These would be three different addresses (not in the fragment of the memory map shown above).

To make sure that you understand pointers, pointer math, and memory maps, try filling out the memory map and the expressions on the side of “Memory Map Practice 1.docx” in the **Exercises** folder. Compare your results with “Memory Map Practice 1 solution.docx”.

## Areas in memory in C

There are several memory areas in C. First, there is the text area. The text area is where the program code, along with any string literals or global constants, is stored. It is illegal to try to change the values in these memory locations and will result in crashing your program in the Visual Studio environment. Looking back at the **memoryMap** function, you might recall that we printed the address of the **memoryMap** itself, and **memoryMap**’s address was far away from the addresses of the local variables. That is because it is stored in the text area.

Variables are stored in one of four locations in RAM, depending on how they are declared/created.

1. Data area – used for global variables that are initialized to a value other than 0.
2. BSS segment (don’t worry what BSS stands for) – used for global variables that are not initialized (defaults to 0).
3. Heap – used for dynamically allocated variables (done at runtime). This is where the keyword **new** in C# and Java put things. We have not yet studied dynamically allocated variables in C, but we will see that C uses the **malloc** function (and others).
4. Stack – used for variables and parameters declared within the function (everything on the stack frame).

You can find a detailed explanation at:  
<https://codingfreak.blogspot.com/2012/03/memory-layout-of-c-program-part-1.html>

Why are there two areas for global variables? When a C program starts up, but before the C library calls your **main** function, it sets all bytes in the BSS segment to zero; this is faster than initializing each variable to 0. In the data area, the C library sets the individual variables to their initial values, again before calling **main()**.

Review: Given the following program, in what area of RAM are each of the variables stored?

#include <stdio.h>

int globalVar;

double pi = 3.14159;

main()

{

char c;

}

Answers: *(highlight and change the font color to see the answers)*

* globalVar – BSS segment (global variable, not initialized – defaults to 0)
* pi – data area (global variable initialized to a value other than 0)
* c – stack (declared in a function, not using malloc or another memory allocation function)

### Static variables

There is a special kind of variable called a static variable, but this is different from global variables defined with the **static** keyword. A static variable is defined in a function, is only known within that function, but exists for the lifetime of the program. Rather than being allocated on the stack, memory for static variables is allocated either in the BSS segment or the data area (same as global variables). They work just like global variables, but their scope is limited to the function that they are declared in.

For instance, looking back at the **memoryMap** function, you will see that we declared a static variable at the top (static int i = 0;). The address of this variable was far from the addresses of the local variables because it would be in the BSS segment (since it is initialized to 0). Try changing it to a value of 1 to see that it moves to a different memory address (since it will now be in the data area) but is still far from the local variables.

It is good programming practice to limit the scope of variables; when you use them, static variables are better to use than global variables.

### More on the stack

Every time a function is called, the program does a number of things to the stack:

* Pushes the address of the current instruction onto the stack.
* Reserves space on the top of the stack frame for the return value.
* Pushes the values of parameters onto the stack.
* Allocates space on top of the stack for all variables defined in the function.

The specific details of how this is done vary from compiler to compiler and OS to OS. It is typically defined in the ABI (application binary interface), specifically the “calling conventions” of an ABI. Collectively, this information for each function is called a “stack frame”.

Suppose we have a program such as:

func1()

{

int someValue;

// do something...

}

func2()

{

int anotherValue;

func1();

}

main()

{

int someValue;

func1();

func2();

}

What happens with the stack?

The stack starts off as being empty.

When the main function is called, the stack will look something like:

|  |  |
| --- | --- |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

This is the stack frame for the **main** function. **main()** starts executing and calls **func1**. When this happens, space is allocated on top of the stack for **func1**’s stack frame:

|  |  |
| --- | --- |
| Local variables for **func1** (**someValue**) | **func1**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in **main** in the text segment) |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

Note that **main**’s variable **someValue** and **func1**’s variable **someValue** are distinct variables, with each occupying a different place in memory!

**func1** executes, and when it is finished, it pops its stack frame off the stack and resumes execution in its caller (at the address stored on its stack frame). The stack now looks like:

|  |  |
| --- | --- |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

Note that **func1**’s variables no longer exist! However, the values that they stored may still be in memory. Execution continues, and **main** calls **func2**. Space is allocated on top of the stack for **func2**’s stack frame:

|  |  |
| --- | --- |
| Local variables for **func2** (**anotherValue**) | **func2**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in **main** in the text segment) |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

We now have space allocated for **func2**’s variables, and they may actually use the same space as **func1**’s variables had!

Execution continues in **func2**, and it calls **func1**. Space is allocated on top of the stack for **func1**’s stack frame. Now, notice that we again have space allocated for **func1**’s variables, but it will be at a different location than the last time **func1** was called. Every time a function is called, new space is allocated for the function’s variables. There is no guarantee that it will occupy the same location in memory as the last time the function was called.

|  |  |
| --- | --- |
| Local variables for **func1** (**someValue**) | **func1**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in **func2** in the text segment) |
| Local variables for **func2** (**anotherValue**) | **func2**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in **main** in the text segment) |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

When **func1** finished execution, it pops its stack frame off the stack, and its variables no longer exist. Execution returns to the location where **func1** was called from (bottom address of **func1**’s stack frame) – somewhere in **func2**. We now have:

|  |  |
| --- | --- |
| Local variables for **func2** (**anotherValue**) | **func2**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in **main** in the text segment) |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

And when **func2** finishes executing, it pops its stack frame off the stack, and its variables no longer exist. Execution returns to the location where **func2** was called from (somewhere in main). We now have:

|  |  |
| --- | --- |
| Local variables for **main** (**someValue**) | **main**’s stack frame |
| Space for parameters (none) |
| Space for return value |
| Address of caller (somewhere in the text segment) |

And finally, when **main** finishes executing, its stack frame is popped off the stack. Execution returns to somewhere in the C library, and the C library finishes up execution and exits the program.

Important stuff to know:

* Local variables are allocated on the stack.
* Local variables may occupy different memory locations every time the function is called.
* Local variables may contain leftover values before they are initialized.

Note that recursive calls generate new stack frames every time the function is called. Thus, if you added the following code at the end of the **memoryMap** function, it would create new stack frames until it runs out of stack space (a condition known as “stack overflow”):

memoryMap(x + 1.0f); // continuous recursive call would cause stack overflow

## Memory map exercises

It is essential to practice your skills for this class since there are only 3 assignments. Thus, you will be given exercises to do. For memory maps, try out the following:

* “Memory Map Practice 1A - Basics.docx” for a practice with memory maps using basic data types (including pointers)



* + See the solutions in “Memory Map Practice 1A - Basics solution.docx”



* “Memory Map Practice 1.docx” for a practice with memory maps using arrays and pointers



* + See the solutions in “Memory Map Practice 1 solution.docx”



* “Memory Map Practice 1B - Arrays and Pointers.docx” for another practice with memory maps using arrays and pointers



* + See the solutions in “Memory Map Practice 1B - Arrays and Pointers solution.docx”



# Input Functions

There are several input functions in C that you should be aware of, along with some of their limitations and security risks. These include:

* scanf (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/scanf-scanf-l-wscanf-wscanf-l?view=msvc-170> or <https://en.cppreference.com/w/c/io/fscanf>)
* scanf\_s (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/scanf-s-scanf-s-l-wscanf-s-wscanf-s-l?view=msvc-170> or <https://en.cppreference.com/w/c/io/fscanf>)
* gets (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/gets-getws?view=msvc-170> or <https://en.cppreference.com/w/c/io/gets>)
* gets\_s (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/gets-s-getws-s?view=msvc-170> or <https://en.cppreference.com/w/c/io/gets>)
* fgets (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/fgets-fgetws?view=msvc-170> or <https://en.cppreference.com/w/c/io/fgets>)
* getc (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/getc-getwc?view=msvc-170> or <https://en.cppreference.com/w/c/io/fgetc>)
* getchar (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/getchar-getwchar?view=msvc-170> or <https://en.cppreference.com/w/c/io/getchar>)

Let’s create a C project with a file named **inputfuncs.c** to test these out. We will include the usual files, but also include **<string.h>**, which contains the string functions in C.

#include <stdio.h>

#include <stdlib.h>

#include <string.h> // contains the string functions in C

Add a **main** function that will call a function named **test1** and return EXIT\_SUCCESS:

int main()

{

test1();

return EXIT\_SUCCESS;

}

## Using **scanf** to read a single string

In **test1**, we will use the **scanf** function to input a string. **scanf** is similar to **printf** in that it includes a string containing specifiers that determine what kind of value to read in, followed by arguments for where the value should be stored. **%s** is the specifier for a string.

Create **test1** as follows. Don’t forget to put the function above **main**, or to prototype the function above **main** if you put **test1** after main.

// Test 1 - use scanf to input a string

void test1()

{

// Create buffers to put data into

char cArray[10];

int iReturnValue;

printf("Enter a string: "); // Note no \n so string will be

// entered on same line as prompt

iReturnValue = scanf("%s", cArray);

printf("String entered: %s\n", cArray);

printf("Return value: %d\n", iReturnValue);

}

If you try to run this program, you will see that you get an error saying “'scanf': This function or variable may be unsafe.”. By default, Visual Studio suggest using the **scanf\_s** function instead. However, not all compilers have **scanf\_s** implemented (it is listed in an optional annex to the C11 specification from 2011 – in fact, some sources like <https://code.fandom.com/wiki/Scanf_s> describe it as “specific to Microsoft compilers”) and code written before 2011 will use **scanf**. Thus, it is important to learn both **scanf** and **scanf\_s**.

To use **scanf** (and some other functions) in Visual Studio, add the following preprocessor directive at the top of your files:

#define \_CRT\_SECURE\_NO\_WARNINGS

Or to include it for all files, go to **Project | Properties > Configuration Properties > C/C++ > Preprocessor** and add **;\_CRT\_SECURE\_NO\_WARNINGS** to the end of the **Preprocessor Definitions** (note that if you switch between **Debug** and **Release** modes, you will have to add it twice – once for each mode).

Now you will be able to run the program:

A screenshot of a computer program

Description automatically generated

There are several things that you can observe by experimenting with the program:

* First, as shown above, **scanf** reads only until the first whitespace character. If that character is the newline character (from pressing **Enter**), it will read the full string. If that character is a space or tab, it stops reading (and leaves the remaining characters, including the whitespace character, on the input stream). Thus for the input “My string” above, it only read “My”.
* Second, **scanf** returns a value which is the number of items read in successfully (in this case 1).
* Third, if you enter a string that is too long for the buffer provided (**cArray** with a size of 10 in this case, so any string with more than 9 characters since the null terminator also needs to be stored), you may encounter an error – in **Debug** mode, it will display a dialog box for a “Debug Error!” indicating that “Stack around the variable 'cArray' was corrupted”. In **Release** mode, no such dialog will be displayed – the program will either run as normal (with an exit code of 0) or exit with a negative error code.

The third example above show the danger of **scanf** for reading strings. If you just uses the %s specifier, it will read into memory starting at the address provided (**cArray** in this case), but will not stop reading until it encounters whitespace, even if this goes beyond the size of the provided buffer!

A better way of reading in a string is to provide a modifier to the **%s** specifier consisting of a number between the **%** and **s** for the maximum size of the string (not including the null terminator). So for instance, if you change your code to:

iReturnValue = scanf("%9s", cArray);

you will find that you no longer get an error, and that it will only read in 9 characters at maximum (leaving any remaining characters on the standard input stream). It is good programming practice to use such a modifier when using **scanf** to read a string.

An alternative available on Microsoft compilers is to use **scanf\_s**. Try changing your code to the following:

iReturnValue = scanf\_s("%s", cArray);

You will see that it compiles and runs but crashes after the **scanf\_s** (regardless of what you enter). You will also see a warning in the **Error List** tab saying “Missing integer argument to 'scanf\_s' …”. When reading a string with **scanf\_s**, you should include the size of the buffer as an argument following the buffer location. Change your code to the following:

iReturnValue = scanf\_s("%s", cArray, (unsigned int)sizeof(cArray));

Note the casting of the size to an unsigned int – this is because the **sizeof** operator produces the special type **size\_t** by default, so you will get a warning without the cast. Now when you run the program, it will return an empty string and a return value of 0 if the string exceeds the buffer size, so you can no longer overwrite memory past the buffer (but you don’t read anything at all if the input is too long).

It is still good programming practice to use a modifier for **%s** when using **scanf\_s**, so a better option would be:

iReturnValue = scanf\_s("%9s", cArray, (unsigned int)sizeof(cArray));

Let’s try doing input and output from separate functions. For output, all we need to do is pass in the address of the start of the buffer and print it out – remember to prototype your function at the top of the file if necessary:

/\*

\* Purpose: Print a message followed by the string passed in.

\* Parameters: cPtr - the address of a buffer containing a string

\* Returns: nothing

\* Assumptions: cPtr contains a null-terminated string.

\*/

void printString(char\* cPtr)

{

printf("The result of reading in the string is:\n");

printf("%s\n", cPtr);

}

Note that C does not have a default commenting style (although plugins similar to Javadoc exist).

We can then call this with:

printString(cArray);

Note that although we call this with an array name, we are just passing the start of the memory block containing the string to be printed, so we store the address in a pointer in the function. A quick review of pointers:

* A pointer uses the \* after a data type to indicate the variable is a pointer and will store an address.
* Pointers are 32 bits or 4 bytes when compiling to an x86 system.
* Pointers are 64 bits or 8 bytes when compiling to an x64 system.

Now let’s create a **readString** function for input. We know that ideally, we should include a modifier to **%s**, so let’s pass that it along with the buffer for the string. Our call in **test1** will look like:

readString(cArray, "6");

printString(cArray);

And our function prototype will look like:

void readString(char\* cPtr, char\* cLengthPtr);

The actual function will look like:

void readString(char\* cPtr, char\* cLengthPtr)

{

// do something here to create a specifier string

printf("Enter a string: ");

scanf("%s", cPtr);

}

This function works but does not create a specifier string to prevent overflow of the buffer. To build a specifier string, we can create an array of chars and build the format specifier from the digit or digits passed in from the string in **cLengthPtr**. One way to do so is to use the string concatenation function **strcat** function (<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/strcat-wcscat-mbscat?view=msvc-170> or <https://en.cppreference.com/w/c/string/byte/strcat>) from **<string.h>** (which we included at the top of the file).

**strcat** takes in a starting null-terminated string followed by another null-terminated string, and copies the second string at the end of the first string (so there must be sufficient space in the first string or it can cause buffer overflow!). Here is one possible solution using **strcat** to build a format specifier from an empty string by concatenating a “%”, the length, and an “s”:

// Create a format specifier

char cFormat[4];

// Initialize cFormat as an empty string

cFormat[0] = '\0';

// Use string concatenation to build up the format specifier

strcat(cFormat, "%");

strcat(cFormat, cLengthPtr);

strcat(cFormat, "s");

If we call the function with **"6"** as the second argument, the format specifier will work out to **"%6s"**. Other methods could also be used to build the string, but note that unlike Java, we cannot just combine strings using the + operator (like **cFormat = "%" + cLengthPtr + "s";**).

To see if you understand the **readString** function, try writing appropriate comments for the function. Compare them with the comments below (currently in white text – change the font color to see the text):

/\*

\* Purpose: read in a string up to a given length

\* Parameters: cPtr - pointer to an array of chars where the string will be stored

\* cLengthPtr - pointer to a string containing the maximum length

\* Returns: nothing

\* Assumptions: cLengthPtr is a string of length 2 (including the null terminator)

\* containing a digit for the length of the string to read in

\*/

(Note that the function is currently limited to one character for the length in the format specifier. To add more characters, declare **cFormat** with a bigger array size.)

## Using **scanf** to read multiple strings

**scanf** can also be used to read in multiple values all at once by including multiple specifiers in the format string. We will try reading in two strings.

Comment out the call to **test1** in **main**, and add a call to **test2**. Then create a **test2** function as follows:

void test2()

{

// Create buffers to put data into

char cArray[10];

char cArray2[10];

int iReturnValue;

printf("Enter two strings separated by a space: ");

// The format specifier includes two strings in this case

iReturnValue = scanf("%9s %9s", cArray, cArray2);

printString(cArray);

printString(cArray2);

printf("Returns: %d\n", iReturnValue);

}

You should notice the following when running the program:

* If your strings are 9 characters or below, you must supply at least two strings separated by whitespace, although any whitespace (one or more spaces, tabs, or newlines) will work. If you only supply one string, the program will wait for further input.
* If you enter more than two strings, the remaining characters are not used (and remain on the standard input stream).
* If your strings are more than 9 characters, only 9 characters are read. If the first string is more than 9 characters, the additional characters will be read in as the second string.
* The return value will be 2.

Here is some sample output showing more than two strings given as input:

A screenshot of a computer

Description automatically generated

## Using **scanf** to read in some integers

Just as **printf** can be used to print data in various formats, **scanf** can read data in various formats. Let’s try to read in some integers and also learn a bit about integer arrays.

Comment out the call to **test2** in **main** and add a call to **test3**. Then write **test3** as follows:

void test3()

{

int iArray[5];

int iSize;

readNumbers(iArray); // should pass in the size of the array

// readNumbers is deliberately poorly written here

printNumbers(iArray, 5);

}

Add a function **readNumbers** that will take in the address of an array of integers. Initially, set it up to just read one int. Here is what your code might look like:

void readNumbers(int\* iPtr)

{

printf("Enter a number: ");

scanf("%d", iPtr);

}

Note the use of the **%d** specifier to look for a decimal number to read into the memory location specified by **iPtr**. Now create a function **printNumbers** that will initially just print one number:

void printNumbers(int\* iPtr, int iSize)

{

printf("The numbers are:\n");

// dereference iPtr to get the value at that address (pointed to)

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

}

You should be able to run the program, enter one number, and see that number printed out.

Let’s now change **readNumbers** to read numbers until a non-number is entered. Note that it will do so regardless of how many numbers are entered, and thus could write fewer numbers than the 5 in **iArray**, or it could write more numbers than can be stored in **iArray**. This is very poor programming practice and done here just to show that we could easily overflow the buffer.

void readNumbers(int\* iPtr)

{

printf("Enter 5 numbers: ");

while (scanf("%d", iPtr))

{

iPtr = iPtr + 1; // or iPtr += 1; or iPtr++;

}

}

Note the following:

* **scanf** returns the number of reads that were successful. If it returns 0, no reads were successful (such as when a non-number is entered for a **%d** specifier).
* Recall that 0 is false in C, so when **scanf** fails to read in a number and returns 0, the **while** loop will end.
* **iPtr = iPtr + 1;** moves the pointer to the next integer using pointer math. “+ 1” here means “Add the size of one integer to the address”, so in this case, we add 4 to the address in i**iPtr**.
* If less than 5 integers are entered followed by a non-integer (such as letters), only those values will be read in. If more than 5 integers are entered, this function will simply keep adding integers to memory, which would corrupt the stack (and be reported as an error in **Debug** mode, although not necessarily in **Release** mode).

Let’s now update **printNumbers** so that it uses **iSize** to print all the values in the array:

void printNumbers(int\* iPtr, int iSize)

{

printf("The numbers are:\n");

for (int i = 0; i < iSize; i++)

{

// dereference iPtr to get the value at that address (pointed to)

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

iPtr++; // increment iPtr to get to the next int in memory

}

}

Note that although **iPtr** is a pointer, we can also use array notation to access the values:

void printNumbers(int\* iPtr, int iSize)

{

printf("The numbers are:\n");

for (int i = 0; i < iSize; i++)

{

printf("%d\n", iPtr[i]);

}

}

Going back to **test3**, you’ll see that both the definition of the array (int iArray[5];) and the call to **printNumbers** (printNumbers(iArray, 5);)use the number 5. It would be better to only define the number once. We could use a **#define** directive, but we can also just calculate the size on an array, if and only if we are in the same scope where it is declared. This can be done as follows:

iSize = sizeof(iArray) / sizeof(int);

printNumbers(iArray, iSize);

The **sizeof** operator will give us the size of **iArray** in bytes, which is then divided by the size of its component integers to get the number of integers in the array. Once again, this ONLY works in the same scope where the array is declared. If you need the size elsewhere, you MUST pass the size of the array to whatever function you are using.

Consider **printNumbers**, for instance. It is tempting to do the following at the top of the code:

iSize = sizeof(iPtr) / sizeof(int);

Try it and see how many numbers are printed out. Run it in both the **x64** and **x86** platforms. Can you explain the results? Try to figure it out, and then read the explanation below (currently in white text – format it using a different font color to see the answer).

**iPtr** does not contain the array – it is just a pointer to an int and contains a memory address. The size of a pointer is ALWAYS 8 bytes on the **x64** platform and 4 bytes on the **x86** platform. Thus, dividing by the size of an integer (4) will always give the result of 2 (on the **x64** platform) or 1 (on the **x86** platform). Since a pointer is just given the address of the start of the array, we MUST pass the size of the array in.

## Using **gets**, **gets\_s**, and **fgets** to read a string

There are other input functions besides **scanf**. One that you should be aware of is **gets** along with its variants **gets\_s** and **fgets**. **gets** gets a string and stores it in the buffer given as an argument.

To try it out, comment out the call to **test3** in **main** and add a call to **test4**, which will be created as follows:

void test4()

{

char cArray[10];

readStringGets(cArray, sizeof(cArray) / sizeof(char));

printString(cArray);

}

Then we need to create **readStringGets** as follows:

void readStringGets(char\* cPtr, int iBufferSize)

{

printf("Enter a big honking string: ");

gets(cPtr);

}

If you run the program, you will see that if you enter a string (including whitespace until a newline) that is 9 characters or less, it works fine. But if the string is longer than 9 characters, it will overflow the buffer given in **cArray** (causing an stack corruption error in **Debug** mode and possibly returning an error code in **Release** mode).

**gets** is thus considered a dangerous function because it can read past the end of the buffer passed to it. In fact, it is considered so dangerous that it was deprecated as of the C99 standard and removed entirely in the C11 standard.

A safer alternative is **get\_s**, which takes in a second argument which is the buffer size:

gets\_s(cPtr, iBufferSize);

If you try running the program and entering more than 9 characters now, the program will exit with an error dialog in **Debug** mode or an error code in **Release** mode. Either way, it will not proceed past the **gets\_s** call (and thus will show no further output). (It is possible to create a constraint handler function to deal with such calls.)

Another safer alternative is **fgets**, which gets a string from a file stream as a third argument, but that file stream can be set to the standard input stream **stdin** to get a value from the console. **fgets** takes in a second argument which is the buffer size, and it will not read past that buffer size (leaving the remaining characters on the stream):

fgets(cPtr, iBufferSize, stdin);

One disadvantage of **fgets** is that if it encounters the newline character and still has space in the buffer, it will actually read the newline character in, which may not be the desired result for the string entered.

Note that you need to be careful when mixing **scanf** with **gets\_s** or **fgets**. **Scanf** will quit reading when it sees the newline character '\n' or any whitespace on the standard input buffer and leaves the newline or whitespace in the input stream. Thus, if the user has entered something using **scanf** and then there is a call to **gets\_s** or **fgets**, the call to **gets\_s** or **fgets** may just read the newline left over (or other characters left over) after the **scanf**.

Try adding the following at the start of **readStringGets**:

int iValue;

printf("Enter an int: ");

scanf("%d", &iValue); // must take address of x so that scanf can change x

printf("You entered %d\n", iValue);

If you run this program, you will see that if you enter a number and hit **Enter**, the following **gets\_s** or **fgets** immediately reads the newline character and thus reads an empty string.

Note also that we must use the address of **iValue** when calling **scanf** with a single variable rather than an array. An array name is already the address of its memory, so **scanf** can read a value into the array. If we just used the single variable name **iValue**, it would by default pass along its value instead of its address. s**canf** would thus be unable to change the value in the current function.

There are several ways to clear the input buffer. If you assume that the user has just hit **Enter** immediately, you can use **getc** to get a single character from the input stream:

getc(stdin); // Gets a character from the stream passed to it

Or you can use **getchar**, which defaults to the standard input stream:

getchar(); // Gets a character from the stdin stream

If you want to clear everything up to and including the next newline (instead of assuming that the user hit **Enter** immediately after entering the **scanf**), you can use the following:

// clear the buffer

int c = getchar();

while (c != '\n')

{

c = getchar();

}

An alternative but less readable version of the code above is as follows. You should be able to read such code because some C programmers will use ode such as the following, but it can be hard to read so you may want to avoid writing such code:

while (getchar() != '\n');

Note the empty loop body (indicated by the **;** at the end) – everything is being done in the **while** condition!

## Warning on getting input from functions

So far, when we have used functions like **readString** or **readStringGets**, we have passed in the buffer where the input is to be stored. What if we try to create the input in the function and then pass it back? Let’s see what happens.

Comment out **test4** in main and add a call to a new function **test5** as follows:

void test5()

{

char\* cPtr = NULL;

cPtr = readStringReturnBuffer();

printString(cPtr);

}

Then create **readStringReturnBuffer** as follows:

char\* readStringReturnBuffer()

{

char cBuffer[256];

printf("Enter a big string, honking or not: ");

gets\_s(cBuffer, sizeof(cBuffer) / sizeof(char));

return cBuffer;

}

As you can see, we create a **char** array of size 256 in the function. Which area of memory is this array stored in? Highlight and change color to see the answer à **stack**

We then read in the string using **gets** into that char array, and return the array, or at least the address of the start of the array (note that we are returning a pointer).

Try running this program in both **Debug** and **Release** mode. In **Debug** mode, you will see it runs, but may prints a bunch of garbage. (You may also notice the warning “warning C4172: returning address of local variable or temporary: cBuffer”.) However, in **Release** mode, it may seem to run without error.

The problem here is that the memory for **cBuffer** is allocated on the stack. We then return the address of memory on the stack. This is a bad idea because the stack frame is popped off when the function is done and will be overwritten when some other function is called.

You can see this in **Debug** mode by adding the following line at the start of **printString**:

char cArbitrary[256] = "Some arbitrary text";

and running the program – it will print out **cArbitrary** instead of the entered text.

Instead, we should pass the buffer into the into the function where we want to read the input, or dynamically allocate memory on the heap (using **malloc** or some related function, which we haven’t covered yet).

# Arrays and Pointers

We should delve further into how arrays are stored in C and how values can be manipulated with pointers.

The first thing to note is that arrays in C are not objects, so they do not have a length property and cannot be traversed using things like the for-each loop in Java. Instead, arrays are just chunks of memory allocated with a certain data type. As such, we need to move to the different memory locations in the array to get the values. For instance, for the array short int sValues[] = { -32768, 9, 32767 }; (note that arrays of numbers can be initialized using a comma-separated list of values inside curly braces, just like in Java), memory would look like:

|  |  |  |  |
| --- | --- | --- | --- |
| **Alias** | **Value of byte** | **Value as short** | **Array item** |
|  | 7F | 7FFF = 32767 |  |
|  | FF | sValues[2] |
|  | 00 | 0009 = 9 |  |
|  | 09 | sValues[1] |
|  | 80 | 8000 = -32768 |  |
| sValues | 00 | sValues[0] |

But since arrays aren’t objects, we can’t pass them to functions. We can only pass the starting address of the array (the start of the chunk of memory). We must also pass in the size of the array, as we have seen, since there is no length property for the array.

## Exercise: Print an array of **int**s

Let’s try a practice exercise using things we’ve already seen. First, create a new project with a main function that calls a **testIntArray** function with an array in it:

#include <stdio.h>

#include <stdlib.h>

void testIntArray()

{

int iArray[] = { 2, 7, 9 }; // initialize an int array using {}

}

int main()

{

testIntArray();

return EXIT\_SUCCESS;

}

Then in **testIntArray**, call a function named **intArrayPrinter** in a separate source file named **array\_pointers.c** (with an appropriate header file). The function should loop through the array, printing each value on a separate line.

Try this practice on your own before looking at the solution below…

(No peeking!)

*Solution:* Let’s start by doing this the wrong way, just so we can see what the problem is. Just like the example of returning stack memory, we can create a version of the code that will compile and run, but not work correctly in all cases. (Diagnosing such problems might be good questions on an exam!)

Create a header file **array\_pointers.h**. You can protect it using the **#pragma once** directive or using a header guard as shown below:

#ifndef ARRAY\_POINTERS\_H

#define ARRAY\_POINTERS\_H

void intArrayPrinter(int[]);

#endif // !ARRAY\_POINTERS\_H

Note that the header file does not need to contain the argument name, so just **int[]** is fine. Also note that we can use array notation to pass the start of the array (although we can also use pointer notation, as we will see later – they are equivalent).

Now we can create a file **array\_pointers.c** containing the function:

#include <stdio.h>

#include "array\_pointers.h"

void intArrayPrinter(int iArrayPtr[])

{

int iSize = sizeof(iArrayPtr) / sizeof(int);

for (int i = 0; i < iSize; i++)

{

printf("The loop counter is %d and the array value is %d\n",

i, iArrayPtr[i]);

}

}

Finally, we can call **intArrayPrinter** from our test function:

#include "array\_pointers.h"

void testIntArray()

{

int iArray[] = { 2, 7, 9 }; // initialize an int array using {}

intArrayPrinter(iArray);

}

(Do you see the error(s) yet?)

The program will compile without problem, but when it runs it will only display either 1 integer or 2 integers from the array. In Visual Studio, you may notice a warning with the line int iSize = sizeof(iArrayPtr) / sizeof(int); saying that you are dividing the size of a pointer by another value. What is happening?

Arrays aren’t objects, so we are just passing the address of the array to the **intArrayPrinter** function, not the array itself. So although we declare the parameter as int iArrayPtr[], it is really a pointer that just stores an address. Using the **sizeof** operator just gets the size of a pointer (4 bytes on the **x86** platform, 8 bytes on the **x64** platform), so then dividing by the size of an **int** just gives us 1 or 2.

This is why we need to pass in the size of the array to the **intArrayPrinter** function. We are just passing the starting address of a chunk of memory to **iArrayPtr**, and without passing the size, we don’t know how big that chunk of memory is. So adjust **intArrayPrinter** as such:

void intArrayPrinter(int iArrayPtr[], int iSize)

{

for (int i = 0; i < iSize; i++)

{

printf("The loop counter is %d and the array value is %d\n",

i, iArrayPtr[i]);

}

}

And adjust the function prototype in the header file to match:

void intArrayPrinter(int[], int);

And finally update the calling function:

int iArray[] = { 2, 7, 9 }; // initialize an int array using {}

int iSize = sizeof(iArray) / sizeof(int);

intArrayPrinter(iArray, iSize);

Note that we *can* use **sizeof** here, since this is where the array is declared. We could also just pass in 3 for **iSize**, but using the formula allows us to modify the number of elements in **iArray** and have the size automatically updated.

Our program now works, but let’s examine it in a bit more detail to understand what is happening. I’m going to show this in **x86** mode with the **Project | Properties > Configuration Properties > Linker > All Options > Randomized Base Address: No (/DYNAMICBASE:NO)** option set to get a consistent set of addresses – your code may come up with different addresses.

In the calling function **testIntArray**, add the following line:

printf("The address of iArray in the calling function is \t%p\n", iArray);

And modify **intArrayPrinter** as follows:

void intArrayPrinter(int iArrayPtr[], int iSize)

{

printf("The address stored in iArrayPtr in the function is \t%p\n", iArrayPtr);

printf("The address of iArrayPtr in the function is \t\t%p\n", &iArrayPtr);

for (int i = 0; i < iSize; i++)

{

printf("The loop counter is %d and the array value is %d"

" and the value's address is %p\n",

i, iArrayPtr[i], &iArrayPtr[i]);

}

printf("\n");

}

My output looks like the following:

A screenshot of a computer

Description automatically generated

Note that the address of **iArray** in the calling function is a value (**0019FE00** in the example above) of the address on the stack. This address will always be the same – you cannot change it. (Try doing iArray++; in the calling function to see that this value cannot be changed.)

The address stored by **iArrayPtr** is the address of **iArray**, but the address of the pointer itself is different. The values in the array are just consecutive chunks of 4 bytes (the size of an **int**) in memory – you can see that their addresses increase by 4 each time (remember to count in hexadecimal!).

Let’s try a second solution. We’ll write a function called **intArrayPrinterWithPointer** this time. It will be called the same way:

intArrayPrinterWithPointer(iArray, iSize);

We could prototype it the same way in the header file, but let’s emphasize that we’re really passing in a pointer:

void intArrayPrinterWithPointer(int\*, int);

We could actually just use the exact same code in this function, but let’s try dereferencing the pointer instead of using array notation. The two notations are equivalent, so iArrayPtr[i] is the same as \*(iArrayPtr + i). For instance, \*(iArrayPtr + 2) would say “Go two **int**s from the start of the array and give me the **int** value at that address”, which is what iArrayPtr[2] says as well (remembering that arrays are zero-based).

void intArrayPrinterWithPointer(int\* iArrayPtr, int iSize)

{

for (int i = 0; i < iSize; i++)

{

// To retrieve the value where the pointer points, we "dereference" it

// using the dereference operator \*

printf("The loop counter is %d and the array value is %d"

" and the value's address is %p\n",

i, \*(iArrayPtr + i), iArrayPtr + i);

// The notation \*(iArrayPtr + i) is exactly the same as iArrayPtr[i]

// Note that because iArrayPtr is int\*, we add i \* sizeof(int) each time

}

The notation iArrayPtr + i is thus also the same as &iArrayPtr[i].

If you run the new version of the function, you should see that its output is exactly the same as the previous version.

Finally, instead of using pointer math to find the right location in the array, we could “walk” the pointer through the array:

void intArrayPrinterWithPointer(int\* iArrayPtr, int iSize)

{

for (int i = 0; i < iSize; i++)

{

printf("The loop counter is %d and the array value is %d"

" and the value's address is %p\n",

i, \*iArrayPtr, iArrayPtr);

iArrayPtr++;

}

// The pointer will point past the end of the array when the loop is complete

printf("The final value of iArrayPtr is %p\n", iArrayPtr);

}

Note that iArrayPtr++ will increment the pointer by the size of one int, not just by one, so it will move to the next element in the array. We still need **iSize**, since there is no indicator of the size of the array otherwise (it is an array of **int**s and not a null-terminated string, where we could tell the end of the **char** array by looking for the null terminator). Finally, note that the pointer actually contains the address of the byte just after the end of the array at the end, since we have walked it through the entire array and past the last entry.

## Exercise: Print a string one character at a time

Let’s try another exercise. In **main**, add a call to the function **testCharArrayPrintPointer**:

testCharArrayPrintPointer();

And create the function **testCharArrayPrintPointer** as follows:

void testCharArrayPrintPointer()

{

}

Finally, write a function called **charArrayPrinterWithPointer** in **array\_pointers.c**. In that function, use a pointer to move along a string that is passed in one character at a time until you reach the end of the string, dereferencing the pointer to display the character at each location. You can use printf("%c", value); or putchar(value); to print out one character at a time. Call the function from the test function with some string passed in.

Try this practice on your own before looking at the solution below (no peeking!)…

Our function prototype in the header file might look like:

void charArrayPrinterWithPointer(char\* cArrayPtr);

Note that in this example, I am showing the name of the parameter, but again, that’s optional in a function prototype. Also note that we don’t have to pass in the size in this case, because we can just look for the null terminator at the end of the string.

We can call **charArrayPrinterWithPointer** from the test function as follows:

charArrayPrinterWithPointer("Hello world!");

The function itself could look like:

void charArrayPrinterWithPointer(char\* cArrayPtr)

{

while (\*cArrayPtr != '\0')

{

printf("%c", \*cArrayPtr);

cArrayPtr++;

}

printf("\n");

}

Note that we dereference the pointer to check its value against the null terminator. If it is not the null terminator, we print the character and then advance **cArrayPtr**. Because it is of type **char\***, incrementing **cArrayPtr** will move to the next **char** (increasing the memory address stored in **cArrayPtr** by one each time).

Note that depending on the version of Visual Studio you are using, you may see error highlighting show up when passing in a literal string like "Hello world!". This error highlighting will not stop the program from compiling or running successfully, but we should examine what it is talking about.

The error will say something about passing a **const char\*** into a **char\*** parameter. Note that this error would go away if you changed your call to the following:

char cArray[] = "Hello world!";

charArrayPrinterWithPointer(cArray);

What is the difference? The second version uses an array of **char**s as a local variable, so it would be stored on the stack and could have its characters changed. The first version passes in a string literal directly, and such string literal constants are stored in the text segment (along with the program instructions) and so cannot have its characters changed. Try the following with both versions of the call at the start of **charArrayPrinterWithPointers**:

\*cArrayPtr = 'A';

You will see the charArrayPrinterWithPointer("Hello world!"); version crashes when it reaches this statement, since you are trying to change data in the text segment of the program, which is illegal. The second version of the call works correctly and prints “Aello world!”, as it has changed the first character in the string to ‘A’.

To see this further, go back to the charArrayPrinterWithPointer("Hello world!"); version, and let’s examine some of the memory addresses in the **charArrayPrinterWithPointer** function. Add the following code at the start of the function (and comment out \*cArrayPtr = 'A';):

printf("The address of cArrayPtr is \t%p\n", &cArrayPtr);

printf("The address of the string literal in cArrayPtr is \t%p\n",

cArrayPtr);

printf(“The character at the address in cArrayPtr is \t%x\n”, \*cArrayPtr);

printf(“The address of the charArrayPrinterWithPointer function is \t%p\n”,

charArrayPrinterWithPointer);

My output for the program is as follows:

A screenshot of a computer

Description automatically generated

Note that the address of the string literal is close to the address of the function, since both are stored in the text segment (recall that the name of a function just prints its address in memory), while the address of **cArrayPtr** is somewhere far away on the stack.

To indicate that the string is not to be changed, we can alter the signature of the function as follows – in the prototype:

void charArrayPrinterWithPointer(const char\* cArrayPtr);

And in **array\_pointers.c**:

void charArrayPrinterWithPointer(const char\* cArrayPtr)

The type const char\* says that the characters being pointed at cannot be changed (note that the value of the pointer can still be changed, as with **cArrayPtr++**). So if you uncomment \*cArrayPtr = 'A';, you will see that a compiler error now occurs saying that “l-value specifies const object”, or in other words, the character cannot be changed.

Finally, let’s look at an alternate version of **charArrayPrinterWithPointers**. We could have written:

void charArrayPrinterWithPointer(const char\* cArrayPtr)

{

while (\*cArrayPtr)

{

putchar(\*cArrayPtr++);

}

putchar('\n');

}

Note that since the value of the null terminator is 0, we can check \*cArrayPtr, which will be false (0) when it reaches the null terminator and true otherwise. **putchar** is a function that outputs a single character to the standard output stream **stdout**.

Finally, note that the evaluation of \*cArrayPtr++; depends on the order of precedence of operators in C. See <https://en.cppreference.com/w/c/language/operator_precedence> for a list. You will see that the postfix **++** has a higher precedence that the indirection (dereference) operator **\***. Thus, the **++** applies to **cArrayPtr** and not to **\*cArrayPtr**, so it increments the pointer and not the character pointed at. Also, postfix **++** (at the end) means that the increment is only applied after the value is used, so we dereference the pointer before it is incremented. This version of the code is not as readable as:

putchar(\*cArrayPtr);

cArrayPtr++;

However, you will see code like the first version, so you should be able to figure it out.

## Printing the bytes in an integer

We have seen that variables are just chunks of memory. We have also seen that different types of pointers dereference different amounts of memory. For instance, an **int\*** pointer dereferences an **int** value (4 bytes), while a **char\*** pointer dereferences a **char** value (1 byte).

Let’s use these observations to print out an **int** as a sequence of bytes. In **main**, add:

testIntBytePrinter();

**testIntBytePrinter** will look like:

void testIntBytePrinter()

{

int iSomeNumber = 2189640;

intBytePrinter(iSomeNumber);

}

Note that you could choose any **int** value for **iSomeNumber**, but we will see that 2189640 has something special about it!

Add void intBytePrinter(int); to the header file. Our function will look like:

void intBytePrinter(int iValue)

{

printf("The int in hexadecimal is %x\n", iValue);

// With a char pointer, we can work one byte at a time

char\* cPtr = (char\*)&iValue;

for (int i = 0; i < sizeof(int); i++)

{

printf("The byte value is %x and the character equivalent is %c\n",

\*cPtr, \*cPtr);

cPtr++;

}

}

What are we doing here? Well, first we print the **int** in hexadecimal (so 2189640 would print as 216948) so that we can compare the values of the bytes to what we print out later.

Then we set a **char\*** variable to the address of the **int** passed in (casting it to a **char\*** to avoid any warnings). Since it is a **char\***, changing the pointer will move one byte at a time, and dereferencing the pointer will give us one byte. So then we can loop through each byte in the int, printing the value of each byte as we change **cPtr** in both hexadecimal and as a character. What do you see for the value 2189640? That’s right, that value has bytes with the codes for **H**, **i**, and **!** (and a 0 which would be the null terminator). So that integer has the same bytes as the string “Hi!”.

Try other values for **iSomeNumber** and you will see different byte values and characters displayed. In general, if you pick a number at random, you may see some weird characters. Also, note that if you choose a number such as 123456789 where a byte is storing a negative number, the **%x** specifier will print it as 4 bytes by default (with leading F’s) – you could change the format specifier to **%hhx**, where the **hh** indicates that only one byte’s worth of data should be printed.

## Encoding and decoding values in a double

We have just seen that you can look at bytes of an **int** individually by using a **char\***. We could do this with any type of value, and with any kind of pointer to modify it. Let’s try an example where we store some values in a **double** as an “encoded” number, and then decode the **double** to get back the original values. In **main**, add:

testEncodeDecode();

**testEncodeDecode** will look like:

void testEncodeDecode()

{

double x = encode();

printf("%g\n", x);

decode(x);

}

So, what we want to do is create an **encode** function that will return a **double** with some values encoded in it. We will print that **double** to see what the encoded number is (note the **%g** specifier just allows us to print large or small floating-point numbers using the exponent). Then we will use **decode** to get back the original values.

Add the following to the header file:

double encode();

void decode(double);

Let’s make **encode** and **decode** simple to start:

double encode()

{

return 0.000000123;

}

void decode(double x)

{

printf("%f\n", x);

}

You will see the program prints **1.23e-07** and the decode function prints **0.000000** (because it uses the **%f** specifier).

Now let’s update **encode** to store some data in a double. The double is just an 8-byte buffer in this case – we will manipulate the individual bytes regardless of the fact that they are part of a double. We will store a 3-letter name, an age in one byte, followed by a student loan amount in two bytes, all within the bytes of the double. Let’s start with the name:

double encode()

{

// Allocate the buffer

double x = 0.0;

unsigned char\* cNamePtr = (unsigned char\*)&x;

// Prompt the user for input

printf("Enter a 3-letter name: ");

scanf\_s("%3s", cNamePtr, NAME\_SIZE);

return x;

}

Notice what we are doing here. We create **x** and return it at the end, but it is just being used as a buffer. To access the bytes in **x**, we create a **unsigned char\*** and make it equal to the address of **x**. Note that Visual Studio will show a red underline for assigning **&x** (the address of a **double**) if we do not explicitly cast it to an **unsigned char\*** (although the program will run and implicitly cast it anyway).

