Samuel Navarro

June 4, 2020

Contents

1 Programming Fundamentals

1.1 Converting Between Decimal and Binary

Hardware Representations

First and foremost, as far as the computer is concerned, there is no way to move "past numbers" because to the computer, everything is a number. A computer stores everything as a series of 0's and 1's. Each 0 or 1 is called a bit, and there are many ways to interpret these bits. This is where types come in. A type is a programming language construct that specifies both a size and an interpretation of a series of bits stored in a computer. For example, the type for working with integers is an int, whose size is typically 32 bits and whose interpretation is an integer number directly represented in binary.

Binary Numbers

Before we delve into how to represent numbers in binary, let us briefly discuss the decimal system, which should be familiar to all of us. A decimal number is a number represented in base 10, in which there are 10 possible values for each digit (0–9). When these digits are concatenated to make strings of numbers, they are interpreted column by column. Beginning at the far right and moving to the left, we have the 1's column, the 10's column, the 100's column, and so forth. The number 348, for example, has 8 ones, 4 tens, and 3 hundreds. The value of each column is formed by taking the number 10 and raising it to increasing exponents. The ones column is actually $10^0 = 1$, the tens column is $10^1 = 10$, the hundreds column is $10^2 = 100$, and so forth. When we see a number in base 10, we automatically interpret it using the process shown in the Figure 1 below, without giving it much thought.

A binary number is a number represented in base 2, in which there are only 2 possible values for each digit (0 and 1). The 0 and 1 correspond to low and high voltage values stored in your computer. Although it might be possible for a computer to store more than two voltage values and therefore support a base larger than 2, it would be extremely difficult to support the 10 voltage values that would be required to support a base 10 number system

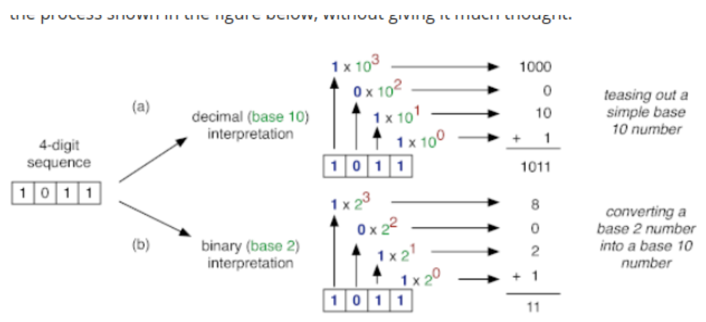

4-digit sequence
1 0 1 1
(a)
decimal (base 10) interpretation
(b)
binary (base 2) interpretation
1 x 10^3
0 x 10^2
1 x 10^1
1 x 10^0
1000
0
10
+ 1
1011
teasing out a simple base 10 number
1 x 2^3
0 x 2^2
1 x 2^1
1 x 2^0
8
0
2
+ 1
11
converting a base 2 number into a base 10 number

Figure 1: binary

in hardware. A familiarity with base 2 is helpful in understanding how your computer stores and interprets data.

Binary numbers are interpreted such that each bit (the name for a binary digit) holds the value 2 raised to an increasing exponent, as shown in the figure part b). We begin with the rightmost bit (also called the least significant bit) which holds the value $2^0 = 1$, or the ones column. The next bit holds the value $2^1 = 2$, or the twos column. In base 10, each column is ten times larger than the one before it. In base 2, each column's value grows by 2. The number 10_2 (the subscript indicates the base) has 1 two and no ones. It corresponds to the value 2 in base 10.

1.2 Looking Under the Hood

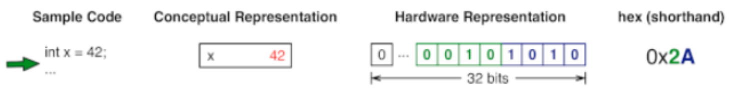


Figure 2: Under the Hood

As mentioned earlier, a type indicates both a size and an interpretation. The Figure 2 above shows you the now-familiar figure with code and its conceptual representation. For this module, we will add a third column, showing you the underlying representation at the hardware level. When you declare a variable x of type int, you should think about this conceptually as a box called x with a value 42 inside. But at a hardware level, the type int means that you have allocated 32 bits dedicated to this variable, and you have chosen for these bits to be interpreted as an integer number in order to yield the value 42.

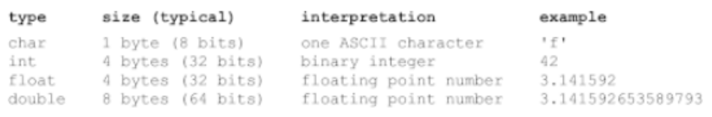
Hexadecimal

As you may well imagine, reading and writing out a series of 32 of 1's and 0's is tedious at best and error-prone at worst. Because of this, many computer scientists choose to write out the values of numbers they are thinking about in binary using an encoding called hexadecimal, or hex for short. Hex is base 16, meaning that it represents a number with a 1's column, a 16's column, a 256's column, and so on. As a hex digit can have 16 possible values (0–15), but our base 10 digits have only 10 possible symbols (0–9) we use the letters A-F to represent the values 10-15 in a single digit. The number eleven, for example, is represented by the single digit 'B' in hex.

Numbers represented in binary can easily be converted to hex by simply grouping them into 4-digit clusters, each of which can be represented by a single hex digit. For example, the 4 rightmost bits in the figure above (colored blue) are 1010, which has the decimal value 10 and the hex value A. The next 4 bits in the figure (colored green) are 0010, which has the decimal value 2 and the hex value 2. The remaining 24 bits in the number are all zeroes. Instead of writing out the entire 32 bit binary sequence, we can use 8 digits of hex (0x0000002A) or the shorthand 0x2A. (In both cases, the leading 0x (interchangeable with just x) indicates that the number is in hex.)

El ejemplo es que en binario, el primer valor es 2^0 *el segundo es* 2^2*, etc.*

1.3 Basic Data Types



type	size (typical)	interpretation	example
char	1 byte (8 bits)	one ASCII character	'f'
int	4 bytes (32 bits)	binary integer	42
float	4 bytes (32 bits)	floating point number	3.141592
double	8 bytes (64 bits)	floating point number	3.141592653589793

Figure 3: Basic data types

Char

A char (pronounced either "car" or "char") is the smallest data type—a mere 8 bits long—and is used to encode characters. With only 8 bits, there are only $2^8 = 256$ possible values for a char (from 00000000 to 11111111). On most machines you will use, these 8 bits are interpreted via the American Standard Code for Information Interchange (or ASCII) character-encoding scheme, which maps 128 number sequences to letters, basic punctuation, and upper- and lower-case letters. A subset of this mapping is shown in the figure below - please don't try to memorize it. Another, much more expressive character-encoding scheme you may encounter (particularly when needing to encode non-English characters) is Unicode (which requires more than 1 byte).

Now if you look at the first line of code in this next Figure 4

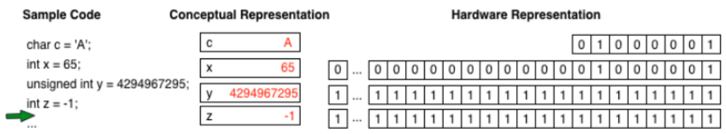

```
char c = 'A';
int x = 65;
unsigned int y = 4294967295;
int z = -1;
...
```

Figure 4: Char types

you can see the char c declared and initialized to the value 'A'. Notice that we wrote A in single quotation marks—these indicate a character literal. In the same way that we could write down a string literal previously, we can also write down a character literal: the specific constant character value we want to use. Writing down this literal gives us the numerical value for A without us having to know that it is 65. If we did need to know, we could consult an ASCII table like the one in the figure above. Being able to write 'A' instead of 65 is another example of abstraction—we do not need to know the ASCII encoding, we can just write down the character we want.

Figure caption: Examples of chars and ints. At first glance, c and x appear identical since they both have the binary value 65. However, they differ in both size (c has only 8 bits whereas x has 32) and interpretation (c's value is interpreted using ASCII encoding whereas x's value is interpreted as an integer). Similarly, y and z are identical in hardware but have differing interpretations because y is unsigned and z is not.

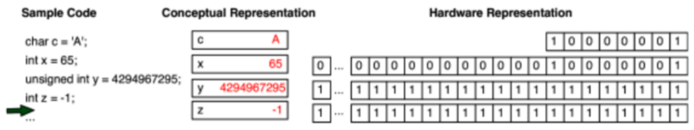
Int

We have said that an int is a 32-bit value interpreted as an integer directly from its binary representation. As it turns out, this is only half of the story—the positive half of the story. If we dedicate all 32 bits to expressing positive numbers, we can express 2^{32} values, from 0 up to 4,294,967,295. We request this interpretation by using the qualifier unsigned in the declaration, as shown in the second line of the figure below.

What about negative numbers? ints are actually represented using an encoding called two's complement, in which half of the 2^{32} possible bit patterns are used to express negative numbers and the other half to express positive ones. Specifically, all numbers with the most significant bit equal to 1 are negative numbers. A 32-bit int is inherently signed (i.e., can have both positive and negative values) and can express values from -2,147,483,648 to 2,147,483,647. Note that both unsigned and signed ints have 2^{32} possible values. For the unsigned int they are all positive; for the signed int, half are positive and half are negative.

In two's complement, the process for negating a number may seem a bit weird, but actually makes a lot of sense when you understand why it is setup this way. To compute negative X, you take the bits for X, flip them (turn 0s into 1s and 1s into 0s), and then add 1. So if you had 4-bit binary and took the number 5 (0101) and wanted negative 5, you would first flip the bits (1010) and then add 1 (1011). Why would computer scientists pick such a strange rule? It turns out that this rule makes it so that you can just add numbers naturally and get the right result whether the numbers are positive or negative. For example -5 + 1 = -4, and in binary 1011 + 0001 is 1100. To see that 1100 is -4, flip the bits (0011) and add 1 (0100) which is 4.

Another pair of qualifiers you may run into are short and long which decrease or increase the total number of bits dedicated a particular variable, respectively. For example, a short int (also referred to and declared in C simply as a short) is often only 16 bits long. Technically, the only requirement that the C language standard imposes is that a short int has fewer than or equal to as many bits as an int, and that a long int has greater than or equal to as many bits as an int.



```
char c = 'A';
int x = 65;
unsigned int y = 4294967295;
int z = -1;
...
```

Figure 5: Int type

Referring to the Figure 5 above, at first glance, c and x appear identical since they both have the binary value 65. However, they differ in both size (c has only 8 bits whereas x has 32) and interpretation (c's value is interpreted using ASCII encoding whereas x's value is interpreted as a signed integer). Similarly, y and z are identical in hardware but have differing interpretations because y is unsigned and z is not.

Float and Double

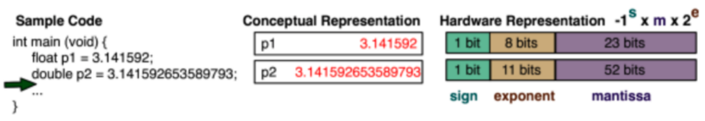
The final two basic data types in C allow the programmer to express real numbers. Since there are an infinite number of real numbers, the computer cannot express them all (that would require an infinite number of bits!). Instead, for values that cannot be represented exactly, an approximation of the value is stored.

If you think about the fact that computers can only store values as 0s and 1s, you may wonder how it is possible to store a real number, which has a fractional part. In much the same way that decimal representations of a number can have a fractional portion with places to the right of a decimal point (the tenth's, hundredth's, thousandth's, etc. places), binary representations of numbers can have fractional portions after the binary point. The places to the right of the binary point are the half's, quarter's, eighth's, etc. places.