Then we get input of at most 3 characters plus the null terminator and store it in **x** using the pointer. Note that somewhere (such as in the header file), we must define **NAME\_SIZE**:

#define NAME\_SIZE 4 // 3 characters plus the null terminator

When working with memory and pointers like this, it can be useful to draw a diagram to show what we are doing. Here is a little table showing what we have done, assuming the name **"Ada"** is entered:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value in Memory**  **(with hex equivalent)** | **Pointer** |
| **x** | 0 |  |
| 0 |  |
| 0 |  |
| 0 |  |
| '\0' = 00 |  |
| 'a' = 61 |  |
| 'd' = 64 |  |
| 'A' = 41 | **ß cNamePtr** |

If you run the program now, you will see that it prints out **3.15345e-317**, which is the **double** equivalent of those bytes. (Don’t worry about how a **double** is represented.)

Next, we want to store the age that the user enters in one byte. To do so, let’s start by adding a #define directive:

#define AGE\_SIZE 1

Did you notice that we are using an **unsigned char** pointer? This explicitly says that we should treat the byte as unsigned so it can contain values from 0 to 255. Add the following code to **encode**:

// Enter an age

printf("Enter an age (less than 256): ");

scanf\_s("%hhu", cNamePtr + NAME\_SIZE);

Note that the **hh** modifier in **%hhu** says that we should work with one byte, and the **u** says it should be unsigned. If a larger number is entered, only the lowest-order byte will be used.

Also note that we use **cNamePtr + NAME\_SIZE** to get at the byte after the space for the name. Because **cNamePtr** is an **unsigned char\***, adding **NAME\_SIZE** just adds that number to the address, so it accesses the fifth byte in the double.

Try running it and enter **"Ada"** and an age of **255**. The output should be **5.41112e-312**.

Note that if an age of **0** is entered, the output would be the same as before (**3.15345e-317**). Try entering other names and ages to see that the output value changes. Our chunk of memory would look like the following:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value in Memory**  **(with hex equivalent)** | **Pointer** |
| **x** | 0 |  |
| 0 |  |
| 0 |  |
| 255 = FF | **ß cNamePtr + 4** |
| '\0' = 00 |  |
| 'a' = 61 |  |
| 'd' = 64 |  |
| 'A' = 41 | **ß cNamePtr** |

The last thing to do in **encode** is to add a student loan amount in two bytes. To get to the correct memory address, we can use **cNamePtr + NAME\_SIZE + AGE\_SIZE**, but since **cNamePtr** is an **unsigned char\***, that would only access one byte. To get two bytes, we need to cast it as an **unsigned short\***:

// Enter the student loan amount

printf("Enter your student loan as an integer <= 65535: ");

scanf\_s("%hu", (unsigned short\*)(cNamePtr + NAME\_SIZE + AGE\_SIZE));

Run it now and try entering **"Ada"**, **255**, and **32769**. The output should be **2.84948e-306**. Our memory chunk would look like the following (note that **32769** = **8001** in hexadecimal):

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value in Memory**  **(with hex equivalent)** | **Pointer** |
| **x** | 0 |  |
| 80 |  |
| 01 | **ß cNamePtr + 4 + 1** |
| 255 = FF | **ß cNamePtr + 4** |
| '\0' = 00 |  |
| 'a' = 61 |  |
| 'd' = 64 |  |
| 'A' = 41 | **ß cNamePtr** |

Note that if you run it with a student loan amount of **0**, the output will be unchanged from before.

Now that we have encoded our double, we should try to get back the values the user entered. This is what our **decode** function should do. We will take in a **double** **x**, but again, we will treat it as just an 8-byte buffer. To do so, we use pointers again. To print out the name, we just create an **unsigned char\*** and give it the address of **x**:

void decode(double x)

{

// Pointer to the name

unsigned char\* cNamePtr = (unsigned char\*)&x;

// Print the string

printf("The name is %s\n", cNamePtr);

}

To get at the age, we need to move the pointer to the correct byte using **cNamePtr + NAME\_SIZE** and then dereference it to get the value at that byte:

// Dereference the pointer to retrieve the single byte representing the age

printf("The age is %d\n", \*(cNamePtr + NAME\_SIZE));

Getting the age is a little bit trickier. We could just move to the correct byte and then cast it as an **unsigned short** all in one step, but it’s easier if we break it into two steps:

unsigned short\* loanPtr = (unsigned short\*)(cNamePtr + NAME\_SIZE + AGE\_SIZE);

// Print the student loan

printf("The student loan is %hu\n", \*loanPtr);

So we add **NAME\_SIZE** and **AGE\_SIZE** to the pointer to get to the correct byte and cast it as an **unsigned short\***, storing the results in **loanPtr**. Then when we dereference **loanPtr**, we will get two bytes.

A key thing to note here: we can treat memory however we like, regardless of how it was declared/defined, by using pointers of different types to point to the portion of memory that we are interested in.

## Exercise: Print a string as **int**s

To practice working with memory in different forms, try taking a string and printing it out as a series of **int** values represented by the bytes of the string:

* Call a function named **testStringToInts()** in main.
* In **testStringToInts**, have the user enter a string including spaces. (Hint: what functions have we used for input? Which take strings with spaces?)
* Then call a function called **printStringAsInts** and pass in the string the user entered.
* Write **printStringAsInts** in **array\_pointers.c** as a function that takes in a string and then prints out the **int** values represented by the bytes of the string, both in hexadecimal and in decimal. (Hint: look up the **strlen** (string length) function – for instance, google “Microsoft strlen” or something similar.)

Try out the following test values:

* Test “abcdefgh” and you should get the hex values 64636261 and 68676665 and the decimal values 1,684,234,849 and 1,751,606,885.
* Test “I/O” and you should get the hex value 4F2F49 and the decimal value 5,189,449.
* You can also try running each of your **int** values through the **intBytePrinter** function to see that you get back the original string.

Consider what limitations your function might have.

Try this practice on your own before looking at the solution below.

(No peeking!)

(Have you tried it yet, even if it didn’t fully work? Try it out!)

In main, add:

testStringToInts();

Since we want to get strings that include spaces, we should use **gets\_s** or **fgets** (not **gets**, since it can cause buffer overruns) to get the user’s input instead of **scanf("%s")** (although there are ways to use **scanf** – see <https://www.geeksforgeeks.org/taking-string-input-space-c-3-different-methods/>, for instance). Let’s use **gets\_s** for now. Then **testStringToInts** would look like:

void testStringToInts()

{

char cBuffer[BUFFER\_SIZE];

printf("Enter a string to convert to integers (spaces allowed): ");

gets\_s(cBuffer, BUFFER\_SIZE);

printStringAsInts(cBuffer);

}

With **BUFFER\_SIZE** defined as some value like:

#define BUFFER\_SIZE 14

Let’s start **printStringToInts** as a simple function that will just print the string that it received. First, prototype it as void printStringAsInts(char\*); or void printStringAsInts(const char\*); in the header file, then write it in **array\_pointers.c** as follows:

void printStringAsInts(const char\* cStringToConvert)

{

printf("'%s' converted to integers is:\n", cStringToConvert);

}

Before we go on, let’s examine the limitation of **gets\_s**. If the user enters a string that is longer than fits in **cBuffer** (including the null terminator), it will fail with an error message or fail silently depending on whether you are in debug or release mode. This prevents buffer overrun, but it’s not that user-friendly.

One way to solve this is to use **fgets** instead. **fgets** is like **gets\_s**, but takes an extra argument which is the file stream where it is looking for input (since it can be used with files, as the **f** suggests). If we use **stdin** (the standard input stream) for input, we can get input from the console as usual. Let’s change **testStringToInts** to use **fgets** instead:

printf("Enter a string to convert to integers (spaces allowed): ");

fgets(cBuffer, BUFFER\_SIZE, stdin);

You will see it runs the same as before, but now if too many characters are entered for the buffer, **fgets** reads only the characters that fit. However, **fgets** will read up to and including the newline character if the newline character fits in the buffer. This can produce undesirable results as well. – for instance, the output from our program for **"small"** is:

A screenshot of a computer

Description automatically generated

One way to search for and eliminate the newline character is as follows:

* Get the length of the string using **strlen** from **<string.h>**
* Take a look at the last character of the string before the null terminator
  + If it is the newline character **'\n'**, change it to the null terminator

Our **testStringToInts** function would now look like:

void testStringToInts()

{

char cBuffer[BUFFER\_SIZE];

int iStringLength;

printf("Enter a string to convert to integers (spaces allowed): ");

fgets(cBuffer, BUFFER\_SIZE, stdin);

iStringLength = strlen(cBuffer);

if (cBuffer[iStringLength - 1] == '\n')

{

cBuffer[iStringLength - 1] = '\0';

}

printStringAsInts(cBuffer);

}

Note that we need #include <string.h> to use **strlen**.

Now let’s complete **printStringAsInts**. The first thing that we need to do is figure out how many **int**s are in the string. To do so, we will use **strlen** again (remember to include **<string.h>** in **array\_pointers.c** as well!) and then divide by the size of an **int**. Let’s test that:

void printStringAsInts(const char\* cStringToConvert)

{

int iStringSize = strlen(cStringToConvert);

int iNumInts = iStringSize / sizeof(int);

printf("Size of string to convert is %d\n", iStringSize);

printf("It contains %d ints\n", iNumInts);

printf("'%s' converted to integers is:\n", cStringToConvert);

}

If you run this function with input **"abcdefgh"**, you will see that it says that the size is 8 and the number of **int**s is 2. If you try **"I/O"**, it will say the size is 3 and the number of **int**s is 0.

Note that the string length does not include the null terminator, just the number of actual characters. If we want to convert the entire **char** array including the null terminator, we should add one to the string length:

int iStringSize = strlen(cStringToConvert) + 1;

Then it will give size 9 and 2 **int**s for **"abcdefgh"** and size 4 and 1 **int** for **"I/O"**. Notice that the division being done is integer division. This is fine since we can’t print out 0.25 of an **int**, but it is a limitation on our function (so far). You were asked to consider what limitations your function might have, so we can just start out with that restriction – we will only print out **int**s from 4-byte chunks of the string.

Now we have a chunk of memory containing a string. We want to print it out as **int**s. The easy way to do so is just to use an **int\*** to point to the start of the string. Dereferencing such a pointer will give us an **int**. We can then just print out each **int** in a simple **for** loop as follows:

int\* iPtr = (int\*)cStringToConvert;

int i;

for (i = 0; i < iNumInts; i++)

{

printf("%x \t%d\n", \*iPtr, \*iPtr);

iPtr++;

}

Note that instead of walking the pointer along with **iPtr++** and dereferencing it at each spot, we could also use **iPtr[i]**.

Again, looking at what is happening in memory will help us:

|  |  |  |  |
| --- | --- | --- | --- |
| **Argument** | **Value in memory** | **Pointer** | **int value** |
|  | ‘\0’ = 0 |  |  |
|  | ‘h’ = 68 | ß **iPtr + 1** | 68676665  or  1,751,606,885 |
|  | ‘g’ = 67 |
|  | ‘f’ = 66 |
|  | ‘e’ = 65 |
|  | ‘d’ = 64 | ß **iPtr** | 64636261  or  1,684,234,849 |
|  | ‘c’ = 63 |
|  | ‘b’ = 62 |
| **cStringToConvert** à | ‘a’ = 61 |

Output from the program would be as follows:

Size of string to convert is 9

It contains 2 ints

'abcdefgh' converted to integers is:

64636261 1684234849

68676665 1751606885

If you want to verify that the numbers are correct, add a call to **intBytePrinter** after the **printf** and see that each **int** breaks down into the component characters:

intBytePrinter(\*iPtr);

Now, what to do about the pesky extra characters? Let’s suppose that input is **"abcdefghij"**. The **'i'** and **'j'** are ignored by the current version of the code because the string length isn’t evenly divisible by 4. The first thing we can do is test whether there are any leftover characters. To do so, we use the **%** remainder operator:

int iRemainingBytes = iStringSize % sizeof(int);

if (iRemainingBytes > 0)

{

printf("There are %d additional bytes\n", iRemainingBytes);

}

One thing we could try is to simply print another **int**. This might seem to work, but it would either go into uninitialized areas of the buffer passed in (resulting in garbage being added) or past the size of the buffer passed in (going into the stack and getting some unknown value). So instead, we can build an int with the remaining characters. Let’s look at memory first:

|  |  |  |  |
| --- | --- | --- | --- |
| **Argument** | **Value in memory** | **Pointer** | **int value** |
|  | *unknown* |  |  |
|  | ‘\0’ = 0 | After another **iPtr++**  ß **iPtr** | 6A69  or  27241 |
|  | ‘j’ = 6A |
|  | ‘i’ = 69 |
|  | ‘h’ = 68 | After **iPtr++**  ß **iPtr** | 68676665  or  1,751,606,885 |
|  | ‘g’ = 67 |
|  | ‘f’ = 66 |
|  | ‘e’ = 65 |
|  | ‘d’ = 64 | ß **iPtr** | 64636261  or  1,684,234,849 |
|  | ‘c’ = 63 |
|  | ‘b’ = 62 |
| **cStringToConvert** à | ‘a’ = 61 |

So after the **for** loop is done, **iPtr** will contain the address of the remaining bytes in **cStringToConvert**. What we need to do is take each remaining byte and put it into an **int** for printing. We can get at each remaining byte by creating a **char\*** which will contain the address in **iPtr**:

char\* cLastBytesPtr = (char\*)iPtr;

Then we can create an **int** initialized to 0 to store the results:

int iNewInt = 0;

And create a **char\*** pointer to that new **int** to get at its individual bytes:

char\* cBuildIntPtr = (char\*)&iNewInt;

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value in memory** | **Pointer** |
| **iNewInt** | 0 |  |
| 0 |  |
| 0 |  |
| 0 | ß **cBuildIntPtr** |

Then we can loop through the remaining bytes, copying the bytes from original spot (by dereferencing **cLastBytesPtr**) into the new **int** (by dereferencing **cBuildIntPtr**).

for (i = 0; i < iRemainingBytes; i++)

{

\*cBuildIntPtr = \*cLastBytesPtr;

// Move to next byte

cLastBytesPtr++;

cBuildIntPtr++;

}

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value in memory** | **Pointer** |
| **iNewInt** | 0 | ß **cBuildIntPtr** |
| ‘\0’ = 0 |  |
| ‘j’ = 6A |  |
| ‘i’ = 69 |  |

Finally, we can print out **iNewInt**.

printf("%x \t%d\n", iNewInt, iNewInt);

intBytePrinter(iNewInt);

The final version of the function would look like:

void printStringAsInts(const char\* cStringToConvert)

{

int iStringSize = strlen(cStringToConvert) + 1;

int iNumInts = iStringSize / sizeof(int);

int\* iPtr = (int\*)cStringToConvert;

int i;

printf("Size of string to convert is %d\n", iStringSize);

printf("It contains %d ints\n", iNumInts);

printf("'%s' converted to integers is:\n", cStringToConvert);

for (i = 0; i < iNumInts; i++)

{

printf("%x \t%d\n", \*iPtr, \*iPtr);

// intBytePrinter(\*iPtr);

iPtr++;

}

int iRemainingBytes = iStringSize % sizeof(int);

if (iRemainingBytes > 0)

{

char\* cLastBytesPtr = (char\*)iPtr;

int iNewInt = 0;

char\* cBuildIntPtr = (char\*)&iNewInt;

printf("There are %d additional bytes\n", iRemainingBytes);

for (i = 0; i < iRemainingBytes; i++)

{

\*cBuildIntPtr = \*cLastBytesPtr;

// Move to next byte

cLastBytesPtr++;

cBuildIntPtr++;

}

printf("%x \t%d\n", iNewInt, iNewInt);

// intBytePrinter(iNewInt);

}

}

Output from the program could look like:

Enter a string to convert to integers (spaces allowed): PointersRFun!!!!!!!

Size of string to convert is 14

It contains 3 ints

'PointersRFun!' converted to integers is:

6e696f50 1852403536

73726574 1936876916

6e754652 1853179474

There are 2 additional bytes

21 33

It would probably be better to move the building of **iNewInt** into a separate function to reduce the complexity of **printStringAsInts** and make it more readable. This change is left as an exercise for you to implement.

# Practices – enums, strings, pointers, and arrays

It is essential in this class that you practice your skills since there are only three assignments. Make sure that you try out the following practices and compare your results to the sample solutions. Note that the sample solutions follow good documentation practices – your code should too!

## enums and strings

This exercise asks you to learn and apply enums in C and various string functions.



*Sample solution:*



## Pointers and arrays

This exercise asks you to apply your knowledge of pointers and arrays.



*Sample solution:*



# Introduction to Pointers to Pointers

In this section, we will introduce pointers to pointers. This topic should give you a greater understanding of pointers and their uses in general, as well as introducing the idea of a pointer to a pointer and how it could be used.

As we know, pointers store an address of a specific location in memory. But pointers are just variables, so they are also stored in memory, so they have an address. Thus, we can store a pointer to a pointer (which stores the address of a pointer).

Let’s start by looking at a function that doesn’t require a pointer (other than a pointer to a string passed in). We will write a function **myIndexOf** that will return the position of the first occurrence in a string passed in of a character passed in, or -1 if the character is not found. This function would be called in **main** like:

#include <stdio.h>

#include <stdlib.h>

int main()

{

char cTestString[] = "It's a beautiful day, isn't it?";

int iLocation;

iLocation = myIndexOf(cTestString, 's');

printf("The position of the character is %d\n\n", iLocation);

return EXIT\_SUCCESS;

}

Here is the comment block and signature for the function:

/\*

\* Function: myIndexOf

\* Purpose: return the zero-based position of the first occurrence in a string

\* of a character passed in or -1 if the character is not found

\* Parameters: cPtr - pointer to the string to find the character in

\* cSeek - character to find

\* Returns: the first position of the character in the string

or -1 if not found

\* Assumptions: cPtr is a null-terminated string

\*/

int myIndexOf(const char\* cPtr, char cSeek)

There is more than one way to write this function, so you should think about how you could write the function yourself before looking at my solution.

My solution will check to see if we are at the end of the string (null terminator equals 0 or false). As long as we are not at the end of the string, I will compare the current character to the character to look for. If it is the character to look for, I will return its location; otherwise, I will move to the next character. If I reach the null terminator, I will return -1.

My code is:

int myIndexOf(const char\* cPtr, char cSeek)

{

int iLocation = 0;

while (\*cPtr) // while we have not reached the null terminator

{

if (\*cPtr == cSeek) // if the character is the one we seek

{

return iLocation; // return its location

}

// move to the next character/location

cPtr++;

iLocation++;

}

return -1; // return -1 if it is not found

}

Now let’s create another function called **myIndexAndCount**. It will determine the location of the first occurrence of a given character in a given string (as before), but also determine the count of how many times that character occurred in the string.

Let’s start with an erroneous way of doing this. We could try calling it in **main** like the following:

char cTestString[] = "It's a beautiful day, isn't it?";

int iLocation;

int iCount = 0;

iLocation = myIndexOfAndCount(cTestString, 's', iCount);

printf("The position of the character is %d\n", iLocation);

printf("The count of the character is %d\n\n", iCount);

The erroneous function could then be written as:

int myIndexOfAndCount(const char\* cPtr, char cSeek, int iCount)

{

int iLocation = -1; // start by assuming we cannot find the character

int iPosition = 0;

while (\*cPtr) // while we have not reached the null terminator

{

if (\*cPtr == cSeek) // if the character is the one we seek

{

if (iLocation == -1) // if it wasn't found yet

{

iLocation = iPosition; // set the location where found

}

iCount++; // increase the count of the number of times found

}

// Move to the next character/location

cPtr++;

iPosition++;

}

printf("In the function, count is %d\n", iCount);

return iLocation;

}

Do you see the problem with this function? If not, try changing the name of the parameter in the function from **iCount** to **iCountInFunction**. That change will make it more obvious that the variable in the function is not the same as the variable in **main** (or whatever the calling function is).

The output would be:

In the function, count is 2

The position of the character is 3

The count of the character is 0

In C, variables are always passed by value. When we pass **iCount** into **myIndexOfAndCount**, we are only passing a copy of its value to the function, which has its own space for **iCountInFunction**. The function updates the count inside the function, but not in the caller.

Let’s examine the stack to see why this function is erroneous:

|  |  |
| --- | --- |
| Stack frame for **myIndexOfAndCount** | **iCountInFunction** = 2 at the end of function |
| Stack frame for **main** | **iCount** = 0 |

You can see that we have two unrelated variables in different locations in memory.

The solution to this problem is to pass a pointer to the count in **main**, so the call will now look like:

iLocation = myIndexOfAndCount(cTestString, 's', &iCount);

The function will still get a copy of the value, but in this case the value is the address of **iCount** in **main**. So **myIndexOfAndCount** will now take a pointer to an **int** and dereferencing that pointer will actually change **iCount** in **main**. We would change the function as follows:

int myIndexOfAndCount(const char\* cPtr, char cSeek, int\* iCountPtr)

{

int iLocation = -1; // start by assuming we cannot find the character

int iPosition = 0;

\*iCountPtr = 0; // start the count at 0

while (\*cPtr) // while we have not reached the null terminator

{

if (\*cPtr == cSeek) // if the character is the one we seek

{

if (iLocation == -1) // if it wasn't found yet

{

iLocation = iPosition; // set the location where found

}

(\*iCountPtr)++; // increase the count of number of times found

}

// Move to the next character/location

cPtr++;

iPosition++;

}

printf("In the function, count is %d\n", \*iCountPtr);

return iLocation;

}

The comment block for the function would be as follows:

/\*

\* Function: myIndexOfAndCount

\* Purpose: return the location of the first occurrence of a given character in

\* a given string, and change the count whose address is passed in to the count

\* of the given character in the given string

\* Parameters: cPtr - string to find the character in

\* cSeek - character to be found

\* iCountPtr - address of the count to be updated

\* Returns: The first position of the character to be found or -1

\* Changes: The int at iCountPtr to the number of occurrences of the character

\*/

Notice that we dereference **iCountPtr** to change the value being pointed at to 0 initially, and then increasing by 1 each time we call (\*iCountPtr)++ (note that the parentheses are required so that the ++ applies to the value being pointed at and not the pointer).

The stack would look like the following:

|  |  |
| --- | --- |
| Stack frame for **myIndexOfAndCount** | **iCountPtr** = address of **iCount** in **main** |
| Stack frame for **main** | **iCount** = 2 at the end of the function |

This example shows one use of pointers – we can pass an address of a value to a pointer if we want the function to change that value. The same thing is true of functions like **scanf** – for instance, to read in an integer, we might have:

int iValue = 0;

scanf("%d", &iValue);

Again, note that we pass the address of **iValue** so that **scanf** can actually change the value in the calling function.

Next, we will try changing a pointer. We will write the same function as before but add a pointer which will be changed to store the address of the *last* occurrence of the character we are seeking. Again, let’s start by writing an erroneous version. We could call it in **main** with the following:

char cTestString[] = "It's a beautiful day, isn't it?";

int iLocation;

int iCount = 0;

char\* cLastPtr = NULL;

iLocation = myIndexOfAndCountAndLastPosition(cTestString, 's', &iCount,

cLastPtr);

printf("The position of the character is %d\n", iLocation);

printf("The count of the character is %d\n\n", iCount);

if (cLastPtr != NULL)

{

printf("The character at location %p is %c\n", cLastPtr, \*cLastPtr);

printf("The string starting at that character is %s\n\n", cLastPtr);

}

Do you see the problem already? **cLastPtr** will always be NULL in this code because the function **myIndexOfAndCountAndLastPosition** does not change the pointer. The last two lines printing the character at the location will never show up.

Let’s write the code for the erroneous function just to show that this problem exists:

int myIndexOfAndCountAndLastPosition(const char\* cPtr, char cSeek, int\* iCountPtr,

char\* cLastPtr)

{

int iLocation = -1; // start by assuming we cannot find the character

int iPosition = 0;

\*iCountPtr = 0; // start the count at 0

cLastPtr = NULL; // make the address of the last position NULL initially

while (\*cPtr) // while we have not reached the null terminator

{

if (\*cPtr == cSeek) // if the character is the one we seek

{

if (iLocation == -1) // if it wasn't found yet

{

iLocation = iPosition; // set the location where found

}

(\*iCountPtr)++; // increase the count of number of times found

cLastPtr = (char\*)cPtr; // set address as current last position

}

// Move to the next character/location

cPtr++;

iPosition++;

}

printf("In the function, count is %d\n", \*iCountPtr);

if (cLastPtr != NULL)

{

printf("In the function, the character pointed at is %c at %p\n",

\*cLastPtr, cLastPtr);

}

return iLocation;

}

Output would be the following (using the **x86** platform with the **Project | Properties > Configuration Properties > Linker > All Options > Randomized Base Address: No (/DYNAMICBASE:NO)** option set):

In the function, count is 2

In the function, the character pointed at is s at 0019FED7

The position of the character is 3

The count of the character is 2

Examining the stack:

|  |  |
| --- | --- |
| Stack frame for function | **cLastPtr** = 0019FED7 at the end of function |
| Stack frame for **main** | **cLastPtr** = NULL |

We see that there are really two variables **cLastPtr**, each with their own values. So how could we change **cLastPtr** in **main**? That’s right – we can pass its address to the function instead of passing the value. In the function, we now have the address of a pointer, so we will have to store it in a pointer to a pointer. Such pointers are indicated using two asterisks: **char\*\* cPtrToPtr**, where the first **\*** indicates a pointer to a **char** and the second **\*** indicates a pointer to that pointer. Dereferencing **cPtrToPtr** once gives us the pointer, while dereferencing it twice like **\*\*cPtrToPtr** gives us the character being pointed at. In **main**, the call would now look like:

iLocation = myIndexOfAndCountAndLastPosition(cTestString, 's', &iCount,

&cLastPtr);

And the correct code would be:

/\*

\* Function: myIndexOfAndCountAndLastPosition

\* Purpose: return the location of the first occurrence of a given character in

\* a given string, and change the count whose address is passed in to the count

\* of the given character in the given string, and change the pointer whose

\* address is passed in to the position of the last character found

\* Parameters: cPtr - string to find the character in

\* cSeek - character to be found

\* iCountPtr - address of the count to be updated

\* cPtrToPtr - address of the pointer to be updated

\* Returns: The first position of the character to be found or -1

\* Changes: The int at iCountPtr to the number of occurrences of the character

\* The pointer at cPtrToPtr to the address of the last occurrence or NULL

\*/

int myIndexOfAndCountAndLastPosition(const char\* cPtr, char cSeek, int\* iCountPtr,

char\*\* cPtrToPtr)

{

int iLocation = -1; // start by assuming we cannot find the character

int iPosition = 0;

\*iCountPtr = 0; // start the count at 0

\*cPtrToPtr = NULL; // make the address of the last position NULL initially

while (\*cPtr) // while we have not reached the null terminator

{

if (\*cPtr == cSeek) // if the character is the one we seek

{

if (iLocation == -1) // if it wasn't found yet

{

iLocation = iPosition; // set the location where found

}

(\*iCountPtr)++; // increase the count of number of times found

\*cPtrToPtr = (char\*)cPtr; //set address as current last position

}

// Move to the next character/location

cPtr++;

iPosition++;

}

printf("In the function, count is %d\n", \*iCountPtr);

if (\*cPtrToPtr != NULL)

{

printf("In the function, the character pointed at is %c at %p\n",

\*\*cPtrToPtr, \*cPtrToPtr);

}

return iLocation;

}

Output would now be:

In the function, count is 2

In the function, the character pointed at is s at 0019FED7

The position of the character is 3

The count of the character is 2

The character at location 0019FED7 is s

The string starting at that character is sn't it?

And the stack would now look like:

|  |  |
| --- | --- |
| Stack frame for function | **cPtrToPtr** = address of **cLastPtr** in **main** |
| Stack frame for **main** | **cLastPtr** = 0019FED7 at the end of function, which is the address of the last 's' in the string |

Key takeaways:

* All values passed to functions in C are passed by value.
* To change the original value of a variable in the calling function, we can pass the address of the variable to a pointer in the function and then dereference the pointer in the function.
* To change the original value of a pointer in the calling function, we can pass the address of the pointer to a pointer to a pointer in the function.
* A pointer to a pointer will have two asterisks like **char\*\* cPtrToPtr**.
* Dereferencing a pointer to a pointer once like **\*cPtrToPtr** will give us the original pointer.
* Dereferencing a pointer to a pointer twice like **\*\*cPtrToPtr** will give us the value pointed at.

# File I/O

Next, let’s examine file input/output in C. The first thing that we should know is that there are several file access modes in C. These include the following:

|  |  |
| --- | --- |
| **Mode** | **Description** |
| r | Opens an existing text file for reading purposes. |
| w | Opens an existing text file for writing. If the file does not exist, then a new file is created. Here your program will start writing content from the beginning of the file (overwriting existing content, if any). |
| a | Opens a text file for writing in appending mode. If the file does not exist, then a new file is created. Here your program will start appending content after the existing file content. |
| r+ | Opens a text file for both reading and writing. |
| w+ | Opens a text file for both reading and writing. It first truncates the file to zero length if the file exists; otherwise, it creates a file if the file does not exist |
| a+ | Opens a text file for both reading and writing. It creates the file if the file does not exist. The reading will start from the beginning, but writing can only be appended |

If you are going to handle binary files, then you will use the following access modes instead of the above-mentioned ones:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

## Opening and closing files

Let’s start by just trying to open and close files. Create a new project with the following **main** function:

#include <stdio.h>

#include <stdlib.h>

int main()

{

testOpenClose();

return EXIT\_SUCCESS;

}

And then create a **testOpenClose** function above **main** as follows:

void testOpenClose()

{

int iErr = demoOpenClose("foo.txt", "r");

printf("The error number returned was %d\n", iErr);

}

Finally, let’s create **demoOpenClose**. You can do so in another file using an appropriate header file.

int demoOpenClose(const char\* cFileName, const char\* cFileMode)

{

}

Before we go on, we should examine some of the aspects of how C handles files. C treats files as streams of input or output, pointed to by a **FILE** pointer (**FILE\***). **FILE** is defined in **<stdio.h>**.

Problems with files are indicated in a value labelled **errno**. Study **errno.h** on Wikipedia before proceeding:

<https://en.wikipedia.org/wiki/Errno.h>

Essentially, there is a global variable named **errno** that gets set to indicate an error. To use **errno**, include the **<errno.h>** header file.

See also the **strerror** function to get a message associated with **errno** (<https://en.cppreference.com/w/c/string/byte/strerror> or <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/strerror-strerror-wcserror-wcserror?view=msvc-170>). Include **<string.h>** to use **strerror**. Note that there is also a more secure version, **strerror\_s**. To use the original **strerror** function, include the **#define \_CRT\_SECURE\_NO\_WARNINGS** definition at the very top of your file (or in the preprocessor definitions for the project.

Thus, your **demoOpenClose** function could look like the following if put in a separate file:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include "file\_io.h"

int demoOpenClose(const char\* cFileName, const char\* cFileMode)

{

FILE\* filePtr = NULL; // Pointer to a file stream

int iErr = EXIT\_SUCCESS; // A variable to store potential error codes

return iErr;

}

Use **fopen** or **fopen\_s** to open a file.

* **fopen** returns a **FILE** pointer that must be assigned.
* If it fails, it returns **NULL** and sets the **errno** variable to indicate an error.
* Parameters to **fopen** are a **const char\*** indicating a path/filename and a **const char\*** indicating the mode (read, write, append, etc.).
* Note that **fopen** also requires the **#define \_CRT\_SECURE\_NO\_WARNINGS** definition (we will examine **fopen\_s** later).

So the body of **demoOpenClose** would be:

filePtr = fopen(cFileName, cFileMode);

if (filePtr != NULL) // Successfully opened the file

{

fclose(filePtr); // Close the file when done

printf("%s was successfully opened and closed.\n", cFileName);

}

else

{

iErr = errno;

printf("Error accessing file %s: %s\n", cFileName, strerror(iErr));

}

The initial output would be the following, since the file **foo.txt** does not exist.

Error accessing file foo.txt: No such file or directory

The error number returned was 2

Change **testOpenClose** to the following:

void testOpenClose()

{

int iErr = demoOpenClose("foo.txt", "w");

printf("The error number returned was %d\n", iErr);

}

You will see the output:

foo.txt was successfully opened and closed.

The error number returned was 0

Changing the mode to "w" created the file successfully. Note that in Visual Studio 2022, the default file location (if a path is not specified) is the same folder as where the source file is located. If you run the program using the **.exe** file from the command prompt, the default folder will be the same folder where the **.exe** file is located.

Look at the created file. You will see that it is a blank file (0 bytes). Open it in **Notepad** and add some content and save it. You will see that if you run your program again, it will wipe out the content and once again leave a file of size 0 bytes (since the mode "w" writes over existing files).

Now add some content like

Hello world

Blah blah blah

This time, save and close your **foo.txt** file and then right-click the file and choose **Properties**. Turn on the **Read-only** attribute. Now try running your program and you will get the following output:

Error accessing file foo.txt: Permission denied

The error number returned was 13

The contents of the file **foo.txt** will be unchanged. You can now change **testOpenClose** back to reading the file:

int iErr = demoOpenClose("foo.txt", "r");

You will see that it successfully opens the file for reading (whether or not the **Read-only** attribute is set).

Let’s try the **fopen\_s** function instead. In **demoOpenClose**, change the **fopen** line to:

fopen\_s(&filePtr, cFileName, cFileMode);

Note that **fopen\_s** takes the address of a **FILE\*** to be stored in a pointer to a pointer so that it can change **filePtr** in the calling program. The pointer to the pointer is needed because **fopen\_s** returns an error code (which we are ignoring) instead of a **FILE\***. You can try all the same tests that we tried before to see that the behavior of the program does not change when we use **fopen\_s**.

## Reading files

Now let’s try reading some data from files. Start by adding a new call to **testRead** in **main**:

int main()

{

//testOpenClose();

testRead();

return EXIT\_SUCCESS;

}

The **testRead** function will look like the following:

void testRead()

{

int iErr = demoRead("foo.txt", "r");

printf("The error number returned was %d\n", iErr);

}

**demoRead** will have the following signature:

int demoRead(const char\* cFileName, const char\* cFileMode);

We’ll also add a #define MAX\_SIZE 100 directive

And **demoRead** will initially just look like our **demoOpenClose** function, with a couple of extra variables thrown in:

int demoRead(const char\* cFileName, const char\* cFileMode)

{

FILE\* filePtr = NULL; // Pointer to a file stream

int iErr = EXIT\_SUCCESS; // A variable to store potential error codes

int intChar, retVal; // Note that you can create 2 variables on one line

char cBuffer[MAX\_SIZE];

char\* retPtr = NULL;

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

fclose(filePtr);

}

else // failed to open file

{

iErr = errno;

printf("Error accessing the file %s: %s\n", cFileName, strerror(iErr));

}

return iErr;

}

Note that in the line if ((filePtr = fopen(cFileName, cFileMode)) != NULL), we are both assigning **filePtr** and then comparing its return value in the same line. This structure is a common pattern in C (and some other languages) to save space. There is nothing wrong with breaking this statement into two lines (one for opening the file and one for the **if**), but you should be able to read such statements.

Now let’s try some input methods. All read operations read data from the file to the memory location indicated. Only some of the methods will read in an additional null terminator to the data in the buffer. Return types vary, so testing for success depends on the function used.

### The **fscanf** function

First, we will look at **fscanf**, which is a file version of **scanf**. Like **scanf**, it returns the number of tokens successfully read. We include the file pointer as the first argument, so after the **fopen** line, include:

retVal = fscanf(filePtr, "%99s", cBuffer); // Quits at first space

if (retVal > 0)

{

printf("retVal: %d\n", retVal);

printf("cBuffer: %s\n", cBuffer);

}

else // failed to read

{

// Check if file stream has an error using the ferror function

if (iErr = ferror(filePtr))

{

printf("Error reading from the file %s: %s\n", cFileName,

strerror(iErr));

}

else // No message, display a generic message

{

printf("Could not retrieve data\n");

}

}

fclose(filePtr);

Assuming that your **foo.txt** file contains “Hello world\nBlah blah blah”, the output from the program would be:

retVal: 1

cBuffer: Hello

The error number returned was 0

Like **scanf**, **fscanf** reads the first token up to any whitespace character, so it returns a value of 1 token read which is **Hello**. Note that the **99** in **"%99s"** is good practice for safety for a buffer of size 100. If you changed the line to:

retVal = fscanf(filePtr, "%s %s", cBuffer, cBuffer);

then it would read 2 tokens (overwriting the first token with the second token since we used **cBuffer** for both tokens) and print **World** as the resulting value of **cBuffer**.

What is there are no tokens to read? Try creating an empty file **bar.txt** (by right-clicking in the File Explorer and choosing **New > Text Document**) and running the program with **"bar.txt"** as the filename passed in. You will see that you get the message **“Could not retrieve data”**. If you print **retVal**, you will see that it is **-1**, which is the value of the special symbol **EOF** or end-of-file. **fscanf** returns **EOF** if a failure occurs before the first assignment.

### The **fgets** function

The second function that we will look at is **fgets**. We have already examined this function as an option for input from the **stdin** stream. Using **fgets** for file input just involves passing in the appropriate file stream as the last argument.

**fgets** will return a pointer to the string read or NULL if no string is read. It will include a null terminator for the string. If there is a newline character at the end of the line, that will be included too.

Change **demoRead** as follows:

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

retPtr = fgets(cBuffer, MAX\_SIZE, filePtr);

if (retPtr != NULL)

{

printf("retPtr: %p\n", retPtr);

printf("cBuffer: %p\n", cBuffer);

printf("cBuffer: %s\n", cBuffer);

}

else // failed to read...

If you run your program with **"foo.txt"**, you will get output like the following:

retPtr: 0000003CF92FF580

cBuffer: 0000003CF92FF580

cBuffer: Hello world

The error number returned was 0

Note that **retPtr** returns the address of the data read in, which is the same address as **cBuffer**. Also note that **fgets** returned the newline character at the end of the line (resulting in the blank line in the output).

If you run your program with **"bar.txt"**, you will get the message **“Could not retrieve data”**.

If you add a second **fgets** statement with **"foo.txt"** like:

retPtr = fgets(cBuffer, MAX\_SIZE, filePtr);

retPtr = fgets(cBuffer, MAX\_SIZE, filePtr);

The output will be **“Blah blah blah”** without a newline character (assuming that you didn’t press **Enter** at the end of the second line when you created your file).

### The **fgetc** function

The **fgetc** function returns a single character read or the special value **EOF**. Since **EOF** may be outside of the range of characters, **fgetc** is treated as reading an **int** rather than a **char**.

Change **demoRead** as follows:

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

intChar = fgetc(filePtr);

if (intChar != EOF)

{

printf("intChar: %d\n", intChar);

printf("intChar: %c\n", intChar);

}

else // failed to read...

Your output for **"foo.txt"** will be the number 72 and the letter **H** (which has the ASCII code **72** in decimal). Your output for **"bar.txt"** will be **“Could not retrieve data”**.

### The **fread** function

The **fread** function will read in a specified number of items of a specified size into memory. Unlike **fgets** or **fscanf("%s")**, it will not append a null terminator after reading a number of characters. **fread** will return the number of items read.

Change **demoRead** as follows:

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

retVal = fread((void\*)cBuffer, sizeof(char), 2, filePtr);

if (retVal > 0)

{

printf("retVal: %d\n", retVal);

printf("cBuffer: %s\n", cBuffer);

}

else // failed to read...

Note that **fread** expects a **void\*** as its first argument. You can cast other types of pointers as a **void\*** to avoid any warnings. The output of the program in **Debug** mode with **"foo.txt"** will be something like:

retVal: 2

cBuffer: He╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠

The error number returned was 0

Two characters are read. Note that because no null terminator was added, the **printf** of **cBuffer** just keeps going until it happens to encounter a value of zero.

The output of the program with **"bar.txt"** would be **“Could not retrieve data”**.

## Writing files

Just as there are several functions for reading from files, there are several functions for writing to files. Let’s create a new function for testing those functions. In **main**, add:

testWrite();

**testWrite** will look like:

void testWrite()

{

int iErr = demoWrite("foobar.txt", "w", "abcdefghi");

printf("The error number was %d\n", iErr);

}

And **demoWrite** will have the function signature:

int demoWrite(const char\* cFileName, const char\* cFileMode, const char\* cDataPtr);

with the following code:

int demoWrite(const char\* cFileName, const char\* cFileMode, const char\* cDataPtr)

{

FILE\* filePtr = NULL;

int iErr = EXIT\_SUCCESS;

int retVal; // Number of writes

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

fclose(filePtr);

}

else // failed to open the file for writing

{

iErr = errno;

printf("Error accessing the file %s: %s\n", cFileName, strerror(iErr));

}

return iErr;

}

### The **fwrite** function

Like **fread**, **fwrite** takes a **void\*** buffer. It then writes a number of items of a specified size from the buffer to a file pointer. It returns the number of items written.

Add the following code to **demoWrite**:

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

retVal = fwrite((void\*)cDataPtr, sizeof(char), strlen(cDataPtr),

filePtr);

if (retVal > 0) // if successful

{

printf("File written: %s\n", cFileName);

printf("retVal: %d\n", retVal);

}

else // failed to write

{

if (iErr = ferror(filePtr))

{

printf("Error writing to the file %s: %s\n",

cFileName, strerror(iErr));

}

else

{

printf("Could not write the data\n");

}

}

fclose(filePtr);

Your output should indicate that the file was written and that the return value is **9**.

You should also have created the file **"foobar.txt"**, and its contents should be 9 bytes: “abcdefghi".

Try changing the mode to **"a"** in **testWrite** and see that the file is now “abcdefghiabcdefghi”.

### The **fprintf** function

**fprintf** is a function that outputs text based on format specifiers and variables passed in, just like **printf**. It will return the number of characters written. For instance, change the following in **demoWrite** (using a mode of **"w"** in **testWrite**):

retVal = fprintf(filePtr, "%s and %d\nHi\n", cDataPtr, 456);

if (retVal > 0) // if successful...

Your output should indicate the file was written with a return value of **21**. The return value comes from the length of **cDataPtr** (9), the length of the text **" and "** (5), the length of the text for **456** (3), and the length of the text **"\nHi\n"** (4).

Your file **"foobar.txt"** should contain the matching output, although the file size may be 23 bytes. This is because the newline in Windows is written as two characters (carriage return and newline). Try removing the two **'\n'** from the **fprintf** and you will see the file size drop to 19 bytes and the contents of the file will change to “abcdefghi and 456Hi”.

Notice that the number 456 is displayed as the text version of the number, not the internal computer representation of the number. We will see how to write the binary version of the number when we look at writing binary files.

### The **fputs** function

Similar to **fgets**, there is a function **fputs** to put a string to a file. Unlike **fwrite** or **fprintf**, it returns 0 if successful. Thus, you could change your code in **demoWrite** to the following:

retVal = fputs(cDataPtr, filePtr); // returns 0 if successful

if (retVal == 0) // check for success for fputs

{

printf("File written: %s\n", cFileName);

printf("retVal: %d\n", retVal);

}

else // failed to write...

You will see that it writes the file with a return value of **0**. File size will be 9 bytes – notice that **fputs** does not add a null terminator or newline to the file output.

### The **fputc** function

The **fputc** function outputs one character to the specified file stream. It returns the value of the character if successful. Thus, you could change your code in **demoWrite** to the following:

retVal = fputc(cDataPtr[0], filePtr);

if (retVal > 0)

{

printf("File written: %s\n", cFileName);

printf("retVal: %d\n", retVal);

}

else // failed to write...

Notice that we use **cDataPtr[0]** to get just one character from the buffer. The return value in the output will be **97** (the decimal ASCII value for the character **'a'**). The file will be of size 1 byte and contain just the letter “a”.

### Writing to binary files

All the examples so far have involved reading from or writing to text files. However, we can also write to and read from binary files, particularly using **fwrite** and **fread**. Binary files will contain the internal bitwise representation of variables in the computer’s memory, so the resulting file will not be made for human consumption, but can save space.

For instance, add the following to **main**:

testWriteBinary();

where **testWriteBinary** contains the following:

void testWriteBinary()

{

int x = 123456789;

int iErr = demoWriteBinary("foobar.bin", "wb", &x);

printf("The number that was passed in (in hex) was %x\n", x);

printf("The error number that was returned was %d\n", iErr);

}

Note that the code above uses the filename **"foobar.bin"** to indicate that it is a binary file (although we could use other extensions like **.dat**), and the mode is **"wb"**.

**demoWriteBinary** would have the following signature:

int demoWriteBinary(const char\* cFileName, const char\* cFileMode, int\* iDataPtr);

And would look like most of our other functions to begin with:

int demoWriteBinary(const char\* cFileName, const char\* cFileMode, int\* iDataPtr)

{

FILE\* filePtr = NULL;

int iErr = EXIT\_SUCCESS;

int retVal; // Number of writes

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

fclose(filePtr);

}

else // failed to open the file for writing

{

iErr = errno;

printf("Error accessing the file %s: %s\n", cFileName, strerror(iErr));

}

return iErr;

}

Now let’s use **fwrite** to write information out in binary. We don’t have to do anything special with **fwrite** – just tell it to output from an **int\*** with output being the size of an **int**:

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

retVal = fwrite((void\*)iDataPtr, sizeof(int), 1, filePtr);

if (retVal > 0) // if successful

{

printf("File written: %s\n", cFileName);

printf("retVal: %d\n", retVal);

}

else // failed to write

{

// if there is an error number

if (iErr = ferror(filePtr))

{

printf("Error writing to file %s: %s\n", cFileName,

strerror(iErr));

}

else

{

printf("Could not write the data\n");

}

}

fclose(filePtr);

}

else // failed to open the file for writing...

The output from the program would be that the file **foobar.bin** was written with a return value of **1** (one integer, which is the size of the item passed into **fwrite**). The number 123456789 in hex would be **75bcd15**.

The size of **foobar.bin** will be 4 bytes (the size of an **int**), which you will notice is shorter than the 9 characters of the text representation. If you open it with **Notepad++**, the contents will be “**NAK**Í[**BEL**”, where **NAK** is the special character for negative acknowledge (ASCII hex value 15), **BEL** is the special character for the bell (ASCII hex value 07), and the characters Í[ having the ASCII hex values CD and 5b. By comparing these characters with the hex value **07 5b cd 15**, we see that the characters correspond to the integer value that the computer is storing (from least significant byte to most significant byte).

Don’t worry if you have trouble understanding the characters in the binary file – binary files are meant for the computer, not for people. You just need to know that an **int** is 4 bytes, and 4 bytes have been written to the file.

## File I/O exercise

To ensure that you understand file I/O in C, particularly for binary files, try doing the following exercise:

Write a function that will write out 3 records of information entered by the user. Each record consists of:

* SIN # (integer)
* Name (char array of MAX\_NAME\_SIZE)

This information is to be entered using **scanf** (remember to get the **'\n'** off the buffer after **scanf**) and **gets\_s** or **fgets** from **stdin** (so spaces can be included in the name).

After entering a record, write the data out to a binary file. Repeat until 3 records have been written out (although your code should be easily modifiable to handle a different number of records). Do some error checking after each write. Close the file. You may want to write the number of records as the first integer in the file (see below) to make it easy to determine how many records are in the file.

Then write code in a separate function to read from the file and display the information contained within. Note that you will either need to write the number of records as the first integer in the file (change your function above as necessary) and read that in, or read each record and then check for end of file after each record using the **feof** function (which you can look up).

If you get that done, modify your read code to get user input of an integer and determine whether that SIN # exists in a record in the file and what its associated name is. (You may want to use the **fseek** function to skip over names that are not associated with the SIN # that was input – you can look up how **fseek** works.)

Here is a sample solution for the exercise – don’t forget to try it on your own first!

First, we will change **main** to call a new test function, **testExerciseWrite**:

testExerciseWrite();

The **testExerciseWrite** function could look like our other test functions (although you may choose a different way of writing it – for instance, you could hardcode the file name instead of passing it in):

void testExerciseWrite()

{

int iErr = exerciseBinaryWrite("exercise.dat", "wb");

printf("The error number returned was %d\n", iErr);

}

The function prototype might then look like the following (with the **#define** directives added as well):

#define MAX\_NAME\_SIZE 20

#define NUM\_RECORDS 3

int exerciseBinaryWrite(const char\* cFileName, const char\* cFileMode);

The actual function would start by opening the file as usual, so it would contain something like the following:

int exerciseBinaryWrite(const char\* cFileName, const char\* cFileMode)

{

FILE\* filePtr = NULL;

int iErr = 0;

int iSin, retSin, retName;

char cNameArray[MAX\_NAME\_SIZE];

unsigned short sNumRecords = NUM\_RECORDS;

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

// TODO: Add code here to get input and save records

fclose(filePtr);

}

else // failed to open file

{

iErr = errno;

printf("Error accessing the file %s: %s\n", cFileName,

strerror(errno));

}

return iErr;

}

Now we can replace the TODO comment above with code. First, we can write the number of records to the file:

// Write out the number of records

// Could check return value to check for a successful write

fwrite((void\*)&sNumRecords, sizeof(unsigned short), 1, filePtr);

If you try this now, you should see that it will create a 2-byte file. Looking at it in **Notepad++** will show the characters **ETX** and **NUL**, corresponding to 0003 in hex (with the least significant byte written first – ETX or End Of Text is the ASCII code for 3). Note that if you open the file in **Notepad**, you will see Asian characters – this particular code is understood by **Notepad** as meaning the characters following are part of the Unicode set representing Asian characters.

Note that using an **unsigned short** will limit the number of records that can be stored to a maximum of 65535 (but storing the number of records only takes 2 bytes). You could have used an **int** or an **unsigned int** instead (but then storing the number of records would take 4 bytes).

Then, we can get input from the user and save each record to the file:

for (int i = 0; i < sNumRecords; i++)

{

printf("Enter SIN number: "); // prompt user for SIN #

scanf("%d", &iSin); // or "%i" - same as "%d"

// Clear the buffer of the newline character

// Note that this will only work if user hit return immediately

// after entering the SIN# - could use loop instead

getc(stdin);

printf("Enter the name: "); // Get the name

gets\_s(cNameArray, MAX\_NAME\_SIZE); // or use fgets

// Write the record to the file

retSin = fwrite((void\*)&iSin, sizeof(int), 1, filePtr);

retName = fwrite((void\*)cNameArray,

sizeof(char) \* MAX\_NAME\_SIZE, 1, filePtr);

// Check the results

if (retSin == 0 || retName == 0)

{

if (iErr = ferror(filePtr))

{

printf("Error writing file %s: %s\n",

cFileName, strerror(iErr));

}

else

{

printf("Could not write data\n");

}

}

}

If you run the program and enter some data, you should find that your file is of the size 74 bytes (assuming that you also set MAX\_NAME\_SIZE to 20 and used an **unsigned short** for the number of records). 2 bytes for the number of records + 3 \* (**sizeof(int)** + MAX\_NAME\_SIZE characters) = 2 + 3\*(4 + 20) = 2 + 3\*24 = 2 + 72 = 74 bytes.

If you open **exercise.dat**, you will see that there are garbage characters after any name that is not of **MAX\_NAME\_SIZE** - 1 (since the null terminator does get saved in this case). These garbage characters just reflect whatever was in **cNameArray** before data entry. For instance, here is one possible run of the program:

A screenshot of a computer

Description automatically generated

Here are the contents of **exercise.dat** (as shown in **Notepad++** to show the special characters):



This solution uses fixed-length records – every name takes up **MAX\_NAME\_SIZE** characters regardless of the name’s size. It would also be possible to use variable-length records, where we saved the size of the name before saving the name and then saved only the number of characters necessary for the name. We will look at an example of variable-length records later on, but for this exercise, we will continue to use fixed-length records.

Note that if you used **fgets** instead of **gets\_s**, your file might look different – you may see some newline characters (if you didn’t get rid of the newline that **fgets** reads in at the end of each line), and you may see the remaining characters from longer names at the end of shorter names. Also, the extra garbage characters may change depending on whether you are in **Debug** or **Release** mode.

Next, we can write a function to read the records back from **exercise.dat**. In **main**, add:

testExerciseRead();

**testExerciseRead** could look like the following. Again, you may have made other design decisions (such as hardcoding the file name **"exercise.dat"**), and we will change it later to allow entry of a SIN # to find.

void testExerciseRead()

{

int iErr = exerciseBinaryRead("exercise.dat", "rb");

printf("The error number returned was %d\n", iErr);

}

The function prototype for **exerciseBinaryRead** would look like:

int exerciseBinaryRead(const char\* cFileName, const char\* cFileMode);

And **exerciseBinaryRead** would start out as:

int exerciseBinaryRead(const char\* cFileName, const char\* cFileMode)

{

FILE\* filePtr = NULL;

int iErr = 0;

int iSin, retSin, retName;

char cNameArray[MAX\_NAME\_SIZE];

unsigned short sNumRecords = 0;

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

// TODO: read the data

fclose(filePtr);

}

else

{

printf("Could not open file\n");

}

// Not really setting this yet, but you (the student)

// can do this boring task as an exercise

return iErr;

}

Because we stored the number of records at the start of the file, we can read that first. Replace the TODO above with the following:

// Read in the number of records

fread((void\*)&sNumRecords, sizeof(unsigned short), 1, filePtr);

Now that we have the number of records, we can do a loop through the records:

// Loop through the records

for (int i = 0; i < sNumRecords; i++)

{

// Read in the SIN #

fread((void\*)&iSin, sizeof(int), 1, filePtr);

fread((void\*)cNameArray, sizeof(char) \* MAX\_NAME\_SIZE,

1, filePtr);

printf("SIN %d has name %s\n", iSin, cNameArray);

}

Note that in this case, we saved the null terminator with each name, so we can print out the names without any extra effort.

Finally, we would like to change the program so that it asks for a SIN # and seeks the corresponding name. Change **testExerciseRead** as follows:

void testExerciseRead()

{

int iSin = 0;

printf("Enter the SIN# of the person to find: ");

scanf("%d", &iSin);

int iErr = exerciseBinaryRead("exercise.dat", "rb", iSin);

printf("The error number returned was %d\n", iErr);

}

This version of the code will read an integer from the user into **iSin**. Note that we can leave the newline or other characters on **stdin**, since we will not be asking for more input. We will need to change **exerciseBinaryRead** so that it takes in an int, so its new function prototype is:

int exerciseBinaryRead(const char\* cFileName, const char\* cFileMode, int iSeek);

And the completed function might look like the following (changes highlighted):

int exerciseBinaryRead(const char\* cFileName, const char\* cFileMode, int iSeek)

{

FILE\* filePtr = NULL;

int iErr = 0;

int iSin, retSin, retName;

char cNameArray[MAX\_NAME\_SIZE];

unsigned short sNumRecords = 0;

int found = 0;

if ((filePtr = fopen(cFileName, cFileMode)) != NULL)

{

// Read in the number of records

fread((void\*)&sNumRecords, sizeof(unsigned short), 1, filePtr);

// Loop through the records

for (int i = 0; i < sNumRecords; i++)

{

// Read in the SIN #

fread((void\*)&iSin, sizeof(int), 1, filePtr);

if (iSin == iSeek)

{

found = fread((void\*)cNameArray,

sizeof(char) \* MAX\_NAME\_SIZE, 1, filePtr);

printf("SIN %d has name %s\n", iSin, cNameArray);

}

else

{

fseek(filePtr, MAX\_NAME\_SIZE, SEEK\_CUR);

}

}

if (!found)

{

printf("SIN # not found\n");

}

fclose(filePtr);

}

else

{

printf("Could not open file\n");

}

// Not really setting this yet, but you (the student)

// can do this boring task as an exercise

return iErr;

}

# Dynamic Memory Allocation

We have seen the limitations of stack memory. Variables that are allocated on the stack only last as long as the stack frame where they are declared, so we can’t safely pass back memory from the stack to a calling function.

The solution to this problem is to use memory from another section of memory – the heap. The heap is a chunk of memory (its size depends on multiple factors), where the memory can be requested and is held on to until it is released. Thus, memory on the heap lasts between function calls.

To access memory from the heap in C, we use functions like **malloc** – see <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/malloc?view=msvc-170> or <https://en.cppreference.com/w/c/memory/malloc>. The **malloc** function is available in **<stdlib.h>** or **<malloc.h>**, so you must include one of those header files in your program.

**malloc** takes in a number of bytes and allocates that amount of space on the heap. It returns a **void\*** (a void pointer to the first byte of that memory. A void pointer can point to any type of memory – for safety, you can cast its result to any pointer type, particularly since **malloc** once returned a **char\***). If it cannot allocate memory (perhaps because there is no space on the heap), **malloc** returns **NULL**.

To explore dynamically allocated memory, let’s start by creating a function **printInts** to print out an array of **int**s. This step will not involve any dynamically allocated memory, and we’ve done this kind of thing before, but we’ll change it slightly to print out a comma-separated list of integers on the same line.

Create the following **main** function in a source file like **program.c**:

#include <stdlib.h>

#include <stdio.h>

#include "dma.h"

int main()

{

int iArray[] = { 10, 20, 30 };

printInts(iArray, sizeof(iArray) / sizeof(int));

return EXIT\_SUCCESS;

}

Then create **dma.h** with the following:

#ifndef DMA\_H

#define DMA\_H

// Function to print an array of ints

void printInts(int\* iArrayPtr, int iSize);

#endif // !DMA\_H

And create the source file **dma.c** that will contain our utility functions like **printInts**. Note that **printInts** here assumes that the size of the array is at least 1, since it will print out **iSize** - 1 elements followed by a comma and then print out the last element followed by a newline:

#include <stdio.h>

#include "dma.h"

// Function to print an array of ints

void printInts(int\* iArrayPtr, int iSize)

{

printf("The integers are: ");

for (int i = 0; i < iSize - 1; i++)

{

printf("%d, ", \*iArrayPtr);

iArrayPtr++;

}

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

}

Your output should be:

The integers are: 10, 20, 30

## Dynamically allocating a block of **int**s

So far, we have just created an array on the stack in **main** and passed a reference to it to the **printInts** function. This step was just to test **printInts**. You can comment those out in **main** and add a call to **testIntDMA** where we will try using some dynamically allocated memory:

testIntDMA();

where **testIntDMA** looks like:

void testIntDMA()

{

int iSize = 3;

int\* iBufferPtr = getInts(iSize);

printInts(iBufferPtr, iSize);

}

The call to **getInts** requires us to return an array. We know that we can’t return it from the stack, so **getInts** will have to return memory allocated on the heap with **malloc**. Add its prototype to **dma.h**:

// Return an array of ints that are dynamically allocated

int\* getInts(int iNumInts);

And in **dma.c**, add the **#include <malloc.h>** or **#include <stdlib.h>** so **malloc** is available, then write code using **malloc** to return an array of **int**s of the correct size:

#include <malloc.h> // Memory allocation - required for DMA

// Return an array of ints that are dynamically allocated

int\* getInts(int iNumInts)

{

int\* iBufferPtr = (int\*)malloc(iNumInts \* sizeof(int));

return iBufferPtr;

}

We will change **getInts** to initialize the memory later, but if you try running this now in **Debug** mode, you will see something like:

The integers are: -842150451, -842150451, -842150451

**malloc** just returns memory from the heap – it does not initialize that memory. The values in memory depend on what was in memory beforehand – in this case, some garbage data.

Note that **malloc** returns a size in bytes, so if we want several **int**s, we must multiply the number of **int**sby the size of an **int**.

Before we change **getInts** to initialize the memory, let’s go back to **testIntDMA**. We got a reference to the memory on the heap and used it to print the integers, but what happens to that memory now? The memory is still allocated on the heap, so we have created a “memory leak” – memory that was allocated but never released. If enough memory is allocated and not released, a memory leak can cause the program to slow down or even crash.

Once we’re done with dynamically allocated memory, free it using the function **free** – see <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/free?view=msvc-170> or <https://en.cppreference.com/w/c/memory/free>. **free** takes in a pointer to the memory to be released, so it should be called as such:

int iSize = 3;

int\* iBufferPtr = getInts(iSize);

printInts(iBufferPtr, iSize);

free(iBufferPtr);

Note that the amount of memory to be freed does not need to be provided. The system tracks memory on the heap, including how much was allocated at a particular spot.

Once we have freed the memory, the pointer still points to the memory that was allocated. So you could still call **printInts(iBufferPtr, iSize);** after **free**. Try it out and you will see that it still prints out data (quite possibly different garbage data) and that there is no compile error – Visual Studio will highlight **iBufferPtr** to say that you are using uninitialized memory, but the program compiles and runs without error.

For safety, it is thus better to set the pointer to **NULL** immediately after freeing the memory:

iBufferPtr = NULL;

//printInts(iBufferPtr, iSize); // would crash - accessing NULL pointer

With the pointer set to **NULL**, any incorrect attempt to access the pointer will crash the program (which is safer than accessing uninitialized memory). If you want to erase the data (perhaps for security reasons), call **memset** (described in a moment) *before* freeing the memory.

Let’s go back to **getInts**. We have noticed that the block of memory on the heap is not initialized. If you want to initialize it all to the same value, you can use the **memset** function – see <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memset-wmemset?view=msvc-170> or <https://en.cppreference.com/w/c/string/byte/memset>.

**memset** is in **<string.h>**, so include that header file. It takes in a pointer to a destination, a value to set in the memory at that destination, and a size. Set the allocated memory to 0 in **getInts** as follows:

memset(iBufferPtr, 0, iNumInts \* sizeof(int));

You will now see the following output:

The integers are: 0, 0, 0

Note that Visual Studio may warn you that **iBufferPtr** may be **NULL**. Recall that **malloc** may return **NULL** if it cannot allocate the memory on the heap. It is best to test for this condition before using this memory. Let’s do so, but also add code to allow the user to enter data into the dynamically allocated array before returning it instead of using **memset**:

// Return an array of ints that are dynamically allocated

int\* getInts(int iNumInts)

{

int\* iBufferPtr = (int\*)malloc(iNumInts \* sizeof(int));

//memset(iBufferPtr, 0, iNumInts \* sizeof(int));

if (iBufferPtr != NULL) // malloc will return NULL if unsuccessful

{

// Load data into the array

for (int i = 0; i < iNumInts; i++)

{

printf("Enter integer %d: ", i + 1);

scanf\_s("%d", iBufferPtr + i);

}

}

return iBufferPtr;

}

We now have a program that gets a dynamically allocated chunk of memory, sets its values to user input, returns it to the calling function, prints out the data from the calling function, and frees the memory when it is done.

Let’s try a few more things in the **testIntDMA** function. What happens if we go beyond the memory that **malloc** allocated? Such code would overflow the buffer, creating potential errors and security vulnerabilities. But C will allow such code. It is up to the programmer to ensure that you never go beyond the bounds of the buffer. Thus, the following is legal code:

int iSize = 3;

int\* iBufferPtr = getInts(iSize);

printInts(iBufferPtr, iSize);

iBufferPtr[4] = 9;

printInts(iBufferPtr, iSize + 3);

free(iBufferPtr);

iBufferPtr = NULL;

The code creates a buffer of 3 **int**s, but then sets the item at the 5th position (**iBuffer[4]** – remember that arrays are zero-based), and prints 6 items. Your input and output may look like the following – note that we are allowed to set and print beyond the buffer with no error:

Enter integer 1: 1

Enter integer 2: 2

Enter integer 3: 3

The integers are: 1, 2, 3

The integers are: 1, 2, 3, -33686019, 9, 115

You may ask why C allows such code. C puts the responsibility for memory management in the hands of the programmer. This allows for efficient programs – no extra processes are needed to check if we are going out of bounds. The extra efficiency is achieved at the cost of the potential errors and security risk if the programmer doesn’t handle the memory correctly.

Comment out those lines. Let’s now try to add another **int** to the end of the array. Call the following after the first **printInts** call:

// Add another int to the array

iBufferPtr = addInt(iBufferPtr, &iSize);

printInts(iBufferPtr, iSize);

Add **addInt** to **dma.h** as follows:

// Add an int to the end of a dynamic array

int\* addInt(int\* iArrayPtr, int\* iSizePtr);

The goal of the **addInt** function is to add space for an additional integer at the end of the dynamically allocated array, allow data entry for that integer, and return the updated space. **addInt** will also take in a pointer to an integer with the current size of the array, and update that size. The code will start as follows:

// Add an int to the end of a dynamic array

// Purpose --> Takes an array, increases its size by one. Asks the user for

// an int and adds it to the end.

// iArrayPtr --> The original dynamically allocated array.

// iSizePtr --> Pointer to the original size of the array.

// Used to increase the passed-in size.

int\* addInt(int\* iArrayPtr, int\* iSizePtr)

{

// Create a new array

int\* iNewPtr = (int\*)malloc((\*iSizePtr + 1) \* sizeof(int));

// Copy the data from old array to new array using the memcpy function

memcpy(iNewPtr, iArrayPtr, (\*iSizePtr) \* sizeof(int));

// Add new int to the end of the new array

printf("Enter another integer: ");

scanf\_s("%d", iNewPtr + \*iSizePtr);

// Increment the size

(\*iSizePtr)++;

return iNewPtr;

}

Note the use of **memcpy** (from **<string.h>**) to copy memory to a destination (**iNewPtr**) from a source **(iArrayPtr**) of a given size. For more on the **memcpy** function, see <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-170> or <https://en.cppreference.com/w/c/string/byte/memcpy>.

The code given will run successfully, but do you see a problem? We create a new chunk of memory using **malloc** in the first line and return it, but what happens to the old chunk of memory? After calling it as **iBufferPtr = addInt(iBufferPtr, &iSize);**, we no longer have a pointer to the old chunk of memory, so we have created a memory leak.

We could avoid this by using a different pointer in the calling function and freeing both pointers there, but if the intended use of **addInt** is to update the buffer, we could also free the original memory in **addInt** as follows:

// Copy the data from old array to new array using the memcpy function

memcpy(iNewPtr, iArrayPtr, (\*iSizePtr) \* sizeof(int));

// Note: The array originally passed in is freed after its contents are copied.

// Free the original array.

free(iArrayPtr);

iArrayPtr = NULL;

We could also add code to check if **malloc** was successful (didn’t return **NULL**) before doing **memcpy** and freeing the old memory, but I will leave that as practice for you.

Another way to avoid this is to use the **realloc** function, available since the C89 specification. See <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/realloc?view=msvc-170> or <https://en.cppreference.com/w/c/memory/realloc>.

Try adding the following to **testIntDMA**:

iBufferPtr = addIntRealloc(iBufferPtr, &iSize);

Add **addIntRealloc** to **dma.h**:

// Add an int to the end of a dynamic array using realloc (C89)

int\* addIntRealloc(int\* iArrayPtr, int\* iSizePtr);

And write **addIntRealloc** as follows:

// Add an int to the end of a dynamic array using realloc (C89)

int\* addIntRealloc(int\* iArrayPtr, int\* iSizePtr)

{

// Reallocates the area of memory by expanding/contracting the old block

// or allocating a new memory block and freeing the old block

int\* iNewPtr = (int\*)realloc(iArrayPtr, (\*iSizePtr + 1) \* sizeof(int));

if (iNewPtr != NULL)

{

// Add new int to the end of the new array

printf("Enter another integer: ");

scanf\_s("%d", iNewPtr + \*iSizePtr);

// Increment the size

(\*iSizePtr)++;

}

else

{

iNewPtr = iArrayPtr;

}

return iNewPtr;

}

Note that **realloc** takes in a pointer to data on the heap and a new size for that data. It will then either expand or contract the buffer if space is available, or it will create a new buffer/copy over the data/free the old buffer. Thus, **realloc** does the work for us of copying data and freeing the old buffer if needed.

**realloc** will return a pointer to the updated buffer or **NULL** if it fails, so we should test that before proceeding (as shown above).

## Detecting memory leaks

Since the system keeps track of memory allocation on the heap (so that it knows how much memory to release with the **free** function), we can actually test if there are any memory leaks after the program has run. To do so, do the following:

* Add the following code to the start of **program.c**:

// for detecting memory leaks

#define \_CRTDBG\_MAP\_ALLOC

#include <crtdbg.h>

This code will first tell the system to keep a map of allocated memory and then add functions in **<crtdbg.h>** that will allow us to work with those functions.

* At the end of **main** (before the **return** statement), add the following:

\_CrtDumpMemoryLeaks();

* Finally, run the program with debugging (**F5** or use the **Local Windows Debugger** on the toolbar – do NOT use **Ctrl+F5** for **Start Without Debugging**).

If you run the program, you can then look at the **Show output from: Debug** pane to see if it reports any memory leaks. Right now, the program should not report anything:

The thread 0x149c has exited with code 0 (0x0).

'2024DMA.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.

'2024DMA.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.

The thread 0x3fb4 has exited with code 0 (0x0).

The thread 0x60d0 has exited with code 0 (0x0).

The program '[20192] 2024DMA.exe' has exited with code 0 (0x0).

But if you comment out the calls to **free** in **testIntDMA** and **addInt**, you will see something like the following (contents of the leaked memory will depend on your input – I used 1 2 3 4 5):

Detected memory leaks!

Dumping objects ->

{83} normal block at 0x00000285E2B7E1F0, 20 bytes long.

Data: < > 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00

{79} normal block at 0x00000285E2B7DE80, 12 bytes long.

Data: < > 01 00 00 00 02 00 00 00 03 00 00 00

Object dump complete.

Note that the first memory leak occurred in **addInt**, losing 12 bytes from the original array when the new array is created and returned. **addIntRealloc** didn’t have a memory leak since **realloc** freed the memory for us (or just resized the block). Then in **testIntDMA**, if the **free(iBufferPtr);** is commented out, there is a second memory leak of size 20 bytes (the final resized array).

You might wonder why we don’t just free all the memory at the end since we can detect where there are memory leaks. In fact, modern operating systems will do so, freeing any resources the program had open before closing and exiting the program. But while the program is running, we could be building up a large amount of memory usage, so we need to free the memory ourselves as we are using it.

Programming languages like Java have “garbage collection”. With garbage collection, in addition to keeping track of memory on the heap (allocated whenever the **new** keyword is used), references to the allocated memory are tracked. When an object no longer has any references to it, its memory is “garbage” and can be “collected” and freed as the program is running. C does not have garbage collection because its goal is to be as efficient as possible, so freeing memory is left to the programmer.

## Dynamically allocating strings

Let’s now turn our attention to creating a function that returns a dynamically allocated string of the minimum required size. Add the following to **main**:

testDynamicString();

**testDynamicString** will look like the following to begin with – we will clean it up later:

void testDynamicString()

{

char\* cFirstPtr = NULL;

char\* cLastPtr = NULL;

cFirstPtr = getStringDynamic("Enter first name: ");

cLastPtr = getStringDynamic("Enter last name: ");

printf("The first name is %s\n", cFirstPtr);

printf("The last name is %s\n", cLastPtr);

}

So our goal is to create a function **getStringDynamic** that will take in a prompt and return a pointer to an array of **char**s containing the string that the user entered. The function prototype will be:

// Return a pointer to a dynamically allocated string or NULL if it fails

char\* getStringDynamic(const char\* cPrompt);

We will also set a **MAX\_STRING\_SIZE** for the function:

#define MAX\_STRING\_SIZE 256

Let’s start by doing it the wrong way – using space on the stack. **getStringDynamic** would look like:

char\* getStringDynamic(const char\* cPrompt)

{

char cReturn[MAX\_STRING\_SIZE];

printf("%s", cPrompt);

gets\_s(cReturn, MAX\_STRING\_SIZE);

return cReturn;

}

You will see that it compiles with no errors (only a warning C4172: returning address of local variable or temporary: cReturn) and runs, but the output may not be correct. Here is one sample run:

Enter first name: Michael

Enter last name: Grzesina

The first name is Grzesina

The last name is Grzesina

Because memory was allocated on the stack for **cReturn** and that memory was returned, it could be overwritten (in the example above, the first name was overwritten by the last name). To avoid that, we need to return a pointer to dynamically allocated memory on the heap and manage it correctly.

Furthermore, the current **getStringDynamic** always returns **MAX\_STRING\_SIZE** bytes. Since we are getting a string from the user, we can determine its size and return only the necessary number of bytes for the string and the null terminator.

Input could still come from an array on the stack which would then be copied to dynamically allocated memory of the correct size, but let’s practice by using dynamically allocated memory for the initial input as well. Our code would start like:

// Return a pointer to a dynamically allocated string or NULL if it fails

char\* getStringDynamic(const char\* cPrompt)

{

// Pointer to a dynamically allocated string

char\* cTempPtr = NULL;

cTempPtr = (char\*)malloc(MAX\_STRING\_SIZE \* sizeof(char));

if (cTempPtr) // same as (cTempPtr != NULL) since NULL is 0 and 0 is false

{

printf("%s", cPrompt);

gets\_s(cTempPtr, MAX\_STRING\_SIZE);

}

return cTempPtr;

}

This code will work correctly, but it still returns **MAX\_STRING\_SIZE** bytes to the calling function. Instead, we would prefer to return the exact size required. Update the code as follows:

// Return a pointer to a dynamically allocated string or NULL if it fails

char\* getStringDynamic(const char\* cPrompt)

{

// Pointer to a dynamically allocated string

char\* cTempPtr = NULL;

// Pointer to a dynamically allocated string of exact size required

char\* cReturnPtr = NULL;

int iSizeDestination;

cTempPtr = (char\*)malloc(MAX\_STRING\_SIZE \* sizeof(char));

if (cTempPtr) // same as (cTempPtr != NULL) since NULL is 0 and 0 is false

{

printf("%s", cPrompt);

gets\_s(cTempPtr, MAX\_STRING\_SIZE);

// Allocate a new buffer based on the actual size of the input

// Always remember to allocate space for the null terminator

iSizeDestination = strlen(cTempPtr) + 1;

cReturnPtr = (char\*)malloc(iSizeDestination \* sizeof(char));

if (cReturnPtr)

{

// Copy the data from temp array to return array

strcpy\_s(cReturnPtr, iSizeDestination, cTempPtr);

}

}

return cReturnPtr;

}

This code should compile and run with no problems. Did you see any mistakes we’ve made? Consider how the memory was allocated and whether it was freed. There are problems both in **getStringDynamic** and **testDynamicString**. The memory dump shows:

Detected memory leaks!

Dumping objects ->

{84} normal block at 0x000001814346E3C0, 9 bytes long.

Data: <Grzesina > 47 72 7A 65 73 69 6E 61 00

{83} normal block at 0x000001814346F0A0, 256 bytes long.

Data: <Grzesina > 47 72 7A 65 73 69 6E 61 00 FE FE FE FE FE FE FE

{82} normal block at 0x000001814346E370, 8 bytes long.

Data: <Michael > 4D 69 63 68 61 65 6C 00

{79} normal block at 0x000001814346EF60, 256 bytes long.

Data: <Michael > 4D 69 63 68 61 65 6C 00 FE FE FE FE FE FE FE FE

Object dump complete.

The blocks of 256 bytes are from **getStringDynamic**, where we took 256 bytes of initial space for **cTempPtr** but never freed them before returning. The blocks of 9 and 8 bytes (size will vary depending on input – I used “Michael” and “Grzesina”) are from **testDynamicString**, which took the dynamically allocated strings of the minimum size and printed them but never freed them.

Modify **getStringDynamic** as follows:

if (cReturnPtr)

{

// Copy the data from temp array to return array

strcpy\_s(cReturnPtr, iSizeDestination, cTempPtr);

}

free(cTempPtr);

cTempPtr = NULL;

Note that we should free the memory for **cTempPtr** even if **cReturnValue** returned **NULL**, since if we are at that point in the code, **cTempPtr** must have allocated the memory successfully. An alternative would be to only free the memory if **cReturnPtr** is not **NULL** and include an **else** clause for if (cReturnPtr) that sets **cReturnPtr** to **cTempPtr** so that something is returned (although it is too large).

We also need to modify **testDynamicString**. In addition to freeing the memory, we can test if the memory was allocated successfully:

void testDynamicString()

{

char\* cFirstPtr = NULL;

char\* cLastPtr = NULL;

cFirstPtr = getStringDynamic("Enter first name: ");

cLastPtr = getStringDynamic("Enter last name: ");

if (cFirstPtr)

{

printf("The first name is %s\n", cFirstPtr);

free(cFirstPtr);

cFirstPtr = NULL;

}

if (cLastPtr)

{

printf("The last name is %s\n", cLastPtr);

free(cLastPtr);

cLastPtr = NULL;

}

}

An alternative way to write **getDynamicString** would be to use **realloc**. Writing the function using **realloc** is left for you as practice.

## Memory allocation exercise

Try the following memory allocation exercise to see if you understand how dynamically allocated memory works, in addition to working with memory with different kinds of pointers.

In **main**, add a function call to **testNameAndSIN** as follows:

testNameAndSIN();

Then in **testNameAndSIN**, do the following:

* Call a function named **getNameAndSIN**. Create **getNameAndSIN** separately so that it will dynamically allocate space for a name of some **MAX\_NAME\_SIZE** plus an **int** for the SIN #, then ask the user to enter a name and an **int** for the SIN # stored in the dynamically allocated space, and finally return the dynamically allocated space. Note that you will use a fixed record size of **MAX\_NAME\_SIZE** for the name and not dynamically allocate just enough room for the name entered.
* Call a function named **printNameAndSIN**. Create **printNameAndSIN** separately so that it will take in a buffer and print out the name and SIN #, assuming that the name is stored in a space of **MAX\_NAME\_SIZE** and the SIN # is stored in the **int** following that space.
* Handle dynamically allocated memory appropriately.

Try it out on your own before looking at the following solution. You will need three parts – the calls and handling dynamic memory in **testNameAndSIN** and the two functions **getNameAndSIN** and **printNameAndSIN**.

Let’s examine **testNameAndSIN** first. We need to call **getNameAndSIN** and store the pointer that it returns. There are two types of memory in the buffer, but we can use a **char\*** to point to the start of the memory buffer (since it starts with an array of **char**s for the name).

char\* cBuffer = getNameAndSIN();

Then we can test if the memory was allocated and returned successfully. If so, we can call **printNameAndSIN** to print the contents of the buffer and then free the buffer:

if (cBuffer)

{

printNameAndSIN(cBuffer);

free(cBuffer);

cBuffer = NULL;

}

Thus, the function prototypes for **getNameAndSIN** and **printNameAndSIN** are as follows:

// A function that creates a buffer with a name (MAX\_NAME\_SIZE) and SIN#

char\* getNameAndSIN();

// A function to display a name and SIN# from a buffer

void printNameAndSIN(char\* cBuffer);

We also need to define **MAX\_NAME\_SIZE**:

#define MAX\_NAME\_SIZE 30

I chose the number 30 arbitrarily. The larger the number, the bigger the name that can be stored, but the more memory is occupied by the record even for small names.

**getNameAndSIN** will need to allocate space. If space if successfully allocated, it needs to read in data for the name and the SIN # in the correct locations. Finally, it will return the memory allocated (or **NULL**):

char\* getNameAndSIN()

{

char\* cNameAndSINPtr = malloc(MAX\_NAME\_SIZE \* sizeof(char) + sizeof(int));

int iLength;

if (cNameAndSINPtr)

{

// Read in the name

printf("Enter the name: ");

fgets(cNameAndSINPtr, MAX\_NAME\_SIZE, stdin);

// fgets puts in the newline character which we might not want.

// You can overwrite it with a null terminator

iLength = strlen(cNameAndSINPtr);

if (cNameAndSINPtr[iLength - 1] == '\n')

{

cNameAndSINPtr[iLength - 1] = '\0';

}

// Read in the SIN#

printf("Enter the SIN#: ");

scanf\_s("%d", (int\*)(cNameAndSINPtr + MAX\_NAME\_SIZE));

}

return cNameAndSINPtr;

}

* Note the memory allocation – it gets a size in **char**s followed by the size of an **int** to get the full record size.
* This version of the function also uses **fgets** and then overwrites any newline character read in – you could have also used **gets\_s**.
* Finally, the SIN# is read in by adding **MAX\_NAME\_SIZE** to the **char** pointer to skip over the name, then casting it as an **int\*** because we wish to read in an **int**’s worth of data.

**printNameAndSIN** is much simpler, since we just need to navigate the buffer passed in. The name is at the start of the buffer and contains a null terminator, so we can just print out the name as usual. To get the SIN #, we again add **MAX\_NAME\_SIZE** to the **char** pointer, cast it to an **int\***, and dereference the **int\*** to get the name out:

void printNameAndSIN(char\* cBuffer)

{

int\* iSINPtr = (int\*)(cBuffer + MAX\_NAME\_SIZE);

printf("The name is %s\n", cBuffer);

printf("The SIN# is %d\n", \*iSINPtr);

}

The program should work as requested now. Note that if you forgot to free the memory in **testNameAndSIN**, you would see a memory leak of size **MAX\_NAME\_SIZE** + 4 (the size of an int) which would contain the name you entered plus the SIN # entered.

## Dynamically allocate a concatenated string

We have already seen **strcat** in Exercise 1. **strcat** concatenates a source string to the end of a destination string, but the destination string must already be big enough to handle the additional source string and must be a modifiable location, so you can’t do **strcat("Hello ", "world");**.

Let’s try writing a function **concatenateStrings** that will return dynamically allocated memory of the correct size for the two strings it is given, combining the two strings together. First, we will add a **testStringConcatenate** function call to **main**:

testStringConcatenate();

The **testStringConcatenate** function will call the **concatenateStrings** function and print out the resulting string while correctly handling dynamic memory:

void testStringConcatenate()

{

char\* cResultString = concatenateStrings("Hello ", "world");

if (cResultString)

{

printf("%s\n", cResultString);

free(cResultString);

cResultString = NULL;

}

}

The function prototype for **concatenateStrings** will be as follows:

// A function to concatenate two strings. No memory is freed within the function.

// Returns a dynamically allocated string.

char\* concatenateStrings(const char\* cStringPtr1, const char\* cStringPtr2);

To concatenate the two strings, we need to figure out the length of each string; allocate enough memory for the dynamically allocated string joining the two strings together (including the null terminator); copy over the first string; and then copy or concatenate the second string. The code could look like:

// A function to concatenate two strings. No memory is freed within the function.

// Returns a dynamically allocated string.

char\* concatenateStrings(const char\* cStringPtr1, const char\* cStringPtr2)

{

int iLengthFirst = strlen(cStringPtr1);

int iLengthSecond = strlen(cStringPtr2);

char\* cReturnString = (char\*)malloc(iLengthFirst + iLengthSecond + 1);

if (cReturnString)

{

// Just to be safe, initialize our string to empty

cReturnString[0] = '\0';

// Copy the first string to destination

strcpy\_s(cReturnString, iLengthFirst + 1, cStringPtr1);

// Copy the second string to the end of the first string

// strcat\_s(cReturnString, iLengthFirst + iLengthSecond + 1, cStringPtr2);

strcpy\_s(cReturnString + iLengthFirst, iLengthSecond + 1, cStringPtr2);

}

return cReturnString;

}

This code should successfully compile and run without memory leaks.

# Review and Miscellaneous

Now is a good time to review some information about memory and file I/O, along with adding some miscellaneous examples about passing values between functions.

## Review of memory and pointers

Let’s start by reviewing memory and pointers. Create a new project with a **main** function and call **testMemory**:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

testMemory();

return EXIT\_SUCCESS;

}

Add the following **testMemory** function:

void testMemory()

{

unsigned short int sArray[] = { 0xFFFF, 0, 0xFEFE, 0x0101, 0x6261, 0x4344 };

int arraySize = sizeof(sArray) / sizeof(unsigned short);

// print the original array

for (int i = 0; i < arraySize; i++)

{

printf("%4x/%5hu\t", \*(sArray + i), \*(sArray + i)); // or sArray[i]

// sArray++; // is illegal - can't change the start of an array

}

printf("\n");

// Change the array

// print the modified array

for (int i = 0; i < arraySize; i++)

{

printf("%4x/%5hu\t", \*(sArray + i), \*(sArray + i)); // or sArray[i]

}

printf("\n");

}

Note that **sArray** contains 6 **unsigned short** values with their hexadecimal values given (starting with **0x**) except for **0**, which is **0x0000** in hex. Printing them with the **%4x** and **%5hu** specifiers will ensure that the hexadecimal representation use 4 characters and the **unsigned short** will print in decimal using 5 characters, so the output will line up nicely.

Note also that **sArray++** is illegal syntax, so we can’t walk **sArray** through the array. Why is this? Because **sArray** is not a pointer, but the actual array in memory. You cannot change its address.

We would like to modify the array by calling a function **modify\_my\_array**. What needs to be passed into this function?