One way we could (but often do not) choose to represent real numbers is fixed point. We could take 32 bits, and interpret them as having the binary point in the middle. That is, the most significant 16 bits would be the "integer" part, and the least 16 bits would be the "fractional" part. While this representation would be conceptually simple, it is also rather constrained—we could not represent very large numbers, nor could we represent very small numbers precisely.

Instead, the most common choice is similar to scientific notion. Recall that in decimal scientific notation, number 403 can be expressed as 4.03×10^2. Computers use floating point notation, the same notation but implicitly in base 2: $m \times 2^e$. m is called the mantissa (though you may also hear it referred to as the significand). e is the exponent.

A float has 32 bits used to represent a floating point number. These 32 bits are divided into three fields. The lowest 23 bits encode the mantissa; the next 8 bits encode the exponent. The most significant bit is the sign bit, s, which augments our formula as follows: $(-1)^s \times m \times 2^e$. (When s = 1, the number is negative. When s = 0, the number is positive.) A double has 64 bits and uses them by extending the mantissa to 52 bits and the exponent to 11 bits. Examples of both a float and a double are shown in the Figure 6 below.



```
int main (void) {
    float p1 = 3.141592;
    double p2 = 3.141592653589793;
    ...
}
```

Figure 6: Double fp type

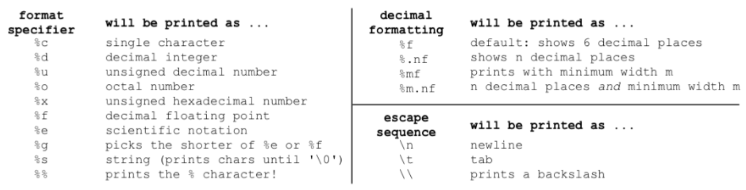
Standards. There would be many possible ways to divide a given number of bits into the mantissa and exponent fields. The arrangement here is part of the IEEE (Institute of Electrical and Electronics Engineers) Standard. Industry standards like these make it possible for engineers from a variety of companies to agree upon a single encoding by which floating point numbers can be represented and subsequently interpreted across all languages, platforms, and hardware products. Part of the IEEE Standard for floating point notation involves two adjustments to the bit-wise representations of a float and a double. These adjustments (normalization and adding a bias) make the actual binary representation of these numbers less accessible to a first time observer. We encourage the interested reader to read the actual IEEE floating point Standard and allow the less curious reader simply to trust that there is a bit-wise encoding for the numbers in the figure above, which is just outside the scope of this course.

Precision. There are an infinite number of values between the numbers 0 and 1. It should be unsurprising, then, that when we use a finite number of bits to represent all possible floating point values, some precision will be lost. A float is said to represent single-precision floating point whereas a double is said to represent double-precision floating point. (Since a double has 64 bits, it can dedicate more bits to both the mantissa and exponent fields, allowing for more precision.)

As we will discuss in the next section, the default print setting for floats and doubles is to print up to 6 decimal places (see figure below). As a consequence, the user has no reason to think that these numbers are not exactly 2.0 and the fact that the neither test for equality to 2.0 passes is simply confusing.

It is important for programmers to understand precision when they choose types for their variables and when they perform tests on variables whose values are assumed to be known. Some programs will need more precision in order to run correctly. Some programs will have to allow for a small degree of imprecision in order to run correctly. Understanding exactly the level of precision required for your code is critical to writing correct code. For every project you begin (or join) it is definitely worth taking a minute to think about the code and how important precision might be in that particular domain. This is true particularly for programs that will ultimately be used to make life-and-death decisions for those who have no say over the precision decisions you are making for them.

It is also important to understand the cost. A double takes up twice as much space as a float. This may not matter for a single variable, but some programs declare thousands or even millions of variables at a time. If these variables do not require the precision of a double, choosing a float can make your code run faster and use less memory with no loss of correctness.



format specifier	will be printed as ...
%c	single character
%d	decimal integer
%u	unsigned decimal number
%o	octal number
%x	unsigned hexadecimal number
%f	decimal floating point
%e	scientific notation
%g	picks the shorter of %e or %f
%s	string (prints chars until '\0')
%%	prints the % character!

decimal formatting	will be printed as ...
%f	default: shows 6 decimal places
%.nf	shows n decimal places
%mf	prints with minimum width m
%m.nf	n decimal places *and* minimum width m

escape sequence	will be printed as ...
\n	newline
\t	tab
\\	prints a backslash

Figure 7: Format specifier

1.4 Expressions have types

Overflow and Underflow

The fact that each type has a set size creates a limit on the smallest and largest possible number that can be stored in a variable of a particular type. For example, a short is typically 16 bits, meaning it can express exactly 2^{16} possible values. If these values are split between positive and negative numbers, then the largest possible number that can be stored in a short is 0111111111111111, or 32767.

What happens if you try to add 1 to this number? Adding 1 yields an unsurprising 1000000000000000. The bit pattern is expected. But the interpretation of a signed short with this bit pattern is -32768, which could be surprising

If the short were unsigned, the same bit pattern 1000000000000000 would be interpreted as an unsurprising 32768.

This odd behavior is an example of overflow: an operation results in a number that is too large to be represented by the result type of the operation. The opposite effect is called underflow in which an operation results in a number that is too small to be represented by the result type of the operation. Overflow is a natural consequence of the size limitations of types.

Note that overflow (and underflow) are actions that occur during a specific operation. It is correct to say "Performing a 16-bit signed addition of 32767 + 1 results in overflow." It is not correct to say "-32768 overflowed." The number -32768 by itself is perfectly fine. The problem of overflow (or underflow) happens when you get -32768 as your answer for 32767 + 1. The operation does not have to be a "math" operation to exhibit overflow. Assignment of a larger type to a smaller type can result in overflow as well. Consider the following code:

In this code, the assignment of x (which is a 32-bit int) to s (which is a 16-

```c
short int s;
int x = 99999;
s = x;
printf("%d\n", s);
```

Listing 1: Overflow

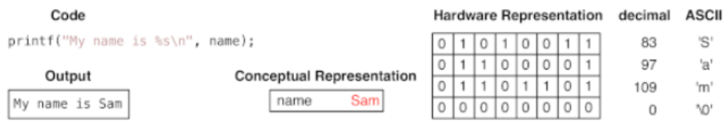
bit short int) overflows—the truncation performed in the type conversion discards non-zero bits. This code will print out -31073, which would be quite unexpected to a person who does not understand overflow.

Whether overflow is a problem for the correctness of your program is context-specific. Clocks, for example, experience overflow twice a day without problems. (That 12:59 is followed by 1:00 is the intended behavior). As a programmer, realize that your choice of type determines the upper and lower limits of each variable, and you are responsible for knowing the possibility and impact of overflow for each of these choices.

1.5 Non-numbers

Strings

A string is a sequence of characters that ends with a special character called the null terminator, which can be written with the character literal '\0' (pronounced "backslash zero") that signals the end of the string. A string is referred to by the location of the first character in memory and each 8-bit character is read until the '\' is detected. A simple drawing of this concept is shown in the figure below:



```
printf("My name is %s\n", name);
```

```
My name is Sam
```

Hardware Representation								decimal	ASCII
0	1	0	1	0	0	1	1	83	'S'
0	1	1	0	0	0	0	1	97	'a'
0	1	1	0	1	1	0	1	109	'm'
0	0	0	0	0	0	0	0	0	'\0'

Figure 8: strings

Strings are not a basic data type in C, meaning you cannot simply declare and use them as you would an int or a double. To give you a tiny glimpse into the complexity of the matter, consider how large a string should be. Is there pre-defined number of bits that should correspond to a string data type? Since each string has a unique number of characters, this does not seem like a choice that can be made up front. In fact, the size of a string

```c
struct rect_tag {
        int left;
        int bottom;
        int right;
        int top;
};

typedef struct rect_tag rect_t;

int main(){
        rect_t myRect;
        myRect.left = 1;
        // more code
}
```

Listing 2: TypeDef

will need to be dynamically determined on a per-string basis. To truly understand how to create and use strings, an understanding of pointers is required. This is one reason why the above figure is deliberately lacking in details—because we haven't yet explained the concepts necessary to show you how to declare and instantiate them. We will delay further discussions of strings until later in the specialization.

2 Introduction

1. Do an instance of the problem.
2. Write down that we just did.
3. Generalize these steps.
4. Test your algorithm. We should work on *different* instances of the problem. The goal is to find out if we mis-generalized before.
5. Translate into code

Guidelines to Tests

1. Try test cases that are qualitatively different from what you used to design your algorithm.

```c
typedef struct rect_tag {
        int left;
        int bottom;
        int right;
        int top;
} rect_t;

int main(){
    rect_t myRect;
        myRect.left = 1;
        // more code
}
```

Listing 3: TypeDef v2

2. Try to find **corner cases**.If your algorithm takes a list off things as an input, try it with an empty list. If you count from 1 to N. Try N=0 or N=1.

3. Try to obtain *statement coverage*, that is, between all of your test cases, each line in the algorithm should be executed at least once.

Repetition

Whenever you have discovered repetition while generalizing your algorithm, it translates into a loop. Typically, if your repetition involves counting, you will use a **for-loop**. Otherwise, if you are sure you always want to do the body at least once, a **do-while** is the most appropriate type. In other cases (which typically align with steps like ”as long as (something)...”) while loops are generally your best bet. If your algorithm calls for you to ”stop repeating things” or ”stop counting” you will want to translate that idea into a **break** statement. Meanwhile, if your algorithm calls for you to skip the rest of the steps in the current repetition, and go back the start of the loop, that translates into a **continue** statement.

Complicated Steps

Whenever you have a complex line in your algorithm—something that you cannot translate directly into a few lines of code—you should call another function to perform the work of that step.

Decision Making

Whenever your algorithm calls for you to make a decision, that will translate into either **if else** or **switch-case**. You will typically only want **switch-case** when you are making a decision based on many possible numerical values of one expression. Otherwise use **if-else**.

3 Writing and Fixing Code

3.1 Compiler

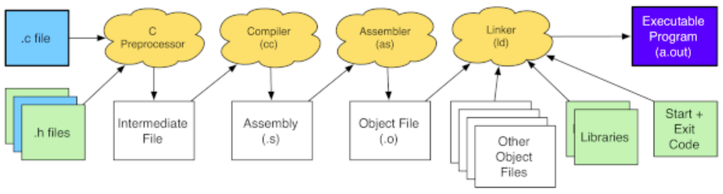


Figure 9: Compiler

Using macro definitions for constants provides a variety of advantages to the programmer over writing the numerical constant directly. For one, if the programmer ever needs to change the constant, only the macro definition must be changed, rather than all of the places where the constant is used. Another advantage is that naming the constant makes the code more readable. The naming of the constant in return EXIT_SUCCESS gives you a clue that the return value here indicates that the program succeeded.

A third advantage of using macro defined constants is portability. While 0 may indicate success on your platform—the combination of the type of hardware and the operating system you have—it may mean failure on some other platform.

3.2 Macros

Macros can also take arguments, however, these arguments behave differently from function arguments. Recall that function calls are evaluated when the program runs, and the values of the arguments are copied into the function's newly created frame. Macros are expanded by the preprocessor (while the program is being compiled, before it has even started running), and the arguments are just expanded textually. In fact, the macro arguments do not have any declared types, and do not even need to be valid C expressions—only the text resulting from the expansion needs to be valid (and well typed).

Macro arguments do not have types to pass it.