For an array in C, you must pass in both the array reference (as a pointer) and the size of the array. If you fail to pass in the size, you do not know how much memory is used. In the function where the array exists, we can use **sizeof(sArray) / sizeof(unsigned short)**, but what we are passing to the function is the address of the start of the array, not the array itself. So our call to **modify\_my\_array** looks like:

// Change the array

modify\_my\_array(sArray, arraySize);

Let’s create **modify\_my\_array** in a separate file **functions.c** with a corresponding header file **functions.h** which will look like:

#ifndef FUNCTIONS\_H // header guard - make sure file is only added once

// if FUNCTIONS\_H is already defined, skip the file

#define FUNCTIONS\_H // if FUNCTIONS\_H was not defined, define it

// so the file will be skipped next time

// Change the array passed in to examine memory

void modify\_my\_array(unsigned short int\* sArrayPtr, int iArraySize);

#endif // !FUNCTIONS\_H

Review the purpose and structure of header files as shown above.

**modify\_my\_array** will start by subtracting one from every item in the array. There are several ways to do this, but I will try walking the pointer through the array. **functions.c** will look like:

#include <stdio.h>

#include "functions.h"

void modify\_my\_array(unsigned short int\* sArrayPtr, int iArraySize)

{

for (int i = 0; i < iArraySize; i++)

{

(\*sArrayPtr)--; // subtract one from each array element

sArrayPtr++; // move to the next element

}

}

Don’t forget to also include **"functions.h"** in your calling file as well.

Note that the loop is moving **sArrayPtr** through the array, one item at a time. The parentheses in **(\*sArrayPtr)--** are required since we want to modify the value, not the pointer.

Your output should currently look like:

ffff/65535 0/ 0 fefe/65278 101/ 257 6261/25185 4344/17220

fffe/65534 ffff/65535 fefd/65277 100/ 256 6260/25184 4343/17219

The only weird thing about this output is that **0** becomes **ffff** or **65535**. That is because we defined the array elements as **unsigned short int**s, which are 2 bytes on our system. As an **unsigned** type, negative numbers can’t be stored, so instead we wrap over to the largest possible value for 2 bytes.

Let’s add some print statements to examine the memory. For demonstration, I will use the **x86** platform so that the addresses will be smaller. I will also set the **Project | Properties > Configuration Properties > Linker > All Options > Randomized Base Address: No (/DYNAMICBASE:NO)** to **No** to get a consistent set of memory addresses.

Change **modify\_my\_array** to the following:

void modify\_my\_array(unsigned short int\* sArrayPtr, int iArraySize)

{

printf("sArrayPtr starts as: %p\n", sArrayPtr);

for (int i = 0; i < iArraySize; i++)

{

(\*sArrayPtr)--; // subtract one from each array element

sArrayPtr++; // move to the next element

}

printf("sArrayPtr becomes: %p\n", sArrayPtr);

}

My output becomes:

ffff/65535 0/ 0 fefe/65278 101/ 257 6261/25185 4344/17220

sArrayPtr starts as: 0019FE00

sArrayPtr becomes: 0019FE0C

fffe/65534 ffff/65535 fefd/65277 100/ 256 6260/25184 4343/17219

A couple of notes:

* **sArrayPtr** exists in the stack frame for **modify\_my\_array**, but it contains the address of **sArray** in the stack frame for **testMemory**.
* At the end of **modify\_my\_array**, **sArrayPtr** is pointing just past the end of the memory for **sArray** because of the last **sArrayPtr++**.

Let’s examine our memory – first, the stack frame for **modify\_my\_array** at the start of the function:

|  |  |  |
| --- | --- | --- |
| **Address** | **Byte Values (in hex)** | **Variable or parameter** |
|  | 00 |  |
|  | 19 |  |
|  | FE |  |
| *Unknown* | 00 | **sArrayPtr** |
|  | 00 |  |
|  | 00 |  |
|  | 00 |  |
| *Unknown* | 06 | **iArraySize** |

Next, the stack frame for **modify\_my\_array** at the end of the function:

|  |  |  |
| --- | --- | --- |
| **Address** | **Byte Values (in hex)** | **Variable or parameter** |
|  | 00 |  |
|  | 19 |  |
|  | FE |  |
| *Unknown* | 0C | **sArrayPtr** |
|  | 00 |  |
|  | 00 |  |
|  | 00 |  |
| *Unknown* | 06 | **iArraySize** |

Next, the stack frame for **testMemory** before calling **modify\_my\_array**. Note that we don’t know exactly where **arraySize** is located – it may be before or after the array, and there may be some space (especially in **Debug** mode) between it and the array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **Byte Value (in hex)** | **Variable** | **Statement** | **Pointer in function** |
|  | 00 |  |  |  |
|  | 00 |  |  |  |
|  | 00 |  |  |  |
| *Unknown* | 06 | **arraySize** |  |  |
| 0019FE0B | 43 |  |  |  |
| 0019FE0A | 44 | [5] |  |  |
| 0019FE09 | 62 |  |  |  |
| 0019FE08 | 61 | [4] |  |  |
| 0019FE07 | 01 |  |  |  |
| 0019FE06 | 01 | [3] |  |  |
| 0019FE05 | FE |  |  |  |
| 0019FE04 | FE | [2] |  |  |
| 0019FE03 | 00 |  |  |  |
| 0019FE02 | 00 | [1] |  |  |
| 0019FE01 | FF |  |  |  |
| 0019FE00 | FF | **sArray** |  | **ß sArrayPtr** |

Finally, the stack frame for **testMemory** after **modify\_my\_array** – I will ignore **arraySize** since it is not changed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **Byte Value (in hex)** | **Variable** | **Statement** | **Pointer in function** |
| 0019FE0C | *Unknown* |  |  | **ß sArrayPtr** |
| 0019FE0B | 43 |  |  |  |
| 0019FE0A | 43 | [5] | **sArrayPtr++** |  |
| 0019FE09 | 62 |  |  |  |
| 0019FE08 | 60 | [4] | **sArrayPtr++** |  |
| 0019FE07 | 01 |  |  |  |
| 0019FE06 | 00 | [3] | **sArrayPtr++** |  |
| 0019FE05 | FE |  |  |  |
| 0019FE04 | FD | [2] | **sArrayPtr++** |  |
| 0019FE03 | FF |  |  |  |
| 0019FE02 | FF | [1] | **sArrayPtr++** |  |
| 0019FE01 | FF |  |  |  |
| 0019FE00 | FE | **sArray** | **sArrayPtr++** |  |

**sArrayPtr** points beyond the end of the array when the loop in **modify\_my\_array** is done. How could we get back the starting value? Simply do some pointer math:

sArrayPtr -= iArraySize;

printf("sArrayPtr is reset: %p\n", sArrayPtr);

Let’s try some other pointer math to practice it and see how we can manipulate values. Create a couple of pointers at the top of **modify\_my\_array** as follows:

char\* cPtr = (char\*)sArrayPtr;

int\* iPtr = (int\*)sArrayPtr;

Note the casting – **cPtr** will work with the array one byte at a time (because the size of a **char** is 1 byte), while **iPtr** will work with the array one int at a time.

Let’s do some stuff with the pointers:

\*cPtr = 0x55; // or 'U'

\*(cPtr + 3) = 0; // same as cPtr[3]

iPtr += 2;

\*iPtr = 0x56555453;

printf("sArrayPtr:\t%p\n", sArrayPtr);

printf("cPtr:\t\t%p\n", cPtr);

printf("iPtr:\t\t%p\n", iPtr);

printf("%x\n", \*(sArrayPtr + 2));

printf("%p\n", sArrayPtr + 2);

printf("%x\n", \*iPtr);

printf("%p\n", &iPtr);

Note that **\*cPtr** will dereference exactly one **char** (one byte), so setting it will not affect the adjacent byte. The same is true of **\*(cPtr + 3)** – it will affect the fourth byte only. Note that we are not changing **cPtr**, just adding a value to it before we dereference it.

Because we do pointer math to **iPtr** with **iPtr += 2;**, it will now point two **int**s from the start of the array, which is 8 bytes on our system. So the first three lines of the new output is:

sArrayPtr: 0019FE00

cPtr: 0019FE00

iPtr: 0019FE08

**sArrayPtr** has been reset to the start of the array, and **cPtr** also points to the start of the array. **iPtr** points two integers from the start of the array, so the address that it contains is 8 more than the other addresses.

When we print **\*(sArrayPtr + 2)**, we do not change **sArrayPtr**, but we add the size of 2 **unsigned short**s to it before dereferencing it, so we get **fefd** as the output (recall that the original value **fefe** had one subtracted from it in the loop at the start of **modify\_my\_array**).

On the other hand, if we just print **sArrayPtr + 2**, we get an address – **0019FE04** – which is 4 bytes from the start of **sArray** (the size of 2 **unsigned short**s).

Next, dereferencing **iPtr** gives us the **int** value at its address, which we set to **56555453**. Since **iPtr** is an **int\***, dereferencing it gives 4 bytes (the size of an **int** on our system).

Note that the address of **&iPtr** is different than the value in **iPtr**. It is the address in the stack frame of **modify\_my\_array** where the pointer is located, while the value in **iPtr** is the address in the stack frame of **testMemory** where the array is located. Output for the last four lines would look like:

fefd **ß value of \*(sArrayPtr + 2)**

0019FE04 **ß address in sArrayPtr + 2**

56555453 **ß value of \*iPtr**

0019FCE4 **ß address of iPtr**

Here is the final stack frame for **modify\_my\_array**:

|  |  |  |
| --- | --- | --- |
| **Address** | **Byte Values (in hex)** | **Variable or parameter** |
|  | 00 |  |
|  | 19 |  |
|  | FE |  |
| *Unknown* | 00 | **sArrayPtr** |
|  | 00 |  |
|  | 00 |  |
|  | 00 |  |
| *Unknown* | 06 | **iArraySize** |
|  | 00 |  |
|  | 19 |  |
|  | FE |  |
| *Unknown* | 00 | **cPtr** |
|  | 00 |  |
|  | 19 |  |
|  | FE |  |
| 0019FCE4 | 08 | **iPtr** |

And the final stack frame for **testMemory**, ignoring **arraySize**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **Byte Value (in hex)** | **Variable** | **Statement** | **Pointer in function** |
| 0019FE0B | 56 |  |  |  |
| 0019FE0A | 55 | [5] |  |  |
| 0019FE09 | 54 |  |  |  |
| 0019FE08 | 53 | [4] | **iPtr += 2;** | **ß iPtr** |
| 0019FE07 | 01 |  |  |  |
| 0019FE06 | 00 | [3] |  |  |
| 0019FE05 | FE |  |  |  |
| 0019FE04 | FD | [2] |  | **ß sPtr + 2** |
| 0019FE03 | 00 |  |  | **ß cPtr + 3** |
| 0019FE02 | FF | [1] |  |  |
| 0019FE01 | FF |  |  |  |
| 0019FE00 | 55 | **sArray** |  | **ß cPtr, sArrayPtr** |

All of which is reflected in the final output:

ffff/65535 0/ 0 fefe/65278 101/ 257 6261/25185 4344/17220

sArrayPtr starts as: 0019FE00

sArrayPtr becomes: 0019FE0C

sArrayPtr is reset: 0019FE00

sArrayPtr: 0019FE00

cPtr: 0019FE00

iPtr: 0019FE08

fefd **ß value of \*(sArrayPtr + 2)**

0019FE04 **ß address in sArrayPtr + 2**

56555453 **ß value of \*iPtr**

0019FCE4 **ß address of iPtr**

ff55/65365 ff/ 255 fefd/65277 100/ 256 5453/21587 5655/22101

A good understanding of arrays, pointers, and pointer math will ensure that you know how to work with memory maps.

## Global variables, local variables, and pass by reference

Let’s continue our review and look at miscellaneous topics by comparing the use of global variables, local variables, and pass-by-reference variables. Let’s start with a simple function called **enterGradesGlobal** called in **main** as follows:

enterGradesGlobal();

and defined in **program.c** (or whatever file you have **main** in) as follows:

void enterGradesGlobal()

{

int grade = getGradeGlobal();

while (grade != 0)

{

printf("Grade is %d and status is %s\n", grade,

grade >= PASSING\_GRADE ? "PASS" : "FAIL");

grade = getGradeGlobal();

}

}

We will also define **PASSING\_GRADE** as **50** in our **functions.h** file:

#define PASSING\_GRADE 50

And add the function prototype for **getGradesGlobal**:

// Get a grade and update the global count of grades - returns the grade

int getGradeGlobal();

Our **getGradeGlobal** function will get and return a grade from 0 to 100. Right now, it won’t update the global count of grades. It will be placed in **functions.c** and look like:

int getGradeGlobal()

{

int grade;

printf("Enter a grade: ");

scanf\_s("%d", &grade);

while (grade < 0 || grade > 100)

{

printf("Invalid grade. Must be between 0 and 100.\n");

printf("Enter a grade: ");

scanf\_s("%d", &grade);

}

return grade;

}

You should now be able to run your program and enter validated grades until you enter a grade of 0. Sample output is as follows:

Enter a grade: 101

Invalid grade. Must be between 0 and 100.

Enter a grade: -1

Invalid grade. Must be between 0 and 100.

Enter a grade: 100

Grade is 100 and status is PASS

Enter a grade: 50

Grade is 50 and status is PASS

Enter a grade: 49

Grade is 49 and status is FAIL

Enter a grade: 0

Suppose that we would also like to keep track of the number of grades entered. We could do so in **enterGradesGlobal**, but let’s suppose that we want to actually keep track of the calls of the function **getGradeGlobal**. One way to do so is with a global variable. Add the following global variable at the top of **program.c**:

int globalCount; // global variables are Bad - try to avoid them

We can then update **enterGradesGlobal** as follows:

void enterGradesGlobal()

{

int grade = getGradeGlobal();

while (grade != 0)

{

printf("Grade is %d and status is %s\n", grade,

grade >= PASSING\_GRADE ? "PASS" : "FAIL");

printf("Grades entered so far: %d\n", globalCount);

grade = getGradeGlobal();

}

printf("Total grades entered: %d\n", globalCount);

}

Running the program now would just display **0** for **globalCount** all the time, so we would like to change **globalCount** in **getGradeGlobal**. The change to the code is straightforward, but there is a problem:

int getGradeGlobal()

{

int grade;

printf("Enter a grade: ");

scanf\_s("%d", &grade);

while (grade < 0 || grade > 100)

{

printf("Invalid grade. Must be between 0 and 100.\n");

printf("Enter a grade: ");

scanf\_s("%d", &grade);

}

globalCount++;

return grade;

}

You will notice that **globalCount** is undefined in **functions.c**, so the code won’t compile. That is because **functions.c** doesn’t know about **globalCount** in the other file. To inform **functions.c** that **globalCount** is defined in the other file, add the following line of code at the top of **functions.c**:

extern int globalCount;

The **extern** keyword indicates that the global variable already exists externally (in another file). The code should now run as follows:

Enter a grade: 100

Grade is 100 and status is PASS

Grades entered so far: 1

Enter a grade: 50

Grade is 50 and status is PASS

Grades entered so far: 2

Enter a grade: 49

Grade is 49 and status is FAIL

Grades entered so far: 3

Enter a grade: 0

Total grades entered: 4

This has achieved our goal, but it used a global variable, which is generally considered to be bad practice – it’s harder to maintain and easier to make errors with. So we could try doing the same thing with a local variable. Add the following to **main**:

enterGradesLocal();

and

void enterGradesLocal()

{

int count = 0; // on the stack frame for enterGradesLocal

int grade = getGradeLocal();

while (grade != 0)

{

printf("Grade is %d and status is %s\n", grade,

grade >= PASSING\_GRADE ? "PASS" : "FAIL");

printf("Grades entered so far: %d\n", count);

grade = getGradeLocal();

}

printf("Total grades entered: %d\n", count);

}

And in **functions.h** and **functions.c**:

// Get a grade and update the count of grades - returns the grade

int getGradeLocal();

// Get a grade and update the count of grades - returns the grade

int getGradeLocal()

{

int grade;

int count = 0; // on the stack frame for getGradeLocal

printf("Enter a grade: ");

scanf\_s("%d", &grade);

while (grade < 0 || grade > 100)

{

printf("Invalid grade. Must be between 0 and 100.\n");

printf("Enter a grade: ");

scanf\_s("%d", &grade);

}

count++;

printf("Local grade count: %d\n", count);

return grade;

}

Do you see the problem already? Running the program produces the following output:

Enter a grade: 100

Local grade count: 1

Grade is 100 and status is PASS

Grades entered so far: 0

Enter a grade: 50

Local grade count: 1

Grade is 50 and status is PASS

Grades entered so far: 0

Enter a grade: 49

Local grade count: 1

Grade is 49 and status is FAIL

Grades entered so far: 0

Enter a grade: 0

Local grade count: 1

Total grades entered: 0

The variable **count** is in **enterGradesLocal** is unrelated to the variable **count** in **getGradeLocal**. A new copy of **count** in **getGradeLocal** is created every time we call **getGradeLocal**, so it always restarts at **0** and is incremented to **1**. Meanwhile, **count** in **getGradeLocal** is always **0**.

We could partially fix this as follows – in **getGradeLocal**, change the declaration of **count** as follows:

static int count = 0; // declared in BSS memory area instead of stack frame

Now the output is:

Enter a grade: 100

Local grade count: 1

Grade is 100 and status is PASS

Grades entered so far: 0

Enter a grade: 50

Local grade count: 2

Grade is 50 and status is PASS

Grades entered so far: 0

Enter a grade: 49

Local grade count: 3

Grade is 49 and status is FAIL

Grades entered so far: 0

Enter a grade: 0

Local grade count: 4

Total grades entered: 0

By using a **static** variable in **getGradeLocal**, we only ever have one copy of **count** in **getGradeLocal**, so the local count is now correct. If all we wanted to do is keep track of the number of times that **getGradeLocal** was called, this would be an acceptable solution. But it still doesn’t pass the value back to the calling function **enterGradesLocal**, where count remains **0** all the time.

Let’s try a third version. In this version, we will pass the address of count to our function to get the grade. (We can’t just pass the value of count, because that would be just like creating a local copy of count.) In **main**, add:

enterGradesPassByRef();

And in **program.c**, define **enterGradesPassByRef** as follows:

void enterGradesPassByRef()

{

int count = 0;

int grade = getGradePassByRef(&count);

while (grade != 0)

{

printf("Grade is %d and status is %s\n", grade,

grade >= PASSING\_GRADE ? "PASS" : "FAIL");

printf("Grades entered so far: %d\n", count);

grade = getGradePassByRef(&count);

}

printf("Total grades entered: %d\n", count);

}

And in **functions.h** and **functions.c**, add:

// Get a grade and update the count of grades passed in - returns the grade

int getGradePassByRef(int\* iCountPtr);

int getGradePassByRef(int\* iCountPtr)

{

int grade;

printf("Enter a grade: ");

scanf\_s("%d", &grade);

while (grade < 0 || grade > 100)

{

printf("Invalid grade. Must be between 0 and 100.\n");

printf("Enter a grade: ");

scanf\_s("%d", &grade);

}

(\*iCountPtr)++;

printf("Local grade count: %d\n", \*iCountPtr);

return grade;

}

Output will be:

Enter a grade: 100

Local grade count: 1

Grade is 100 and status is PASS

Grades entered so far: 1

Enter a grade: 50

Local grade count: 2

Grade is 50 and status is PASS

Grades entered so far: 2

Enter a grade: 49

Local grade count: 3

Grade is 49 and status is FAIL

Grades entered so far: 3

Enter a grade: 0

Local grade count: 4

Total grades entered: 4

This is known as “pass by reference”. We have a **count** variable in **enterGradesPassByRef**, and we pass its address to a pointer in the function **getGradePassByRef**. This allows us to dereference the pointer in **getGradePassByRef** and change the value of the **count** variable in the calling function. So now we can return the grade, but also change the count.

Pass by reference is often useful in cases where we want to avoid global variables, but update more than one thing (if only one thing needs to be updated, it can often be changed by using the return value).

## Additional file I/O examples

Next in our miscellaneous topics, let’s examine some other options when doing file I/O. We will try writing out a set of records using various functions and processes, and then reading back those records. In **main**, add:

writeRecords();

And in **program.c**, add:

void writeRecords()

{

FILE\* filePtr = NULL;

errno\_t err = fopen\_s(&filePtr, "data.dat", "wb");

char messageBuffer[BUFFER\_SIZE];

if (err == 0)

{

//writeData(filePtr);

fclose(filePtr);

}

else

{

strerror\_s(messageBuffer, BUFFER\_SIZE, errno);

printf("Error %d opening file: %s\n", errno, messageBuffer);

}

}

You will also need to define **BUFFER\_SIZE** (in **program.c** or **functions.h**):

#define BUFFER\_SIZE 100

And in **program.c**, add the following includes for **errno** and **strerror\_s**:

#include <errno.h>

#include <string.h>

You will notice several things about **writeRecords**:

* We are opening the file directly from here, as opposed to our other examples where we passed in the file name and file mode and opened the file in the called function. Here, we will eventually pass the open file pointer to a function called **writeData** (currently commented out).
* We are using **fopen\_s**, which returns an error code and thus needs to pass the address of the file pointer (to a pointer to a file pointer) so that **filePtr** can be changed here.
* We are using **strerror\_s**, which changes a buffer of **BUFFER\_SIZE** to include the error message, as opposed to **strerror**, which just returns the string with the error message.

If you run this program, it should create an empty file named **data.dat**. If you change the properties of **data.dat** so that it is read-only, on the next run, you should see an error 13 saying that permission is denied to write to that file.

Uncomment **writeData(filePtr);** and let’s write that function. In **functions.h**, add the prototype:

// Asks for # and name and write records to file passed in

void writeData(FILE\* filePtr);

And add a definition for a **MAX\_NAME\_SIZE**:

#define MAX\_NAME\_SIZE 30

We will now define **writeData** to get a number of records and write them out:

void writeData(FILE\* filePtr)

{

int id;

char cNameBuffer[MAX\_NAME\_SIZE];

printf("Enter an id number (0 to quit): ");

scanf\_s("%d", &id);

getchar(); // get the newline character off stdin

while (id != 0)

{

printf("Enter a name: ");

//gets(cNameBuffer); // insecure because it may cause buffer overflow

gets\_s(cNameBuffer, MAX\_NAME\_SIZE);

fwrite((void\*)&id, sizeof(int), 1, filePtr);

fwrite((void\*)cNameBuffer, sizeof(char), MAX\_NAME\_SIZE, filePtr);

// should check for successful writing but I'll leave that to you

printf("Enter an id number (0 to quit): ");

scanf\_s("%d", &id);

getchar(); // get the newline character off stdin

}

}

Notes about **writeData**:

* It will write an indefinite number of records to the file (since it will write records until the user enters 0). Therefore, we cannot write the number of records at the start of the file as easily as we could if we knew the number of records. (It would be possible to write the number of records at the start and update that value using **fseek** to get between the start and end of the file.)
* The records being written are currently fixed-length records – each name is of **MAX\_NAME\_SIZE**, regardless of the size of the name entered.

Here is some output from the program:

Enter an id number (0 to quit): 123456789

Enter a name: Mike Grzesina

Enter an id number (0 to quit): 987654321

Enter a name: Coralee Kaban

Enter an id number (0 to quit): 999999999

Enter a name: Max Number And Name Size!!!!!

Enter an id number (0 to quit): 0

And the corresponding contents of **data.dat**:

Í[Mike Grzesina þþþþþþþþþþþþþþþþ±hÞ:Coralee Kaban þþþþþþþþþþþþþþþþÿÉš;Max Number And Name Size!!!!!

Now how can we read the records back? Because there is no number of records written at the start of the file, we have a couple of choices:

* Because the records are fixed length, we could find the file size in bytes and divide by the record size in bytes.
* Or we can read each record, looking for the end of file. This is what we will do.

So in **main**, comment out **writeRecords();** (so we don’t overwrite the file) and add a **readRecords** call:

//writeRecords();

readRecords();

In **program.c**, add **readRecords**:

void readRecords()

{

FILE\* filePtr = NULL;

errno\_t err = fopen\_s(&filePtr, "data.dat", "rb");

char messageBuffer[BUFFER\_SIZE];

if (err == 0)

{

readData(filePtr);

fclose(filePtr);

}

else

{

strerror\_s(messageBuffer, BUFFER\_SIZE, errno);

printf("Error %d opening file: %s\n", errno, messageBuffer);

}

}

Note that **readRecords** is almost the same as **writeRecords** except that we are using the file mode **"rb"**. We could probably write a utility function to do both read and write, but we will leave them as separate functions.

In **functions.h**, add the **readData** function prototype:

// Read data from the file passed in

void readData(FILE\* filePtr);

And in **functions.c**, add the **readData** function:

void readData(FILE\* filePtr)

{

int id;

char cNameBuffer[MAX\_NAME\_SIZE];

fread((void\*)&id, sizeof(int), 1, filePtr);

while (!feof(filePtr))

{

printf("ID %d read in\n", id);

fread((void\*)cNameBuffer, sizeof(char), MAX\_NAME\_SIZE, filePtr);

printf("Name %s read in\n", cNameBuffer);

fread((void\*)&id, sizeof(int), 1, filePtr);

}

}

Notes on the **readData** function:

* Note that we try to read in the **id** before we test for the end of file using the **feof** function. This is because **feof** only returns true when an attempt has been made to read past the end of the file. Try it out with an empty file instead of **data.dat**. You will see that the program continues to work correctly. But if you move the first **fread** into the **while** loop, it will not recognize that you are at the end of the file.
* Note that we are reading in names of the maximum size which include the null terminator, so we can just print out the name afterwards.

So far, we have used fixed-length records, which means that all the records use the maximum space for the name. This could be wasteful if we have a lot of shorter names, especially if **MAX\_NAME\_SIZE** is set to a large value. What could be done as a solution to this problem?

Instead of using fixed-length records, we could use variable-length records where only the characters needed for the name (including the null terminator) are stored. But then how do we know how long the name will be? We could just read until the null terminator, or we could save the name size in the file before the name. We will try the approach of saving the name size.

Because the **MAX\_NAME\_SIZE** is less than 256, we can store the name size in an **unsigned char**. Change **writeData** as follows:

void writeData(FILE\* filePtr)

{

int id;

unsigned char nameLength;

char cNameBuffer[MAX\_NAME\_SIZE];

printf("Enter an id number (0 to quit): ");

scanf\_s("%d", &id);

getchar(); // get the newline character off stdin

while (id != 0)

{

printf("Enter a name: ");

//gets(cNameBuffer); // insecure because it may cause buffer overflow

gets\_s(cNameBuffer, MAX\_NAME\_SIZE);

fwrite((void\*)&id, sizeof(int), 1, filePtr);

nameLength = (unsigned char)strlen(cNameBuffer);

fwrite((void\*)&nameLength, sizeof(unsigned char), 1, filePtr);

fwrite((void\*)cNameBuffer, sizeof(char), nameLength + 1, filePtr);

// should check for successful writing but I'll leave that to you

printf("Enter an id number (0 to quit): ");

scanf\_s("%d", &id);

getchar(); // get the newline character off stdin

}

}

And change **readData** to read in only the specified name size:

void readData(FILE\* filePtr)

{

int id;

char cNameBuffer[MAX\_NAME\_SIZE];

unsigned char nameLength;

fread((void\*)&id, sizeof(int), 1, filePtr);

while (!feof(filePtr))

{

printf("ID %d read in\n", id);

fread((void\*)&nameLength, sizeof(unsigned char), 1, filePtr);

fread((void\*)cNameBuffer, sizeof(char), nameLength + 1, filePtr);

printf("Name %s read in\n", cNameBuffer);

fread((void\*)&id, sizeof(int), 1, filePtr);

}

}

Here is a sample run of both **writeRecords** and **readRecords**:

Enter an id number (0 to quit): 123456789

Enter a name: Mike Grzesina

Enter an id number (0 to quit): 987654321

Enter a name: Coralee Kaban

Enter an id number (0 to quit): 999999999

Enter a name: Max Number And Name Size!!!!!

Enter an id number (0 to quit): 0

ID 123456789 read in

Name Mike Grzesina read in

ID 987654321 read in

Name Coralee Kaban read in

ID 999999999 read in

Name Max Number And Name Size!!!!! read in

And here are the contents of the file:

Í[Mike Grzesina ±hÞ:Coralee Kaban ÿÉš;Max Number And Name Size!!!!!

Note that the file no longer contains any extra characters with the names. The file size has dropped from 102 bytes to 73 bytes.

Let’s update **readData** so that it looks for a desired id number.

void readData(FILE\* filePtr)

{

int id, iNumberToFind;

char cNameBuffer[MAX\_NAME\_SIZE];

unsigned char nameLength;

do

{

printf("Enter an id number to find (0 to quit): ");

scanf\_s("%d", &iNumberToFind);

fread((void\*)&id, sizeof(int), 1, filePtr);

while (!feof(filePtr))

{

printf("ID %d read in - %s\n", id,

id == iNumberToFind ? "FOUND" : "not found");

fread((void\*)&nameLength, sizeof(unsigned char), 1, filePtr);

// skip over the name from current location

fseek(filePtr, nameLength + 1, SEEK\_CUR);

fread((void\*)&id, sizeof(int), 1, filePtr);

}

fseek(filePtr, 0, SEEK\_SET); // SEEK\_SET is the start of the file

} while (iNumberToFind != 0);

}

Notes:

* This version checks all the id numbers for a matching id and just displays **FOUND** when it finds a match. We could change the program to stop when it finds a matching id, or only display the matching ids.
* This version skips over the names entirely using **fseek**. We could print out the name when the id is found.
* To return to the start of the file for each search, we use the **fseek** function with an offset of **0** from the start of the file (**SEEK\_SET**).

For more information on **feof** and **fseek**, see:

* <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/feof?view=msvc-170> or <https://en.cppreference.com/w/c/io/feof>
* <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/fseek-fseeki64?view=msvc-170> or <https://en.cppreference.com/w/c/io/fseek>

# Pointers to Pointers

We have previously looked at one example of pointers to pointers, but this topic deserves more attention.

A pointer is a variable that stores a memory address. But since pointers are variables, they are also stored in memory, and thus have an address. A pointer to a pointer is a variable that stores the memory address of a pointer.

Dereferencing a pointer gives us the type of value that is being pointed at. Dereferencing a pointer to a pointer thus gives us a pointer.

Pointers to pointers are required when we are trying to change a pointer in the calling function in the called function (just like we need a pointer to an **int** to change an **int** in the calling function). Pointers to pointers are also required when we have an array of pointers.

## Example: using command-line arguments

You may recall that in C, the main function can be declared as **int main(int argc, char\* argv[])**. The parameters given there access the command line arguments. **argc** is the number of command-line arguments, and **argv** is an array of pointers to strings containing those command-line arguments. **argv** is really a pointer to an array of pointers! The main function can in fact be written as **int main(int argc, char\*\* argv)**.

Let’s try it out. Create a new project with a **main** function as follows. To make the addresses simpler to view, I will use the **x86** platform with **Project | Properties > Configuration Properties > Linker > All Options > Randomized Base Address: No (/DYNAMICBASE:NO)** set.

int main(int argc, char\*\* argv)

{

printf("There are %d command-line arguments\n", argc);

printf("argv is \t%p\n", argv);

for (int i = 0; i < argc; i++)

{

printf("%p\t%p\t%c\t%s\n", argv + i, argv[i], argv[i][0], argv[i]);

}

return EXIT\_SUCCESS;

}

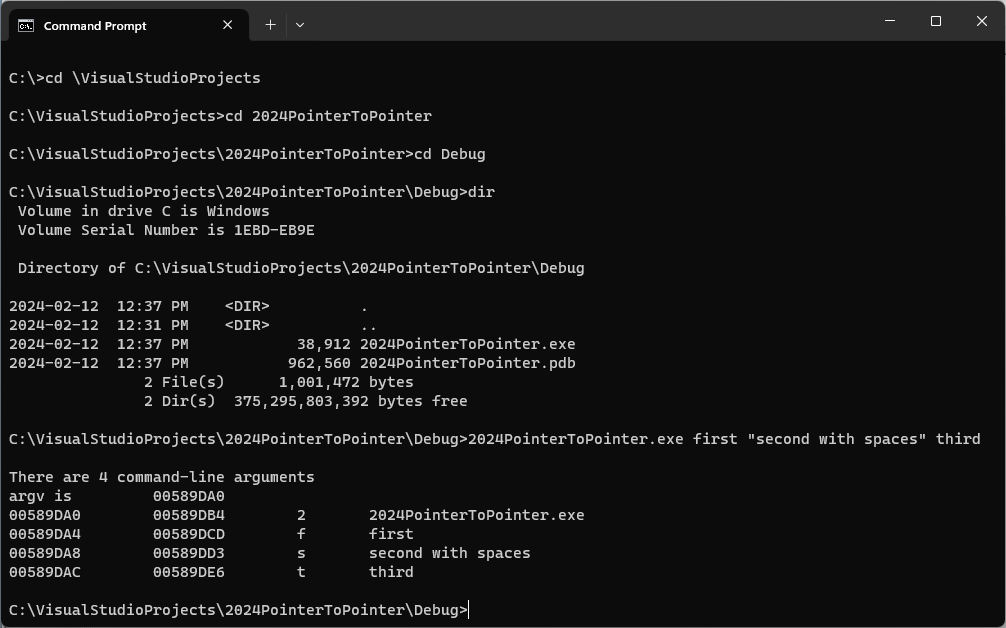
If you run this in Visual Studio, you will see it defaults to one command-line argument which is the path and name of the program:

A screenshot of a computer

Description automatically generated

Note that my project name is **2024PointerToPointer**, and the **.exe** file defaults to that name.

You can change the command-line arguments in Visual Studio (**Project | Properties > Configuration Properties > Debugging > Command Arguments**) or try running the program at the command prompt to type command-line arguments in as follows:



You can see that **argv** contains a location in memory (**00589DA0** in this example) which is the start of an array of pointers that we see on the next lines (**00589DA0**, **00589DA4**, **00589DA8**, **00589DAC** in this example). Each of those addresses in turn contains the address of a string, and we can double-dereference **argv** to get the first character in each string, **argv[i][0]**.

Thus, **argv** is a pointer to an array of **char** pointers, each of which points to a string in the command-line arguments. (The addresses of the strings in the example are **00589DB4**, **00589DCD**, **00589DD3**, and **00589DE6**.)

## Example: trimming characters from a dynamically allocated string

Let’s try a second example of using pointers to pointers. In this example, we will use a pointer to a pointer to change the pointer in the calling function.

The function that we will create will trim a number of characters from the right edge of a dynamically allocated string. Add the following to **main**:

testTrimRight();

**testTrimRight** will create a dynamically allocated string, and then it will call a function **trimRight** to change that string to a new dynamically allocated string using a pointer to a pointer to the passed-in string. It will look like the following (note that you will need to add #define \_CRT\_SECURE\_NO\_WARNINGS to use **strcpy** instead of **strcpy\_s**):

void testTrimRight()

{

char cArray[] = "Michael";

char\* cPtr = (char\*)malloc((strlen(cArray) + 1) \* sizeof(char));

strcpy(cPtr, cArray);

printf("cPtr: %s\n", cPtr);

printf("The address stored by cPtr is %p\n", cPtr);

printf("The address of cPtr is %p\n\n", &cPtr);

trimRight(&cPtr, 3);

printf("cPtr is now: %s\n", cPtr);

printf("The address stored by cPtr is %p\n", cPtr);

printf("The address of cPtr is %p\n\n", &cPtr);

free(cPtr);

cPtr = NULL;

}

Make sure you add #include <string.h> for **strlen** and **strcpy**. You can add #include <malloc.h> if you want, but **malloc** is also included in **<stdlib.h>**.

You can see that **cPtr** is a pointer to a string of **char**s. We would like to change that pointer so that it points to a new string, so we need to pass its address to the **trimRight** function (just like you need to pass the address of an **int** to change the value of that **int** in a function). What stores the address of a pointer? A pointer to a pointer!

So **trimRight** will have the following signature and description:

/\*

Purpose: trim a number of characters from the right end of the

(dynamically allocated) string passed in.

cPtrToPtr --> a pointer to a pointer that points to a string

(pointer to the string will be changed)

iNumChars --> the number of characters to chop off the right end of the string

\*/

void trimRight(char\*\* cPtrToPtr, int iNumChars)

Note the “pointer to a pointer” notation for **cPtrToPtr**.

**trimRight** will look like:

void trimRight(char\*\* cPtrToPtr, int iNumChars)

{

int iDiff = 0;

// Create a temp pointer to the original array

char\* cOriginalPtr = \*cPtrToPtr; // dereferencing a pointer to a pointer

// gives us a pointer

printf("Address in cOriginalPtr is %p\n", cOriginalPtr);

printf("Address in cPtrToPtr is %p\n\n", cPtrToPtr);

// Difference in size of the new and old string;

// could check to ensure nonnegative number

iDiff = strlen(cOriginalPtr) - iNumChars;

// Allocate space for the new array

\*cPtrToPtr = (char\*)malloc((iDiff + 1) \* sizeof(char));

// Copy the data from the old string to the new string

memcpy(\*cPtrToPtr, cOriginalPtr, iDiff \* sizeof(char));

// Add the null terminator to the end of the new string using

// double dereference - character at the pointer pointed to

\*(\*cPtrToPtr + iDiff) = '\0';

// Free the old string

free(cOriginalPtr);

cOriginalPtr = NULL;

}

The output could look as follows:

cPtr: Michael

The address stored by cPtr is 006B0F50

The address of cPtr is 0019FDF8

Address in cOriginalPtr is 006B0F50

Address in cPtrToPtr is 0019FDF8

cPtr is now: Mich

The address stored by cPtr is 006B0F88

The address of cPtr is 0019FDF8

Note that **cPtr** is at an address on the stack frame for **testTrimRight** (**0019FDF8**) and contains an address on the heap (**006B0F50**) which contains the string "Michael". In the **trimRight** function, **cPtrToPtr** contains the address of **cPtr**, so dereferencing **cPtrToPtr** gives us the pointer in the calling function, which we can assign to **cOriginalPtr**.

We can then change **\*cPtrToPtr** to change the pointer in the calling function to the newly allocated memory (at **006B0F88**) which we assign "Mich". To add the null terminator to the end of "Mich", we double-dereference **cPtrToPtr** to get at the character pointed to by the pointer which **cPtrToPtr** points to. We then free the original string so that we don’t have a memory leak.

Back in the calling function, we see that **cPtr** now points to "Mich" at the address **006B0F88**, since **cPtr** was changed by the **trimRight** function.

Try drawing out the memory to ensure that you understand what is happening. You should have the stack frame for **testTrimRight** containing **cPtr**, the stack frame for **trimRight** containing **cPtrToPtr** and **cOriginalPtr**, and memory on the heap containing "Michael" and "Mich".

## Example: finding a character in a string and pointing to it

Suppose that we want a function to find a character in a given string and return the location of that character in the string. Furthermore, we also want a pointer to the position of the character. Since we are returning the location of the character, we can’t also return a pointer. Instead, we use a pointer to a pointer to change the pointer in the calling function.

Start by adding the following to **main**:

testFindChar();

**testFindChar** will look like the following:

void testFindChar()

{

char cArray[] = "Coralee";

char\* cAddressOfChar;

int index = findChar(cArray, 'e', &cAddressOfChar);

printf("Index of character = %d\n", index);

printf("The address of cArray is %p\n", cArray);

printf("The address of the character is %p\n", cAddressOfChar);

}

Note that we will pass in three things to **findChar**: the string to search, the character to search for, and the *address* of a pointer that we wish to change in **findChar**.

Here is the signature and description of **findChar**:

// Write a function findChar that takes a string and a char to find in the string.

// The function will also take a pointer to a pointer to which the function will

// assign the address of the first instance of the char in the string. If not found

// in the string, the pointer will be assigned NULL. The function will also return

// the number of chars into the string that the char was found, or -1 if not found.

int findChar(char\* cStringPtr, char cCharToFind, char\*\* cPtrPtr)

Try writing this function on your own. You can start by just finding the character and returning its location, and then add changing the pointer to the right position.

Your result should look something like:

int findChar(char\* cStringPtr, char cCharToFind, char\*\* cPtrPtr)

{

int location = 0;

while (\*cStringPtr) // != '\0'

{

if (\*cStringPtr == cCharToFind)

{

\*cPtrPtr = cStringPtr;

return location;

}

location++;

cStringPtr++;

}

\*cPtrPtr = NULL;

return -1;

}

Note the use of **\*cPtrPtr** to change the contents of the pointer whose address was passed in from the calling function.

The output of the program could be:

Index of character = 5

The address of cArray is 0019FE04

The address of the character is 0019FE09

Note that running **findChar** changed **cAddressOfChar** to **0019FE09**, which is the position of the first 'e' in "Coralee".

If we change the function call to findChar(cArray, 'z', &cAddressOfChar), then the output will be:

Index of character = -1

The address of cArray is 0019FE04

The address of the character is 00000000

Running **findChar** this time changed **cAddressOfChar** to **NULL**.

## Example: working with a 2D jagged array

A 2-dimensional array is another example of where we need a pointer to a pointer. The pointer to a pointer will contain the address of an array of pointers which will point to the actual data. This arrangement is similar to Java, which uses an array of array references to create a multidimensional array. We will try this out using a jagged array.

Add the following to **main**:

testJaggedArray();

**testJaggedArray** will start out looking like the following – we will add to it and clean it up later:

void testJaggedArray()

{

int iRowSizes[] = { 4, 6, 5 };

// Calculate the number of rows in iRowSizes

int iRows = sizeof(iRowSizes) / sizeof(int);

// Create the jagged array

char\*\* cPtrPtr = createJaggedArray(iRowSizes, iRows);

printArrayOfStrings(cPtrPtr, iRows);

}

**iRowSizes** is an array containing the size of each row in the jagged array. **iRows** is thus the number of rows in the jagged array. We use a pointer to a pointer to contain the address of an array of pointers, each of which will point to an array of data (which will be **char**s in this case so we can treat them as strings, although a jagged array could contain **int**s or **double**s or any other data type).

**printArrayOfStrings** is pretty straightforward:

void printArrayOfStrings(char\*\* cPtrPtr, int iRows)

{

for (int i = 0; i < iRows; i++)

{

printf("String %d is: %s\n", i, cPtrPtr[i]);

}

}

For each row, we just find the pointer to the row of data and print the string at that location. **cPtrPtr** points to an array of pointers, each of which point to a string, so we can print them in a simple loop.

**createJaggedArray** will actually allocate the jagged 2D array and return a pointer to a pointer:

// Return a dynamically allocated 2D jagged array based on information passed in

// iRowSizes --> array of the size of each row

// iNumRows --> number of rows

char\*\* createJaggedArray(int iRowSizes[], int iNumRows)

{

char\*\* cPtrPtr = (char\*\*)malloc(iNumRows \* sizeof(char\*));

// Create each row

for (int i = 0; i < iNumRows; i++)

{

cPtrPtr[i] = allocateRow(iRowSizes[i]);

}

return cPtrPtr;

}

With **allocateRow** as follows:

char\* allocateRow(int iRowSize)

{

char\* cPtr = (char\*)malloc(iRowSize \* sizeof(char));

cPtr[0] = '\0';

return cPtr;

}

Note the line malloc(iNumRows \* sizeof(char\*)) in **createJaggedArray** creates space for an array of **char** pointers, which is stored in a pointer to a pointer. **allocateRow** then allocates the correct number of **char**s for each row.

The program will now print out three empty strings since the rows are just initialized to contain the null terminator. To update this, we can go into **testJaggedArray** and after creating the jagged array, we can manipulate it as follows:

// Add some data to each row

// Index into each location char by char using brackets

cPtrPtr[0][0] = 'M';

cPtrPtr[0][1] = 'e';

cPtrPtr[0][2] = 'l';

cPtrPtr[0][3] = '\0';

We could also use pointer notation and index in using pointer math:

// Index in using pointer math

\*(\*(cPtrPtr + 1) + 0) = 'C';

\*(\*(cPtrPtr + 1) + 1) = 'h';

\*(\*(cPtrPtr + 1) + 2) = 'r';

\*(\*(cPtrPtr + 1) + 3) = 'i';

\*(\*(cPtrPtr + 1) + 4) = 's';

\*(\*(cPtrPtr + 1) + 5) = '\0';

Finally, because the jagged array contains arrays of **char**, we can use **strcpy** to copy in values (note that this wouldn’t work with **int** or **double** arrays, although you could use **memset** to copy in an array of values):

// Use strcpy - only because this is a jagged array of chars

strcpy(cPtrPtr[2], "Mark");

Our program now prints out the three names in the jagged array:

String 0 is: Mel

String 1 is: Chris

String 2 is: Mark

Have you noticed the problem with **testJaggedArray**? It uses dynamically allocated memory for the jagged array, but never frees the memory. To free the memory, we must first free each row (allocated by **allocateRow**), then free the array of pointers to the rows (allocated in **createJaggedArray**). The code for this step is as follows:

// Free ALL the memory created for the jagged array

// Free each row first

for (int i = 0; i < iRows; i++)

{

free(cPtrPtr[i]);

cPtrPtr[i] = NULL;

}

free(cPtrPtr);

cPtrPtr = NULL;

The output is the same, but we no longer have memory leaks.

## Exercise: get an array of doubles and point to the largest element

To practice with pointers to pointers, try the following exercise. First, add the following to **main**:

testDoubleArrayWithMax();

**testDoubleArrayWithMax** will start out as:

void testDoubleArrayWithMax()

{

}

Now create a function **getDynamicDoubles** that asks the user for a number of **double** values and reads in that number of **double**s (using **"%lf"** in **scanf**) into a dynamically allocated array and return that array. Also change an **int** so that it will contain the size of the array allocated. Also change a pointer so that it will point to the largest element in that array.

In **testDoubleArrayWithMax**, call **getDynamicDoubles** appropriately and then print out the array and the max value.

Try this exercise out on your own.

To start, you can skip the portion that changes a pointer to point to the largest element in the array (which requires a pointer to a pointer). Then add that in.

**testDoubleArrayWithMax** will start out looking like the following:

int iArraySize;

double\* dArray = getDynamicDoubles(&iArraySize);

for (int i = 0; i < iArraySize; i++)

{

printf("Double #%d: %lf\n", i, dArray[i]);

}

**getDynamicDoubles** will start out as:

double\* getDynamicDoubles(int\* iArraySizePtr)

{

double\* dArray;

printf("How many doubles would you like to enter? ");

scanf\_s("%d", iArraySizePtr);

dArray = (double\*)malloc(\*iArraySizePtr \* sizeof(double));

for (int i = 0; i < \*iArraySizePtr; i++)

{

printf("Enter a double: ");

scanf\_s("%lf", dArray + i); // or &dArray[i]

}

return dArray;

}

Now update **testDoubleArrayWithMax** to include a pointer to the largest value whose address will be passed into **getDynamicDoubles**, and while we’re making changes, free the array of **double**s to avoid a memory leak:

int iArraySize;

double\* dMaxPointer = NULL;

double\* dArray = getDynamicDoubles(&iArraySize, &dMaxPointer);

for (int i = 0; i < iArraySize; i++)

{

printf("Double #%d: %lf\n", i, dArray[i]);

}

printf("The max value is at %p and is %lf\n", dMaxPointer, \*dMaxPointer);

free(dArray);

dArray = NULL;

And update **getDynamicDoubles** to check for the largest value – there are several ways to do so:

double\* getDynamicDoubles(int\* iArraySizePtr, double\*\* maxPtrPtr)

{

double dMaxValue = -DBL\_MAX; // smallest double from <float.h>

double\* dArray;

printf("How many doubles would you like to enter? ");

scanf\_s("%d", iArraySizePtr);

dArray = (double\*)malloc(\*iArraySizePtr \* sizeof(double));

for (int i = 0; i < \*iArraySizePtr; i++)

{

printf("Enter a double: ");

scanf\_s("%lf", dArray + i); // or &dArray[i]

if (i == 0)

{

dMaxValue = dArray[i];

\*maxPtrPtr = dArray + i; // or &dArray[i]

}

else if (dArray[i] > dMaxValue)

{

dMaxValue = dArray[i];

\*maxPtrPtr = dArray + i; // or &dArray[i]

}

}

return dArray;

}

Sample output would be:

How many doubles would you like to enter? 5

Enter a double: 1.3

Enter a double: 0.4

Enter a double: 2.7

Enter a double: -9.9

Enter a double: 0

Double #0: 1.300000

Double #1: 0.400000

Double #2: 2.700000

Double #3: -9.900000

Double #4: 0.000000

The max value is at 00631908 and is 2.700000

### Using AI to help program in C

On your assignments, you are required to hand in your own work. Thus, you should not be using generative AI tools like ChatGPT (<https://chat.openai.com/>) or Google’s Gemini (<https://gemini.google.com/app>) for help coding your files. And since your midterm and final exams will be written with no Internet access, you will not have such tools available there, so it is important that you learn to code on your own.

However, such tools can be valuable in practicing with programming, and evaluating the code produced by such tools is a useful skill. Thus, let’s see what ChatGPT produces as an answer to this question.

I prompted ChatGPT with “Create a program in C that will include a function that will ask the user for a number of doubles and read that number of doubles into a dynamically allocated array and return that array. The function should also change an int so that it will contain the size of the array allocated. Also change a pointer so that it will point to the largest element in the array.”

Here is what ChatGPT produced – note that it may produce different results for you (reformatted to match our standards, and with the **main** function name changed to **testChatGPTDoubleArrayWithMax**):

double\* read\_doubles(int\* size, double\*\* largest)

{

printf("Enter the number of doubles: ");

scanf("%d", size);

double\* arr = (double\*)malloc(\*size \* sizeof(double));

\*largest = &arr[0]; // Initialize largest pointer to first element

for (int i = 0; i < \*size; i++)

{

printf("Enter a double: ");

scanf("%lf", &arr[i]);

// Update largest pointer if new value is larger

if (\*largest < &arr[i])

{

\*largest = &arr[i];

}

}

return arr;

}

testChatGPTDoubleArrayWithMax()

{

int size;

double\* largest;

double\* arr = read\_doubles(&size, &largest);

printf("Array elements:\n");

for (int i = 0; i < size; i++)

{

printf("%.2lf ", arr[i]);

}

printf("\nSize of array: %d\n", size);

printf("Largest element: %.2lf\n", \*largest);

free(arr);

}

ChatGPT’s comments on the code were:

“The **read\_doubles** function prompts the user for the number of doubles to be read, dynamically allocates an array of that size, and reads the doubles into the array. It also updates the **size** variable to contain the size of the array and the **largest** pointer to point to the largest element in the array.

In the **main** function, the read\_**doubles** function is called and the resulting array, size, and largest element are printed to the console. Finally, the dynamically allocated array is freed.

Note that this program assumes that the user will enter valid input and does not include error handling for invalid input.”

Compare this result with our code to see the similarities and differences. The code for **testChatGPTDoubleArrayWithMax** is very similar to our **testDoubleArrayWithMax** except for the variable names and format of the printout.

**read\_doubles** is similar to **getDynamicDoubles**, but it cleverly sets the largest pointer to the address of the first element in the array, automatically setting its value as the current maximum. However, instead of finding the maximum value in the array, it finds the maximum address in the array!

This error is one of the drawbacks of using AI – if you just tested it with **2.0**, **4.0**, and **6.0**, you might think it gives the right answer (**6.0**). But if you reverse those numbers, you will see it gives the answer **2.0**. Proper testing and understanding of the code generated by AI is important.

I updated my ChatGPT prompt as follows “That program points `largest` to the largest address in the array, not the largest value in the array. Can you correct it?”

ChatGPT then created the correct code, including the following change:

// Update largest pointer if new value is larger

if (arr[i] > \*\*largest) {

\*largest = &arr[i];

}

Note that in our program and in ChatGPT’s result, the results of **malloc** are not tested to see if it is **NULL**.

I tried the same prompt in Gemini, and it produced a similar but slightly different program (it read in all the doubles first, then after that searched for the largest without the initial error that ChatGPT had) and it also tested the results of **malloc** for **NULL** as follows:

// Allocate memory for the array

double \*array = malloc(\*size \* sizeof(double));

if (array == NULL) {

fprintf(stderr, "Error allocating memory for array\n");

exit(1);

}

## Exercise: swapping bytes in an int and pointing to the high-order byte

Here is another exercise for working with pointers to pointers. First, add the following to **main**:

testSwapBytesWithPointer();

**testSwapBytesWithPointer** will start out as:

void testSwapBytesWithPointer()

{

}

Now, create a function **swapBytesWithPointer** that will take an **int** and swap its bytes (as if it were Big-Endian instead of Little-Endian) and change a char pointer to the new high-order byte (formerly the low-order byte). For instance, if **0x65666768** is passed in, the result would be **0x68676665** and the pointer would point to the byte containing **0x68**.

Call this function from **testSwapBytesWithPointer** and print out the initial value and the results of **swapBytesWithPointer**.

Try this exercise out on your own.

**testSwapBytesWithPointer** should look something like:

void testSwapBytesWithPointer()

{

int iValue = 0x65666768;

char\* cBytePointer = NULL;

printf("Initial value: %x\t%d\n", iValue, iValue);

swapBytesWithPointer(&iValue, &cBytePointer);

printf("Changed value: %x\t%d\n", iValue, iValue);

printf("New high-order byte: %x\n", \*cBytePointer);

}

Note that we pass the address of **iValue** to the **swapBytesWithPointer** function so that it can actually change the **int** in **iValue**. Similarly, we pass the address of **cBytePointer** to the **swapBytesWithPointer** function so that it can change the pointer in **cBytePointer**.

One solution for **swapBytesWithPointer** is as follows:

void swapBytesWithPointer(int\* iValuePtr, char\*\* cBytePtrPtr)

{

int iOldValue = \*iValuePtr;

char\* cBytePointerOld = (char\*)&iOldValue;

char\* cBytePointerNew = (char\*)iValuePtr;

cBytePointerNew[3] = cBytePointerOld[0];

cBytePointerNew[2] = cBytePointerOld[1];

cBytePointerNew[1] = cBytePointerOld[2];

cBytePointerNew[0] = cBytePointerOld[3];

\*cBytePtrPtr = cBytePointerNew + 3;

}

In this solution, we keep track of the original integer value in **iOldValue**, which allows us to change the value which is pointed at in **iValuePtr** while still keeping track of the original byte sequence. Then we use **char\*** variables to both the original and new value to alter the order of the bytes in the new value. Finally, we dereference our pointer to a pointer in **cBytePtrPtr** to set it to the high-order byte.

Output for this example would be:

Initial value: 65666768 1701209960

Changed value: 68676665 1751606885

New high-order byte: 68

### Using AI to help program in C

I will repeat my warnings and advice for those that may not have read the last section:

On your assignments, you are required to hand in your own work. Thus, you should not be using generative AI tools like ChatGPT (<https://chat.openai.com/>) or Google’s Gemini (<https://gemini.google.com/app>) for help coding your files. And since your midterm and final exams will be written with no Internet access, you will not have such tools available there, so it is important that you learn to code on your own.

However, such tools can be valuable in practicing with programming, and evaluating the code produced by such tools is a useful skill. Thus, let’s see what ChatGPT produces as an answer to this question.

I prompted ChatGPT with the following: “Create a program in C that will include a function that will take an int and swap its bytes (as if it was Big-Endian instead of Little-Endian) and change a char pointer to the new high-order byte (formerly the low-order byte). For instance, if 0x65666768 was passed in, the result would be 0x68676665 and the pointer would point to the byte containing 0x68.”

Here is the code it gave me – again, you may get different results (reformatted to match our standards and with **main** renamed to **testChatGPTSwapBytes**):

void swap\_int\_bytes(int\* num, char\*\* high\_order\_byte) {

char\* bytes = (char\*)num;

char temp;

// swap bytes

temp = bytes[0];

bytes[0] = bytes[3];

bytes[3] = temp;

temp = bytes[1];

bytes[1] = bytes[2];

bytes[2] = temp;

// update pointer to high-order byte

\*high\_order\_byte = &bytes[0];

}

testChatGPTSwapBytes()

{

int num = 0x65666768;

char\* high\_order\_byte;

printf("Before swap: 0x%08x\n", num);

swap\_int\_bytes(&num, &high\_order\_byte);

printf("After swap: 0x%08x\n", num);

printf("High-order byte: 0x%02x\n", \*high\_order\_byte);

}

Its explanation is as follows: “The **swap\_int\_bytes** function takes an **int\*** and a **char\*\*** as arguments. The **int\*** is a pointer to the integer whose bytes we want to swap. The **char\*\*** is a pointer to a **char\*** that will be updated to point to the high-order byte of the swapped integer.

Inside the function, we cast the **int\*** to a **char\*** so that we can manipulate the individual bytes of the integer. We then swap the bytes by copying the bytes to temporary variables and then swapping them in place. Finally, we update the **char\*\*** to point to the first byte of the swapped integer, which is now the high-order byte.

In main, we declare an **int** variable and initialize it to the value **0x65666768**, which is **efgh** in little-endian order. We then call **swap\_int\_bytes** and pass it a pointer to **num** and a pointer to **high\_order\_byte**. After the function call, we print out the swapped integer and the high-order byte using **printf** with the **%08x** and **%02x** format specifiers, respectively.”

Note that ChatGPT used a different method of swapping the bytes, using a **char temp** to swap bytes 0 and 3 and then 1 and 2. Also, ChatGPT was pointing to the wrong byte – it is pointing to the low-order byte in the result, not the high-order byte in the result (perhaps because it chose to display the high-order byte from the original number?).

Trying it in Gemini produced similar code, including the same error (although again, you may get different results if you try it).

# Midterm Exam Notes

The midterm exam will cover everything up to this point except for file I/O (which will be covered on the final exam).

To prepare for the midterm exam, you should try out the various exercises that have been provided throughout the course. In addition, here is a midterm review exercise:



And a sample solution:



## Using AI to prepare for the midterm exam

I will repeat my warnings and advice for those that may not have read the last sections:

On your assignments, you are required to hand in your own work. Thus, you should not be using generative AI tools like ChatGPT (<https://chat.openai.com/>) or Google’s Gemini (<https://gemini.google.com/app>) for help coding your files. And since your midterm and final exams will be written with no Internet access, you will not have such tools available there, so it is important that you learn to code on your own.

However, such tools can be valuable in practicing with programming, and evaluating the code produced by such tools is a useful skill. Furthermore, such tools can generate sample questions for you to use in studying for the midterm exam. You will have to be careful, as some questions may be too difficult or involve topics that we haven’t covered yet, but with careful selection and repeated prompting, you may be able to get good review questions.

Here's an example – I prompted ChatGPT with “Provide a sample exam question using the C programming language to write a function that will work with dynamic memory.”

The result was as follows:

Certainly, here's a sample exam question involving dynamic memory allocation in C:

Write a C function **resizeArray** that dynamically resizes an integer array to a new specified size. The function should have the following prototype:

cCopy code

int \*resizeArray(int \*arr, int oldSize, int newSize);

The function should take three parameters:

* **arr**: a pointer to the current integer array.
* **oldSize**: an integer representing the current size of the array.
* **newSize**: an integer representing the desired new size of the array.

The function should return a pointer to the resized array. If resizing is successful, the function should copy the contents of the old array into the new array up to the minimum of **oldSize** and **newSize**. If **newSize** is smaller than **oldSize**, the function should truncate the array; if **newSize** is larger, the function should allocate additional memory and set any new elements to zero.

Your implementation should correctly handle dynamic memory allocation and deallocation.

Provide your function definition and an example demonstrating its usage in a program.

And here is the sample solution and the explanation provided:

#include <stdio.h>

#include <stdlib.h>

int\* resizeArray(int\* arr, int oldSize, int newSize)

{

int\* resizedArr = (int\*)realloc(arr, newSize \* sizeof(int));

if (resizedArr == NULL)

{

// Memory allocation failed, return original array

printf("Memory allocation failed. Returning original array.\n");

return arr;

}

// Copy contents from the old array to the new array

int elementsToCopy = oldSize < newSize ? oldSize : newSize;

for (int i = 0; i < elementsToCopy; i++)

{

resizedArr[i] = arr[i];

}

// If newSize is greater than oldSize, set new elements to zero

if (newSize > oldSize)

{

for (int i = oldSize; i < newSize; i++)

{

resizedArr[i] = 0;

}

}

return resizedArr;

}

int main()

{

int size = 5;

int\* arr = (int\*)malloc(size \* sizeof(int));

if (arr == NULL)

{

printf("Memory allocation failed.\n");

return 1;

}

// Initialize array

for (int i = 0; i < size; i++)

{

arr[i] = i + 1;

}

printf("Original array:\n");

for (int i = 0; i < size; i++)

{

printf("%d ", arr[i]);

}

printf("\n");

// Resize array to a new size

int newSize = 8;

arr = resizeArray(arr, size, newSize);

printf("Resized array:\n");

for (int i = 0; i < newSize; i++)

{

printf("%d ", arr[i]);

}

printf("\n");

// Free dynamically allocated memory

free(arr);

return 0;

}

In this implementation, the **resizeArray** function uses **realloc** to dynamically resize the array to the new specified size. It copies the contents of the old array into the new array and sets any new elements to zero if the new size is larger than the old size. If memory allocation fails during resizing, it returns the original array. Finally, the **main** function demonstrates the usage of **resizeArray** by initializing an array, resizing it, and printing the contents of both the original and resized arrays.

Note that in an exam, you might be asked to provide only the function (not the code in main calling the function). Also note that the question could have been answered differently – for instance, you could have called **memcpy** from **<string.h>** to copy the elements of the array from the old to the new array, and you could have used **memset** to set the new elements to 0. See the following example:

memcpy(resizedArr, arr, elementsToCopy \* sizeof(int));

memset(resizedArr + oldSize, 0, (newSize - oldSize) \* sizeof(int));

Another sample exam question generated by the prompt “Provide a sample exam question in the C programming language that involves writing a function that uses pointers to pointers” is to transpose elements in a matrix. Here is a variation of the question:

Write a program that will dynamically allocate a 2D array of values (a matrix) and fill it with the values 1, 2, 3, …. Then write a function **transposeMatrix** that will return a different dynamically allocated 2D array of values with the values transposed.

Here is a sample solution:

void printMatrix(int\*\* matrixPtr, int rows, int cols)

{

for (int i = 0; i < rows; i++)

{

printf("[\t");

for (int j = 0; j < cols; j++)

{

printf("%d\t", matrixPtr[i][j]);

}

printf("]\n");

}

printf("\n");

}

void freeMatrix(int\*\* matrixPtr, int rows)

{

for (int i = 0; i < rows; i++)

{

free(matrixPtr[i]);

matrixPtr[i] = NULL;

}

free(matrixPtr);

matrixPtr = NULL;

}

int\*\* transposeMatrix(int\*\* matrixPtr, int rows, int cols)

{

int\*\* transposedMatrixPtr = malloc(cols \* sizeof(int\*));

for (int i = 0; i < cols; i++)

{

transposedMatrixPtr[i] = malloc(rows \* sizeof(int));

for (int j = 0; j < rows; j++)

{

transposedMatrixPtr[i][j] = matrixPtr[j][i];

}

}

return transposedMatrixPtr;

}

void testTransposeMatrix()

{

int rows = 5;

int cols = 3;

int count = 1;

int\*\* matrixPtr = malloc(rows \* sizeof(int\*));

for (int i = 0; i < rows; i++)

{

matrixPtr[i] = malloc(cols \* sizeof(int));

for (int j = 0; j < cols; j++)

{

matrixPtr[i][j] = count++;

}

}

printMatrix(matrixPtr, rows, cols);

int\*\* transposedMatrixPtr = transposeMatrix(matrixPtr, rows, cols);

printMatrix(transposedMatrixPtr, cols, rows);

freeMatrix(matrixPtr, rows);

freeMatrix(transposedMatrixPtr, cols);

}

Note again that on the exam, you might be asked to provide only a single function (like **transposeMatrix**) rather than all the code.

# Bitwise Operators

Recall that we can operate on any byte. So for instance, given:

short sValue = 0x1234;

char\* cPtr = &sValue;

we could print out **sValue** or each byte in **sValue** as follows:

printf("sValue: %x\n", sValue); // prints 1234 (hex)

printf("sValue: %d\n", sValue); // prints 4660 (decimal)

printf("Byte 1: %x\t%d\n", \*cPtr, \*cPtr); // prints 34 (hex) and 52 (decimal)

printf("Byte 2: %x\t%d\n", \*(cPtr + 1), \*(cPtr + 1));

// prints 12 (hex) and 18 (decimal)

But what if we want to change the individual bits in a number? Since the computer is byte-addressable, we can’t do so with just pointers. Instead, we need to use bitwise operators to do so.

## General notes on bitwise operations

A bitwise operation operates on one or more bit patterns at the level of their individual bits. It is a fast, primitive action directly supported by the processor, and is used to manipulate values for comparisons and calculations.

According to <https://en.wikipedia.org/wiki/Bitwise_operation> (retrieved on February 27, 2024), “On simple low-cost processors, typically, bitwise operations are substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition. While modern processors usually perform addition and multiplication just as fast as bitwise operations, … bitwise operations do commonly use less power because of the reduced use of resources.”

## Bitwise NOT

Bitwise NOT or bitwise inversion is a unary operator that flips the bits. Before we look at the bitwise operator, let’s recall the logical operators. In Boolean logic, we use **NOT a** to indicate that we should reverse the truth value of **a**. In C, the logical operator for **NOT** is **!**. The truth table for **NOT** is as follows, using 0 for false and 1 for true:

|  |  |  |
| --- | --- | --- |
| **Logical** |  | **!a** |
| **Boolean** | **a** | **NOT a** |
|  | 0 | 1 |
|  | 1 | 0 |

The bitwise operator for NOT is the tilde (**~**). It does the same thing as the logical operator, but operates on every bit of variable that it is applies to. So for some variable **var1**, if **var1** = 0x12 = 0001 0010, then **~var1** = 1110 1101 = 0xED.

Note that in C, you can apply the logical operator **!** to any variable since any non-zero value is treated as true, but it would just result in either **0** for false or **1** for true. Bitwise NOT applies to every bit individually.

Let’s try an example in code.

If we have the function **bitInversion** as follows:

void bitInversion()

{

unsigned char a = 0x48;

unsigned char result = ~a;

printf("Bitwise inversion of a is %x\n", result);

}

then the output would be Bitwise inversion of a is b7, since 0x48 = 0100 1000 and we then flip the bits to get 0xB7 = 1011 0111.

It would be useful to see the individual bits of the numbers. Here is a function in C to do so. Don’t worry about how it works yet – we will examine that later. Just know that you can pass in a value and the size of that value in bytes and it will display the bits of the value:

// Print a value as a string of bits (up to 32 bits)

void showBits(int value, int bytes)

{

unsigned int filter = 1;

int bit;

char\* bitString = (char\*)malloc(33);

// Null terminator

bitString[32] = '\0';

for (bit = 0; bit < bytes \* 8; bit++)

{

// Determine the value of the current bit

if (value & filter)

{

bitString[31 - bit] = '1';

}

else

{

bitString[31 - bit] = '0';

}

filter \*= 2; // Move to the next bit to the left

}

printf("The value %d in binary is \t%s\n",

value, &bitString[32 - bytes \* 8]);

free(bitString);

bitString = NULL;

}

We can then modify the **bitInversion** function as follows:

void bitInversion()

{

unsigned char a = 0x48;

unsigned char result = ~a;

printf("Bitwise inversion of a is %x\n", result);

showBits(a, sizeof(char));

showBits(result, sizeof(char));

}

And the output is now:

The value 72 in binary is 01001000

The value 183 in binary is 10110111

## Bitwise AND

Bitwise AND is a binary operator that compares each bit in the first operand with each bit in the second operand. If both bits in the compared position are 1, the bit in the resulting binary representation is 1; otherwise, the result is 0.

Let’s review the Boolean AND first, carried out in C by the logical operator **&&**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Logical** |  |  | **!a** | **a && b** |
| **Boolean** | **a** | **b** | **NOT a** | **a AND b** |
|  | 0 | 0 | 1 | 0 |
|  | 0 | 1 | 1 | 0 |
|  | 1 | 0 | 0 | 0 |
|  | 1 | 1 | 0 | 1 |

The bitwise operator for **AND** is a single ampersand (**&**). Don’t confuse it with the logical operator **&&**! If we have **var1** = 0x12 = 0001 0010 and **var2** = 0x34 = 0011 0100, then **var1 & var2** = 0001 0000 = 0x10. (Note that **var1 && var2** would just be 1, since two non-zero truthy values are combined to result in true.)

Let’s try it in code:

void bitAND()

{

unsigned char a = 0x48;

unsigned char b = 0xB8;

unsigned char result = a & b;

printf("Bitwise AND of a and b is %x\n", result);

showBits(a, sizeof(char));

showBits(b, sizeof(char));

showBits(result, sizeof(char));

}

Output will be:

Bitwise AND of a and b is 8

The value 72 in binary is 01001000

The value 184 in binary is 10111000

The value 8 in binary is 00001000

With a carefully constructed *mask* (a bit pattern that controls which bits will be displayed), we can use bitwise AND to preserve bits as they are or turn some bits off. Any bit combined with **&** will have the same value if the bit in the mask is **1**, and will be turned off (set to 0) if the bit in the mask is **0**.

So for instance, given an **int** **x** = 0x030B1A0F, we could check if the bit in the 4’s position is set on by using a mask with all 0’s except for a 1 in the 4’s position (so 0x00000004). We could set the bit in the 1’s position to 0 by using a mask with all 1’s except for a 0 in the 1’s position (so 0xFFFFFFFE). In code:

int x = 0x30B1A0F;

int maskFindBit = 0x00000004;

int maskSetBitToZero = 0xFFFFFFFE;

showBits(x, sizeof(int));

showBits(x & maskFindBit, sizeof(int));

showBits(x & maskSetBitToZero, sizeof(int));

This code would result in the following output (reformatted with a few extra spaces to line things up):

The value 51059215 in binary is 00000011000010110001101000001111

The value 4 in binary is 00000000000000000000000000000100

The value 51059214 in binary is 00000011000010110001101000001110

Let’s now reexamine the code in **showBits**. We start by creating a 33-character null terminated string:

char\* bitString = (char\*)malloc(33);

// Null terminator

bitString[32] = '\0';

We set a filter to 1 to look for a 1 in the 1’s position:

unsigned int filter = 1;

Then we loop through each bit in the number based on the number of bytes provided:

for (bit = 0; bit < bytes \* 8; bit++)

We use bitwise AND to check if the current bit (starting with the 1’s bit) is set to 1:

// Determine the value of the current bit

if (value & filter)

Because we want the first bit to be the *last* character in the string, we need to set the character as follows:

bitString[31 - bit] = '1';

Then we move to the next bit position by adjusting our filter to have the bit 1 in the next power of 2:

filter \*= 2; // Move to the next bit to the left

We only set the characters for the size of the value provided in bytes, so when we print the string, we have to start from the appropriate spot. This spot is given by **32** (the last character) minus the number of bytes times 8 (to get the number of bits that we set). We then take the address of that byte to get the string position:

printf("The value %d in binary is \t%s\n",

value, &bitString[32 - bytes \* 8]);

The bit string is no longer needed, so we free the memory allocated:

free(bitString);

bitString = NULL;

So in summary, what we do is look at each bit from the right to the left, checking if it is a 1 or a 0 and then setting the corresponding location in the string (right to left) for the number of bytes specified as the size of **value**.

## Bitwise OR

Bitwise OR is a binary operator that compares each bit in the first operand with each bit in the second operand. The result in each position is 1 if the first bit is 1 OR the second bit is 1 OR both bits are 1. Otherwise (if both bits are 0), the result is 0.

Let’s review the Boolean OR first, carried out in C by the logical operator **||**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Logical** |  |  | **!a** | **a && b** | **a || b** |
| **Boolean** | **a** | **b** | **NOT a** | **a AND b** | **a OR b** |
|  | 0 | 0 | 1 | 0 | 0 |
|  | 0 | 1 | 1 | 0 | 1 |
|  | 1 | 0 | 0 | 0 | 1 |
|  | 1 | 1 | 0 | 1 | 1 |

The bitwise operator for **OR** is a single pipe (**|**). Don’t confuse it with the logical operator **||**! If we have **var1** = 0x12 = 0001 0010 and **var2** = 0x34 = 0011 0100, then **var1 | var2** = 0011 0110 = 0x36. (Note that **var1 || var2** would just be 1, since two non-zero truthy values are combined to result in true.)

Let’s try it in code:

void bitOR()

{

unsigned char a = 0x48;

unsigned char b = 0xB8;

unsigned char result = a | b;

printf("Bitwise OR of a and b is %x\n", result);

showBits(a, sizeof(char));

showBits(b, sizeof(char));

showBits(result, sizeof(char));

}

The output would be:

Bitwise OR of a and b is f8

The value 72 in binary is 01001000

The value 184 in binary is 10111000

The value 248 in binary is 11111000

With a carefully constructed mask, we can use bitwise OR to preserve bits as they are or turn some bits on. Any bit combined with **|** will have the same value if the bit in the mask is **0**, and will be turned on (set to 1) if the bit in the mask is **1**.

So for instance, given an **int** **x** = 0x030B1A0F, we could set the first 2 bytes to 1’s and preserve the other bits using a mask of **0xFFFF0000**:

int x = 0x030B1A0F;

int mask = 0xFFFF0000; // set first 2 bytes to 1's and preserve other bits

printf("Masked value:\t%x\n", x | mask);

showBits(x, sizeof(int));

showBits(x | mask, sizeof(int));

With output as follows (with a few spaces added to line things up):

Masked value: ffff1a0f

The value 51059215 in binary is 00000011000010110001101000001111

The value -58865 in binary is 11111111111111110001101000001111

Note that because **showBits** uses an **int** for the value, the value of the masked number is a negative value (-58865) because the first bit is set to 1 by the mask.

## Bitwise XOR

Bitwise exclusive or (XOR) is a binary operator that compares each bit in the first operand with each bit in the second operand. The result in each position is 1 if only the first bit is 1 OR if only the second bit is 1, but will be 0 if both are 0 or both are 1.

Exclusive OR can be used to model to the state of something like a two-way switch. It is also used in encryption, where the ciphertext C = plaintext XOR key and then the plaintext P = ciphertext XOR key.

Let’s review the Boolean XOR first – note that in C, there is no logical XOR operator (in Java, you can use **^** to work on Boolean values as well):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Logical** |  |  | **!a** | **a && b** | **a || b** |  |
| **Boolean** | **a** | **b** | **NOT a** | **a AND b** | **a OR b** | **a XOR b** |
|  | 0 | 0 | 1 | 0 | 0 | 0 |
|  | 0 | 1 | 1 | 0 | 1 | 1 |
|  | 1 | 0 | 0 | 0 | 1 | 1 |
|  | 1 | 1 | 0 | 1 | 1 | 0 |

The bitwise operator for **XOR** is a the caret (**^**).If we have **var1** = 0x12 = 0001 0010 and **var2** = 0x34 = 0011 0100, then **var1 ^ var2** = 0010 0110 = 0x26.

Let’s try it in code:

void bitXOR()

{

unsigned char a = 0x48;

unsigned char b = 0xB8;

unsigned char result = a ^ b;

printf("Bitwise XOR of a and b is %x\n", result);

showBits(a, sizeof(char));

showBits(b, sizeof(char));

showBits(result, sizeof(char));

}

The output would be:

The value 72 in binary is 01001000

The value 184 in binary is 10111000

The value 240 in binary is 11110000

XOR with a full complement mask (FF or all 1’s) will flip the bits in the other operand. XOR with a mask of 0’s will preserve the bits in the other operand. So the code:

showBits(b, sizeof(char));

showBits(0xFF ^ b, sizeof(char));

showBits(0x00 ^ b, sizeof(char));

would produce the output:

The value 184 in binary is 10111000

The value 71 in binary is 01000111

The value 184 in binary is 10111000

## Bit shift operators

There are also binary operators for shifting bits. No comparison is made – bits are simply moved left or right by the amount of the second operand with 0’s (or sometimes 1’s, as we will see) moved in for padding.

Shifting right *n* positions is equivalent to dividing by 2 to the power *n* (2*n*). Shifting left *n* positions is equivalent to multiplying by 2 to the power *n*.

The operator to shift right is **>>** and the operator to shift left is **<<**.

Let’s try it in code:

void shiftOperators()

{

short x = 37;

// Display x in binary

printf("No shift in x\n");

showBits(x, sizeof(short));

// Note that bit shifting left is the same as multiplying by a power of 2

printf("Shift x 1 bit to the left\n");

showBits(x << 1, sizeof(short));

// Note that bit shifting right is the same as dividing by a power of 2

// (integer division)

printf("Shift x 2 bits to the right\n");

showBits(x >> 2, sizeof(short));

}

The output would be:

No shift in x

The value 37 in binary is 0000000000100101

Shift x 1 bit to the left

The value 74 in binary is 0000000001001010

Shift x 2 bits to the right

The value 9 in binary is 0000000000001001

As you can see, the value 37 is doubled by shifting it 1 bit to the left. The additional bit on the right edge is filled with a 0.

The value 37 is divided by 4 by shifting 2 bits to the right. The additional bits on the left edge are filled with 0. The bits on the right edge (which correspond to the remainder of 1 from integer division by 4) are lost.

Note that bit shifting does not change the original value but rather produces a new value. So x << 1 produces the new value 74 and does not change the value of **x**, which is still 37.

Let’s examine what happens when we have a **1** in the leftmost (sign) bit in a signed value:

int z = 1;

printf("Shift z 30 bits to the left\n");

showBits(z << 30, sizeof(int));

printf("Shift z 31 bits to the left and then 1 bit back to the right\n");

showBits(z << 31 >> 1, sizeof(int));

The output produced is:

Shift z 30 bits to the left

The value 1073741824 in binary is 01000000000000000000000000000000

Shift z 31 bits to the left and then 1 bit back to the right

The value -1073741824 in binary is 11000000000000000000000000000000

The left shift of 30 moves the value of 1 thirty places to the left. So a left shift of 31 moves the 1 bit to the leftmost position. But shifting it back 1 bit to the right does not produce the same result as a left shift of 30 – instead, a 1 is shifted in as padding. What happened?

If the value is a signed value, then the padding for a right shift is determined by the value of the sign bit. If the sign bit is set (to 1), then the padding will be 1’s; otherwise, the padding will be 0’s. This is why we often use unsigned values when doing bit operations. Try changing the code as follows and note that the padding is always 0:

unsigned int z = 1;

One note about Java: Java also has all the bitwise operators we’ve described, including the bit shift operators. However, Java does not have unsigned data types, so it introduced another bit shift operator, **>>>**, which always shifts in 0’s on the left (even if the sign bit is 1).

Note that the bit shift operators produce a result that is an **int** (or an **unsigned int**) unless they are implicitly or explicitly cast as another type. Thus, the following code:

signed char y = -128;

printf("y without shifting\n");

showBits(y, sizeof(char));

printf("Shift y left by 1\n");

showBits(y << 1, sizeof(char));

printf("Shift y right by 1\n");

showBits(y >> 1, sizeof(char));

produces the output:

y without shifting

The value -128 in binary is 10000000

Shift y left by 1

The value -256 in binary is 00000000

Shift y right by 1

The value -64 in binary is 11000000

Notice that shifting **y** left by 1 resulted in the value **-256** although displaying it as just one byte displays all 0’s. To display that result as the value **0**, change the code as follows:

showBits((unsigned char)(y << 1), sizeof(char));

Also note again that since **y** is a **signed char**, shifting to the right used a 1 for padding. This does correspond to integer division (-128 divided by 2 is -64).

By using proper masks, we can use bit shifting to get the bits from any part of a value. For instance, consider the following code:

unsigned char value = 0xD3;

unsigned char mask = 0x0F;

// Masking demo

printf("Value\n");

showBits(value, sizeof(char));

printf("Mask\n");

showBits(mask, sizeof(char));

// The lower "nibble" of value

printf("The lower nibble of value is\n");

showBits(value & mask, sizeof(char));

// The upper nibble of value

printf("The upper nibble of value is\n");

showBits((value >> 4) & mask, sizeof(char));

The output would be as follows:

Value

The value 211 in binary is 11010011

Mask

The value 15 in binary is 00001111

The lower nibble of value is

The value 3 in binary is 00000011

The upper nibble of value is

The value 13 in binary is 00001101

The mask has all 1’s in the bottom 4 bits. Groups of 4 bits within a byte are referred to as a “nibble”, so this mask when used with bitwise AND gives us the low-order nibble of **value**. To find out what the high-order nibble of **value** contains, we can use the bit shift operator **>>** to shift the upper nibble into the lower nibble and then mask it as before.

(Note that the mask is not really needed to find the upper nibble because **value** is an **unsigned char** so the upper 4 bits are padded with 0’s. If **value** were a **signed char**, we would need the mask to set the upper 4 bits to 0.)

## Bitwise operator exercises

To fully understand bitwise operations, you need to practice with them. Here are a few questions that you can use for practice:

1. Write a function **swapNibbles** that will take an **unsigned char** and make the low-order 4 bits into the high-order 4 bits and also make the high-order 4 bits into the low-order 4 bits. So for instance, passing in 11010011 would return 00111101. The signature would be:

unsigned char swapNibbles(unsigned char a)

1. Write a function **blendNibbles** that will take two **unsigned char**s and return a result which contains the high-order nibble of the first **unsigned char** and then (as the low-order nibble of the result) the high-order nibble of the second **unsigned char**. So for instance, given 10001111 and 01101110, the return value would be 10000110. The signature would be:

unsigned char blendNibbles(unsigned char first, unsigned char second)

1. Write a function called **reverseBits**. Given an unsigned integer, return another unsigned integer where the bits are reversed. So if 0x0000008D is passed in, 0xB1000000 would be returned (0000000000000000000010001101 would return 10110001000000000000000000000000). The signature would be:

unsigned int reverseBits(unsigned int data)

Note that this is tricker than questions 1 and 2, which just involve some simple bitwise operations!

1. Check out the chart at <https://en.cppreference.com/w/c/language/operator_precedence> for the precedence of operators in C. Write a function called **testPrecedence**. Write some test cases to test the precedence of various operators. For example, for **x | y << 1**, does **|** or **<<** have precedence? The signature would be:

void testPrecedence()

1. Write a function that will ask the user for a y/n answer asking if they speak various languages. For example:

Do you speak English (y/n)?

Do you speak French (y/n)?

Do you speak Plains Cree (y/n)?

Have the function return a single **unsigned char** with bits set for each answer (1 for yes, 0 for no). You will have to consider how to create appropriate masks to set a *flag* for each language separately. You will call it as follows:

unsigned char languages = languageQuestions();

Then write a second function that will take in an **unsigned char** and will print greetings for each language that the **unsigned char** has bits set for. For example:

Hello

Bonjour

Tan’si

You will have to use the flags that you created to determine which languages are set. You will call this second function as follows:

printGreetings(languages);

1. Create a new project (or write functions within this project) that will ask for an input filename and an output filename. Then read each byte from the input file (use **fgetc**), combine that byte with a byte from the file **random.pad** (in the **Exercises** folder or below) using XOR. Write the combined byte to the output file (use **fputc**).

That will create an encrypted file using *random pad* or *one-time pad* encryption (see <https://en.wikipedia.org/wiki/One-time_pad>) which can be decrypted by using the encrypted file as the input filename.

Try it on the encrypted file **message.ciphertext** to decode the secret message that I created!





### Question 1 solution

*Question:* Write a function **swapNibbles** that will take an **unsigned char** and make the low-order 4 bits into the high-order 4 bits and also make the high-order 4 bits into the low-order 4 bits. So for instance, passing in 11010011 would return 00111101.

To create this function, we need to do two things:

* Move each nibble (from high to low and low to high) – this can be accomplished using bit shift operators. Since we are dealing with an **unsigned char**, all the padding shifted in will be 0’s, so we don’t need to worry about masking the results.
* Combine the results – this can be accomplished using the bitwise OR operator.

The code is as follows. We shift left by 4 to move the low-order bits into the high-order bits, and shift right by 4 to move the high-order nibble into the low-order bits. Because the bit shift operators are padded with 0’s, we can simply use bitwise OR to combine the two.

unsigned char swapNibbles(unsigned char a)

{

return (a << 4) | (a >> 4);

}

Here is some test code. Change the value of **a** to ensure that it works with other bit patterns – for instance, the bit pattern listed in the question would be **0xD3**.

unsigned char a = 0xAB;

printf("%x swapped is %x\n", a, swapNibbles(a));

showBits(a, sizeof(unsigned char));

showBits(swapNibbles(a), sizeof(unsigned char));

The output would be:

ab swapped is ba

The value 171 in binary is 10101011

The value 186 in binary is 10111010

(Highlights show the swapped nibbles.)

### Question 2 solution

*Question:* Write a function **blendNibbles** that will take two **unsigned char**s and return a result which contains the high-order nibble of the first **unsigned char** and then (as the low-order nibble of the result) the high-order nibble of the second **unsigned char**. So for instance, given 10001111 and 01101110, the return value would be 10000110.

To create this function, we need to:

* Mask the first value to get just the high-order nibble.
* Shift the second value to push the high-order nibble into the low-order nibble – because we are using an **unsigned char**, the padding will be 0’s.
* Combine the two nibbles using bitwise OR.

The code is:

unsigned char blendNibbles(unsigned char first, unsigned char second)

{

return (first & 0xF0) | (second >> 4);

}

Some test code follows:

unsigned char a = 0xAC;

unsigned char b = 0xDC;

printf("Blended value is %x\n", blendNibbles(a, b));

printf("Blended value is %x\n", blendNibbles(b, a));

Sample output would be as follows – note that the order of the operands determines the order of the nibbles in the result:

Blended value is ad

Blended value is da

### Question 3 solution

*Question:* Write a function called **reverseBits**. Given an unsigned integer, return another unsigned integer where the bits are reversed. So if 0x0000008D is passed in, 0xB1000000 would be returned (0000000000000000000010001101 would return 10110001000000000000000000000000).