However, what happens if we attempt SQUARE(z-y)? If this were a function, we would evaluate z-y to a value and copy it into the stack frame for a call to SQUARE, however, this is a macro expansion, so the preprocessor works only with text. It expands the macro by replacing x in the macro definition with the text z-y, resulting in $z - y * z - y$. Note that this will compute $z - (y * z) - y$, which is not z-y squared.

The SQUARE macro is a bit contrived—we would be better to write the multiplication down where we need it—but highlights the nature (and dangers) of textual expansion of macros. They are quite powerful and can be used for some rather complex things, but you will not need them for most things you write in this specialization.

The Actual Compiler

The output of the preprocessor is stored in a temporary file, and passed to the actual compiler.

Compiler Errors

Dealing With Compilation Errors: Tip 1 Remember that the compiler can get confused by earlier errors. If later errors are confusing, fix the first error, then try to recompile before you attempt to fix them.

Dealing With Compilation Errors: Tip 2 If parts of an error message are completely unfamiliar, try to ignore them and see if the rest of the error message(s) make sense. If so, try to use the part that makes sense to understand and fix your error. If not, search for the confusing parts on Google and see if any of them are relevant.

3.3 Assembling

The next step is to take the assembly that the compiler generated and assemble it into an object file. gcc invokes the assembler to translate the assembly instructions from the textual/human readable format into their numerical encodings that the processor can understand and execute. This translation is another example of the rule that "everything is a number"—even the instructions that the computer executes are numbers.

While it is possible to get error messages at this stage, it should not happen for anything you will try to do in this specialization. Generally errors here are limited to cases in which you explicitly write the specific assembly level instructions that you want into your program (which is possible in C, but limited to very advanced situations) and make errors in those.

The important thing to understand about this step is that it results in an object file. The object file contains the machine-executable instructions for the source file that you compiled, but is not yet a complete program. The object file may reference functions that it does not define (such as those in the C library, or those written in other files). You can request that gcc stop after it assembles an object file by specifying the -c option. By default, the name of the object file will be the name of the .c file with the .c replaced by .o. For example, gcc -c xyz.c will compile xyz.c into xyz.o. If you wish to provide a different name for the object file, use the -o option followed by the name you want. For example, gcc -c xyz.c -o awesomeName.o will produce an object file called awesomeName.o.

This ability to stop is important for large programs, where you will split the code into multiple different C files. Splitting the program into large files is primarily useful to help you keep the code split into manageable pieces. However, each source file can be individually compiled to an object file, and those object files can then be linked together (as we will discuss in the next section). If you change code in one file, you can recompile only that file (to generate a new object file for it), and re-link the program without recompiling any of the other source files. For programs that you write in this specialization, this will not make a difference. However, when you write real programs, you may have tens or hundreds of thousands of lines of code split across dozens or hundreds of files. In this case, the difference between recompiling one file and recompiling all files (especially if optimizations are enabled) may be the difference between tens of seconds and tens of minutes.

3.4 Linking

The final step of the process is to link the program. Linking the program takes one or more object files and combines them together with various libraries, as well as some startup code, and produces the actual executable binary. The object files refer to functions by name, and the linker's job is to resolve these references—finding the matching definition. If the linker cannot find the definition for a name (called a "symbol") that is used, or if the same symbol is defined multiple times, it will report an error.

Linker errors—indicated by the fact that they are reported by ld (the linker's name)—are typically less common than other compiler errors. If you encounter an unresolved symbol error, it means that you either did not define the symbol, did not include the object file that defines the symbol in the link, or that the symbol was specified as only visible inside that object file (which can be done by using the static keyword—a feature we will not delve into). If you encounter errors from duplicate definitions of a symbol, first make sure you did not try to name two different functions with the same name. Next, make sure you did not include any files twice on the compilation command line. Finally, make sure you do not #include a .c file—only header files—and that you only include function prototypes in the header file, not the function's definition (there are some advanced exceptions to these rules, but if you are at that stage, you should understand the linking process and static keyword well enough to fix any problems).

3.5 More Tools

Recall that the input to make is a Makefile, which contains one or more rules that specify how to produce a target from its prerequisites (the files it depends on). The rule is comprised of the target specification, followed by a colon, and then a list of the prerequisite files. After the list of prerequisites, there is a newline, and then any commands required to rebuild that target from the prerequisites. The commands may appear over multiple lines; however, each line must begin with a TAB character (multiple spaces will not work, and accidentally using them instead of a TAB is often the source of problems with a Makefile).

Once all files which a target depends on are ensured to be up to date, make checks if the target itself needs to be (re)built. First, make check if the target file exists. If not, it must be built. If the target file already exists, make compares its last-modified time (which is tracked by all major filesystems) to the last-modified times of each of the prerequisites specified in the rule. If any dependency is newer than the target file, then the target is out of date, and must be rebuilt. Note that if any of the prerequisites were rebuilt in this process, then that file will have been modified more recently than the target, thus forcing the target to be rebuilt.

In Listing 4 we see three targets: myProgram, oneFile.o and anotherFile.o If we just type make, then make will attempt to (re)build myProgram, as that is the first target in the file, and thus the default. This target depends on oneFile.o and anotherFile.o, so the first thing make will do is make the oneFile.o target (much as if we had typed make oneFile.o).

After processing oneFile.o, make does a similar process for anotherFile.o.

```
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
oneFile.o: oneFile.c oneHeader.h someHeader.h
    gcc -std=gnu99 -pedantic -Wall -c oneFile.c
anotherFile.o: anotherFile.c anotherHeader.h someHeader.h
    gcc -std=gnu99 -pedantic -Wall -c anotherFile.c
```

Listing 4: Makefile

```
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
oneFile.o: oneFile.c oneHeader.h someHeader.h
    gcc $(CFLAGS) -c oneFile.c
anotherFile.o: anotherFile.c anotherHeader.h someHeader.h
    gcc $(CFLAGS) -c anotherFile.c
```

Listing 5: Make - Variables

After that completes, it checks if it needs to build myProgram (that is, if either myProgram does not exist, or either of the object files that it depends on are newer than it). If so, it runs the specified gcc command. If not, it will produce the message: `make: 'MyProgram' is up to date.`

Variables

When we have the same flags in every instruction. If we wanted to change these options (e.g., to turn on optimizations, or add a debugging flag), we would have to do it in every place.

Clean

A common target to put in a Makefile is a clean target. The clean target is a bit different in that it does not actually create a file called clean (it is therefore called a "phony" target). Instead, it is a target intended to remove the compiled program, all object files, all editor backups (*.c *.h), and any other files that you might consider to be cluttery.

Generic rules

Our example Makefile improved slightly when we used a variable to hold the compilation flags. However, our Makefile still suffers from a lot of repetition,

```
.PHONY: clean
clean:
    rm -f myProgram *.o *.c~ *.h~
```

Listing 6: Make - Clean

```
# A good start, but we lost the dependencies on the header files
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
%.o: %.c
    gcc $(CFLAGS) -c $<
.PHONY: clean
clean:
    rm -f myProgram *.o *.c~ *.h~
```

Listing 7: Make - Generic Rules

and would be a pain to maintain if we had more than a few sources files. If you look at what we wrote, we are doing pretty much the same thing to compile each of our .c source files into an object file.

In make, we can write generic rules. A generic rule lets us specify that we want to be able to build (something).o from (something).c, where we represent the something with a percent-sign (%).

In Listing 7 we have replaced the two rules we had for each object file with one generic rule. It specifies how to make a file ending with .o from a file of the same name, except with .c instead of .o. In this rule, we cannot write the literal name of the source file, as it changes for each instance of the rule. Instead, we have to use the special built-in variable $ <, which make will set to the name of the first prerequisite of the rule (in this case, the name of the .c file).

However, we have introduced a significant problem now. We have made it so that our object files no longer depend on the relevant header files. If we were to change a header file, then make might not rebuild all of the relevant object files. Such a mistake can cause strange and confusing bugs, as one object file may expect data in an old layout but the code will now be passed data in a different layout. We could make every object file depend on every header file (by writing %.o : %.c *.h), however, this approach is overkill—we would definitely rebuild everything that we need to when we

```
# This fixes the problem
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
%.o: %.c
    gcc $(CFLAGS) -c $<
.PHONY: clean
clean:
    rm -f myProgram *.o *.c˜ *.h˜
oneFile.o: oneHeader.h someHeader.h
anotherFile.o: anotherHeader.h someHeader.h
```

Listing 8: Make - Better Generic Rules

change a header file, because we would rebuild every object file, even if we only need to rebuild a few.

Here, we still have the generic rule, but we also have specified the additional dependencies separately. Even though it looks like we have two rules, make understands that we are just providing additional dependence information because we have not specified any commands. If we did specify commands in the, they would supersede the generic rules for those targets.

Managing all of this dependency information by hand would, of course, be tedious and error-prone. The programmer would have to figure out every file which is transitively included by each source file, and keep the information up to date as the code changes. Instead, there is a tool called makedepend which will edit the Makefile to put all of this information at the end. In its simplest usage, makedepend takes as arguments all of the source files (i.e., all of the .c and/or .cpp files), and edits the Makefile. It can also be given a variety of options, such as -I path to tell it to look for include files in the specified path. See man makedepend for more details.

Built-in generic rules

By default, CFLAGS (flags for the C-compiler) and CPPFLAGS (flags for the C preprocessor143), as well as TARGET_ARCH (flags to specify what architecture to target) are empty. By default CC (the C-compiler command) is cc (which may or may not be gcc depending on how our system is configured). The defaults for any of these variables (or any other variables) can be overridden by specifying their values in our Makefile. Note that $@ in OUTPUT_OPTION is a special variable which is the name of the current target (much like $ < is the name of the first prerequisite).

```
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
.PHONY: clean depend
clean:
    rm -f myProgram *.o *.c~ *.h~
depend:
    makedepend anotherFile.c oneFile.c
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

Listing 9: Built-in Generic Rules

In Listing 9 the default rule will result in:

```
gcc -std=gnu99 -pedantic -Wall -c -o something.o something.c
```

Built-in Functions

Our Makefile is looking more like something we could use in a large project, but we have still manually listed our source and object files in a couple places. If we were to add a new source file, but forget to update the makedepend command line, we would not end up with the right dependencies for that file when we run make depend. Likewise, we might forget to add object files in the correct places (e.g., if we add it to the compilation command line, but not the dependencies for the entire program, we may not rebuild that object file when needed).

We can fix these problems by using some of make's built-in functions to automatically compute the set of .c files in the current directory, and then to generate the list of target object files from that list. The syntax of function calls in make is $(functionName arg1, arg2, arg3). We can use the $(wildcard pattern) function to generate the list of .c files in the current directory: SRCS = $(wildcard *.c). Then we can use the $(patsubst pattern, replacement, text) function to replace the .c endings with .o endings: OBJS = $(patsubst %.c, %.o, $(SRCS)). Once we have done this, we can use $(SRCS) and $(OBJS) in our Makefile.

We could, however, be a little bit fancier. In a real project, we likely want to build a debug version of our code (with no optimizations, and -ggdb3 to turn on debugging information—see the next module for more info about debug-

```
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall
SRCS=$(wildcard *.c)
OBJS=$(patsubst %.c,%.o,$(SRCS))
myProgram: $(OBJS)
    gcc -o $@ $(OBJS)
.PHONY: clean depend
clean:
    rm -f myProgram *.o *.c~ *.h~
depend:
    makedepend $(SRCS)
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

Listing 10: Built-in Functions

ging), and an optimized version of our code that will run faster (where the compiler works hard to produce improve the instructions that it generates, but those transformations generally make debugging quite difficulty). We could change our CFLAGS back and forth between flags for debugging and flags for optimization, and remember to make clean each time we switch. However, we can also just set our Makefile up to build both debug and optimized object files and binaries with different names.