To create this function, we need to:

* Look at each bit (like we did in **showBits**). We can do so by using a mask with a 1 set in just the bit position that we want to examine, then using bitwise AND to determine if that bit is 1 or not.
* Set the opposite bit in the reversed value. To make it easier, we can start with a result of 0. Then we only need to set the bits that are 1’s in the reversed value. To set the bit, we just use bitwise OR to combine the result with a mask that sets the opposite bit.

The code would be as follows:

unsigned int reverseBits(unsigned int data)

{

unsigned char totalBits = sizeof(data) \* 8; // determine number of bits

unsigned int reverse = 0; // start with all 0's in the result

unsigned int temp = 0;

for (int i = 0; i < totalBits; i++) // for each bit

{

temp = data & (1 << i); // get the bit in position i using a mask

if (temp) // if the bit is a 1

{

// set the opposite bit in the result

// Note that bitwise operators can be combined with =

reverse |= 1 << (totalBits - 1 - i);

}

}

return reverse;

}

Some sample code follows:

unsigned int x = 0x0012FF7C;

unsigned int rev = reverseBits(x);

showBits(x, sizeof(int));

showBits(rev, sizeof(int));

With sample output as follows (reformatted to line things up):

The value 1245052 in binary is 00000000000100101111111101111100

The value 1056917504 in binary is 00111110111111110100100000000000

### Question 4 solution

*Question:* Check out the chart at <https://en.cppreference.com/w/c/language/operator_precedence> for the precedence of operators in C. Write a function called **testPrecedence**. Write some test cases to test the precedence of various operators. For example, for **x | y << 1**, does **|** or **<<** have precedence?

Here is some sample code for the situation described above and a couple of other situations – try out other precedence cases on your own:

void testPrecedence()

{

unsigned char value1 = 1;

unsigned char value2 = 1;

// Prints a 1

showBits(value1 | value2, sizeof(char));

// Prints 11 if shift has precedence, or prints 10 if | has precedence

// From the precedence chart or from the example, << has precedence

showBits(value1 | value2 << 1, sizeof(char));

// Since << has predence followed by AND, result is 11

showBits(value1 << 1 | value1 & value2, sizeof(char));

// Adding parentheses makes the result 1

showBits((value1 << 1 | value1) & value2, sizeof(char));

// Adding parentheses here makes the result 10

showBits(value1 << (1 | value1 & value2), sizeof(char));

// XOR has precedence over OR, so result is 111

showBits(0x07 | 0x05 ^ 0x02, sizeof(int));

// Adding parentheses makes the result 101

showBits((0x07 | 0x05) ^ 0x02, sizeof(int));

}

Sample output:

The value 1 in binary is 00000001

The value 3 in binary is 00000011

The value 3 in binary is 00000011

The value 1 in binary is 00000001

The value 2 in binary is 00000010

The value 7 in binary is 00000000000000000000000000000111

The value 5 in binary is 00000000000000000000000000000101

A good guideline for working with precedence of bitwise operators is that if you feel that the expression is too complex, use parentheses or break the expression into multiple statements for readability.

### Question 5 solution

*Question:* Write a function that will ask the user for a y/n answer asking if they speak various languages. For example:

Do you speak English (y/n)?

Do you speak French (y/n)?

Do you speak Plains Cree (y/n)?

Have the function return a single **unsigned char** with bits set for each answer (1 for yes, 0 for no). You will have to consider how to create appropriate masks to set a *flag* for each language separately.

First, we will have to decide which bit to set for each language. (Feel free to use different or additional languages if you want.) I will set the lowest bit for English, the second bit for French, and the third bit for Plains Cree. Thus, I create the following masks:

#define ENGLISH\_SIGNAL 0x01 // 0000 0001

#define FRENCH\_SIGNAL 0x02 // 0000 0010

#define CREE\_SIGNAL 0x04 // 0000 0100

I will also need to get an answer from the user, so I’ll set a buffer size for the answer:

#define ANSWER\_SIZE 5

We will start with a result set to 0 to indicate that no languages are known. Then we ask the user about their knowledge of each language. If the answer is 'y', we can use bitwise OR to set the appropriate bit with the signals defined above. The code is as follows:

unsigned char languageQuestions()

{

unsigned char languages = 0;

char answer[ANSWER\_SIZE];

printf("Do you speak English (y/n)? ");

gets\_s(answer, ANSWER\_SIZE);

if (answer[0] == 'y')

{

languages |= ENGLISH\_SIGNAL;

}

printf("Do you speak French (y/n)? ");

gets\_s(answer, ANSWER\_SIZE);

if (answer[0] == 'y')

{

languages |= FRENCH\_SIGNAL;

}

printf("Do you speak Plains Cree (y/n)? ");

gets\_s(answer, ANSWER\_SIZE);

if (answer[0] == 'y')

{

languages |= CREE\_SIGNAL;

}

return languages;

}

*Question:* Then write a second function that will take in an **unsigned char** and will print greetings for each language that the **unsigned char** has bits set for. For example:

Hello

Bonjour

Tan’si

You will have to use the flags that you created to determine which languages are set.

For this question, we use the flags defined above to test if the bits are set in the **unsigned char** passed in. The test will use bitwise AND to isolate the bit we are interested in. We test each language separately and print out the corresponding greetings. The code is as follows:

void printGreetings(unsigned char languages)

{

if (languages & ENGLISH\_SIGNAL)

{

printf("Hello!\n");

}

if (languages & FRENCH\_SIGNAL)

{

printf("Bonjour!\n");

}

if (languages & CREE\_SIGNAL)

{

printf("Tan'si!\n");

// <https://apihtawikosisan.com/2011/04/tansi/>

}

}

Sample test code for the functions would be as follows:

unsigned char languages = languageQuestions();

printGreetings(languages);

With the following output – try different answers to see that the corresponding greetings are printed out:

Do you speak English (y/n)? y

Do you speak French (y/n)? n

Do you speak Plains Cree (y/n)? y

Hello!

Tan'si!

### Question 6 solution

*Question:* Create a new project (or write functions within this project) that will ask for an input filename and an output filename. Then read each byte from the input file (use **fgetc**), combine that byte with a byte from the file **random.pad** (in the **Exercises** folder or below) using XOR. Write the combined byte to the output file (use **fputc**).

For my solution, I will create a new project and practice good documentation. First, for the file:

/\*

Filename: program.c

Author: Sample Solution (solution9999)

Course: COSC 292 (Advanced Programming 2)

Instructor: Michael Grzesina

Date: Winter 2024

Purpose: Practice XOR by creating a program to encode/decode files

using random pad encryption

\*/

Then I need to include the standard I/O library and the standard library, and define a size for the buffer where the user will enter filenames:

#include <stdio.h>

#include <stdlib.h>

#define BUFFER\_SIZE 256

Here is my documentation for the **main** function:

/\*

Function: main

Purpose: Run program that will ask for an input filename and an output

filename and open the input file (for reading), the output file

(for writing), and the file "random.pad" (for reading),

to be used by an encryption function.

Parameters: argc - number of command-line arguments (not used)

argv - array of command-line arguments (not used)

Returns: 0 (EXIT\_SUCCESS) if program exits successfully

Changes: Nothing

Assumptions: Files exist or are created in the proper folder

\*/

I could do some of the file I/O separately, but I will put it all in **main**. First, let’s open the input file:

int main(int argc, char\*\* argv)

{

char fileName[BUFFER\_SIZE];

FILE\* inputFile = NULL;

FILE\* outputFile = NULL;

FILE\* padFile = NULL;

int err;

printf("Enter an input filename: ");

gets\_s(fileName, BUFFER\_SIZE);

err = fopen\_s(&inputFile, fileName, "rb");

if (err == 0)

{

// TODO: open the output file and do some more stuff

fclose(inputFile);

}

else

{

fprintf(stderr, "Couldn't read input file\n");

}

return EXIT\_SUCCESS;

}

As you can see, I get the user to enter a filename into a buffer using **gets\_s**. Then I open the file using **fopen\_s**. If it opens successfully, we will do some more stuff later, but for now we just close the file. Otherwise, we print an error message and exit.

Note that I am using **fprintf** to print the error message to **stderr**. **stderr** is the standard error stream, which is by default console output (same as **stdout**). Printing to **stderr** does allow more flexibility if one of **stderr** or **stdout** is redirected.

Now we get to the “open the output file and do some more stuff” part. The next phase is opening the output file. Since it’s so similar to opening the input file, we could break this code out into a separate function, but for this simple example, I’ll just open it directly here:

printf("Enter an output filename: ");

gets\_s(fileName, BUFFER\_SIZE);

err = fopen\_s(&outputFile, fileName, "wb");

if (err == 0)

{

// TODO: open the random pad file and do some more stuff

fclose(outputFile);

}

else

{

fprintf(stderr, "Couldn't create output file\n");

}

Since this code is basically the same as opening the input file, I will provide no further details.

Next, we open the random pad file. In practice, we would always want a different random pad for each input file, but for this exercise, I will just hardcode the filename as “random.pad” and always use the same random pad.

err = fopen\_s(&padFile, "random.pad", "rb");

if (err == 0)

{

encryptMessage(inputFile, outputFile, padFile);

fclose(padFile);

}

else

{

fprintf(stderr, "Couldn't open random pad\n");

}

Opening the random pad file is the same as before except that we are not getting user input for the filename.

Once all files are opened, we call **encryptMessage** with the three **FILE\*** variables for the open files. The documentation for **encryptMessage** is as follows:

/\*

Function: encryptMessage

Purpose: Take bytes from an input file and a pad file and combine them

using XOR and then save them to an output file

Parameters: inputFile - file pointer to opened input file (for reading)

outputFile - file pointer to opened output file (for writing)

padFile - file pointer to opened pad file (for reading)

Returns: Nothing

Changes: outputFile to an encrypted or unencrypted message

Assumptions: padFile contains at least as many bytes as inputFile

\*/

I am going to do this in a simple manner, reading and writing one byte at a time. This method does a lot of file I/O, so it is not the best approach, but it is the easiest. (We could also read the entire input file and pad file in, make sure that the pad file contains enough bytes to encode the entire input file, do the encryption in memory, and then write out the entire output file.)

You may have noticed that we haven’t done any bitwise operations yet. Most of the code for this question is for file I/O. The encryption is simple: take each byte from the input file and each byte from the pad file and using bitwise XOR to combine them. If the pad is random, the results of encryption will be random characters (no patterns because even common letters like ‘e’ are encoded differently every time based on the pad byte). The code is:

void encryptMessage(FILE\* inputFile, FILE\* outputFile, FILE\* padFile)

{

int inputChar, padChar, outputChar;

inputChar = fgetc(inputFile); // prime the loop by trying to read

while (!feof(inputFile)) // while we haven't reached the end of the file

{

padChar = fgetc(padFile); // get the pad character

outputChar = inputChar ^ padChar; // XOR with the input byte

fputc(outputChar, outputFile); // write out encrypted/decrypted byte

inputChar = fgetc(inputFile); // try to read the next byte

}

}

Note that this method could encode any type of file, including Word documents or images. We will just use some simple text files since the size of our “random.pad” file is only 1024 bytes. (If you want some more file I/O practice, try creating your own random pad by randomly generating values from 0 to 255 and saving the values!)

Suppose that our text file contains:

This is some input.

eeeeeeeeeeeeeeeeee.

I have included a string of e’s to emphasize the random nature of the encoding.

Running this text file with “random.pad” will produce the following results:

TçX½µ\*Ä¢ÑA¾•ë#sb(

'ö{dgâôüÿOÌ<kÿ¢©áb

Note that the characters all appear to be random, including the string of all e’s. This would be the secret file that we would send to our recipient. If intercepted, it just looks like random bytes – it cannot be decoded without the random pad.

To decode, our recipient would need to have the same random pad. (One of the main problems with this encryption method is how to get the random pad securely from the encoder to the recipient.) Then they would just run the program using the encrypted file as the input file. Using XOR with the same bytes will restore the original characters.

This example shows how XOR can be used for encryption, even though most of it is just practice with file I/O!

## Representing masks in binary

In the examples above, we have represented our masks in hexadecimal (for instance, we used the mask **0x04** for 0000 0100). In C23 (the 2023 standard for C) and in some earlier compilers (including Visual Studio 2022 because it was already in the C++ standard), you can also use a binary literal using **0b** in front, such as **0b00000100** or just **0b100**.

You can even use *tick separators* (the apostrophe or ') to break up a long binary or hexadecimal value, or even a decimal value – for instance, **65'535** = **0xFF'FF** or **0b1111'1111'1111'1111**.

The C23 standard also includes a **%b** format specifier for **printf** or **scanf** to print or read binary values. Much easier than our **showBits** function (but less educational, and not supported by Visual Studio 2022).

For more details, see:

Binary Notation in C23: <https://c-for-dummies.com/blog/?p=6173>

Tick Separators: <https://c-for-dummies.com/blog/?p=5675>

Or <https://thephd.dev/c-the-improvements-june-september-virtual-c-meeting#n2626---digit-separators>

# Typedefs and Function Pointers

## Type Synonyms Using **typedef**

**typedef** is a tool to provide a synonym for an existing type. The **typedef** is handled by the compiler (not the preprocessor).

Besides purely aesthetic issues (making the program nicer to look at and easier to read), there are two main reasons for using **typdef**:

1. To parameterize a program against portability issues. If typedefs are used for data types that may be machine-dependent, only the typedefs need to change when the program is moved. One common situation is to use typedef names for various integer quantities, then make an appropriate set of choices from **short**/**int**/**long**/**long long** for each host machine. Typedefs like **size\_t** from the library are examples.
2. To provide better documentation for the program – a type called **TreePtr** may be easier to understand than one declared only as a pointer to a complicated structure.

To illustrate the first reason, create a program as follows:

#include <stdio.h>

#include <stdlib.h>

int main()

{

int iArray[] = { 45, 56, 626262, 30, 2 };

size\_t iSize = sizeof(iArray) / sizeof(int);

printf("Size of iSize: %zu\n", sizeof(iSize));

return EXIT\_SUCCESS;

}

If you run this program using the **x64** platform, you will see that it prints out a size of **8**. But if you run the program using the **x86** platform, you will see that it prints out a size of **4**. This difference is because **size\_t** is set with **typedef** in **<stdio.h>** and **<stdlib.h>** and has different settings (**unsigned long long** versus **unsigned int**) for different environments (**x64** versus **x86**).

You can see the actual **typedef** for **size\_t** by hovering over **size\_t** in the code above.

### Syntax

**typedef** *ExistingType NewType*;

Note the **;** at the end since it is a compiler command not a preprocessor directive.

A **typedef** declaration does not create a new type in any sense – it merely adds a new name for some existing type. Also, there are not any new semantics. Variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, **typedef** is like **#define**, except that since it is handled by the compiler, it can copy with textual substitutions that are beyond the capabilities of the preprocessor. For example, **typedef** can be used with function pointers (see next section).

### Examples

#### Example 1

**typedef** **int** Length;

This statement makes the name Length a synonym for **int**. The type Length can be used in declarations, casts, etc. in exactly the same ways that the **int** type can be used:

Length len, maxLen;

Length\* lengths[];

To try it out in code, add a header file to your program and add the declaration for the Length type in the header file. (We often put **typedef** statements in a header file so they will be available across multiple source files.)

#ifndef FUNCS\_H

#define FUNCS\_H

typedef int Length;

#endif // !FUNCS\_H

Then we can use Length in our program as follows:

#include "funcs.h"

void testTypedefs()

{

Length x = 3;

Length y = 5;

printf("x plus y = %d\n", x + y);

}

Running this function will print **8** since Length variables are just **int**s with a different name, so they can be added and printed using the **"%d"** specifier.

#### Example 2

**typedef float** REAL;

REAL x; // same as **float**

#### Example 3

**typedef char\*** String;

The statement above makes String a synonym for **char\*** or a **char** pointer, which may then be used in declarations and casts like the following:

String p, linePtr[MAXLINES]; // declaration

**int** strcomp(String, String); // function prototype

p = (String)**malloc**(100); // cast

Trying this out in code would add the following to our header file:

typedef char\* String;

And the following code:

String p;

p = "Hello world";

printf("The string is %s\n", p);

printf("The first character is %c\n", \*p);

Output would be:

The string is Hello world

The first character is H

Note that String can be referenced using the **"%s"** specifier and can be dereferenced (**\*p**) like any other pointer.

#### Example 4

**typedef unsigned short** UShort;

In code, we would add that statement to the header file:

typedef unsigned short UShort;

And we could then use it in code as follows:

UShort s = 60000;

printf("The value of s is %hu\n", s);

#### Example 5

We can use **typedef** with **enum**:

**typedef enum** Boolean { False, True } BOOL;

BOOL areWeDoneYet = False;

## Function Pointers in C

Recall that we can print the address of a function by printing the function’s name using the **"%p"** specifier. We can have pointers to functions that store this address and can then call the function using those function pointers.

A **typedef** can make function pointers easier to use. The basic syntax is:

**typedef** *returnType* (\**SOMENAME*)(*parameter list*);

where *returnType* is the return type of the function to be pointed to, *SOMENAME* is the name of the function pointer, and *parameter list* is the list of the parameter types of the function to be pointed to.

Where are function pointers used?

1. Callback functions – pass a function to another function. That function will “call back” the function that you used as a function pointer. This usage will be the primary example that we will use for this class.
2. Event handlers – event handlers are implemented internally with function pointers.
3. Functions in objects (methods) are implemented with function pointers.

### Example

**typedef int** (\*FNPTR\_TYPE)(**int**);

would be a declaration of a type named FNPTR\_TYPE where the function takes an **int** as a parameter and returns an **int**.

Let’s try that in code. The following code in our header file defines the function pointer type by providing the signature for the function pointer:

typedef int (\*FNPTR\_TYPE)(int);

In our main source code file, we could then have multiple functions that use that signature:

int doSquare(int x)

{

return x \* x;

}

int doAdd(int x)

{

return x + x;

}

int doNothing(int someNumber)

{

return someNumber;

}

We could then define a function that would take in a function pointer to operate on the data in an array passed to it. We could put this in a separate source file:

#include <stdio.h>

#include "funcs.h"

void iterateNumbers(int iArray[], int iSize, FNPTR\_TYPE myFunc)

{

int iVal = 0;

for (int i = 0; i < iSize; i++)

{

iVal = myFunc(iArray[i]); // use myFunc like any other function

printf("The value %d after the operation is %d\n", iArray[i], iVal);

}

printf("\n");

}

And then include the function prototype in our header file:

void iterateNumbers(int iArray[], int iSize, FNPTR\_TYPE myFunc);

Note that the third parameter to **iterateNumbers** is a function pointer. This parameter can be used like any other function within the body of the **iterateNumbers** function, but it will call whatever function is passed into it. So in **main**, we could have:

iterateNumbers(iArray, iSize, doSquare);

iterateNumbers(iArray, iSize, doAdd);

iterateNumbers(iArray, iSize, doNothing);

Note that we pass a function name into **iterateNumbers** as the last argument. The name of the function stores its address, so we don’t need the **&** address of operator.

Output from these calls would be:

The value 45 after the operation is 2025

The value 56 after the operation is 3136

The value 626262 after the operation is 1362068708

The value 30 after the operation is 900

The value 2 after the operation is 4

The value 45 after the operation is 90

The value 56 after the operation is 112

The value 626262 after the operation is 1252524

The value 30 after the operation is 60

The value 2 after the operation is 4

The value 45 after the operation is 45

The value 56 after the operation is 56

The value 626262 after the operation is 626262

The value 30 after the operation is 30

The value 2 after the operation is 2

Note that each call to **iterateNumbers** uses a different function, and thus the values printed at the end are different for each call.

Do we need to match the signature of the function pointer? No, it will still compile (with a warning) and run, but may produce unpredictable results. If you do so, you need to make sure that there are no conversion errors from one type to the next. In general, using functions that do not have matching signatures is not a good idea.

To illustrate, add the following function to the main source file:

int dontMatchSignature(short x)

{

return (int)x;

}

And call it in **main** as follows:

iterateNumbers(iArray, iSize, dontMatchSignature);

You will see the output is:

The value 45 after the operation is 45

The value 56 after the operation is 56

The value 626262 after the operation is -29098

The value 30 after the operation is 30

The value 2 after the operation is 2

Note that it compiled and ran, but the value that couldn’t fit in a **short** ended up being printed as a negative number when converted to an **int**.

## typedef and function pointer exercises

Here are some questions that you can use to practice with **typedef** declarations and function pointers.

1. Write a sort function that will sort in ascending order or descending order based on a comparison function passed in. The function’s signature will be:

void sortInts(int iArray[], int iSize, COMPARER comp);

where **iArray** is an array of **int**s to be sorted, **iSize** is the number of elements in the array, and **comp** is a function pointer of type **COMPARER** that compares two **int**s.

When called, pass in one of two comparison functions. The comparison function should return a value greater than zero when the first item is higher in the sort order.

int compareToAscending(int first, int second)

{

return first - second;

}

int compareToDescending(int a, int b)

{

return b - a;

}

*Note: You can use a sort algorithm from a previous class, or find one on the Internet such as* [*https://www.geeksforgeeks.org/Insertion-sort*](https://www.geeksforgeeks.org/Insertion-sort) *or* [*https://www.geeksforgeeks.org/bubble-sort*](https://www.geeksforgeeks.org/bubble-sort) *(or use a fancier sort to practice your use of recursion!).*

***Extra challenge:*** *Write an additional comparison function that will sort the integers by their last digit (hint: use* ***% 10*** *to get the last digit).*

1. Some basic encrypting can be done using the XOR operator. Write an encrypting function with the signature:

void Encryptor(String s, BYTE key);

Use typedefs to create String (**unsigned char\***) and BYTE (**unsigned char**).

This function will loop through the string and encrypt its values one **char** at a time using the key and the XOR bit operator. You can test with the following code:

// Encrypting key can be any character.

BYTE key = '&';

unsigned char cMessage[] = "Rob Miller";

// Calling the method the first time will encrypt the string.

Encryptor(cMessage, key);

// Note that some characters may be converted to one of the

// "unprintable" characters. In this case, printing the message

// may produce garbled results.

printf("Encrypted message is %s\n", cMessage);

// Calling the method a second time will decrypt the message.

Encryptor(cMessage, key);

printf("Decrypted message is %s\n", cMessage);

1. Recall that a jagged array can be implemented with pointers and pointers to pointers. In this exercise, we will create a 2D rectangular array (not jagged). The code to manage the array will consist of:
   1. A function that will loop through each row of the array and call a function pointer on each row:

void twoDArrayRowModifier(int\*\* iArray, int iNumRows, ROW\_FUNC rf, int iRowSize);

The first parameter is the 2D array, and **ROW\_FUNC** is a typedefed function pointer that operates on each row.

For each row, call the function pointer passing in the current row. Note that since the row might be modified, assign the return value from **rf** back to the current row.

* 1. A typedefed function pointer that returns an **int\*** and takes an **int\*** (current row) and an **int** (row size).
  2. Several functions that will do something to each row:

//Create a new row using malloc.

//Return the pointer to the row.

int\* createRows(int\* iPtr, int iRowSize);

//Print the values of the current row.

int\* printRows(int\* iPtr, int iRowSize);

//Free the row and return null

int\* freeRows(int\* iPtr, int iRowSize);

You can add other functions like **enterRows** to enter values into the row and **doubleRows** to double the values in the row.

* 1. Some test code:

void testTwoDArray()

{

int\*\* iArray;

int iNumRows = 3;

int iRowSize = 3;

iArray = (int\*\*)malloc(iNumRows \* sizeof(int\*));

//Create each row

twoDArrayRowModifier(iArray, iNumRows, createRows, iRowSize);

// Can call enterRows and doubleRows here...

//Print each row

twoDArrayRowModifier(iArray, iNumRows, printRows, iRowSize);

//Free each row

twoDArrayRowModifier(iArray, iNumRows, freeRows, iRowSize);

free(iArray);

iArray = NULL;

}

### Question 1 solution

We will need the following **typedef** in our header file:

// The COMPARER function returns an int and takes in two ints.

typedef int (\*COMPARER)(int, int);

This function pointer can be used in the sort function as follows – the simple but inefficient insertion sort is used here:

// Insertion sort from https://www.geeksforgeeks.org/Insertion-sort

// comp should return > 0 when item is higher in the sort order

void sortInts(int iArray[], int iSize, COMPARER comp)

{

int i, key, j;

for (i = 1; i < iSize; i++)

{

key = iArray[i];

j = i - 1;

while (j >= 0 && comp(iArray[j], key) > 0)

{

iArray[j + 1] = iArray[j];

j = j - 1;

}

iArray[j + 1] = key;

}

}

*Alternative sort using bubble sort – also simple but inefficient:*

// https://www.geeksforgeeks.org/bubble-sort

void bubbleSortInts(int iArray[], int iSize, COMPARER comp)

{

int i, j;

for (i = 0; i < iSize - 1; i++)

{

// Last i elements are already in place

for (j = 0; j < iSize - i - 1; j++)

{

if (comp(iArray[j], iArray[j + 1]) > 0)

{

int temp = iArray[j];

iArray[j] = iArray[j + 1];

iArray[j + 1] = temp;

}

}

}

}

The sort method can be called from **main** as follows:

// When called, pass in one of the two compareTo methods:

sortInts(iArray, iSize, compareToAscending);

printf("Sorted ascending: ");

printArray(iArray, iSize);

sortInts(iArray, iSize, compareToDescending);

printf("Sorted descending: ");

printArray(iArray, iSize);

Which calls **sortInts** using the names of the comparison functions and uses the helper function **printArray**:

void printArray(int\* iArray, int iSize)

{

for (int i = 0; i < iSize; i++)

{

printf("%d ", iArray[i]);

}

printf("\n");

}

Producing the output for int iArray[] = { 45, 56, 626262, 30, 2 }:

Sorted ascending: 2 30 45 56 626262

Sorted descending: 626262 56 45 30 2

Answer to extra challenge: The following function takes the last digit of the two parameters using **% 10**, and then subtracts the second one from the first one so that it will sort in ascending order by the last digit:

int compareLastDigit(int a, int b)

{

int lastDigitA = a % 10;

int lastDigitB = b % 10;

return lastDigitA - lastDigitB;

}

Called from **main**:

sortInts(iArray, iSize, compareLastDigit);

printf("Sorted by last digit: ");

printArray(iArray, iSize);

It produces the following output for int iArray[] = { 45, 56, 626262, 30, 2 }:

Sorted by last digit: 30 626262 2 45 56

### Question 2 solution

First, we add the typedefs and function prototype to our header file:

typedef unsigned char\* String;

typedef unsigned char BYTE;

void Encryptor(String s, BYTE key);

Then we can loop through the string and XOR each byte:

void Encryptor(String s, BYTE key)

{

while (\*s) // \*s != '\0'

{

\*s ^= key;

s++;

}

}

Note that since String is equivalent to **char\***, we can dereference it (to find the current character until we reach the null terminator) and increment it (to move to the next character). We also dereference it to XOR it with the key and assign the resulting value back to the byte.

With the test code, it produces the following output:

Encrypted message is tIDkOJJCT

Decrypted message is Rob Miller

There are also other ways to write the **Encryptor** function, such as the following:

void Encryptor(String s, BYTE key)

{

int i = 0;

while (s[i]) // s[i] != '\0'

{

s[i] ^= key;

i++;

}

}

Note that array notation is fine for a String variable, since it is really just a pointer.

### Question 3 solution

First, we should do part (b). To typedef a function pointer that returns an **int\*** and takes an **int\*** (current row) and an **int** (row size), we add the following **typedef** to our header file along with the function prototype:

typedef int\* (\*ROW\_FUNC)(int\*, int);

void twoDArrayRowModifier(int\*\* iArray, int iNumRows, ROW\_FUNC rf, int iRowSize);

Now we can go back to part (a). All that is needed here is a loop to go through every row and call the row function passed in on it (using it as a callback function). We assign it back to the current row in case the row function changes the row:

void twoDArrayRowModifier(int\*\* iArray, int iNumRows, ROW\_FUNC rf, int iRowSize)

{

for (int i = 0; i < iNumRows; i++)

{

iArray[i] = rf(iArray[i], iRowSize);

}

}

We can now do part (c), writing functions that match **ROW\_FUNC**. First, **createRows** will just use **malloc** to create enough space for the row:

// Create a new row using malloc.

// Return the pointer to the row.

int\* createRows(int\* iPtr, int iRowSize)

{

return (int\*)malloc(iRowSize \* sizeof(int));

}

Next, **printRows** will print the values in each row. We already have a helper function to print an array, so we can just call that:

//Print the values of the current row.

int\* printRows(int\* iPtr, int iRowSize)

{

printArray(iPtr, iRowSize);

return iPtr;

}

Finally, we can free the rows in **freeRows**:

//Free the row and return null

int\* freeRows(int\* iPtr, int iRowSize)

{

free(iPtr);

return NULL;

}

Output from the test code would then be something like the following – note the values that show up will depend on what is in the space on the heap that **malloc** accessed:

-842150451 -842150451 -842150451

-842150451 -842150451 -842150451

-842150451 -842150451 -842150451

To actually place values in each row and then double them, we can add the following code to **testTwoDArray**:

// Can call enterRows and doubleRows here...

twoDArrayRowModifier(iArray, iNumRows, enterRows, iRowSize);

twoDArrayRowModifier(iArray, iNumRows, doubleRows, iRowSize);

The function **enterRows** is as follows:

// BONUS FUNCTION

// Assign values to each row

int\* enterRows(int\* iPtr, int iRowSize)

{

for (int i = 0; i < iRowSize; i++)

{

// iPtr[i] = i; // if we just want to count through the row

printf("Enter integer value %d for current row: ", (i + 1));

scanf("%d", iPtr + i); // or &iPtr[i], but iPtr + i is better

}

return iPtr;

}

Note that we just go through each item in the given row and ask the user for a value (or if you just want to count through the row, set the item to **i**):

// BONUS FUNCTION

// Assign values to each row

int\* enterRows(int\* iPtr, int iRowSize)

{

for (int i = 0; i < iRowSize; i++)

{

// iPtr[i] = i; // if we just want to count through the row

printf("Enter integer value %d for current row: ", (i + 1));

scanf("%d", iPtr + i); // or &iPtr[i], but iPtr + i is better

}

return iPtr;

}

Finally, **doubleRows** just goes through each item in the given row and multiplies it by 2. Note that although the array pointer does not change, we still return it so that it can be assigned back to the array pointer in the calling function (same as in **enterRows** and **printRows**):

// BONUS FUNCTION

// Double the values in each row

int\* doubleRows(int\* iPtr, int iRowSize)

{

for (int i = 0; i < iRowSize; i++)

{

\*(iPtr + i) \*= 2; // or iPtr[i] \*= 2;

}

return iPtr;

}

Sample input and output follows:

Enter integer value 1 for current row: 1

Enter integer value 2 for current row: 2

Enter integer value 3 for current row: 3

Enter integer value 1 for current row: 10

Enter integer value 2 for current row: 20

Enter integer value 3 for current row: 30

Enter integer value 1 for current row: 100

Enter integer value 2 for current row: 200

Enter integer value 3 for current row: 300

2 4 6

20 40 60

200 400 600

# Structures in C

A structure is a user-defined data type in the C language that allows data of different types to be combined into a single entity. It is somewhat like an array in that multiple items can be stored, but an array holds data of similar types only that are referenced by an index. A structure, on the other hand, can store data of any type, with data accessed by name. In the C language, a structure is declared using the keyword **struct**.

Structs, also known as records, are similar to classes in Java and C# with the exception that structs do not contain methods in C (but they do in C++). Structs contain attributes (primitive types, pointers, arrays, function pointers, other structs). There is no support for inheritance or polymorphism.

## Defining a **struct**

The **struct** keyword is used to define a structure. **struct** defines a new data type that is a collection of primitive and user-defined data types.

The basic syntax for declaring a struct is:

struct [structure\_tab]

{

// member variable 1

// member variable 2

...

} [structure variables];

Where [structure\_tab] is the name of the type of structure – similar to a class name in Java or C#.

// member variable 1 Is a variable representing a primitive, an array, a pointer, a function pointer, or even another **struct**. No practical limit on the number of member variables. Same idea as an attribute in an object. Access modifiers such as private or public do not apply.

[structure variables] are rarely used, but if only a few instance of the struct are required, the instance variables can be declared here.

For example, suppose we want to track information regarding a person’s birth date. The following **struct** declared a BirthDate **struct** type:

struct BirthDate

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

};

If we only want a few instances of this **struct**, we could add structure variable names:

struct BirthDate

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

} B1, B2;

Note that although this syntax is valid, adding variables names is rarely used.

Now that we have a struct **defined**, how is it declared?

As a variable:

struct BirthDate bd1;

As a parameter to a function:

void DisplayBirthDate(struct BirthDate bd)

## Using **typedef** to define a **struct**

Note that putting the keyword **struct** is required prior to the **struct**’s type in both cases above. Putting the keyword **struct** in front of every single declaration becomes tedious. Fortunately, there is a way around it using a **typedef**.

For example, the previously declared BirthDate **struct** could be declared as follows:

typedef struct

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

} BirthDateTD;

Now, how is the **struct** declared? Note the absence of the keyword **struct**.

As a variable:

BirthDateTD bd = { "Mike", 'm', 31, 12, 1980 };

As a parameter to a function:

void DisplayBirthDate(BirthDateTD bd)

Note that default values can be assigned using an initialization list in curly braces. This is similar to an array. When you think about it, a **struct** is similar to an array in that both take up a continuous block of memory and represent multiple values, but in a **struct**, each member is possibly a different data type (accessed by the member name), whereas in an array, each member is of the same data type (accessed by an index).

All the examples of **struct**s so far are declared on the stack. Often, we will need to dynamically allocate a **struct**. **struct**s can be declared on the heap using the function **malloc**:

BirthDateTD\* bdPtr = NULL;

bdPtr = (BirthDateTD\*)malloc(sizeof(BirthDateTD));

In the first line of code, a pointer is created and assigned NULL. Note that this does *not* allocate memory for the **struct**. The second line of code actually allocates memory for the **struct**.

## Accessing attributes of a **struct**

The method used to access the attributes of a **struct** depends on the type of reference that you have to the **struct**.

### A **struct** reference

Given a struct variable that is *not* a pointer, use the familiar dot syntax.

So for the structure declared as:

BirthDateTD bd1;

The **day** attribute can be accessed as follows:

bd1.day = 10;

### A pointer to a **struct**

Given a pointer to a **struct**, the pointer must be dereferenced first, and then the attribute can be accessed. For instance, for:

BirthDateTD\* bdPtr = NULL;

bdPtr = (BirthDateTD\*)malloc(sizeof(BirthDateTD));

The **day** attribute can be accessed as follows:

(\*bdPtr).day = 31;

Note that this is a clumsy way of accessing an attribute, as we need to dereference in parentheses (because the dot operator has precedence over the dereference operator otherwise), and then use the dot operator. Fortunately, for pointers to **struct**s, we can use the *arrow operator* as follows:

bdPtr->day = 31;

## **struct** examples

Let’s try out some of these examples in code.

### Basic **struct**s and **typedef**

**struct**s are generally defined in a header file (so they can be made available in multiple source files). So for instance, we could have the following in a file **structs.h**:

#ifndef STRUCTS\_H

#define STRUCTS\_H

#define NAME\_SIZE 5

// How to define a struct without a typedef.

// We rarely create structs this way - too cumbersome.

struct BirthDate

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

};

// Example of declaring a struct using a typedef

typedef struct

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

} BirthDateTD;

#endif // !STRUCTS\_H

In our **program.c** source file, we could then have a **main** function like:

int main()

{

printf("Size of struct BirthDate is: %zu\n", sizeof(struct BirthDate));

printf("Size of BirthDateTD is: %zu\n", sizeof(BirthDateTD));

return EXIT\_SUCCESS;

}

How big is a **struct**? If you run the code above, you will see that it shows a size of **20**. The size of the **struct** depends on the order that the attributes are declared, and sometimes the type of attributes.

By default (on this system), **struct**s will align on 4-byte boundaries (or 8-byte boundaries if you are using 8-byte data types like pointers on an **x64** system or **long long int** values). With 4-byte boundaries, an **int** will always align on a 4-byte boundary.

The 5 bytes for **cName** (based on the **NAME\_SIZE** defined as 5) will actually take up 8 bytes, even though only 5 are required. With **cGender** placed immediately after **cName**, the gender will fit in the remaining 3 bytes assigned to **cName**, thus giving us a total of 5 + 1 + *2 for alignment* + 4 + 4 + 4 = 20 bytes.

But if **cGender** is added as the last attribute, **cGender** will take up an entire 4-byte chunk. You can see this by changing the order of the attributes in the header file. The resulting size will be **24** bytes (5 + *3 for alignment* + 4 + 4 + 4 + 1 + *3 for alignment*). So with the byte alignment set to anything other than 1, the order that you declare attributes can affect the size of the overall structure.

In C, we can set the byte alignment for **struct**s using the **#pragma pack(*value*)** directive. For instance, go to the header file and change the second definition (using the **typedef**) as follows:

#pragma pack(push) // Used to store the default byte alignment

#pragma pack(1) // Set the byte alignment to 1

// Example of declaring a struct using a typedef

typedef struct

{

// Attributes

char cName[NAME\_SIZE];

char cGender;

int day;

int month;

int year;

} BirthDateTD;

#pragma pack(pop) // Used to restore the default byte alignment

You will see that its size now prints as **18** (5 + 1 + 4 + 4 + 4), and will stay at **18** even if **cGender** is moved to the end.

Now let’s try setting up a **struct**. Add a function **testStructureBasicFormat** as follows and call it in **main**:

void testStructureBasicFormat()

{

// Does this create a struct or a reference to a struct? A struct

struct BirthDate bd1;

// Try to add a value to a struct member.

// It works. This struct is created on the stack.

bd1.day = 17;

}

Now let’s add a separate source file **structs.c** and create a function to display a basic **struct** – don’t forget to prototype **DisplayBirthDateBasic** in **structs.h**!

#include <stdio.h>

#include "structs.h"

void DisplayBirthDateBasic(struct BirthDate bd)

{

printf("\nName is %s\n", bd.cName);

printf("Gender is %c\n", bd.cGender);

printf("Date is %d %d %d\n", bd.day, bd.month, bd.year);

}

Call that function from **testStructureBasicFormat**:

DisplayBirthDateBasic(bd1);

You will see that the **day** attribute displays as **17**, but the other attributes will just display random data (which will depend on whether you are in debug or release mode and what is in the stack memory at the time that the **struct** is declared). If you do not initialize a **struct**’s members, they will still exist, but will not be set to anything meaningful.

You can set all the initial values using an initializer in curly braces:

struct BirthDate bd2 = { "Ada", 'f', 21, 3, 1987 };

DisplayBirthDateBasic(bd2);

Note that all the members now display the corresponding data. But what happens if your initializer doesn’t contain all the right data or the data in the right order? Try the following:

struct BirthDate bd2 = { "Ada", 21, 3, 1987 };

You will see that the name displays correctly, but other attributes are set inconsistently – it uses 21 to set the gender, then 3 and 1987 to set the day and month, and the year is not set (so the year may be 0 or some random value).

Now try the following:

struct BirthDate bd2 = { "Ada", 21, 3, 1987, 'f' };

You will see the values are again wrong, this time with the year set to 102 (the ASCII value for **'f'**). So as you can see, C does not check if your initializer is in the correct order or has the correct number of values – it will just set the members that you have provided values for in the order that they appear. Be careful when using **struct** initializers!

Fix your initializer, and then try accessing the attributes as follows:

// Access an attribute

bd2.month = 4;

DisplayBirthDateBasic(bd2);

You should see that even after a **struct** has been initialized, we can still access and change the member values.

A **struct** is just a block of memory, so although it would not generally be good practice, you could also manipulate the memory directly. For instance, try the following code:

\*(&bd2.day + 2) = 1999;

DisplayBirthDateBasic(bd2);

You will see that the year now displays as **1999** (since day is an **int**, adding 2 to its address adds the size of two **int**s, which moves it past the month and into the year).

Note that we had to use the keyword **struct** in the variable and function declarations above. As mentioned previously, we can avoid this extra keyword by using a **typedef** to create a struct. We have already defined **BirthDateTD** as a **struct** type using a **typedef**, so we can try using it.

Add a function **testStructureDeclarationsWithTypedef** as follows (and call it from **main**):

void testStructureDeclarationsWithTypedef()

{

BirthDateTD bd = { "Mike", 'm', 31, 12, 1980 };

DisplayBirthDateTypedef(bd);

}

We will also need a function in **structs.c** (prototyped in **structs.h**) to display the birth date as follows:

void DisplayBirthDateTypedef(BirthDateTD bd)

{

printf("\nName is %s\n", bd.cName);

printf("Gender is %c\n", bd.cGender);

printf("Date is %d %d %d\n", bd.day, bd.month, bd.year);

}

The results will be similar to those before (with the new data), showing that we can declare a variable, pass in a parameter, or use an initializer with the variable using **typedef**.

What happens if we try to do assignment with a **struct**? Try modifying **testStructureDeclarationsWithTypedef** as follows:

void testStructureDeclarationsWithTypedef()

{

BirthDateTD bd = { "Mike", 'm', 31, 12, 1980 };

// Can we do assignment?

BirthDateTD bd1 = bd;

// Modify bd1 – does it also modify bd?

bd1.day = 1;

DisplayBirthDateTypedef(bd);

DisplayBirthDateTypedef(bd1);

}

You will see that we can assign a **struct** to another **struct** (BirthDateTD bd1 = bd;). When we change the day in the new **bd1**, it does *not* change the day in **bd**. This shows that we have created a new copy of the **struct** with its own memory. **struct** variables are not just references, but rather copies of the block of data. (Note how this is different from objects in Java, where a Date variable, for instance, is just a reference to the Date object, and assigning it to another Date variable still refers to the same object.)

Similarly, we can ask if **struct**s are pass by value or pass by reference when we pass them into a function. Try the following:

// Are structs pass by value or pass by reference?

AlterBirthDate(bd1);

DisplayBirthDateTypedef(bd1);

with the function **AlterBirthDate** as follows:

void AlterBirthDate(BirthDateTD bd)

{

printf("\nSize of passed in BirthDateTD is %zu\n", sizeof(bd)); // size 18

bd.year = 2001;

DisplayBirthDateTypedef(bd); // will display 2001

}

You will see that the size of the passed in **BirthDateTD** is **18**, so we are passing in a copy of the **struct**. The call to display the **struct** will print **2001** in the **AlterBirthDate** function, but it will still print **1980** in the calling function.

What if we want to change an attribute of the current **struct** in another function? Just like an **int** or other primitive, we can pass the address of the **struct** to the other function. Try the following:

AlterBirthDateByRef(&bd1);

DisplayBirthDateTypedef(bd1);

with the function **AlterBirthDateByRef** as follows:

void AlterBirthDateByRef(BirthDateTD\* bdPtr)

{

printf("\nSize of passed in bdPtr: %zu\n", sizeof(bdPtr));

(\*bdPtr).year = 2023;

}

Note that we now pass in a pointer to the **struct**, so we are still referring to the same struct as in the calling function. The size of the pointer will be 8 bytes (on the **x64** platform) or 4 bytes (on the **x86** platform), since we are not passing in a **struct**. So to change the value of the attribute, we need to dereference the pointer first.

The code (\*bdPtr).year = 2023; is clumsy. We can replace it with the arrow operator as follows:

bdPtr->year = 2023;

### Dynamically allocated **struct**s

What if we want to dynamically allocate a **struct**? Let’s try it with a **testDynamicallyAllocatedStructs** function (called from **main**):

void testDynamicallyAllocatedStructs()

{

BirthDateTD\* bdPtr = NULL;

// Try to access an attribute.

// Crashes program - no memory allocated.

bdPtr->day = 12;

}

Note that creating a pointer does *not* create memory for the **struct**, so the code above would crash the program. Update it as follows:

void testDynamicallyAllocatedStructs()

{

BirthDateTD\* bdPtr = NULL;

bdPtr = (BirthDateTD\*)malloc(sizeof(BirthDateTD));

if (bdPtr != NULL)

{

// Set the day

bdPtr->day = 13;

// Set the name

strcpy\_s(bdPtr->cName, NAME\_SIZE, "Mike");

DisplayBirthDateTypedef(\*bdPtr);

// Uninitialized variables are just whatever was on heap.

// If we do declare a dynamically allocated struct, we must free it.

// Make sure to free dynamically allocated attributes first, if any.

free(bdPtr);

bdPtr = NULL;

}

}

Note the following:

* Don’t forget to include **<string.h>** for the **strcpy** or **strcpy\_s** function.
* We use **malloc** to allocate the size of the **struct**.
* **malloc** can fail, so we can test if it is NULL before using the memory.
* We set the value using the arrow operator since we have a pointer to the **struct**’s memory.
* We dereference the pointer to call **DisplayBirthDateTypedef** because it takes a **struct** instance, not a pointer.
* Uninitialized variables are just whatever was on the heap.
* Remember to free the **struct** (and dynamically allocated attributes, if any) when done to avoid a memory leak.

### **struct**s with dynamically allocated attributes

Add the following **struct** to **structs.h**:

typedef struct

{

// Some of the attributes are dynamically allocated

char\* cNamePtr;

int iStudentNum;

// Array of marks

int\* iMarksPtr;

// Number of marks in the marks array

int iMarksCount;

} student;

If you check the size of **student**, you will see that it varies depending on whether you are on an **x64** platform (where it will be 32 bytes, since it aligns the attributes on 8-byte boundaries to match the 8-byte addresses) or an **x86** platform (where it will be 16 byes, 4 bytes per attribute).

Note that **student** does not contain a fixed length for the name – instead, we just use a pointer as an attribute. This is because we will dynamically allocate the name and assign the dynamically allocated memory to the pointer.

Similarly, the array of marks will be dynamically allocated and assigned to the pointer. Note that we need to also have an attribute for the length of the array, since otherwise we will not know how big the dynamically allocated memory is. (We don’t require such an attribute for the name because we can look for the null terminator to determine the size of the name.)

Write a **testStudent** function as follows (and call it from **main**):

void testStudent()

{

int iMarksArray[] = { 95, 100, 85 };

student\* sPtr = CreateStudent("Mike Grzesina", 123456789, iMarksArray,

sizeof(iMarksArray) / sizeof(int));

DisplayStudent(\*sPtr);

FreeStudent(sPtr);

sPtr = NULL;

}

We will now need to add functions for **CreateStudent**, **DisplayStudent**, and **FreeStudent**.

**CreateStudent** will be similar to the **new** keyword in Java or C# in that it actually allocates the memory for the student. It is also like a constructor that sets all the attributes.

Don’t forget to include **<stdlib.h>** or **<malloc.h>** for **malloc**, and **<string.h>** for **strcpy** and **strlen**.

// A function that creates a student and initializes the attributes.

// This is similar to the keyword new in C# or Java. It allocates the instance

// AND it is also like a constructor that sets all the attributes.

student\* CreateStudent(char\* cNamePtr, int iNum, int\* iMarksPtr, int iMarkCount)

{

// Allocate memory on the heap

student\* sPtr = (student\*)malloc(sizeof(student));

if (sPtr != NULL)

{

// Allocate room for the name

sPtr->cNamePtr = (char\*)malloc((strlen(cNamePtr) + 1) \* sizeof(char));

strcpy(sPtr->cNamePtr, cNamePtr); // like this.cNamePtr = cNamePtr;

// Assign student number

sPtr->iStudentNum = iNum; // like this.iStudentNum = iStudentNum;

// Marks array

sPtr->iMarksPtr = (int\*)malloc(iMarkCount \* sizeof(int));

memcpy(sPtr->iMarksPtr, iMarksPtr, iMarkCount \* sizeof(int));

// Set the number of marks

sPtr->iMarksCount = iMarkCount;

}

return sPtr;

}

Notes:

* We start by allocating memory for the **student**.
* Then we allocate memory for the name – we could test the results of **malloc** here as well. We allocate memory based on the length of the name passed in plus one for the null terminator.
* We also dynamically allocate memory for the number of marks, with the size based on the number of marks (**iMarkCount**) passed in.
* The student number and number of marks do not need dynamically allocated memory since they are just primitive values.

**DisplayStudent** will simply display the attributes, including all the marks. We will write this using a student instance, not a pointer. This is a design decision. It ensures that a copy of the student is made, so that if the function messes up the **struct** somehow (such as changing the name to uppercase), it will not mess up the original **struct**.

// A function to display the date stored in a student instance.

// Takes a student instance, not a pointer. Sometimes this is done as a copy

// of the student is made. If the function messes up the struct somehow, it

// will not mess up the struct passed in.

void DisplayStudent(student s)

{

printf("\nName: %s\n", s.cNamePtr);

printf("Student number: %d\n", s.iStudentNum);

printf("Marks: ");

for (int i = 0; i < s.iMarksCount; i++)

{

printf("%d ", s.iMarksPtr[i]);

}

printf("\n");

}

Note that in the calling function, we have a pointer to a **struct**, so to pass in a **struct**, we dereferenced the pointer:

DisplayStudent(\*sPtr);

Finally, we need to free the student. We will create a function to do so since we need to free the dynamically allocated name and marks array. We can also free the **struct** itself, although this is a design decision since we should only be freeing the **struct** if it is dynamically allocated.

The **FreeStudent** function is similar to a *destructor* in C++, which is a function where the programmer is supposed to release memory and clean up any other resources. Note that this task would be done in Java or C# by the automatic garbage collection.

// This function will free the memory allocated for a student instance.

// In C++, there is a function called a destructor where the programmer is

// supposed to release memory and clean up any other resources - this is similar.

void FreeStudent(student\* sPtr)

{

// Free the dynamically allocated name and marks array

free(sPtr->cNamePtr);

sPtr->cNamePtr = NULL;

free(sPtr->iMarksPtr);

sPtr->iMarksPtr = NULL;

// Do we free the struct itself? Only if the struct is dynamically allocated.

// This is a design decision.

free(sPtr);

sPtr = NULL;

}

Note that we must free the attributes first – if we free the struct first, we no longer have access to the attributes!

### Pointers to pointers to **struct**s

Sometimes we will want to have a pointer to a pointer to a **struct**, particularly if we have an array of **struct**s (so we need a pointer to each **struct** pointer. For instance, suppose we want to have a classroom structure that will contain a dynamically allocated array of student pointers. This **struct** could look like the following:

typedef struct

{

// A dynamically allocated array of student pointers

student\*\* sPtrPtr;

// How many students are in the classroom

unsigned short sNumStudents;

} classroom;

We can test the classroom by populating it with students. Let’s start by creating a **GetStudent** function to ask the user for student data:

// Get student data from the user

student\* GetStudent()

{

char cName[MAX\_NAME\_SIZE];

int iMarks[2];

int iStudentNum;

// Get the name

printf("Enter the student name: ");

gets\_s(cName, MAX\_NAME\_SIZE);

// Get the student number

printf("Enter the student number: ");

scanf("%d", &iStudentNum);

// Get two marks

printf("Enter two marks separated by a space: ");

scanf("%d %d", iMarks, iMarks + 1);

getc(stdin); // Clear any newline characters for next time

return CreateStudent(cName, iStudentNum, iMarks, 2);

}

Be sure to define the **MAX\_NAME\_SIZE** as some appropriate value like 256. Note that this version of **GetStudent** just hardcodes the number of marks as 2 – we could ask the user for how many marks to enter and then dynamically allocate space for those marks, read in each mark, and then call **CreateStudent** (but we couldn’t return the results of **CreateStudent** directly, since we would have to free the space for the marks).

You can test **GetStudent** in **main** as follows:

// Test GetStudent() function

student\* sPtr = GetStudent();

DisplayStudent(\*sPtr);

FreeStudent(sPtr);

Now we can easily populate a classroom with a **PopulateClassroom** function:

// Ask the user to populate the classroom. First ask how many students.

// Then ask for data for each student.

classroom PopulateClassroom()

{

// Create an instance of the classroom on the stack

classroom c;

printf("How many students are in the classroom? ");

scanf("%hu", &c.sNumStudents); // or &(c.sNumStudents) - same

getc(stdin);

// Need to allocate space for the array of student pointers

c.sPtrPtr = (student\*\*)malloc(c.sNumStudents \* sizeof(student\*));

for (int i = 0; i < c.sNumStudents; i++)

{

c.sPtrPtr[i] = GetStudent();

}

return c;

}

Note that we allocate space for an array of student pointers (**student\***), then get a student for each element in that array. Also note the design decision to create a classroom instance on the stack rather than dynamically allocating the classroom itself.

We can display the classroom with a **DisplayClassroom** function:

// Display each student in the classroom

void DisplayClassroom(classroom c)

{

printf("\nDisplaying the classroom:\n");

for (int i = 0; i < c.sNumStudents; i++)

{

DisplayStudent(\*(c.sPtrPtr[i])); // or \*c.sPtrPtr[i] - same

}

}

Again, note that we are passing in a classroom **struct**, not a pointer to a classroom. We need to dereference each of the student pointers in the array since **DisplayStudent** takes a student instance, not a student **struct**.

We will also need to free the classroom with a **FreeClassroom** function:

void FreeClassroom(classroom\* classPtr)

{

// Free students

for (int i = 0; i < classPtr->sNumStudents; i++)

{

FreeStudent(classPtr->sPtrPtr[i]);

classPtr->sPtrPtr[i] = NULL;

}

// Free classroom

free(classPtr->sPtrPtr);

classPtr->sPtrPtr = NULL;

// We could also free the classroom - this is a design decision.

// In this case, we used classroom structs, not a dynamically allocated

// classroom, so it would be inappropriate to free the classroom.

}

Note that we need to free each student first (by calling **FreeStudent**, which frees the name and marks array as well) and then free the array of student pointers (in that order).

Finally, we can create a **testClassroom** function and call it from **main**:

void testClassroom()

{

classroom c = PopulateClassroom();

DisplayClassroom(c);

FreeClassroom(&c);

}

## Unions

A union is a data type that is similar in syntax to a **struct**. A union can have several attributes; however, only one is assigned at a time. The memory for a union is equal to that of its largest member. A union is declared using the keyword **union**.

Example: Suppose we want to store how much fertilizer a farmer purchased. Fertilizer is stored in various units. The following union could be used to store the amount purchased:

typedef union

{

unsigned char truckloads;

unsigned int bags;

double tons;

} FertilizerQuantity;

The size of this union is 8 bytes, since the largest attribute is the **double** for tons. We can then test various aspects of the union – create a **testFertilizerUnion** function and call it from **main**:

void testFertilizerUnion()

{

FertilizerQuantity fq;

// How big is the union in bytes?

printf("Size of union is %zu\n", sizeof(fq)); // 8 bytes - largest is double

// Assign a value to truckloads

fq.truckloads = 3;

printf("The value of truckloads is: %d\n", fq.truckloads);

printf("The value of bags is: %u\n", fq.bags);

printf("The value of tons is: %f\n", fq.tons);

printf("The address of truckloads is: %p\n", &fq.truckloads);

printf("The address of bags is: %p\n", &fq.bags);

printf("The address of tons is: %p\n", &fq.tons);

}

You will notice a couple of things:

* We can access **fq.bags** and **fq.tons** although nothing was assigned to them; however, their results will be meaningless since only one byte was set by **fq.truckloads**.
* All three attributes share the same memory address.

Now try adding the following after the assignment to truckloads:

// Assign a value to bags

fq.bags = 400;

You will see that **fq.bags** now prints correctly as **400**, but **fq.truckloads** displays **144** (the lower byte of **400**).

Now try assigning a value to tons:

// Assign a value to tons

fq.tons = 12345.6789;

You will see that the printout for **fq.tons** is now correct, but the other values print out as different numbers since their memory was written over by the **double** for **fq.tons**.

Sometimes a union can make sense in both of its representations. For instance, consider the following **colorVal** union based on the **pixel32** struct:

typedef struct {

unsigned char a; // transparency value (or alpha channel)

unsigned char b; // blue

unsigned char g; // green

unsigned char r; // red

} pixel32;

typedef union {

// This is 4 bytes

unsigned int val;

// This is also 4 bytes

pixel32 components;

} colorVal;

In this case, **val** in the union represents the entire 32-bit color value, while the **components** in the union allow each byte to be accessed separately to determine a single value of the color. Consider the following **testColorConversion** function:

void testColorConversion()

{

colorVal c;

printf("Size of pixel32 is %zu\n", sizeof(pixel32));

printf("Size of colorVal is %zu\n\n", sizeof(colorVal));

c.val = 0x2A3BFF1C;

c.components.a = 0xFF;

printf("The whole int is %x\n", c.val);

printf("The Red component is %x\n", c.components.r);

printf("The Green component is %x\n", c.components.g);

printf("The Blue component is %x\n", c.components.b);

printf("The alpha channel (transparency) is %x\n", c.components.a);

}

The output of this function would be as follows:

Size of pixel32 is 4

Size of colorVal is 4

The whole int is 2a3bffff

The Red component is 2a

The Green component is 3b

The Blue component is ff

The alpha channel (transparency) is ff

Notes:

* The **pixel32** struct is 4 bytes large (one for each **unsigned char** or byte).
* The **colorVal** union is also 4 bytes large (since both the **pixel32** components and the **int** val are 4 bytes).
* When **c.val** is set, it sets the entire **int**, which also sets each individual color component.
* When **c.components.a** is set, it changes one byte of the **int**.

## Review of **struct**s and **union**s

Some review questions on **struct**s and **union**s:

1. Create a **struct** representing a computer. The struct should contain space for a 9-digit computer ID (plus a null terminator), an **int** for the size of RAM, a character for a unit of RAM (like ‘K’ for kilobyte or ‘G’ for gigabyte), and a pointer to dynamically allocated memory for a computer description.

The struct might look like the following:

typedef struct {

char id[10]; // 9-character id plus null terminator

int ramSize;

char ramUnit;

char\* descriptionPtr;

} Computer;

1. Given the **struct** above, how much space (in bytes) would the **struct** take up? Assume that you’re working on an **x86** system where pointers are 4 bytes, and the default byte alignment is 4 bytes.

10 bytes + *2 for alignment* for **id**, 4 bytes for **ramSize**, 1 byte + *3 for alignment* for **ramUnit**, and 4 bytes for the pointer = 24 bytes

1. Suppose the Computer **struct** was rearranged as shown below. How big would the **struct** be (using the same assumptions)?

typedef struct {

char id[10]; // 9-character id plus null terminator

char ramUnit;

int ramSize;

char\* descriptionPtr;

} Computer;

10 + 1 + *1 for alignment* for **id** and **ramUnit** together, 4 for **ramSize**, 4 for pointer = 20 bytes

*Note that the same attributes in a different order can lead to a different size.*

1. What would the size be if the **struct** was changed as follows?

#pragma pack(push)

#pragma pack(1)

typedef struct {

char id[10]; // 9-character id plus null terminator

int ramSize;

char ramUnit;

char\* descriptionPtr;

} Computer;

#pragma pack(pop)

Because the byte alignment has been set to 1 by the **#pragma pack(1)** directive, the size would be 10 + 4 + 1 + 4 = 19 bytes

1. Create a function that will take values passed in for the various Computer attributes and assign them to a **struct** to be returned. Create a function that will display the various attributes of a Computer that is passed in. Show how they would be called from **main**. Handle dynamically allocated memory properly.

Computer assignValues(char\* id, int ram, char unit, char\* description)

{

Computer retVal;

strcpy(retVal.id, id);

retVal.ramSize = ram;

retVal.ramUnit = unit;

retVal.descriptionPtr = (char\*)malloc(strlen(description) + 1);

strcpy(retVal.descriptionPtr, description);

return retVal;

}

void displayValues(Computer c)

{

printf("The %s computer has id %s with %d RAM of type %c\n",

c.descriptionPtr, c.id, c.ramSize, c.ramUnit);

}

From **main**:

char myId[] = "A12345678";

Computer myComputer = assignValues(myId, 64, 'K', "Commodore 64");

displayValues(myComputer);

// ... could continue to use myComputer as desired

free(myComputer.descriptionPtr); // dynamically allocated description

1. Create a function that will take values passed in for the various Computer attributes and assign them to a dynamically allocated **struct** to be returned. Using the **displayValues** function above, display the values. Handle dynamically allocated memory properly.

Computer\* assignDynamic(char\* id, int ram, char unit, char\* description)

{

Computer\* retPtr = (Computer\*)malloc(sizeof(Computer));

strcpy(retPtr->id, id);

retPtr->ramSize = ram;

retPtr->ramUnit = unit;

retPtr->descriptionPtr = (char\*)malloc(strlen(description) + 1);

strcpy(retPtr->descriptionPtr, description);

return retPtr;

}

From **main**:

char myId[] = "A87654321";

Computer\* myComputerPtr = assignDynamic(myId, 32, 'G', "modern PC");

displayValues(\*myComputerPtr);

// ... could continue to use myComputerPtr as desired

free(myComputerPtr->descriptionPtr); // free description

free(myComputerPtr); // free dynamically allocated struct

1. How would you define a **union** that would contain options for memory to be stored in bytes (as a **long long int**, in MB (as an **int**), in GB (as a **short**), or in **TB** (as an **unsigned char**)?

typedef union {

long long int bytes;

int megabytes;

short gigabytes;

char terabytes;

} MemorySize;

1. What would the size of the **union** given above be?

The size would be 8 bytes (as big as its largest member, the **long long int**).

1. Is the following valid code? Will the printout make sense?

MemorySize m;

m.terabytes = 16;

m.bytes = 1234567890;

printf("%d\n", m.terabytes);

Yes, this is valid code – it will compile and run without issue. No, the printout will not make sense because **terabytes** and **bytes** share the same memory, so the value of 16 for **terabytes** is overwritten by the value for **bytes**.

## Memory map practices involving **struct**s

It is important to know how memory is manipulated using structs. See the following two exercises and their solutions.

* “Memory Map Practice 2.docx” for a practice with memory maps using structs, arrays, and pointers



* + See the solutions in “Memory Map Practice 2 solution.docx”



* “Memory Map Practice 2A - structs and Pointers.docx” for more practice with structs, arrays, and pointers



* + See the solutions in “Memory Map Practice 2A - structs and Pointers solution.docx”



# Application: Manipulating Bitmap Images

We have covered many topics and done a variety of small examples, but to fully understand structs, typedefs, function pointers, bitwise operators, pointers, dynamic memory allocation, and file I/O, we will put them all together in an application that will involve manipulating 24-bit bitmap images in the **.BMP** format.

For more information on the bitmap image format, see the following (summarized below):

* <https://en.wikipedia.org/wiki/BMP_file_format>
* <https://learn.microsoft.com/en-us/windows/win32/gdi/bitmap-storage>
* <https://learn.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-bitmapfileheader>
* <https://learn.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-bitmapinfoheader>

## Bitmap Image Format

There are numerous variations on the bitmap image format. For our application, the bitmap will have the following properties:

* 24 bits per pixel (so in Paint, **Save as… > Other formats > Save as type: 24-bit Bitmap**)
* No alpha channel (for transparency)
* No compression

## Bitmap File Header

This block of bytes is at the start of the file and is used to identify the file. A typical application reads this block first to ensure that the file is actually a BMP file and that it is not damaged. The first two bytes of the BMP file format are the character “B” and then the character “M” in ASCII encoding. All the integer values are stored in little-endian format (that is, least significant byte first). The size of this header is 14 bytes.

|  |  |  |
| --- | --- | --- |
| Offset (in bytes) | Size (in bytes) | Description |
| 0 | 2 | The header field used to identify the BMP and DIB is 0x42 0x4D in hexadecimal, same as “BM” in ASCII. BM is used for Windows 3.1x, 95, NT, …, etc. |
| 2 | 4 | The size of the BMP file in bytes |
| 6 | 2 | Reserved; actual value depends on the application that created the image |
| 8 | 2 | Reserved; actual value depends on the application that created the image |
| 10 | 4 | The offset, that is the starting location, of the byte where the bitmap image data (pixel array) can be found |

## DIB Header (bitmap information header)

This block of bytes tells the application detailed information about the image, which will be used to display the image on the screen. The block also matches the header used internally by Windows and OS/2 and has several different variants. All of them contain a dword (32-bit) field, specifying their size, so that an application can easily determine which header is used in the image.

For the purpose of our application, we will use the 40-byte BITMAPINFOHEADER for Windows NT, 3.1x, or later.

The format of the BITMAPINFOHEADER is as follows (all values are stored as unsigned integers, unless explicitly noted):

|  |  |  |
| --- | --- | --- |
| Offset (in bytes) | Size (in bytes) | Description |
| 14 | 4 | The size of this header, in bytes (40) |
| 18 | 4 | The bitmap width in pixels (signed integer) |
| 22 | 4 | The bitmap height in pixels (signed integer) |
| 26 | 2 | The number of color planes (must be 1) |
| 28 | 2 | The number of bits per pixel, which is the color depth of the image. Typical values are 1, 4, 8, 16, **24** (which is what we will use), and 32. |
| 30 | 4 | The compression method being used |
| 34 | 4 | The T size of the image. This is the size of the raw bitmap data. |
| 38 | 4 | The horizontal resolution of the image (pixels per meter, signed integer) |
| 42 | 4 | The vertical resolution of the image (pixels per meter, signed integer) |
| 46 | 4 | The number of colors in the color palette, or 0 to default to 2n |
| 50 | 4 | The number of important colors used, or 0 when every color is important; generally ignored |

Note that the image width and height can be negative for top-down images. We will assume that the image width and height are positive (so images will be stored “bottom-up”, starting in the lower left corner, going from left to right, and then row by row from the bottom to the top of the image).

## Pixel Array

Following the DIB header is the pixel array. For this application, it is assumed that the pixel stored a red, green, and blue component, and that each component is 1 byte in size (3 bytes or **24** bits per pixel). Note that some images could be 4 bytes per pixel (32 bits) if they have an alpha channel (used for transparency).

Consider an image that is two pixels by two pixels:

|  |  |
| --- | --- |
| Pixel 3 | Pixel 4 |
| Pixel 1 | Pixel 2 |

What would this image data look like in the file? At the end of each row, there might be padding. The bitmap standard requires that each row must end on a 4-byte boundary:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b | g | r | b | g | r | 0 | 0 | b | g | r | *Etc.* |
| Pixel 1 | | | Pixel 2 | | | Padding | | Pixel 3 | | | *…* |

How big is this padding? How do we calculate it? For a 24-bit bitmap, the formula is as follows:

Padding (bytes) = image width (in pixels) % 4

A universal padding formula that will work for any image depth is as follows:

Padding (bytes) = width \* sizeof(PIXEL) % 4 == 0 ? 0 : (4 - width \* sizeof(PIXEL) % 4)

We must consider padding when reading in the image AND when we are looping through or otherwise referencing the array of pixels.

## Bitmap Image Structures

To start our project, we will create a header file named **bitmap.h** that contains the structure of a bitmap image. We will begin with the usual header guards and also include the standard I/O library and standard library so that we don’t need to include them elsewhere:

#ifndef BITMAP\_H

#define BITMAP\_H

#include <stdio.h> // for I/O functions

#include <stdlib.h> // for malloc and other standard definitions

#endif // !BITMAP\_H

Then we will add some **typedef** statements to declare the types of values that will be used in the bitmap headers – these are standard Microsoft definitions:

typedef unsigned char BYTE; // 1 byte or 8 bits, unsigned

typedef unsigned short WORD; // 2 bytes or 16 bits, unsigned

typedef unsigned int DWORD; // 4 bytes or 32 bits, unsigned

// could also use unsigned long on our system since that is also 32 bits

typedef signed int LONG; // 4 bytes or 32 bits, signed

// could also use signed long on our system since that is also 32 bits

Next, we will create the structures for the file header and the pixels – note that we need to pack these to go along 1-byte boundaries so that no extra space is left between members, since there is no space left between them in the definition of a bitmap file:

#pragma pack(push) // Used to store the default byte alignment

#pragma pack(1) // Set the byte alignment to 1

typedef struct

{

// Bitmap file header - 14 bytes

WORD wType; // Should be 'B' 'M'

DWORD dwFileSize;

WORD wReserved1;

WORD wReserved2;

DWORD dwDataOffset; // Should be 54 for our application

// DIB header (bitmap information header)

DWORD dwHeaderSize; // Should be 40 for our application

LONG lWidth;

LONG lHeight;

WORD wPlanes; // Should be 1

WORD wBitCount; // Should be 24 for our application

DWORD dwCompression; // Should be 0 for our application

DWORD dwImageSize;

LONG lXPelsPerMeter;

LONG lYPelsPerMeter;

DWORD dwClrUsed;

DWORD dwClrImportant;

} BITMAPHDR;

typedef struct

{

// Strangely, blue comes first in the pixel...

BYTE bBlu, bGrn, bRed;

} PIXEL;

#pragma pack(pop) // Used to reset the default byte alignment

Finally, we can create a structure that represents the entire image. This structure can also be put in the **#pragma pack(1)** area, but it does not need to be there, since it only contains pointers that align along the boundaries and does not contain data from the bitmap image:

// A struct that represents the entire image

typedef struct

{

BITMAPHDR\* bmHDR;

PIXEL\* bmData;

} IMAGE;

Let’s do a simple test of these structs by checking their sizes. Create **program.c** and add the following code:

#include "bitmap.h"

int main()

{

printf("Size of BITMAPHDR: %zu\n", sizeof(BITMAPHDR));

printf("Size of PIXEL: %zu\n", sizeof(PIXEL));

printf("Size of IMAGE: %zu\n", sizeof(IMAGE));

return EXIT\_SUCCESS;

}

Your output should be 54 bytes for the size of BITMAPHDR, 3 for PIXEL, and either 16 or 8 for IMAGE (depending on whether you are compiling to an **x64** or **x86** platform).

Try commenting out the **#pragma pack(1)**. What sizes do you get now? How about if you switch between the **x64** and **x86** platform? (Don’t forget to uncomment them before we proceed!)

## Reading the file

Next, we will read the file in. Before proceeding, we should have some standard images to work with. Unzip the following 24-bit bitmap images:



Images should include **michael.bmp** and **ron.bmp** (both 150×150, so 2 bytes of padding required at the end of each row), **nopaddingcat.bmp** (1200×1199, so no padding), **face.bmp** (259×194, so 3 bytes of padding), and **bike.bmp** (257×192, so 1 byte of padding). You can verify the sizes by looking at the properties of the files (**Alt+Enter**) and looking at the **Details** tab. Note that sizes are given as width×height.

Next, we will add the code to open the file, read the header in, and print the header. In **main**, add the following:

TestLoadWriteImage();

The **TestLoadWriteImage** function (which will currently only open and close the file) will look like:

void TestLoadWriteImage()

{

FILE\* infile = GetFile("Enter file to open: ", "rb");

IMAGE img = { NULL, NULL };

if (infile != NULL)

{

fclose(infile);

}

else

{

printf("The file did not open properly for reading: %s\n",

strerror(errno));

}

}

Add the following at the top of **program.c**:

#define \_CRT\_SECURE\_NO\_WARNINGS // to use strerror (or else use strerror\_s)

#include <string.h> // for strerror

#include <errno.h> // for errno

Now, let’s write the **GetFile** function. We will include it in a separate file named **bitmap.c**, so first add the function prototype to **bitmap.h**:

// Open a file given a prompt for the user (asking for the filename) and a file mode

FILE\* GetFile(const char\* cPrompt, const char\* cMode);

In **bitmap.c**, add the following:

#define \_CRT\_SECURE\_NO\_WARNINGS // to use older functions (fopen instead of fopen\_s)

#include "bitmap.h"

#define MAX\_FILENAME\_SIZE 256

// Open a file given a prompt for the user (asking for the filename) and a file mode

FILE\* GetFile(const char\* cPrompt, const char\* cMode)

{

FILE\* aFile = NULL;

char cFileName[MAX\_FILENAME\_SIZE];

// Get the name from the user

printf("%s", cPrompt);

gets\_s(cFileName, MAX\_FILENAME\_SIZE);

// Open the file with the given name and mode passed in

aFile = fopen(cFileName, cMode);

return aFile;

}

Now try running the program, giving file names that exist (in the appropriate path – copy the images into the same folder as the source files) and file names that don’t exist. Nothing will happen if the file opens successfully, but an error message will print if the file does not exist.

Next let’s try reading in and printing information from the header. Update **TestLoadWriteImage** with the following in place of just fclose(infile);:

ReadImage(&img, infile);

fclose(infile);

if (img.bmHDR != NULL)

{

PrintHeader(img.bmHDR);

}

Add definitions for **ReadImage** and **PrintHeader** and the related **ReadHeader** function to **bitmap.h** as follows:

// Read an image into the image pointer from the already opened file pointer

void ReadImage(IMAGE\* imgPtr, FILE\* infile);

// Read the image header into the image pointer from the already opened file pointer

void ReadHeader(IMAGE\* imgPtr, FILE\* infile);

// Print out various header values to see if they have been correctly read in

void PrintHeader(BITMAPHDR\* headerPtr);

Then add the following to **bitmap.c**:

// Read an image into the image pointer from the already opened file pointer

void ReadImage(IMAGE\* imgPtr, FILE\* infile)

{

// Read in the header

ReadHeader(imgPtr, infile);

// Later, read in the data...

}

// Read the image header into the image pointer from the already opened file pointer

void ReadHeader(IMAGE\* imgPtr, FILE\* infile)

{

// Allocate memory for the bitmap header

imgPtr->bmHDR = (BITMAPHDR\*)malloc(sizeof(BITMAPHDR));

if (imgPtr->bmHDR != NULL) // memory successfully allocated

{

if (fread(imgPtr->bmHDR, sizeof(BITMAPHDR), 1, infile) != 1)

{

// No success reading, but malloc worked

free(imgPtr->bmHDR);

imgPtr->bmHDR = NULL;

printf("Reading header from file did not work\n");

}

}

}

Finally, we will add a **PrintHeader** function to **bitmap.c**. We will simply print out the values to compare them ourselves, but you could add Boolean conditions that would check if the file is valid for our application:

// Print out various header values to see if they have been correctly read in

void PrintHeader(BITMAPHDR\* headerPtr)

{

printf("First two characters: %x\n", headerPtr->wType); // should be 0x4d42

printf("File size: %d\n", headerPtr->dwFileSize); // check file properties

printf("Data offset: %d\n", headerPtr->dwDataOffset); // should be 54

printf("Header size: %d\n", headerPtr->dwHeaderSize); // should be 40

printf("Dimensions: %d by %d\n", headerPtr->lWidth, headerPtr->lHeight);

// check file properties to verify width and height are correct

printf("Planes: %d\n", headerPtr->wPlanes); // should be 1

printf("Color depth: %d\n", headerPtr->wBitCount); // should be 24

printf("Compression: %d\n", headerPtr->dwCompression); // should be 0

printf("Image size: %d\n", headerPtr->dwImageSize); // file size minus 54

printf("Colors used: %d\n", headerPtr->dwClrUsed); // should be 0

}

Run the program now to see that it works and that our bitmap files are valid. (For further testing, you could create a text file with less than 54 characters and see that it displays the correct error message, or one with 54 or more characters and see that the **PrintHeader** results are just garbage.)

Note that you could also check the structure of our bitmap files by opening them in **Notepad++** and looking at the various byte values. The first two bytes should simply display as “BM”. Many of the rest of the bytes will be unreadable, but you can see where there are values of 0 (showing up as NUL), and you will see the digit 6 and the symbol ( corresponding to the ASCII characters for the values 54 and 40.

Next, we will read the image data in. Replace the comment // Later, read in the data... with the following:

if (imgPtr->bmHDR != NULL) // header was read in successfully

{

// Read in the data

ReadData(imgPtr, infile);

}

Add **ReadData**’s prototype to **bitmap.h**:

// Read the image data into the image pointer from the already opened file pointer

void ReadData(IMAGE\* imgPtr, FILE\* infile);

And then add the **ReadData** function to **bitmap.c**:

// Read the image data into the image pointer from the already opened file pointer

void ReadData(IMAGE\* imgPtr, FILE\* infile)

{

// The padding, in bytes, for the image is:

unsigned int padding = imgPtr->bmHDR->lWidth % 4;

// Calculate the image size in bytes = size of row \* number of rows

unsigned int imageSize = (imgPtr->bmHDR->lWidth \* sizeof(PIXEL) + padding)

\* imgPtr->bmHDR->lHeight;

printf("Calculated image size: %d\n", imageSize);

if (imageSize == imgPtr->bmHDR->dwImageSize)

{

// Allocate memory for pixel data

imgPtr->bmData = (PIXEL\*)malloc(imageSize);

if (imgPtr->bmData != NULL) // memory successfully allocated

{

// Read in the image data

if (fread(imgPtr->bmData, imageSize, 1, infile) != 1)

{

// Failed to read image data

printf("Reading data from file did not work");

}

}

else

{

free(imgPtr->bmHDR);

imgPtr->bmHDR = NULL;

}

}

else

{

printf("Invalid image\n");

free(imgPtr->bmHDR);

imgPtr->bmHDR = NULL;

}

}

Again, you could use fake or altered files to test the various conditions.

Has anyone noticed any problems with our code? Consider the following cases:

* The data is allocated but we failed to read the image data
* The file was read successfully

In either of those cases, we allocate memory for both the header and the data, but never free it. Test this using the code to detect memory leaks – to the top of **program.c**, add:

// for detecting memory leaks

#define \_CRTDBG\_MAP\_ALLOC

#include <crtdbg.h>

And in **main**, add the following before the **return** statement:

\_CrtDumpMemoryLeaks();

Remember to **Start With Debugging** (**F5**) or choose **Local Windows Debugger** from the toolbar. You will find two memory leaks – a 54-byte block starting with “BM” from the bitmap header, and a block that is the size of the image data.

To prevent these memory leaks, let’s write a function called **FreeImage** which will free both the header block and the data block. First, add the prototype to **bitmap.h**:

// Free the image header and image data from the passed-in image pointer

void FreeImage(IMAGE\* imgPtr);

Then, add the code for the function to **bitmap.c**:

// Free the image header and image data from the passed-in image pointer

void FreeImage(IMAGE\* imgPtr)

{

// Free the image

free(imgPtr->bmHDR);

imgPtr->bmHDR = NULL;

if (imgPtr->bmData != NULL)

{

free(imgPtr->bmData);

imgPtr->bmData = NULL;

}

}

Then, add the call to **FreeImage** in **ReadData** (before the **printf** saying that reading data from file did not work):

// Failed to read image data

FreeImage(imgPtr);

printf("Reading data from file did not work");

And also in **TestLoadWriteImage** in **program.c** – note that we must use **&img** since **img** is an **IMAGE** struct, not a pointer to a struct:

if (img.bmHDR != NULL)

{

PrintHeader(img.bmHDR);

FreeImage(&img);

}

If you run the code now, you should see that there are no memory leaks.

## Writing the file

Next, instead of simply printing the header information, we should write the file out. Initially, this will just make an exact copy of the file, but later we will add code to manipulate the image before writing the file out. Add the following to **TestLoadWriteImage** in **program.c** just after calling **PrintHeader** and before freeing the image:

PrintHeader(img.bmHDR);

// Manipulate the image...

FILE\* outfile = GetFile("Enter file to save to: ", "wb");

if (outfile == NULL)

{

printf("Error opening file for writing: %s\n",

strerror(errno));

}

else

{

WriteImage(&img, outfile);

fclose(outfile);

}

FreeImage(&img);

Now we must create the **WriteImage** function. Start by adding it to **bitmap.h**:

// Write the image header and image data from the image pointer provided

// into the already opened file

void WriteImage(IMAGE\* imgPtr, FILE\* outfile);

Now let’s add it to **bitmap.c**. We’ll start by doing it the wrong way:

// Write the image header and image data from the image pointer provided

// into the already opened file

void WriteImage(IMAGE\* imgPtr, FILE\* outfile)

{

if (fwrite(imgPtr, sizeof(IMAGE), 1, outfile) != 1)

{

printf("Failed to write image\n");

}

}

Run the program and check the file size of the resulting file. What is it? Why is it that size?

You should see that the resulting file is 16 bytes (if you compiled to an **x64** platform) or 8 bytes (if you compiled to an **x86** platform), regardless of the size of the image that you used. That is because **imgPtr** is a pointer to an **IMAGE** struct, and **sizeof(IMAGE)** is just the size of the two pointers in the struct! We must save the data that is being pointed to on the heap, not the **IMAGE** struct. We must save both the header data and the image data. Update your code as follows:

if (fwrite(imgPtr->bmHDR, sizeof(BITMAPHDR), 1, outfile) != 1)

{

printf("Failed to write image header\n");

}

If you run the code now, you should see that the file is always 54 bytes in size, and if you look at the file using Notepad++, you should see the header information in its binary form. Next, let’s save the image data:

else

{

// Write the image data - size has already been verified

DWORD imageSize = imgPtr->bmHDR->dwImageSize;

if (fwrite(imgPtr->bmData, imageSize, 1, outfile) != 1)

{

printf("Failed to write the image data\n");

}

}

Note that the size we write out is not **sizeof(PIXEL)** although **imgPtr->bmData** is a **PIXEL\***, since it is pointing to multiple pixels plus any padding necessary. If we wrote out **sizeof(PIXEL)**, our file size would always be 57 bytes since **sizeof(PIXEL)** is three bytes.

Run your program and verify that it now correctly copies image files.

## Manipulating the image

Now that we have loaded the data into memory, we can manipulate the image. To do so, we will define functions that will operate on a pixel and then move through the image data, changing each pixel.

First, we must create a function pointer for the functions that operate on a pixel. In **bitmap.h**, add the following function pointer type definition:

// Function pointers for editing a bitmap image

typedef void (\*BM\_FUNC\_PTR)(PIXEL\*);

Do you remember how to read this definition? It is creating a name **BM\_FUNC\_PTR** which will be used for functions that take a **PIXEL\*** as their only parameter and return nothing (**void**).

In **program.c**, add a function named **GrayscaleAverage** at the top of the file which matches the function pointer definition. This function will change a pixel’s blue, green, and red colors to an average of all three colors, resulting in a gray color for the pixel:

void GrayscaleAverage(PIXEL\* pixelPtr)

{

BYTE gray = (pixelPtr->bBlu + pixelPtr->bGrn + pixelPtr->bRed) / 3;

pixelPtr->bBlu = gray;

pixelPtr->bGrn = gray;

pixelPtr->bRed = gray;

}

Now we can call a **ManipulateImage** function in **TestLoadWriteImage** and pass in this pixel function:

// Manipulate the image...

ManipulateImage(&img, GrayscaleAverage);

Add **ManipulateImage** to **bitmap.h**:

// Manipulate each pixel in a bitmap image provided using the function provided

void ManipulateImage(IMAGE\* imgPtr, BM\_FUNC\_PTR pixelFunc);

Now we need to create **ManipulateImage** in **bitmap.c** to go through every pixel in the image data, applying the function to each pixel as we go. We will start by not worrying about the padding:

// Manipulate each pixel in a bitmap image provided using the function provided

void ManipulateImage(IMAGE\* imgPtr, BM\_FUNC\_PTR pixelFunc)

{

PIXEL\* currentPixel = imgPtr->bmData;

for (LONG i = 0; i < imgPtr->bmHDR->lHeight; i++) // for each row

{

for (LONG j = 0; j < imgPtr->bmHDR->lWidth; j++) // for each pixel

{

pixelFunc(currentPixel); // manipulate the pixel

currentPixel++; // move to the next pixel

}

}

}

Try out the program on an image with no padding (like **nopaddingcat.bmp**). You should see that it correctly creates a grayscale image. Then try out the program on an image with padding (like **ron.bmp**). It will look mostly gray, but the left and right edges will have some color (and there may be other subtle differences – examine the images using Paint at maximum zoom).

Now we should account for the padding. Recall that there are *width (in pixels) % 4* bytes of padding at the end of each row. Note that these are bytes, not pixels. Thus, we need to calculate the padding, and then change the pixel pointer to a **BYTE** pointer, add the padding, and then change it back to a pixel pointer. So we change **ManipulateImage** as follows:

// Manipulate each pixel in a bitmap image provided using the function provided

void ManipulateImage(IMAGE\* imgPtr, BM\_FUNC\_PTR pixelFunc)

{

PIXEL\* currentPixel = imgPtr->bmData;

// The padding, in bytes, for the image is:

BYTE padding = imgPtr->bmHDR->lWidth % 4;

for (LONG i = 0; i < imgPtr->bmHDR->lHeight; i++) // for each row

{

for (LONG j = 0; j < imgPtr->bmHDR->lWidth; j++) // for each pixel

{

pixelFunc(currentPixel); // manipulate the pixel

currentPixel++; // move to the next pixel

}

// At the end of the current row, move padding BYTEs forward

currentPixel = (PIXEL\*)(((BYTE\*)currentPixel) + padding);

}

}

Try out the resulting code on an image with padding to see that it now works correctly.