The output of Listing 11 will be both myProgram (the optimized version), and myProgram-debug (which is compiled for debugging).

3.6 Compiler Options

We strongly recommend compiling with at least the following warning options:

-Wall -Wsign-compare -Wwrite-strings -Wtype-limits -Werror

These options will help catch a lot of mistakes, and should not pose an undue burden on correctly written code.

Recent versions of gcc also support an option -fsanitize=address which will generate code that includes extra checking to help detect a variety of problems at runtime. Using this option is also strongly recommended during your development cycle. However, we will note that (at least with gcc 4.8.2

```makefile
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall -O3
DBGFLAGS = -std=gnu99 -pedantic -Wall -ggdb3 -DDEBUG
SRCS=$(wildcard *.c)
OBJS=$(patsubst %.c,%.o,$(SRCS))
DBGOBJS=$(patsubst %.c,%.dbg.o,$(SRCS))
.PHONY: clean depend all
all: myProgram myProgram-debug
myProgram: $(OBJS)
    gcc -o $@ -O3 $(OBJS)
myProgram-debug: $(DBGOBJS)
    gcc -o $@ -ggdb3 $(DBGOBJS)
%.dbg.o: %.c
    gcc $(DBGFLAGS) -c -o $@ $<
clean:
    rm -f myProgram myProgram-debug *.o *.c~ *.h~
depend:
    makedepend $(SRCS)
    makedepend -a -o .dbg.o  $(SRCS)
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

Listing 11: Debug and Optimization

```c
typedef struct template{
  int x, y, width, height;
} rectangle;

rectangle intersection(rectangle r1, rectangle r2) {
  rectangle r;
  r.x = max(r1.x, r2.x);
  r.y = max(r1.y, r2.y);
  r.width = min(r1.width, r2.width);
  r.height = min(r1.height, r2.height);
  return r;
}
```

Listing 12: Typedef

and valgrind 3.10.0) you cannot run a program built with this option in valgrind. The two tools detect different, but overlapping sets of problems, so use of both is a good idea—they just have to be used separately.

3.7 Rectangle Assignment

Definition of a `typedef struct`

4 Testing

4.1 Testing

We say that a Test Case is good if a program **fails** it.

Black Box Testing

Without seeing the code, what test cases might you come up with?. Without even starting the code, we would like to imagine examples where the code would fail and start step 1 with this in mind.

4.1.1 Test-driven Development

Practical Tips for Designing Test Cases

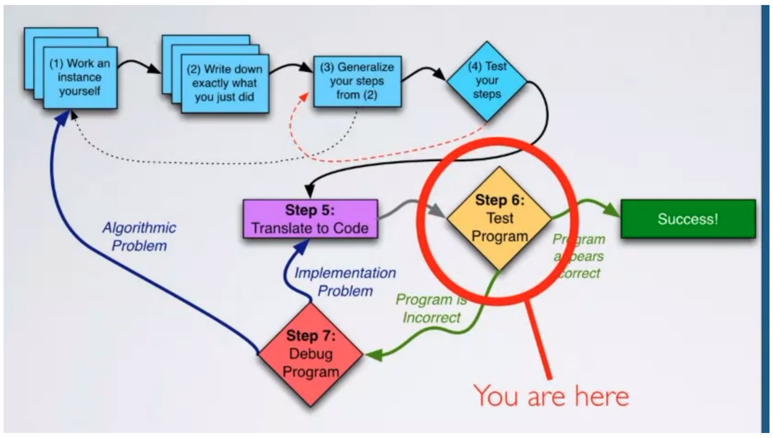


Figure 10: Test in the 7 Steps

Writing good tests cases requires a lot of thought: you need to think about a wide variety of things that can go wrong. Here are some suggestions to guide your thinking, as well as important lessons. First, some thoughts for testing error cases:

- Make sure your tests cover every error case. Think about all the inputs that the program cannot handle, i.e., ones where you would expect to receive an error message. If the program requires a number, give it "xyz", which is not a valid number. An even better test case might be "123xyz" which starts out as a number but isn't entirely a number. How the program should handle this depends on the requirements of the program, but you want to make sure it handles it correctly.
- Be sure to test "too many" as well as "too few". If a program requires exactly N things, test it with at least one case greater than N and at least one case with fewer than N. For example, if a program requires a line of input with exactly 10 characters, test it with 9 and with 11.
- Any given test case can only test one "error message and exit" condition. This means that if you want to test two different error conditions, you need two different test cases: one for each error condition. Suppose that our program requires a three letter string of lower-case letters. We might be tempted to test with "aBcD" to test two things at once, but this will only test one (and believing you tested both is problematic!) To see why this rule exists, think about the algorithm to check for these errors:

Check if the input string does not have exactly three letters

```
If it does not then:

print "Error: the input needs to be 3 letters"

exit the program

Check if the input string is made up entirely of lowercase letters

If it does not then:

print "Error: the input needs to be all lowercase letters"

exit the program
```

Violating one input condition will cause the program to exit! This is also true of other situations where the program rejects the input, even if does not exit

- Test exactly at the boundary of validity. If a program requires between 7 and 18 things, you should have test cases with 6, 7,8, 17, 18, and 19 things. You need to make sure that 6 and 19 are rejected while 7, 8, 17, and 18 are accepted. Testing exactly at the boundaries is important because of the common "off by one" mistake—maybe the programmer wrote ¡ when he should have written ¡= or ¿= when he should have written ¿ or something similar. If you test with values that are right at the boundary, you will find these mistakes.
-

However, testing is not just about checking error handling. You want to make sure that the algorithm correctly handles valid inputs too. Here are some suggestions:

- Think carefully about whether or not there are any special cases where one particular input value (or set of values has to be treated unusually). For example, in poker an Ace is usually ranked the highest, however, it can have the lowest ranking in an "Ace Low Straight" (5 4 3 2 A). If you are testing code related to poker hands, you would want to explicitly test this case, since it requires treating an input value differently from normal
- Think carefully about the requirements, and consider whether something could be misinterpreted, easily mis-implemented, or have variations which could seem correct. Suppose your algorithm works with

sequences of decreasing numbers. You should test with a sequence like 7 6 6 5 4, which has two equal numbers in it. Checking equal numbers is a good idea here, since people might have misunderstood whether the sequence is strictly decreasing (equal numbers don't count as continuing to decrease) or non-increasing (equal numbers do count).

- Think about types. What would happen if the programmer used the wrong type in a particular place? This could mean that the programmer used a type which was too small to hold the required answer (such as a 32-bit integer when a 64-bit integer is required), used an integer type when a floating point type is required, or used a type of the wrong signedness (signed when unsigned is required or vice versa).
- Consider any kind of off-by-one error that the programmer might have been able to make. Does the algorithm seem like it could involve counting? What if the programmer was off by one at either end of the range she counted over? Does it involve numerical comparison? What if ¡ and ¡= (or ¿ and ¿=) were mixed up?
- Whenever you have a particular type of problem in mind, think about how that mistake would affect the answer relative to the correct behavior, and make sure they are different. For example, suppose you are writing a program that takes two sequences of integers and determine which one has more even numbers in it. You are considered that the programmer might have an off-by-one error where he accidentally misses the last element of the sequence. Would this be a good test case?

Sequence 1: 2 3 5 6 9 8

Sequence 2: 1 4 2 8 7 6

This would not be a good test case for this particular problem. If the program is correct, it will answer "Sequence 2" (which has 4 compared to 3). However, if the algorithm mistakenly skips the last element, it will still answer "Sequence 2" (because it will count 3 elements in Sequence 2, and 2 elements in Sequence 1). A good test case to cover this off-by-one-error would be

Sequence 1: 2 3 5 6 9 8

Sequence 2: 1 4 2 8 7 3

Now a correct program will report a tie (3 + 3) and a program with this particular bug will report Sequence 2 as having more even numbers.

Consider all major categories of inputs, and be sure you cover them.

- For numerical inputs, these would generally be negative, zero, and positive. One is also usually a good number to be sure you cover
- For sequences of data, your tests should cover an empty sequence, a single element sequence, and a sequence with many elements.
- For characters: lowercase letters, uppercase letters, digits, punctuation, spaces, non-printable characters
- For many algorithms, there are problem-specific categories that you should consider. For example, if you are testing a function related to prime numbers (e.g., isPrime), then you should consider prime and composite (not prime) numbers as input categories to cover.
- When you combine two ways to categorize data, cover all the combinations. For example, if you have a sequence of numbers, you should test with an empty list, a one element sequence with 0, a one element sequence with a negative number, a one element sequence with a positive number, and have each of negative, zero, and positive numbers appearing in your many-element sequences.
- An important corollary of the previous rules is that if your algorithm gives a set of answers where you can list all possible ones (true/false, values from an enum, a string from a particular set, etc), then your test cases should ensure that you get every answer **at least once**. Furthermore, if there are other conditions that you think are important, you should be sure that you get all possible answers for each of these conditions. For example, if you are getting a yes/no answer, for a numeric input, you should test with a negative number that gives yes, a negative number that gives no, a positive number that gives yes, a positive number that gives no, and zero [zero being only one input, will have one answer].
- All of this advice is a great starting point, but the most important thing for testing is to think carefully—imagine all the things that could go wrong, think carefully about how to test them, and make sure your test cases are actually testing what you think they are testing.

4.1.2 White Box Testing

Another testing methodology is white box testing. Unlike black box testing, white box testing involves examining the code to devise test cases. One

```
1  int aFunction(int a, int b, int c) {
2    int answer = 0;
3    if (a < b) {
4      answer = b - a;
5    }
6    if (b < c) {
7      answer = answer * (b - c);
8    }
9    else {
10     answer = answer + 42;
11   }
12   return answer;
13 }
```

Listing 13: Testing Code

consideration in white box tests is test coverage—a description of how well your test cases cover the different behaviors of your code.

Note that while white box and black box testing are different, they are not mutually exclusive, but rather complimentary. One might start by forming a set of black box test cases, implement the algorithm, then create more test cases to reach a desired level of test coverage.

We will discuss three levels of test coverage: **statement coverage**, **decision coverage**, and **path coverage.**

Statement coverage means that every statement in the function is executed. Statement coverage is a minimal starting point, but if we want to test our code well, we likely want a stronger coverage metric. To see how statement coverage falls short, notice that we have not tested any cases where the a ¡ b test evaluates to false—that is, a case where we do not enter the body of the if statement (line 4).

A stronger level of coverage is **decision coverage**—in which all possible outcomes of decisions are exercised. For boolean tests, this means we construct a test case where the expression evaluates to true, and another where it evaluates to false. If we have a switch/case statement, we must construct at least one test case per choice in the case statement, plus one for the default case, even if it is implicit—that is, if we do not write it down, and it behaves as if we wrote `default: break;`.

Decision coverage corresponds to having a suite of test cases which covers

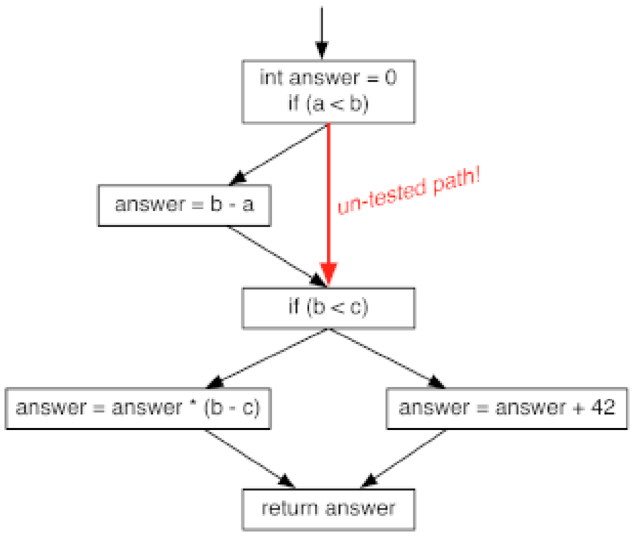


Figure 11: Control Flow Graph

every edge in the graph. In our graph, if a < b is true, we will take the left edge coming out of the first block (corresponding to going into the “then” clause of the if/else). If it is false, we will take the right edge, which skips over this clause, and as there is no else clause, goes to the next if statement. This second edge—representing the path of our execution arrow when a < b is false—is highlighted in red in the figure above, since it represents the shortcomings of our test cases so far with respect to decision coverage.

We can obtain **decision coverage** on this function by adding a test case where $a = 5, b = 2, c = 1$.

And even stronger type of test covergae is **path coverage**. To achieve path coverage, our test cases must span all possible valid paths through the control flow graph (following the direction of the arrows). The Figure 12 shows the four paths through our example control flow grpah. Note that there is one path (shown in red) which we have not yet covered. This path corresponds to the case where a¡b is false, but b¡c is true. Our test cases which gave us decision coverage tested each of these conditions separately, but none of the tests tested both of these together at the same time—where our execution would follow the red path. We can add another test case (e.g., a=5, b=22, c=99) to gain path coverage.

The number of paths through the control flow graph is exponential in the number of conditional choices—if we have 6 if/else statements one after the

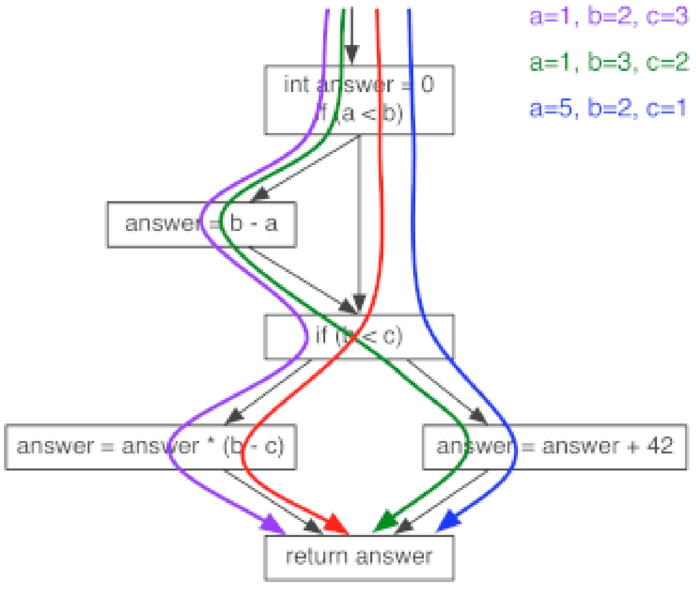


Figure 12: Control Flow Graph

other, then there are 64 possible paths through the control flow graph. If we increase the number of if/else statements from 6 to 16, then the number of paths grows to 65536—quite a lot!

The answer to the correct level of test coverage is depeding of your needs. If you are testing a piece of critical software which will be deployed when you finish testing, only achieving statement coverage would be woefully insufficient. Here, you would likely want to aim for more than just the minimum to achieve path coverage

4.1.3 Generating Test Cases

One approach could be to generate test cases according to some algorithm. If the function we are testing takes a single integer as input, we might iterate over some large range and use each integer in that range as test case.

Another possibility is to generate the test cases pseudo-randomly (called *random testing*). This has some issues if the randomly generated cases are generated by the same seed.

One tricky part about generating test cases algorithmically is that we need some way to verify the answer—and the function we are trying to test is what computes that answer. We obviously cannot trust our function to check itself, leaving us a seeming "chicken and egg" problem. In a very few cases, it may be appealing to write two versions of the function which can be used to check each other. This approach is appropriate when you are writing a complex implementation in order to achieve high performance, but you could also write a simpler (but slower) implementation whose correctness you would more readily be sure of. Here, it makes sense to implement both, and test the complex/optimized algorithm against the simpler/slower algorithm on many test cases.

We may, however, be able to test other properties of the system to increase our confidence that it is correct. Even without knowing the right answer, we can still test that certain properties of the system are obeyed: every message we send should be delivered to the proper destination, that delivery should happen exactly one time, no improper destinations should receive the messages, the delivery should occur in some reasonable time bound, an so on.

Checking these properties does not check that the program gave the right answer (e.g., it may have routed the message by an incorrect but workable path), but it checks for certain classes of errors on those test cases.

Mutation Testing is when you got broken implementations from making small changes in the correct implementation which could represent reasonable mistakes in programming.

4.2 Debugging

Debugging should be an application of the *scientific method*:

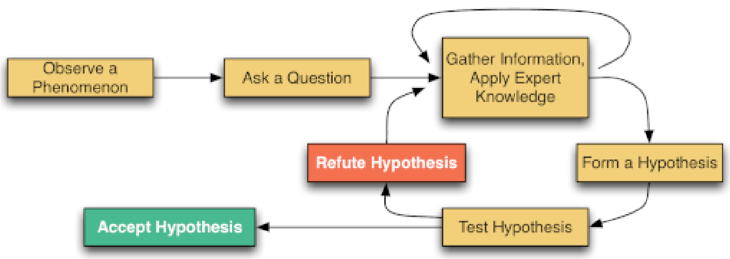


Figure 13: Debugging

Testing our hypothesis may proceed in a variety of ways—sometimes by a combination of them—such as:

Constructing Test Cases

Sometimes our hypothesis will suggest a new test case (in the sense of the testing we discussed in the Testing lesson), that we can use to produce the error under a variety of circumstances. Generally, these follow from the specific cases that your hypothesis makes predictions about: “My program will (something) with inputs that (something).” When your hypothesis suggests this sort of result, construct a test case with the values you were thinking of, and see if your program exhibits that behavior. Also, construct other test cases that do not meet the conditions to see if you were too specific.

Inspecting the internal state of the program

We can use the same tools that we used for gathering more information (print statements, or the debugger) to inspect the internals of the program. This sort of testing is particularly useful when our hypothesis suggests something about the behavior or our program in its past—before it reaches the point where we observe the symptom. This past may be recent (“. . . but that means in the previous iteration of the loop. . . ” or “then x had to be odd on the last line. . . .”) or the distant past (“. . . but for that to be the case, I had to have deleted (something) from (some data structure). . . ” or “. . . then y’s value has to have changed in a way I did not expect. . . ”). Here we want to use the debugger to inspect the state we are interested in and see if it agrees with our hypothesis. Frequently, when we take this approach, discovering the surprising change in state will give us a clue to the problem.

Adding Asserts

We earlier discussed asserts as a testing tool, however, they are also useful for debugging. If our hypothesis suggests we are violating an invariant of our algorithm, we can often write an assert statement to check this invariant. If the assert does not fail, we refute the hypothesis. If the assert fails, it not only gives us confidence in our hypothesis, **but also makes our program fail faster—the closer it fails to the actual source of the problem, the easier it is to fix.**

Code inspection

When evaluating whether to modify the current code or throw it away and start fresh, you should not consider how much time you have already put into it, but rather how much time it will take to fix the current code versus redesigning it from scratch. Note that this is not intended to suggest you should throw away your code and redesign it from scratch every time there is an error. Instead, you should be contemplating the time required for both options, and making a rational tradeoff.

The fact that the program crashes sooner than we expect should not cause us to reject the hypothesis immediately. We must instead consider the possibility that there is another error, which is triggered by overlapping conditions. When faced with such a possibility, we have two options.

4.3 Debugging Tools

Getting Started with gdb

The first step in using gdb (or most any other debugging tool) is to compile the code with debugging symbols—extra information to help a debugging tool understand what the layout of the code and data in memory is—included in the binary. The -g option to gcc requests that it include this debugging information, but if you are using gdb in particular, you should use -ggdb3, which requests the maximum amount of debug information (e.g., it will include information about preprocessor macro definitions) in a gdb-specific format. Note that if you compile your program in multiple steps (object files, then linking), you should include -ggdb3 at all steps.

start: Begin (or restart) the program's execution. Stop the program (to accept more commands) as soon as execution enters main.

run: This command runs the program (possibly restarting it). It will not stop unless it encounters some other condition that causes it to stop (we will learn about these later).

step: Advance the program one "step", in much the same way that we would advance the execution arrow when executing code by hand. More specifically, gdb will execute until the execution arrow moves to a different line of source code, whether that is by going to the next line, jumping in response to control flow, or some other reason. In particular, step will go into a function called by the current line. This command can be abbreviated s.

next: Advance the program one line of code. However, unlike step, if the current line of code is a function call, gdb will execute the entire called function without stopping. This command can be abbreviated n.

print: The print command takes an expression as an argument, evaluates that expression, and prints the results. Note that if the expression has side-effects, they will happen, and will affect the state of the program (e.g., if you do print x = 3, it will set x to 3, then print 3). You can put /x after print to get the result printed in hexadecimal format. This command can be abbreviated p (or p/x to print in hex). Every time you print the value of an expression, gdb will remember the value in its own internal variables which are named $1, $2, etc (you can tell which one it used, because it will say which one it assigned to when it prints the value—e.g., $1 = 42). You can use these $ variables in other expressions if you want to make use of these values later. gdb also has a feature to let you print multiple elements from an array—if you put @number after an lvalue, gdb will print number values starting at the location you named. This feature is most useful with arrays—for example, if a is an array, you can do p a[0]@5 to print the first 5 elements of a.

display: The display command takes an expression as an argument, and displays its value every time gdb stops and displays a prompt. For example display i will evaluate and print i before each (gdb) prompt. You can abbreviate this command disp.

If you hit enter without entering any command, gdb will repeat the last command you entered. This feature is most useful when you want to use step or next multiple times in a row.

Note that if you need to pass command line arguments to your program, you can either write them after the start or run command (e.g., run someArg anotherArg), or you can use set args to set the command line arguments.

When you print expressions, gdb uses the variables in the current scope. However, sometimes, you might want to inspect variables in other frames further up the stack. You can instruct gdb to select different frames with up and down, which move up and down the stack specifically.

One particularly common use of up is when your program stops in a failed assert. When this happens, gdb will stop deep inside the C library, in the code that handles assert. However, you will want to get back to your own code, which is a few stack frames up. You can use up a few times until gdb returns to a frame corresponding to your code.

Controlling Execution

The next and step commands give you the basic ability to advance the state of the program, however, there are also more advanced commands for controlling the execution. If we are debugging a large, complex program, we may not want to step through every line one-by-one to reach the point in the program where we want to gather information.

One of the most useful ways to control the execution of our program is to set a breakpoint on a particular line. A breakpoint instructs gdb to stop execution whenever the program reaches that particular line. You can set a breakpoint with the break command, followed by either a line number, or a function name (meaning to set the breakpoint at the start of that function).

Once we have a breakpoint set, we can run the program (or continue, if it is already started), and it will execute until the breakpoint is encountered (or some other condition which causes execution to stop)

By default, breakpoints are *unconditional breakpoints—gdb* will stop the program and give you control anytime it reaches the appropriate line. Sometimes, however, we may want to stop under a particular condition. For example, we may have a **for** loop which executes 1,000,000 times, and we need information from the iteration where i is 250,000. With an unconditional breakpoint, the program would stop, and we would need to continue many times before we got the information we wanted. We can instead, use a *conditional breakpoint*—once where we give gdb a C expression to evaluate to determine if it should give us control, or let the program continue to run.

We can put a condition on a breakpoint when we create it with the break command by writing if after the location, followed by the conditional expression. We can also add a condition later (or change an existing condition) with the cond command. For example, if we want to make a breakpoint on line size for the condition i==25000, we could tell gdb:

```
(gdb) break 7 if i==250000
```

Alternatively, if the breakpoint already existed, for example, as breakpoint 1, we could write

```
cond 1 i==250000
```

If we write a cond command with no expression, then it makes a breakpoint unconditional. We can also `enable` or `disable` breakpoints (by their numeric id). A disabled breakpoint still exists (and can be re-enabled later), but has no effect—it will not cause the program to stop. We can also `delete` a breakpoint by its numeric id. We can also use `clear breakpointname` . You can use the `info breakpoints` command (which can be abbreviated i b) to see the status of current breakpoints.

Two other useful commands to control the execution of the program are `until`, which causes a loop to execute until it finishes (gdb stops at the first line after the loop), and `finish` (which can be abbreviated fin), which finishes the current function—i.e., causes execution until the current function returns.

Watchpoints

Another incredibly useful feature of gdb is a **watchpoint**—the ability to have gdb stop when the value of a particular expression changes. For example, we can write watch i, which will cause gdb to stop whenever the value of i changes. When gdb stops in response to a watchpoint, it will print the old value of the expression and the new value.

Watchpoints can be a particularly powerful tool when you have pointer-related problems, and values of variables are changing through aliases. However, sometimes, the alias we have when we setup the watchpoint may go out of scope before the change we are interested in happens. For example, we may want to watch ***p**, but p is a local variable, whose scope ends before the value changes. Whenever we face such a problem, we can print p, which will give us the pointer in a gdb variable (e.g., $6), and then we can use that $-variable (which never goes out of scope—it lives until we restart gdb) to set our watchpoint: watch *$6.

Signals

Whenever your program receives a signal, gdb will stop the program and give you control. There are three particularly common signals that come up during debugging. The first is SIGSEGV, which indicates a segmentation fault. If your program is segfaulting, then just running it in gdb can help you gather a lot of information about what is happening. When the segfault happens, gdb will stop, and your program will be on the line where the segfault happened. You can then begin inspecting the state of the program (printing out variables) to see what went wrong.

Another common signal is SIGABRT, which happens when you program calls abort() or fails an assert. As with segfaults, if your code is failing asserts, then running it in gdb can be incredibly useful—you will get control of the program at the point where assert causes the program to abort, and (after going up a few frames back into your own code), see exactly what was going on when the problem happened.

The other signal that is useful is SIGINT, which happens when the program is interrupted—e.g., by the user pressing Control-c. If your program is getting stuck in an infinite loop, you can run it in gdb, and then after a while, press Control-c. You can then see where the program is, and what it is doing. You are not guaranteed to be in the right place (you may interrupt the program before it gets into the infinite loop), but if you wait sufficiently long, you will typically end up where you want. You can then see what is happening, and why you are not getting the behavior you want.

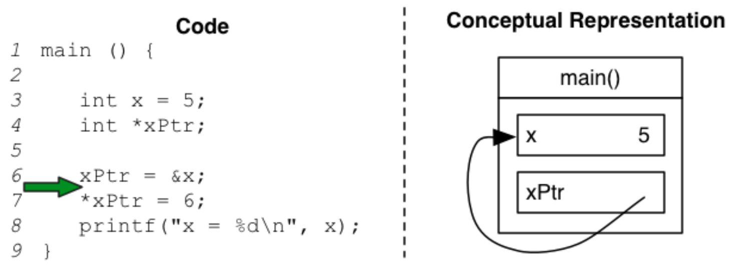
5 Pointers, Arrays, and Recursion

5.1 Pointers Conceptually

Pointers are way of referring to the *location* of a variable. Instead of storing a value like 5 or the character 'c', a pointer's value is the location of another variable. Later in this chapter we will discuss how this location is stored in hardware (i.e., as a number). Conceptually, you can think of the location as an arrow that points to another variable.

Declaring a pointer

In C, "pointer" (by itself) is not a type. It is a type constructor—a language construct which, when applied to another type, gives us a new type. In particular, adding a star (*) after any type names the type, which is a pointer to that type. For example, the code char * my_char_pointer; (pronounced "char star my char pointer") declares a variable with the name my_char_pointer with the type pointer to a char (a variable that points to a character). The declaration of the pointer tells you the name of the variable and type of variable that this pointer will be pointing to.



```
main () {

   int x = 5;
   int *xPtr;

   xPtr = &x;
   *xPtr = 6;
   printf("x = %d\n", x);
}
```

Figure 14: Pointer

Assigning to a pointer

As with all other variables, we can assign to pointers, changing their values. In the case of a pointer, changing its value means changing where it points—what its arrow points at. Just like other variables, we will want to initialize a pointer (giving it something to point at) before we use it for anything else. If we do not initialize a pointer before we use it, we have an arrow pointing at some random location in our program, and we will get bad behavior from our program—if we are lucky, it will crash.

The address-of operator can be applied to any lvalue (recall that an lvalue is an expression that "names a box") and evaluates to an arrow pointing at

```c
int x = 3;
int *p;
p = &x;
*p = 4; // poner 4 en x
int y = *p; // el valor de y es a donde apunta p que es 4.
int *q = &y; // Address of y
//Determina el valor de la caja adonde apunta q, que es y equivalente
// al valor de donde apunta p, el cual es x=4 + 1. Es decir, y=5;
*q = *p + 1;
// Cambia la flecha de q hacia donde apunta la flecha de p
// (caja de x)
q = p;
```

Listing 14: Pointer Example

the box named by the lvalue. The only kind of lvalue we have seen so far is a variable, which names its own box. Accordingly, for a variable x, the expression &x gives an arrow pointing at x's box. It is important to note that the address of a variable is not itself an lvalue, and thus not something that can be changed by the programmer. The code &x = 5; will not compile. **A programmer can access the location of a variable, but it is not possible to change the location of a variable.**

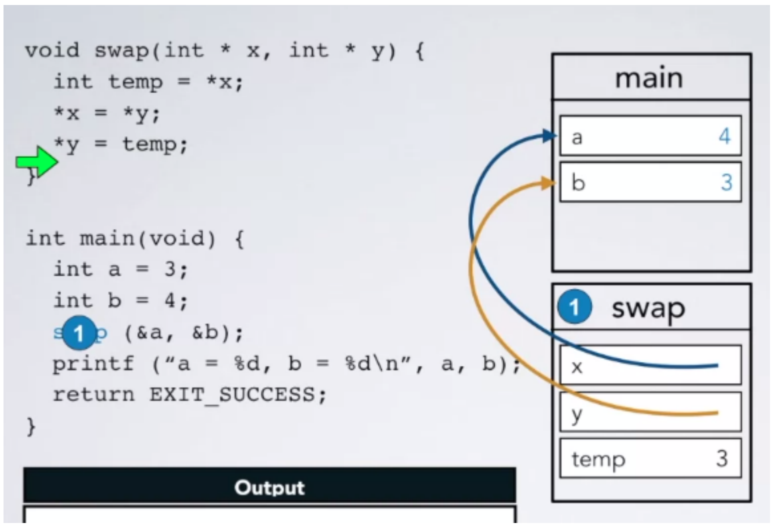
When you are writing code with pointers, carefully planning—by drawing pictures—is even more important in order to write correct code.

Assignment statements in C (x = y;) can be broken down into two parts. The left-hand side (here, x) is called the **lvalue**. This is the box that will have a new value placed inside it. Before this chapter, **lvalues** were simply variables (e.g., x or y). With the introduction of pointers, we can also use the a derferenced pointer as an **lvalue**—if p is a pointer, then *p is an lvalue.

The right-hand side (the y in the statement x = y;) is called the **rvalue**. This is the value that will be placed inside the box/lvalue. With the introduction of pointers, we add two new types of expressions that can appear in **rvalues**: the address of an **lvalue** (&x), and dereferencing a pointer (*p).

A nice explanation of pointers in in video pointers.mp4 in my local computer.

One important fact to consider is that pointers are transitive in the sense that if I declare a pointer in main, pass it to a function or copy it to a local variable and change its value then if the value is accessed in any code-location the updated value will appear.



```
void swap(int * x, int * y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int a = 3;
  int b = 4;
  swap (&a, &b);
  printf ("a = %d, b = %d\n", a, b);
  return EXIT_SUCCESS;
}
```

Figure 15: Corrected Swap

That's the reason why the output of Listing 15 is:

In f, *a = 3, b = 4 In g, x = 7, *y = 8 Back in f, *a = 7, b = 0 In main: x = 7, y = 4

B was previously set to 8 but in g it was changed to 0, so the answear, Back in f, reflects that change.

5.2 Pointers in Hardware

Pointers under the Hood

In a conceptual drawing, representing a pointer with an arrow is an effective way of showing what the pointer points to. From a technical perspective, however, an arrow does not make much sense. (How exactly does one represent an arrow in hardware? After all, everything is a number—and arrows do not seem like numbers.)

The mechanics of pointers make a little more sense when we look under the hood at the hardware representation. By now you are familiar with the idea that data is stored in your computer in bytes. Some data types, like characters, require 1 byte (8 bits), and some data types, like integers, require 4 bytes18 (32 bits). When we draw boxes for our variables, we do not necessarily think about how big the box is, but that information is implicit in the type of the variable.

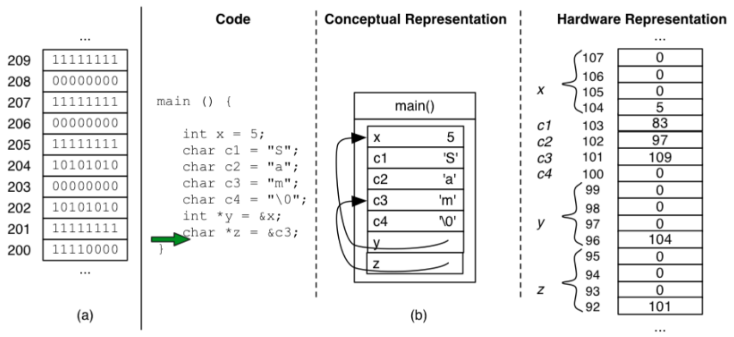
```c
#include <stdio.h>
#include <stdlib.h>

void g(int x, int * y) {
  // g(7, &b)
  printf("In g, x = %d, *y = %d\n", x, *y);
  x++; // x = 8
  *y = *y - x; // y = 0
  y = &x;  // y apunta a x que es 8
}

void f(int * a, int b) {
  printf("In f, *a = %d, b = %d\n", *a, b);
  *a += b; // x=7
  b *= 2;  // y = 8
  g(*a, &b);
  printf("Back in f, *a = %d, b = %d\n", *a, b);
}


int main(void) {
  int x = 3;
  int y = 4;
  f(&x, y);
  printf("In main: x = %d, y = %d\n", x, y);
  return EXIT_SUCCESS;
}
```

Listing 15: Pointers exercise



```c
main () {

    int x = 5;
    char c1 = "S";
    char c2 = "a";
    char c3 = "m";
    char c4 = "\0";
    int *y = &x;
    char *z = &c3;
}
```

Figure 16: Pointers in Hardware

Addressing

A computer keeps track of all of the data by storing it in memory. The hardware names each location in memory by a numeric address. As each different address refers to one byte of data, this type of storage is called byte-addressable memory. A visualization of what this looks like is shown in section (a) of the figure above. **Each box represents a byte** (1 byte = 8 bits) of memory, and each byte has an address shown immediately to the left of it. For example, the address 208 contains one byte of data, with the value 00000000.

Section (b) in the figure shows the code, conceptual representation, and hardware representation of the declaration and initialization of one 4-byte integer, four 1-byte characters, and finally two 4-byte pointers. Each variable has a base address. For example, the address of x is 104 and the address of c3 is 101. The variable x is small enough that it can be expressed within the first byte of its allocated space in memory. If it were larger (or simply negative), however, the bytes associated with addresses 105-107 would be non-zero. On a 32-bit machine, addresses are 32 bits in size. **Therefore pointers are always 4 bytes in size, regardless of the size of the data they point to.**

With this more concrete understanding of memory and addresses, the hardware representation of pointers becomes clear: pointers store the addresses of the variable they point to. The final variables, y and z, in (b) in the above figure show just that. The variable y is an integer pointer initialized to point to x. The conceptual drawing shows this as an arrow pointing to the box labeled x. The hardware drawing shows this as a variable that has the value 104, the base address of x. The variable z is declared as a pointer to a char. Although a character is only 1 byte, an address is 32 bits, and so z is 4 bytes in size and contains the value 101, the location of c3. (If these variables were located in memory locations with higher addresses, the numerical values of the addresses would be larger, and there would be non-zero values across all four bytes of the pointers y and z.)

Pointers are variables whose values are addresses. An important consequence of this fact is that a pointer can only point to addressable data. Expressions that do not correspond to a location in memory cannot be pointed to, and therefore a program cannot attempt to use the ampersand ("address of") operator for these expressions. For example, neither 3 nor (x+y) are expressions that represent an addressable "box" in memory. Consequently, lines like these are illegal: int *ptr = &3; and int *ptr = &(x + y);. Note that 3 and (x+y) are not lvalues—they do not name boxes, which is why they cannot have an address. The compiler would reject this code, and we would need to correct it—primarily by thinking carefully about what we were trying to do (drawing a picture). A corollary to this rule is that an assignment statement can only assign a variable that corresponds to a location in memory. So expressions such as 3 = 4; or x+y+z = 2 will also trigger a compiler error.

A Program's View of Memory

On a 32-bit machine, where addresses are 32 bits in size, the entire memory space begins at 0x00000000 (in hex, each 0 represents 4 bits of 0) and ends at 0xFFFFFFFF (recall that 0x indicates hexadecimal, and that each F represents four binary 1's). Every program has this entire address space at its disposal and there is a convention for how a program uses this range of addresses. the figure below depicts where in memory a program's various components are stored.

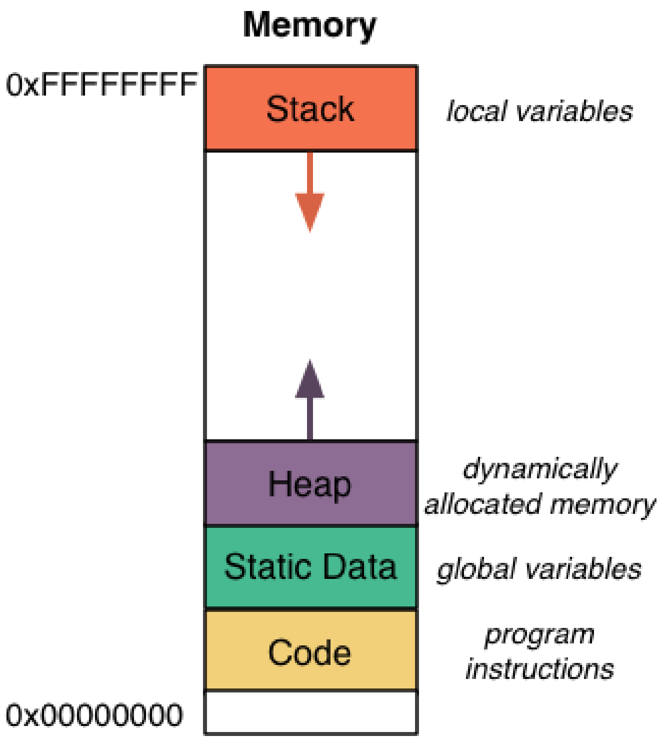


Figure 17: Memory

Code

Given the repeated claims that everything is a number it should come as no surprise that a program itself can be represented by a series of numbers.

Producing these numbers is the job of a compiler, which takes a program in a language like C and converts it into a bunch of numbers (called object code) which is readable by the computer. An instruction in C which adds two numbers together, for example, might be encoded as a 32-bit instruction in object code. Some of the 32 bits will tell the machine to perform addition, some of the bits will encode which two numbers to add, and some of the bits will tell the processor where it should store the computed sum. The compiler converts the C code into object code and also assigns each encoded instruction a location in memory. These encoded program instructions live in the Code portion of memory, shown in yellow in the figure above.

Static Data

The static data area contains variables that are accessible for the entire run of the program (e.g. global variables). Unlike a variable that is declared inside a function and is no longer accessible when the function returns **(But I do believe that pointers can)**, static variables are accessible until the entire program terminates (hence, the term static). Conceptually, a static variable's box remains "in the picture" for whole program, whereas other are usable for only a subset of the program's lifetime. These variables are placed in their own location in memory, just past the code portion of the program, shown in the figure above.

The final two sections of memory are for two different types of program data that are available at specific times during a program's execution. The Heap (in purple) stores dynamically allocated data. The Stack (in orange) stores the local variables declared by each function. The stack is divided into stack frames that are available from starting when the function is called, and last until it returns. The conceptual drawings throughout this book have primarily shown pictures of stack frames as boxes (one for each function) with local variables. In actuality, the stack is one contiguous piece of memory; each stack frame sits below the stack frame of the function that called it.

With these details about how pointers work and how code is placed in memory, we can show a more detailed view of what happens during the execution of our swap function. The next video shows the same swap function's execution, but with a contiguous view of the stack and memory addresses as the values of pointers. It also shows the code of the program stored in memory in 4-byte boxes. The return address field of the stack—previously depicted in our conceptual drawings as a blue call site location–is also made explicit in this video. The return address is the address of the instruction that should be executed next after the function being called completes and returns.

The calling convention —the specific details of how arguments are passed to and values returned from functions resembles an x86 machine; the arguments to a function reside in the stack frame of the function that called it. This is slightly different from the conceptual drawings we have shown previously, in which the arguments were placed in the stack frame of the function being called. For a conceptual drawing, this is both sufficient and easy to understand. Hardware details will differ slightly for every target architecture.

For a good example see pointers_in_hardware.mp4

NULL

At the bottom of the Figure 17, there is a blank space below the code segment, which is an invalid area of the program's memory. You might wonder why the code does not start at address 0, rather than wasting this space. The reason is that programs use a pointer with the numeric value of 0—which has the special name, NULL to mean "does not point at anything." By not having any valid portion of the program placed at (or near) address 0, we can be sure that nothing will be placed at address 0. This means no properly initialized pointer that actually points to something would ever have the value NULL.

Knowing that a pointer does not point at an actual thing is useful for a variety of reasons. For one, we may sometimes have algorithms which need to answer "there is no answer". For example, if we think about our closest point example discussed earlier when the set of points is empty, we must return "there is no answer"—with pointers, we can return a pointer to a point, and return NULL in the "there is no answer" case. We may also have functions whose job it is to create something that return NULL if they fail to do so we will see functions that open files (so we can read data from the disk) which fail in this manner. Later in the course we will see how to dynamically allocate memory, which also return NULL if the memory cannot be allocated.We will also begin to learn about linked structures which store data with pointers from one item to the next, and use NULL to indicate the end of the sequence.

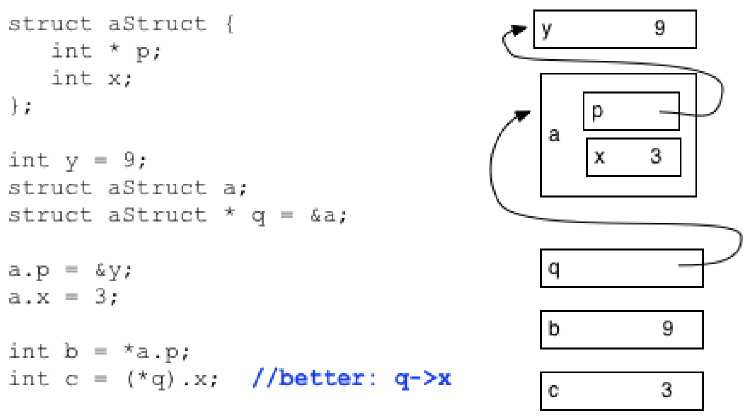
When we use NULL, we will represent it as an arrow with a flat head (indicating that it does not point at anything). Whenever we have NULL, we can use the pointer itself (which just has a numeric value of 0), however, we cannot follow the arrow (as it does not point at anything). If we attempt to follow the arrow (i.e., dereference the pointer), then our program will crash with a segmentation fault —an error indicating that we attempted to access memory in an invalid way (in fact, if our program tries to access any region of memory that is "blank" in the figure above, it will crash with a segmentation fault).

The NULL pointer has a special type that we have not seen yet— **void ***. A void pointer indicates a pointer to any type, and is compatible with any other pointer type—we can assign it to an **int ***, a **double***, or any other pointer type we want. Likewise, if we have a variable of type **void ***, we can assign any other pointer type to it. Because we do not know what a **void *** actually points at, we can neither dereference it (the compiler has no idea what type of thing it should find at the end of the arrow), nor do pointer arithmetic on it.

5.3 Pointers to Sophisticated Types

Pointers to Structs

In this figure note that *a.b dereferences the **b** field inside of struct a. Dereferencing q, then selecting field x requires either parenthesis, or the -¿ operator.



```
struct aStruct {
   int * p;
   int x;
};

int y = 9;
struct aStruct a;
struct aStruct * q = &a;

a.p = &y;
a.x = 3;

int b = *a.p;
int c = (*q).x;   //better: q->x
```

Figure 18: Pointers to Structs

When we have pointers to structs, we can just use the * and . operators that we have seen so far, however, the order of operations means that . happens first. If we write *a.b, it means *(a.b)—a should be a struct, which we look inside to find a field named b (which should be a pointer), and we dereference that pointer. If we have a pointer to a struct c, and we want to refer to the field d in the struct at the end of the arrow, we would need parenthesis, and write (*c).d (or the -¿ operator we will learn about momentarily).

In the figure above, we have a struct which has a field p (which is a pointer to an int), and a field x which is an int. We then declare y (an int), a (a struct), and q (a pointer to a struct), and initialize them. When we write *a.p, the order of operations is to evaluate a.p (which is an arrow pointing at y), then dereference that arrow. If we wrote *q.x, we would receive a compiler error, as q is not a struct, and the order of operations would say to do q.x first (which is not possible, since q is not a struct). We could write parenthesis, as in the figure ((*q).x).

However, pointers to structs are incredibly common, and the above syntax gets quite cumbersome, especially with pointers to structs which have pointers to structs, and so on. For (*q).x, it may not be so bad, but if we have (*(*(*q).r).s).t it becomes incredibly ugly, and confusing. Instead, we should use the -¿ operator, which is shorthand for dereferencing a pointer to a struct and selecting a field—that is, we could write q-¿x (which means exactly the same thing as (*q).x). For our more complex example, we could instead write q-¿r-¿s-¿t (which is easier to read and modify).

Pointers to Pointers

We can have pointers to pointers (or pointers to pointers to pointers...etc). For example, an **int**** is a pointer to a pointer to an int. An **int***** is a pointer to a pointer to a pointer to an int. We can have as many levels of "pointer" as we want (or need), however, the usefulness drops of quite quickly (**int**** is quite common, **int***** moderately common, but neither author has ever had a use for an **int*************). The rules for pointers to pointers are no different from anything else we have seen so far: the * operator dereferences a pointer (follows an arrow), and the & operator takes the address of something (gives an arrow pointing at that thing).

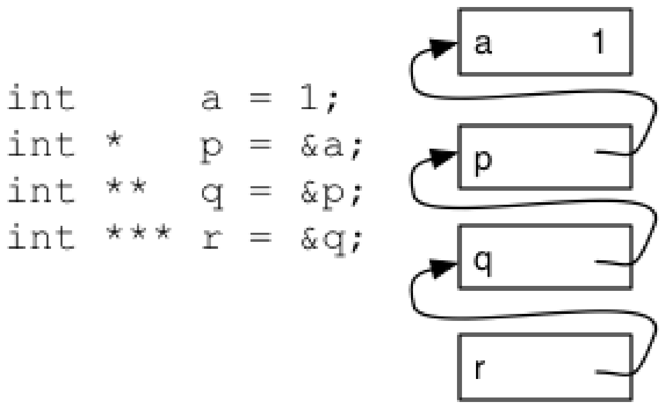
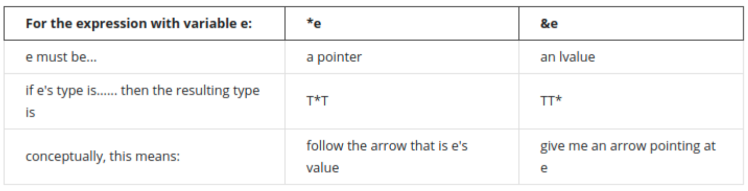

```
int       a = 1;
int *     p = &a;
int **    q = &p;
int ***   r = &q;
```

Figure 19: Pointers to Pointers

The Figure 19 illustrates pointers to pointers. Here we have 4 variables, a (which is an **int**), p (which is an **int***), q (which is an **int ****), and r (which is an **int*****). If we were to write *r, it would refer to q's box (because we would follow the arrow from r's box, and end up at q's box). We could write **r, which would refer to p's box—because *r is an arrow pointing at p, and the second * dereferences that pointer. Likewise, ***r would refer to a's box. It would be a compiler error to write ****r, because that would attempt to follow the arrow in a's box, which is not an arrow, but rather a number (sure, everything is a number—but our types have told the compiler that a is just a plain number, not a number that means an arrow).

You may wonder why we might want pointers to pointers. One answer to this question is "for all the same reasons we want pointers"— **a pointer gives us the ability to refer to the location of a thing, rather than to have a copy of that thing**. Anytime we have a variable that tells us the location of a thing, we can change the original thing through the pointer. Just as we might want to write swap for integers, we might also want to write swap for **int** *s (in which case, our swap function would take **int** **s as parameters).

Notice how the types work out (i.e. match up so that the compiler type check the program). Whenever we take the address of an lvalue of type T, we end up with a pointer of type T * (e.g., p has type **int** *, so &p has type **int** **). Whenever we take the address of something, we "add a star" to the type. This rule makes intuitive sense, because we have a pointer to whatever we had before. Whenever we dereference an expression of type T*, we end up with a value of type T—we "take a star off the type", because we followed the arrow.



For the expression with variable e:	*e	&e
e must be...	a pointer	an lvalue
if e's type is...... then the resulting type is	T*T	TT*
conceptually, this means:	follow the arrow that is e's value	give me an arrow pointing at e

Figure 20: Pointers to Pointers 2

The table above summarizes these rules about pointers. Note that * and & are inverse operations—if we write *&e or &*e, they both just result in e (whenever they are legal). The first would mean "give me an arrow pointing at e, then follow it back (to e)", while the second would mean "follow the arrow that is 's value, then give me an arrow pointing at wherever you ended up."

Const

As we discuss pointers to various types, it is good time to introduce the notion of const data—data which we tell the compiler we are not allowed to change. When we declare a variable, we can add const to its type to specify that the compiler should not allow us to change the data:

When we have pointers, there are two different things that can be const: the data that the pointer points at (what is in the box at the end of the arrow), or the pointer itself (where it points). If we write:

```
const int * p = &x;
```
[1]

We have declared p as a pointer to a const int—that is, p points at a int, and we are not allowed to change that int. We can change where p points (e.g., p = &y; is legal—if y is an int). However, changing the value in the box that p points at (e.g., *p = 4;) is illegal— **we have said that the int which p points at is const**. If we do try to write something like *p = 4;, we will get a compiler error like this:

```
assignment of read-only location '*p'
```

we can achieve exactly the same effect by writing:

```
int const * p = &x;
```

If we want to specify that we can change *p, but not p itself, we would write

```
int * const p = &x;
```
[2]

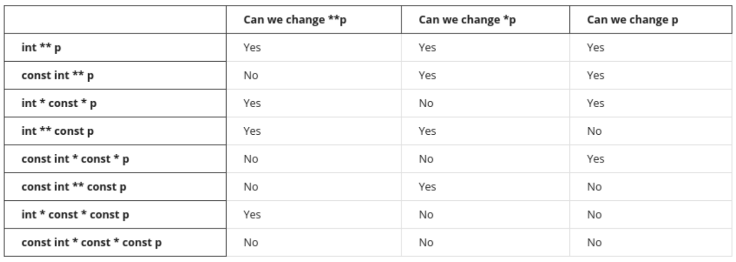
This declaration says that p is a const pointer to a (modifiable) int. Writing *p=4; would be legal, but writing p =&y; would be illegal. If we so desire, we can combine both to prevent changing either where the pointer points, or the value it points at:

```
const int * const p = &x;
```

Note that a declaration of const tells the compiler to give us an error only if we try to change the data through the variable declared as const, or perform an operation where the const-ness gets dropped. For example, the following is legal:

```
 int x = 3;
const int * p = &x;
x = 4;
```

[1] **The int which p points at would not change**
[2] **The arrow would not change**



	Can we change **p	Can we change *p	Can we change p
int ** p	Yes	Yes	Yes
const int ** p	No	Yes	Yes
int * const * p	Yes	No	Yes
int ** const p	Yes	Yes	No
const int * const * p	No	No	Yes
const int ** const p	No	Yes	No
int * const * const p	Yes	No	No
const int * const * const p	No	No	No

Figure 21: Const

Here, we are not allowed to change *p, however, the value we find at *p can still be changed by assigning to x (since x is not const, it is not an error to assign to it). However, if we write:

```
const int y = 3;
int * q = &y;
*q = 4;
```

then we will receive a compiler warning (which we should treat as an error):

initialization discards 'const' qualifier from pointer target type [enabled by default]

The error is on line 2, in which we assign &y (which has type const int *) to q (which has type int *)—discarding the const qualifier (const is called a qualifier because it modifies a type). This snippet of code is an error because *q=4; (on line 3) would be perfectly legal (q is not declared with the const qualifier on the type it points to), but would do something we have explicitly said we do not want to do: modify the value of y.

Novice programmers often express some confusion at the fact that the first example is legal and the second is not—in both cases, we have tried to declare a variable and a pointer (to that variable), with one const and the other not. We have then tried to modify the value in that box through whichever is not const—but one is ok, and the other is not. These rules do actually make sense: in the second case, we have said "y cannot be modified" then we try to say "q is a pointer (which I can use to modify or read a value) to y"—that clearly violates what we said about "y" (that it cannot be modified). In the first case, however, we are saying "x is a variable that can be modified" and then "p is a pointer, which we can only use to read the value it points at, not modify it"—this does not impose new (nor violate existing) restrictions on "x", only tells us what we can and cannot do with "p".

5.4 Aliasing and Arithmetic

Aliasing

In our discussion of pointers, we have alluded to the fact that we may now have multiple names for the same box; however, we have not explicitly discussed this fact. For example, in the figure below, we have four names for a's box: a, *p, **q, and ***r. Whenever we have more than one name for a box, we say that the names alias each other—that is, we might say *p aliases **q. We can see this in Figure 19; we have multiple names for the same box.

Aliasing plays a couple of important roles in our programming. First, when we are working through Step 2 of our programming process, we may find that we changed a value, but there are multiple ways we can name what value we changed. When we write down precisely what we did, it is crucial to think about which name we used to find the box when we worked the problem by hand in Step 1. If we are not sure, we should make note of the other names we might have used—then if we have trouble generalizing, we can consider whether we should have be using some other name instead.

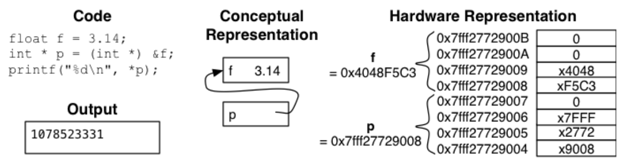
When we are debugging (or executing code by hand), it is also important to understand aliasing. Novice C programmers often express surprise and confusion at the fact that a variable changes its value without being directly assigned to: "I wrote x = 4;, then look I don't assign to x anywhere in this code, but now it is 47!" Generally such behavior indicates that you alias the variable in question (although you may not have meant to). If you have problems of this type, using watch commands in gdb can be incredibly helpful.

If we want to live dangerously, we can even have aliases with different types. Consider the following code:

```
float f = 3.14;
int * p = (int *) &f; Generally a bad idea
printf("%d", *p);
```

Here, f is an alias for *p, although f is declared to be a float, while p is declared to be a pointer to an int (so we have told the compiler that *p is an int). What will happen when you run this code? Your first reaction might be to say that it prints 3, the following (similar-ish) code would print 3:

```
float f = 3.14;
int x = (int) f;
printf("%d", x);
```



```
float f = 3.14;
int * p = (int *) &f;
printf("%d\n", *p);
```

```
1078523331
```

An example of aliasing with different types. Storing the floating point bit encoding for *3.14* in *f*, then reading it out as though it represented an integer.

Figure 22: Aliasing dif types

However, if we run the first code snippet on the computer, we will get 1078523331! This result may seem like the computer is just giving us random nonsense, however, what happened is perfectly logical (if a bit low level). When we cast a float to an int (in the second snippet of code), we ask the compiler to insert instructions which convert from a float to an int. However, when we dereference a pointer to an int, the compiler just generates instructions to read the bit pattern at that location in memory, as something that is already an int. In the first snippet of code, initializing float f = 3.14; writes the bit pattern of the floating point encoding of 3.14 into f's box. Without going into the details, the floating point encoding of 3.14 works out to 0100 0000 0100 1000 1111 0101 1100 0011 (= 0x4048F5C3 in hex, 1078523331 in decimal). When we dereference the pointer as an int, the program reads out that bit pattern, interprets it as though it were an integer, and prints 1078523331 as output, as the figure above illustrates.

This last example is not something you need to use in programs you write, but rather a caution against abusing pointers. Unless/until you understand exactly what is happening here and have a good reason to do it, you should not cast between pointer types.

Pointer Arithmetic

Like all types in C, pointers are variables with numerical values. The precocious programming student may wonder if the value of variables can be manipulated the way one can manipulate the value of integers and floating point numbers. The answer is a resounding Yes!

Consider the following code:

Lines 1–3 initialize the values of 3 variables of various types (integer, floating point number, and pointer-to-an-integer). Lines 5–7 add 1 to each of these variables. For each type, adding 