### Pixel manipulation exercises

To practice, you should write more pixel functions and try to manipulate the images with each of them (or even try to manipulate the image twice in a row with different pixel functions). Note that if you remove the padding calculation, the results on padded images may be much more dramatic that in the grayscale formula! Here are some pixel functions to try:

1. Write a GrayscaleFormula function that will calculate the gray level (used for blue, green, and red) using the formula *11% \* blue + 59% \* green + 30% \* red*. This formula should create a grayscale image that looks better because it matches some of the characteristics of the human eye.
2. Write a Negative function that will flip all the bits in each of the blue, green, and red colors (or reverse the intensity of the blue/green/red colors).
3. Shift the bits of each color over to the right by 1 to “ghost” the image darker.
4. Shift the bits of each color over to the right by 2 and make the two bits that are shifted in 1’s to “ghost” the image but make it brighter.
5. Keep the red its usual value and make the green/blue values 0.
6. Keep the green its usual value and make the red/blue values 0.
7. Keep the blue its usual value and make the red/green values 0.
8. Make the red (or green or blue) the grayscale value and the other values 0.
9. Write a SwapNibbles function that swaps the nibbles of each color.
10. Rotate the red/green/blue components.
11. Come up with a pixel manipulation function of your own!

#### Selected pixel manipulation exercise solutions

1. Write a GrayscaleFormula function that will calculate the gray level (used for blue, green, and red) using the formula *11% \* blue + 59% \* green + 30% \* red*. This formula should create a grayscale image that looks better because it matches some of the characteristics of the human eye.

void GrayscaleFormula(PIXEL\* pixelPtr)

{

BYTE gray = (BYTE)(pixelPtr->bBlu \* 0.11 + pixelPtr->bGrn \* 0.59

+ pixelPtr->bRed \* 0.3);

pixelPtr->bBlu = gray;

pixelPtr->bGrn = gray;

pixelPtr->bRed = gray;

}

And in **TestLoadWriteImage**:

// Manipulate the image...

ManipulateImage(&img, GrayscaleFormula);

1. Write a Negative function that will flip all the bits in each of the blue, green, and red colors (or reverse the intensity of the blue/green/red colors).

void Negative(PIXEL\* pixelPtr)

{

// Two ways to achieve: subtract from 255...

pixelPtr->bBlu = 255 - pixelPtr->bBlu;

// Be careful with your math calculations

// that you do not overflow the pixel component

// Use the bitwise inversion operator...

pixelPtr->bGrn = ~pixelPtr->bGrn;

pixelPtr->bRed = ~pixelPtr->bRed;

}

And in TestLoadWriteImage, if we want to try #1 and #2 in succession:

// Manipulate the image...

ManipulateImage(&img, GrayscaleFormula);

ManipulateImage(&img, Negative);

1. Shift the bits of each color over to the right by 1 to “ghost” the image darker.

void GhostDarker(PIXEL\* pixelPtr)

{

pixelPtr->bBlu = pixelPtr->bBlu >> 1;

pixelPtr->bGrn = pixelPtr->bGrn >> 1;

pixelPtr->bRed = pixelPtr->bRed >> 1;

}

1. Shift the bits of each color over to the right by 2 and make the two bits that are shifted in 1’s to “ghost” the image but make it brighter.

void GhostBrighter(PIXEL\* pixelPtr)

{

pixelPtr->bBlu = pixelPtr->bBlu >> 2 | 0b11000000;

pixelPtr->bGrn = pixelPtr->bGrn >> 2 | 0b11000000;

pixelPtr->bRed = pixelPtr->bRed >> 2 | 0xC0;

}

1. Keep the red its usual value and make the green/blue values 0.

void RedOnly(PIXEL\* pixelPtr)

{

pixelPtr->bBlu = 0;

pixelPtr->bGrn = 0;

}

1. Keep the green its usual value and make the red/blue values 0.

*Left as an exercise for you (since it’s so close to #5). Or try making the other values 255!*

1. Keep the blue its usual value and make the red/green values 0.

*Left as an exercise for you (since it’s so close to #5). Or try making the other values 255!*

1. Make the red (or green or blue) the grayscale value and the other values 0.

void RedOnlyGrayAverage(PIXEL\* pixelPtr)

{

BYTE gray = (pixelPtr->bBlu + pixelPtr->bGrn + pixelPtr->bRed) / 3;

pixelPtr->bBlu = 0;

pixelPtr->bGrn = 0;

pixelPtr->bRed = gray;

}

1. Rotate the red/green/blue components. Note that you could switch the order of rotation.

void RotateColors(PIXEL\* pixelPtr)

{

BYTE temp = pixelPtr->bBlu;

pixelPtr->bBlu = pixelPtr->bGrn;

pixelPtr->bGrn = pixelPtr->bRed;

pixelPtr->bRed = temp;

}

1. Write a SwapNibbles function that swaps the nibbles of each color.

void SwapNibbles(PIXEL\* pixelPtr)

{

pixelPtr->bBlu = (pixelPtr->bBlu >> 4) | (pixelPtr->bBlu << 4);

pixelPtr->bGrn = (pixelPtr->bGrn >> 4) | (pixelPtr->bGrn << 4);

pixelPtr->bRed = (pixelPtr->bRed >> 4) | (pixelPtr->bRed << 4);

}

## Manipulating two images

Now that we have manipulated one image, we can reuse a lot of our code (to open files, read images, and save images) to manipulate two images (using a function that manipulates the pixels from both of the images). First, update **main** as follows:

//TestLoadWriteImage();

TestTwoImages();

Then write **TestTwoImages** as follows (we could add testing for successful reads, etc., but this is adequate):

void TestTwoImages()

{

FILE\* infile1 = GetFile("Enter 1st file to open: ", "rb");

FILE\* infile2 = GetFile("Enter 2nd file to open: ", "rb");

FILE\* outfile = GetFile("Enter filename to save: ", "wb");

IMAGE img1 = { NULL, NULL };

IMAGE img2 = { NULL, NULL };

if (infile1 != NULL && infile2 != NULL && outfile != NULL)

{

// Read in the images

ReadImage(&img1, infile1);

ReadImage(&img2, infile2);

// Could check for successful read...

// Close input files

fclose(infile1);

fclose(infile2);

// Manipulate the first image to combine it with the second image

ManipulateTwoImages(&img1, &img2, Blend);

WriteImage(&img1, outfile);

fclose(outfile);

FreeImage(&img1);

FreeImage(&img2);

}

else

{

printf("One of the files didn't open\n");

}

}

Now write a Blend function that will make the first pixel into the average of the first and second pixels:

// Two-pixel functions

void Blend(PIXEL\* p1, PIXEL\* p2)

{

p1->bBlu = p1->bBlu / 2 + p2->bBlu / 2;

p1->bGrn = p1->bGrn / 2 + p2->bGrn / 2;

p1->bRed = p1->bRed / 2 + p2->bRed / 2;

}

We should also create a function pointer **typedef** for two-pixel functions in **bitmap.h** – what should it look like? Try it yourself before looking at the solution below:

typedef void (\*BM\_TWO\_PIXELS)(PIXEL\*, PIXEL\*);

Finally, we need the code for **ManipulateTwoImages**. This will be a little tricky since the images may not be the same size. We only want to traverse the pixels for the smaller width and smaller height. Rather than doing somewhat complicated pointer math, let’s write a helper function to get a pixel from an image. Add the following to **bitmap.h**:

// Combine pixels from two bitmap images provided using the function provided

void ManipulateTwoImages(IMAGE\* img1, IMAGE\* img2, BM\_TWO\_PIXELS twoPixelFunc);

// Helper function to get a pixel pointer from a particular spot in an image

PIXEL\* GetPixel(int row, int col, IMAGE\* imgPtr);

Then write the helper function in **bitmap.c**:

// Helper function to get a pixel pointer from a particular spot in an image

PIXEL\* GetPixel(int row, int col, IMAGE\* imgPtr)

{

BYTE padding = imgPtr->bmHDR->lWidth % 4;

// Offset in bytes = size of the row in bytes \* number of rows

// + number of bytes preceding the desired pixel in the row

unsigned int offset = (imgPtr->bmHDR->lWidth \* sizeof(PIXEL) + padding) \* row

+ col \* sizeof(PIXEL);

// Offset is in bytes, so we need to do some casting of pointers

return (PIXEL\*)((BYTE\*)imgPtr->bmData + offset);

}

Using that helper function, we can cycle through the pixels in the smaller dimensions in **ManipulateTwoImages** in **bitmap.c**:

// Combine pixels from two bitmap images provided using the function provided

void ManipulateTwoImages(IMAGE\* img1, IMAGE\* img2, BM\_TWO\_PIXELS twoPixelFunc)

{

// Find the smaller height and width - they may be from different images

int height = (img1->bmHDR->lHeight < img2->bmHDR->lHeight) ?

img1->bmHDR->lHeight : img2->bmHDR->lHeight;

int width = (img1->bmHDR->lWidth < img2->bmHDR->lWidth) ?

img1->bmHDR->lWidth : img2->bmHDR->lWidth;

// Goal: Go through each pixel and call the twoPixelFunc on the pixels

for (int i = 0; i < height; i++)

{

for (int j = 0; j < width; j++)

{

twoPixelFunc(GetPixel(i, j, img1), GetPixel(i, j, img2));

}

}

}

Try it out using two images of the same size (like **michael.bmp** and **ron.bmp**) and two images of different sizes (try each order, so smaller first and then bigger first). The images should be blended together so that you can see shadowy versions of both images in one picture!

### Manipulating two images exercises

To practice, you should write some additional two-image manipulation functions and call them.

1. Write a function Hide that takes in two pixels and combines the top 4 bits (or top nibble) of the first pixel with the top 4 bits of the second pixel, but the top 4 bits of the second pixel will become the bottom 4 bits of the resulting pixel.

The result should be that the first image can be seen fairly clearly, but there should be a faint second image visible as well.

If you use the SwapNibbles function from the previous exercises on the resulting picture, the portion where the two pictures overlap will be reversed so that the second picture shows up but the first picture is only faintly visible!

1. Write a function Overwrite that will set the first pixel’s values to the second pixel’s values.

Note that is the images are the same size, this will just result in the second image being saved. If the images are not of the same size, the second image will take up space in the first image (bottom left).

Try changing just the blue or just the red or just the green!

#### Selected two image manipulation exercise solutions

1. Write a function Hide that takes in two pixels and combines the top 4 bits (or top nibble) of the first pixel with the top 4 bits of the second pixel, but the top 4 bits of the second pixel will become the bottom 4 bits of the resulting pixel.

void Hide(PIXEL\* p1, PIXEL\* p2)

{

p1->bBlu = (p1->bBlu & 0xF0) | (p2->bBlu >> 4);

p1->bGrn = (p1->bGrn & 0xF0) | (p2->bGrn >> 4);

p1->bRed = (p1->bRed & 0xF0) | (p2->bRed >> 4);

}

1. Write a function Overwrite that will set the first pixel’s values to the second pixel’s values.

void Overwrite(PIXEL\* p1, PIXEL\* p2)

{

p1->bBlu = p2->bBlu;

p1->bGrn = p2->bGrn;

p1->bRed = p2->bRed;

}

## Resizing the image’s canvas

Next, we will try resizing the width and height that the image takes up. Note that we will not resize the image – that is a harder problem than we want to try. Instead, we will just change the width and height of the “canvas” the image is displayed on.

This is the equivalent of going into Paint and choosing **File | Image Properties** and changing the **Width:** and **Height:** fields in the ensuing dialog box. If the dimensions we choose are bigger than the initial canvas, it will add some blank space. If the dimensions that we choose are smaller than the initial canvas, it will cut off the image so only the bottom left of the image is displayed.

To begin, change **main** as follows:

//TestLoadWriteImage();

//TestTwoImages();

TestResizeCanvas();

Then add the **TestResizeCanvas** function. I will skip testing for successful file reads and memory allocation, and assume we are resizing to 200 by 200 pixels:

void TestResizeCanvas()

{

FILE\* infile = GetFile("Enter file to open: ", "rb");

IMAGE img = { NULL, NULL };

ReadImage(&img, infile);

fclose(infile);

IMAGE img2 = ResizeCanvas(&img, 200, 200);

FILE\* outfile = GetFile("Enter file to save to: ", "wb");

WriteImage(&img2, outfile);

fclose(outfile);

FreeImage(&img);

FreeImage(&img2);

}

If you prefer a bit more control, add the following instead of just using 200 by 200:

printf("Original dimensions: %d by %d\n",

img.bmHDR->lWidth, img.bmHDR->lHeight);

LONG newWidth, newHeight;

printf("Enter new dimensions separated by a space: ");

scanf("%d %d", &newWidth, &newHeight);

while (getchar() != '\n'); // clear buffer for GetFile

IMAGE img2 = ResizeCanvas(&img, newWidth, newHeight);

Now add **ResizeCanvas** to the **bitmap.h** file – note that it returns an **IMAGE** struct:

// Resize the canvas of an image to the width and height given

IMAGE ResizeCanvas(IMAGE\* originalImgPtr, LONG newWidth, LONG newHeight);

Finally, we will write **ResizeCanvas** to **bitmap.c**. Let’s go over the solution strategy first:

* First, create a new **IMAGE** struct with the header and data pointers set to **NULL**. This is the struct that we will return.
* Next, allocate memory for a new **BITMAPHDR** in the new image.
* Copy over the original image’s **BITMAPHDR** data into the new image’s **BITMAPHDR**, since many of the struct’s members will have the same values.
* Set the new values:
  + Set the width and length to the width and height passed in.
  + Calculate and set the new image size based on the new width and new height passed in. Remember to account for the padding!
  + Calculate and set the new file size.
* Using the new image size, allocate memory for the image data.
* Copy over the original image data where appropriate.
  + Determine the smaller width and height from the original width and height and the new passed-in values.
  + Loop through the rows and pixels in each row, setting the pixels of the new image to the pixels of the original image.

Try it on your own, and then look at the following solution:

// Resize the canvas of an image to the width and height given

IMAGE ResizeCanvas(IMAGE\* originalImgPtr, int newWidth, int newHeight)

{

IMAGE imgNew = { NULL, NULL };

DWORD newImageSize = 0;

BYTE padding = 0;

LONG width, height;

imgNew.bmHDR = (BITMAPHDR\*)malloc(sizeof(BITMAPHDR));

if (imgNew.bmHDR) // != NULL

{

memcpy(imgNew.bmHDR, originalImgPtr->bmHDR, sizeof(BITMAPHDR));

imgNew.bmHDR->lWidth = newWidth;

imgNew.bmHDR->lHeight = newHeight;

padding = newWidth % 4;

newImageSize = (newWidth \* sizeof(PIXEL) + padding) \* newHeight;

imgNew.bmHDR->dwImageSize = newImageSize;

imgNew.bmHDR->dwFileSize = newImageSize + sizeof(BITMAPHDR);

imgNew.bmData = (PIXEL\*)malloc(newImageSize);

if (imgNew.bmData) // != NULL

{

height = (newHeight < originalImgPtr->bmHDR->lHeight) ?

newHeight : originalImgPtr->bmHDR->lHeight;

width = (newWidth < originalImgPtr->bmHDR->lWidth) ?

newWidth : originalImgPtr->bmHDR->lWidth;

for (int i = 0; i < height; i++)

{

for (int j = 0; j < width; j++)

{

\*GetPixel(i, j, &imgNew) = \*GetPixel(i, j, originalImgPtr);

}

}

}

else

{

free(imgNew.bmHDR);

imgNew.bmHDR = NULL;

}

}

return imgNew;

}

Try running it and test it with various situations (width and height smaller than the original, width and height bigger than the original). If the canvas is expanded, what color is the new space? Does it differ if you run it in **Release** mode instead of **Debug** mode?

In fact, the new space could have any data in it – we are not guaranteed that the heap memory that is allocated to the new image data has been set to any particular value. We might want to change the new space to a specific color. There are a couple of ways to do so.

First, we could use **memset** to set each byte in the data to a specific value, such as 0 (black) or 255 (white – note that this would set the padding to 255 as well, which is allowed.) In **ResizeCanvas**, add:

if (imgNew.bmData) // != NULL

{

memset(imgNew.bmData, 255, newImageSize);

This only works if we want to set the red/green/blue values to the same value. If we want a specific color, such as green, we must set each individual pixel as follows:

if (imgNew.bmData) // != NULL

{

//memset(imgNew.bmData, 255, newImageSize);

PIXEL startPixel = { 0, 255, 0 }; // green pixel

InitializePixelData(&imgNew, startPixel);

Where **InitializePixelData** has its function prototype in **bitmap.h**:

// Purpose: Given an image with new dynamically allocated data,

// initialize each pixel to some background color.

void InitializePixelData(IMAGE\* imgPtr, PIXEL desiredPixel);

And the corresponding code in **bitmap.c**:

// Purpose: Given an image with new dynamically allocated data,

// initialize each pixel to some background color.

// Parameters:

// imgPtr -> An image with no pixel values set

// desiredPixel -> A pixel (set of colors) to assign to each pixel in the image

void InitializePixelData(IMAGE\* imgPtr, PIXEL desiredPixel)

{

PIXEL\* pixelPtr = imgPtr->bmData;

BYTE padding = imgPtr->bmHDR->lWidth % 4;

// Loop through the array of pixels

for (LONG i = 0; i < imgPtr->bmHDR->lHeight; i++)

{

for (LONG j = 0; j < imgPtr->bmHDR->lWidth; j++)

{

//\*GetPixel(i, j, imgPtr) = desiredPixel;

\*pixelPtr = desiredPixel;

pixelPtr++;

}

// At the end of the row, move padding BYTEs forward

pixelPtr = (PIXEL\*)((BYTE\*)pixelPtr + padding);

}

}

## Bitmap manipulation exercises

So far, we have walked you through the solutions to most of the code. To challenge you to really understand the bitmap image code, here are a couple of functions you can add. Use the **bitmap.c** and **bitmap.h** files completed in class as your starting files (and **program.c** to call it, or copy them over to a different project and create a new **program.c**). Add function prototypes as necessary to the **bitmap.h** file and code the following functions in **bitmap.c**:

1. Cropping an image reduces the image to the size indicated by bounding points in the lower left and upper right of a rectangle. Code a function to crop a bitmap image given lower left and upper right pixel co-ordinates as well as an image itself. A new image is returned.

The function prototype is as follows:

// Given an image and a lower left and upper right coordinate,

// crop the image.

IMAGE Crop(IMAGE\* imgPtr, int xLeft, int yLeft, int xRight, int yRight);

1. Rotating an image by an angle Θ (theta) measured in radians involves the use of formulas learned in the MATH 282 class. So, assuming that the bottom left corner of the image is the origin (0, 0), then the formulas to rotate about the origin are (x, y is the original location, and the new location is x', y'):

x' = x \* cos Θ – y \* sin Θ

y' = y \* sin Θ + x \* cos Θ

To rotate about a specific point (Px, Py), we first translate to the origin, then rotate, and then translate back. The formulas then become:

x' = (x – Px) \* cos Θ – (y – Py) \* sin Θ + Px

y' = (x – Px) \* sin Θ + (y – Py) \* cos Θ + Py

Note that you will have to **#include <math.h>** to use the **sin** and **cos** functions.

The basic idea is to take a pixel from the original image, calculate its new position, and set it into the new image. Now supposing the new image is the same size as the original image, this would imply that some pixels will end up “out” of the range. In this case, the pixel is discarded.

Write a function that will take a pointer to an image and an angle (in radians) and return a new image, the same size as the original image but rotated around the center of the image. The function prototype is as follows:

// Given an image and an angle in radians,

// rotate the image counterclockwise

IMAGE RotateImage(IMAGE\* imgPtr, double theta);

Here are some example images. First an original bike image:



Then, the bike rotated 45° counterclockwise:



Things to note:

* + Notice the faint white or gray or black pixels in the result. This is called aliasing. Why do you think these pixels exist? Make an attempt to solve this problem.
  + Notice that locations where no image data exists are gray. Why? Try running your project in **Release** mode – what happens? How can you control the color of the locations where no image data exists?

### Solutions to bitmap manipulation exercises

Here are sample solutions to the bitmap manipulation exercises, along with a solution to the aliasing problem and an additional enhanced solution for rotating the image in a larger canvas:

1. First, we add the function prototype to **bitmap.h**:

// Given an image and a lower left and upper right coordinate,

// crop the image.

IMAGE Crop(IMAGE\* imgPtr, int xLeft, int yLeft, int xRight, int yRight);

Then we can add code in **main** to call a test function:

TestCropImage();

And add code for **TestCropImage** in **program.c** – note that we could have additional testing to see if the file is read in correctly, or ask the user to input the cropping coordinates, or check that the cropping coordinates are in range:

void TestCropImage()

{

FILE\* infile = GetFile("Enter file to open: ", "rb");

IMAGE img = { NULL, NULL };

ReadImage(&img, infile);

fclose(infile);

IMAGE imgCrop = Crop(&img, 50, 40, 125, 110);

// coordinates for cropping back wheel in bike.bmp

FILE\* outfile = GetFile("Enter file to save to: ", "wb");

WriteImage(&imgCrop, outfile);

fclose(outfile);

FreeImage(&img);

FreeImage(&imgCrop);

}

Then we can write the **Crop** function in **bitmap.c**:

// Given an image and a lower left and upper right coordinate,

// crop the image.

// Parameters:

// imgPtr -> a pointer to an image to crop

// xLeft -> column value of the left coordinate

// yLeft -> row value of the left coordinate

// xRight -> column value of the right coordinate

// yRight -> row value of the right coordinate

// Returns: the cropped image

IMAGE Crop(IMAGE\* imgPtr, int xLeft, int yLeft, int xRight, int yRight)

{

DWORD imageSize;

IMAGE tmpImg = { NULL, NULL };

// If the cropped image's header was allocated

if (tmpImg.bmHDR = (BITMAPHDR\*)malloc(sizeof(BITMAPHDR))) // != NULL

{

// Copy the header from the original image to the cropped image

memcpy(tmpImg.bmHDR, imgPtr->bmHDR, sizeof(BITMAPHDR));

// Set the height and width of the cropped image

tmpImg.bmHDR->lWidth = xRight - xLeft;

tmpImg.bmHDR->lHeight = yRight - yLeft;

// Calculate the padding of the cropped image

int padding = tmpImg.bmHDR->lWidth % 4;

imageSize = (tmpImg.bmHDR->lWidth \* sizeof(PIXEL) + padding)

\* tmpImg.bmHDR->lHeight;

tmpImg.bmHDR->dwImageSize = imageSize;

tmpImg.bmHDR->dwFileSize = imageSize + sizeof(BITMAPHDR);

// If memory for the cropped image was allocated

if (tmpImg.bmData = (PIXEL\*)malloc(imageSize)) // != NULL

{

// For each row in the cropped image

for (int i = 0; i < tmpImg.bmHDR->lHeight; i++)

{

// For each column in the cropped image

for (int j = 0; j < tmpImg.bmHDR->lWidth; j++)

{

\*GetPixel(i, j, &tmpImg) =

\*GetPixel(yLeft + i, xLeft + j, imgPtr);

}

}

}

else

{

free(tmpImg.bmHDR);

tmpImg.bmHDR = NULL;

}

}

// Return the image

return tmpImg;

}

Note that you could also write **Crop** to take an upper left and a lower right coordinate with appropriate changes to the calculations.

1. First, we add the function prototype to **bitmap.h**:

// Given an image and an angle in radians,

// rotate the image counterclockwise

IMAGE RotateImage(IMAGE\* imgPtr, double theta);

Then we can add code in **main** to call a test function:

TestRotateImage();

Don’t forget to include **math.h**:

#include <math.h> // for sin and cos

Add the following definition to **bitmap.h**:

#define PI 3.14159265358979323846

And add code for **TestRotateImage** in **program.c** – note that we could have additional testing to see if the file is read in correctly, or ask the user to input the value for the rotation:

void TestRotateImage()

{

FILE\* infile = GetFile("Enter file to open: ", "rb");

IMAGE img = { NULL, NULL };

ReadImage(&img, infile);

fclose(infile);

IMAGE imgRot = RotateImage(&img, (45.0 \* PI) / 180.0);

FILE\* outfile = GetFile("Enter file to save to: ", "wb");

WriteImage(&imgRot, outfile);

fclose(outfile);

FreeImage(&img);

FreeImage(&imgRot);

}

Then we can write the **RotateImage** function in **bitmap.c**:

// Given an image and an angle in radians,

// rotate the image counterclockwise

IMAGE RotateImage(IMAGE\* imgPtr, double theta)

{

DWORD imageSize = 0;

IMAGE img = { NULL, NULL };

// Allocate space for the header

img.bmHDR = (BITMAPHDR\*)malloc(sizeof(BITMAPHDR));

if (img.bmHDR)

{

memcpy(img.bmHDR, imgPtr->bmHDR, sizeof(BITMAPHDR));

imageSize = imgPtr->bmHDR->dwImageSize;

img.bmData = (PIXEL\*)malloc(imageSize);

if (img.bmData)

{

// Determine the center of rotation (middle of image)

LONG oX = imgPtr->bmHDR->lWidth / 2;

LONG oY = imgPtr->bmHDR->lHeight / 2;

// Copy pixel to new image

for (LONG pY = 0; pY < imgPtr->bmHDR->lHeight; pY++)

{

for (LONG pX = 0; pX < imgPtr->bmHDR->lWidth; pX++)

{

// Calculate position of the current pixel

// in the new image

LONG x = (int)(cos(theta) \* (pX - oX)

- sin(theta) \* (pY - oY) + oX);

LONG y = (int)(sin(theta) \* (pX - oX)

+ cos(theta) \* (pY - oY) + oY);

// If it is in bounds,

// assign to the new image

if (x >= 0 && x < img.bmHDR->lWidth &&

y >= 0 && y < img.bmHDR->lHeight)

{

\*GetPixel(y, x, &img) =

\*GetPixel(pY, pX, imgPtr);

}

}

}

}

else // If there was a problem allocating memory for data,

// release memory that was allocated

{

free(img.bmHDR);

img.bmHDR = NULL;

printf("Failed to allocate data for image");

}

}

else

{

printf("Header for image could not be allocated");

}

return img;

}

This version of the code will rotate the image, but it will leave dots that are not colored in (and therefore are gray or black or some random color from uninitialized memory) because not all pixels in the rotated image are included in the set of rotated points.

One solution to this problem is to initialize the background of the new image to some crazy color (unlikely to be in the actual image, so not just white or black). Doing so will mean that the pixels that were not included in the rotated pixels that we set will have that crazy color. Here we choose a bright green:

if (img.bmData)

{

// Crazy color to make anti-aliasing easier

PIXEL startPixel = { 0, 255, 128 };

// Initialize background to crazy color

InitializePixelData(&img, startPixel);

Note that if you run the program now, the dots (and the extra space in the background) will show up in that bright green color. So, after the rotation is done, we can now search for those dots and replace them with an appropriate color for the image. Doing so is called “anti-aliasing”. We can write a new function to do so. First, add it in **bitmap.h**:

// Given a rotated image, fill in the "holes"

// with the average of the neighbouring colors

void AntiAlias(IMAGE\* imgPtr);

Next, call it after the rotation is done:

AntiAlias(&img);

}

else // If there was a problem allocating memory for data,

// release memory that was allocated

Then we can write the **AntiAlias** function:

// Rotating the image leaves "holes" in the rotated image.

// This function looks for those holes

// and fills them with the average of its neighbours if they exist.

// Code assumes that the background was "pre-initialized"

// to the given background color.

void AntiAlias(IMAGE\* imgPtr)

{

// Crazy color to make anti-aliasing easier

PIXEL p = { 0, 255, 128 };

PIXEL\* pCurrent = NULL;

PIXEL\* pLeft = NULL;

PIXEL\* pRight = NULL;

// If the image is not null

if (imgPtr != NULL)

{

// if the image header and data are not null

if (imgPtr->bmHDR != NULL && imgPtr->bmData != NULL)

{

// Loop through the array of pixels

for (LONG i = 0; i < imgPtr->bmHDR->lHeight; i++)

{

for (LONG j = 0; j < imgPtr->bmHDR->lWidth; j++)

{

pCurrent = GetPixel(i, j, imgPtr);

// If pixel is the background color

if (memcmp(pCurrent, &p, sizeof(PIXEL)) == 0)

{

// Get left pixel if it exists

if (j > 0)

{

pLeft = GetPixel(i, j - 1, imgPtr);

}

// Get right pixel if it exists

if (j < imgPtr->bmHDR->lWidth - 1)

{

pRight = GetPixel(i, j + 1, imgPtr);

}

// If both exists

if (pLeft && pRight)

{

pCurrent->bBlu = pLeft->bBlu / 2

+ pRight->bBlu / 2;

pCurrent->bGrn = pLeft->bGrn / 2

+ pRight->bGrn / 2;

pCurrent->bRed = pLeft->bRed / 2

+ pRight->bRed / 2;

}

// If only the left exists

else if (pLeft)

{

pCurrent->bBlu = pLeft->bBlu;

pCurrent->bGrn = pLeft->bGrn;

pCurrent->bRed = pLeft->bRed;

}

// If only the right exists

else if (pRight)

{

pCurrent->bBlu = pRight->bBlu;

pCurrent->bGrn = pRight->bGrn;

pCurrent->bRed = pRight->bRed;

}

pLeft = NULL;

pRight = NULL;

}

}

}

}

}

}

BONUS FUNCTION: The previous **RotateImage** function rotated the image in the original canvas size, so parts of the rotated image were cut off. We could also use some fancy math to determine how big the new canvas should be to allow the entire rotated image to show up.

Add the following to **bitmap.h**:

// Given an image and an angle in radians,

// rotate that image counterclockwise in a larger canvas

IMAGE FancyRotateImage(IMAGE\* imgPtr, double theta);

Change the function call in **program.c**:

//IMAGE imgRot = RotateImage(&img, (45.0 \* PI) / 180.0);

IMAGE imgRot = FancyRotateImage(&img, (30.0 \* PI) / 180.0);

And finally, add the following code in **bitmap.c**:

// Given an image and an angle in radians,

// rotate that image counterclockwise in a larger canvas

// Parameters: imgPtr -> an image to rotate

// theta -> an angle measured in radians

// Returns: rotated image in a canvas big enough for the rotation

IMAGE FancyRotateImage(IMAGE\* imgPtr, double theta)

{

IMAGE img = { NULL, NULL };

// Allocate space for the header

img.bmHDR = (BITMAPHDR\*)malloc(sizeof(BITMAPHDR));

memcpy(img.bmHDR, imgPtr->bmHDR, sizeof(BITMAPHDR));

// Allocate space for the image

DWORD imageSize = 0;

BYTE padding = 0;

// Determine the center of rotation (middle of input)

LONG oX = imgPtr->bmHDR->lWidth / 2;

LONG oY = imgPtr->bmHDR->lHeight / 2;

double alpha = atan(oY / (double)oX);

// Calculate the diagonal size of the image

// that will be maximum height/width of rotated image

int maxDimension = (int)ceil(sqrt(4.0 \* oX \* oX + 4.0 \* oY \* oY));

// Convert the angle theta back to degrees, then modulus it by 360,

// store the result in "angle". The "angle" will determine

// which quadrant the width size of the image rotates into.

// It, in turn, determines the formula for calculating

// the exact width and height of the rotated image

// in order not to lose any pixels.

int angle = (int)ceil(theta \* 180.0 / PI) % 360;

if (angle <= 90 || (angle > 180 && angle <= 270))

{

img.bmHDR->lWidth = abs(maxDimension \* cos(-alpha + theta));

img.bmHDR->lHeight = abs(ceil(maxDimension \* sin(alpha + theta)));

}

else

{

img.bmHDR->lHeight = abs(maxDimension \* sin(-alpha + theta));

img.bmHDR->lWidth = abs(ceil(maxDimension \* cos(alpha + theta)));

}

// Calculate the padding

padding = img.bmHDR->lWidth % 4;

// Calculate the image size = # of rows \* size of each row

imageSize = img.bmHDR->lHeight \* (sizeof(PIXEL)

\* img.bmHDR->lWidth + padding);

img.bmHDR->dwImageSize = imageSize;

img.bmHDR->dwFileSize = imageSize + sizeof(BITMAPHDR);

img.bmData = (PIXEL\*)malloc(imageSize);

// Crazy color to make anti-aliasing easier

PIXEL startPixel = { 0, 255, 128 };

// Initialize background to crazy color

InitializePixelData(&img, startPixel);

// Copy pixels to the new image

for (LONG pY = 0; pY < imgPtr->bmHDR->lHeight; pY++)

{

for (LONG pX = 0; pX < imgPtr->bmHDR->lWidth; pX++)

{

// Calculate the position of the current pixel

// in the new image.

// Due to the increased width and height,

// x needs to translated by (lWidth / 2 - oX) and

// y needs to be translated by (lHeight/2 - oY)

LONG x = (LONG)(cos(theta) \* (pX - oX)

- sin(theta) \* (pY - oY)) + oX

+ img.bmHDR->lWidth / 2.0 - oX;

LONG y = (LONG)(sin(theta) \* (pX - oX)

+ cos(theta) \* (pY - oY)) + oY

+ img.bmHDR->lHeight / 2.0 - oY;

if (x >= 0 && x < img.bmHDR->lWidth

&& y >= 0 && y < img.bmHDR->lHeight)

{

\*GetPixel(y, x, &img) = \*GetPixel(pY, pX, imgPtr);

}

}

}

AntiAlias(&img);

return img;

}

# Makefiles

So far, we have used Visual Studio to edit, compile, and run our programs. Although this makes these tasks simple, we don’t understand what the development environment is doing to complete this process. In this section, we will manually compile a program into an executable using a *makefile* (a file containing instructions for compiling and linking project files) and the Microsoft compiler **cl.exe**.

For this exercise, we will need some files in a project. So to begin, let’s create a Visual Studio 2022 project with some statistics files.

## Statistics project

Consider a statistics project (has some stats functions) that contains the following files:

* **mathfuncs.h** (prototypes for all functions)
* **mathfuncs.c** (implementation of math functions)
* **display.c** (implementation of functions to display the results)
* **stats.c** (main function and test code)

Let’s create the code for these files. First, we will create **stats.c** as follows:

#include <stdlib.h>

#include "mathfuncs.h"

int main(void)

{

double data[] = { 12.0, 13.6, 12.4, 10.9 }; // sample numbers

double dAvg, dStDev;

int iSize = sizeof(data) / sizeof(double);

dAvg = average(data, iSize); // calculate average of numbers

dStDev = stDev(data, iSize); // calculate standard deviation of numbers

display(dAvg, dStDev); // It can be a good idea to separate display

// of results from the rest of the code so we

// could, for instance, display it in a GUI later.

return EXIT\_SUCCESS;

}

Then we will create **mathfuncs.h** with the function prototypes for each of the functions called:

#ifndef MATHFUNCS\_H

#define MATHFUNCS\_H

double average(double data[], int iSize);

double stDev(double data[], int iSize);

double square(double dNum); // helper function for stDev

void display(double dAvg, double dStDev);

#endif // !MATHFUNCS\_H

Next, let’s create **display.c**:

#include <stdio.h>

#include "mathfuncs.h"

void display(double dAverage, double dStdDev)

{

printf("Average = %-10.21f\n", dAverage);

printf("St.Dev. = %-10.21f\n", dStdDev);

}

Finally, we will create the statistics functions in **mathfuncs.c**. Note that the standard deviation is a measure of the spread of data. It is not important to know what the standard deviation is for this example, but here’s a brief explanation:

The standard deviation of a set of data *xi* is calculated using the following formula, where is the mean (or average) of the set of data:

For instance, for the sample numbers given { 12.0, 13.6, 12.4, 10.9 }, the mean is 12.225 and standard deviation is approximately 1.114675. If we subtract 10 from the first number and add 10 to the last number resulting in { 2.0, 13.6, 12.4, 20.9 }, the mean stays the same but the standard deviation increases to approximately 7.783048, reflecting the greater spread of the data about the mean.

So here is the code for **mathfuncs.c**:

#include <math.h> // for sqrt function

#include "mathfuncs.h"

double average(double data[], int iSize)

{

double dSum = 0.0;

for (int i = 0; i < iSize; i++)

{

dSum += data[i];

}

return dSum / iSize;

}

double stDev(double data[], int iSize)

{

double dAvg = average(data, iSize);

double dSum = 0.0;

for (int i = 0; i < iSize; i++)

{

dSum += square(data[i] - dAvg);

}

return sqrt(dSum / (iSize - 1));

}

double square(double dNum)

{

return dNum \* dNum;

}

Running the program gives the following results:

A black screen with white text

Description automatically generated

Changing the data to { 2.0, 13.6, 12.4, 20.9 } gives the following updated results:

A black screen with white text

Description automatically generated

## Purpose of a makefile

The purpose of a makefile is to manage the compilation and linking of software projects.

Recall that compiling and linking proceed as follows:

* *source.c* and *source.h* files are compiled to produce object files (**.obj** or **.o** extension). Object files contain code that is understandable to the machine.
* The linker takes all the object files and puts them together in a single executable file. It resolves references to functions like **printf** included in library files (**.a** or **.lib**). A **.lib** file is essentially a collection of object files.

Large software projects can have many source files. Each file needs to be compiled. Some files depend on other files as well. To manage the compiling of files, a *makefile* is used. The makefile is a text file (called **makefile** with no file extension) that lists how files are compiled and linked and the dependencies between files.

The makefile is executed by a utility called **make** (**nmake** in the Microsoft world). Before looking at a makefile, it is important to understand how to compile and link a file on the command line.

## Command-line compiling and linking

The C/C++ compiler provided by Microsoft is simply called **cl.exe**. It is actually an integrated compiler and linker. For example, suppose that you have a source file called **code.c**. To compile and link this source code, the following syntax would be used:

**cl code.c**

The output of this file is called **code.exe** by default (or **a.exe** or **a.out** depending on the compiler and environment).

Compiler switches may also be used. For example:

**-c** or **/c** 🡪 compile only, do not file

**-o** *file* 🡪 place output in a file called *file* (or **/Fe:***file*)

**/Wall** 🡪 turn on all warnings

Switches are case-sensitive and may vary depending on the compiler/linker. To compile the **code.c** file to an object file, we would use:

**cl -c code.c**

**Note:** it is important to have the system path variable include the compiler, include files (headers), library files, and any other files required by the compiler/linker. The **Developer Command Prompt for VS 2022** (from the **Start** menu > **Visual Studio 2022** folder) has all these paths set for you automatically.

## Creating a makefile

Consider our statistics project. To review, we have the following files:

* **mathfuncs.h** (prototypes for all functions)
* **mathfuncs.c** (implementation of math functions)
* **display.c** (implementation of functions to display the results)
* **stats.c** (main function and test code)

The following *dependency graph* shows the dependencies between files when compiling and linking:

Diagram

Description automatically generated

For example, **display.obj** (or **display.o** as shown above) is dependent on the files **display.c** and **mathfuncs.h**. The file **stats.exe** is dependent on **display.obj**, **mathfuncs.obj**, and **stats.obj**. All the dependencies are highlighted in the following dependency graph:

Diagram

Description automatically generated

Each dependency is implemented separately as a target in the makefile.

The basic syntax used in a makefile is:

***target*: [dependency list]**

**<Tab>*command to execute***

Note that **Tab** MUST be used. You cannot indent using spaces. This is a common error and hard to catch if typing this out in a simple text editor such as Notepad.

For example, to implement the dependencies listed above, the following targets are required (color-coding not required – just used to illustrate the dependencies shown above):

**statistics: display.obj mathfuncs.obj stats.obj**

**cl display.obj mathfuncs.obj stats.obj**

**display.obj: display.c mathfuncs.h**

**cl -c display.c**

**mathfuncs.obj: mathfuncs.c mathfuncs.h**

**cl -c mathfuncs.c**

**stats.obj: stats.c mathfuncs.h**

**cl -c stats.c**

To run the makefile, use **nmake** as follows: from the directory where the **makefile** exists and which also includes the source files, type:

**nmake**

The first target will then be built. Notice the order in which the files are built. Since the target **statistics** is dependent on the other targets, those targets are built first. The **.obj** files will be visible now if you view the folder where the **makefile** exists.

The executable is called **display.exe**, named by default. This can be changed by passing the switch to the linker as follows:

**statistics: display.obj mathfuncs.obj stats.obj**

**cl -o statistics.exe display.obj mathfuncs.obj stats.obj**

### Rebuilding a project

Suppose that the file **stats.c** has changed (for instance, to change the sample data as described previously). Should all the files be rebuilt? No, the other files do not need recompiling. **nmake** can check the timestamps on the files to see if they have recently changed. It only calls the required targets. In this case, the targets **stats.obj** and **statistics** would be called.

Alternatively, the user can run a specified target. Suppose that the file **stats.c** is changed and needs to be compiled for testing (but we do not want to link the entire project at this time). **nmake** could be called as follows:

**nmake stats.obj**

### Other targets

Other targets can be added as well. For example, for administrative purposes, a target called **clean** or **remove** can be added:

**clean:**

**del \*.obj \*.exe**

Note that this target is not run automatically when we just type **nmake** since none of the other targets are dependent on it. Instead, it can only be run by typing **nmake clean**.

### Constants/macros

Constants/macros can be used to represent things that repeat or change. For example, what if we switch compilers? We would need to replace **cl** with something else in multiple locations. Instead, we can use a macro:

**CC=cl**

Then replace **cl** with **$(CC)**

Other variables may need to be changed as well. **cl.exe** uses **.obj** whereas **gcc** uses **.o** extensions for object files. So the updated **makefile** might look like the following:

CC=cl

EX=obj

statistics: display.$(EX) mathfuncs.$(EX) stats.$(EX)

$(CC) -o statistics.exe display.$(EX) mathfuncs.$(EX) stats.$(EX)

display.$(EX): display.c mathfuncs.h

$(CC) -c display.c

mathfuncs.$(EX): mathfuncs.c mathfuncs.h

$(CC) -c mathfuncs.c

stats.$(EX): stats.c mathfuncs.h

$(CC) -c stats.c

clean:

del \*.$(EX) \*.exe

Now if you want to compile with **gcc** (Unix/Linux), simple change the **CC=cl** to **CC=gcc** and **EX=obj** to **EX=o**

Note that there are many more options available for **cl.exe** – try **cl /?** for a list of them. There is also a separate linker available – try **link.exe** for a list of options.

You may also note that **cl.exe** says that the **-o** option is deprecated and will be removed in a future version, although it still works now. It is shown here because the **-o** option works with **gcc** as well. The other ways to change the output filename with **cl.exe** are **/Fe***filename* or **/Fe:***filename*.

Finally, this is only a brief overview of makefiles. For more information, you can find many tutorials and examples online, including:

* <https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
* <https://www.gnu.org/software/make/manual/make.html>
* <https://www.cs.usask.ca/staff/oster/makefiles.html>
* <https://learn.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=msvc-170>
* <https://learn.microsoft.com/en-us/cpp/build/reference/creating-a-makefile-project?view=msvc-170>

# Final Exam Notes

The final exam will cover everything in the class with an emphasis on material covered since the midterm exam and file I/O which was covered before the midterm exam but was not on the midterm exam.

To prepare for the final exam, you should try out the various exercises that have been provided throughout the course. In addition, here is a final review exercise:



And a sample solution:

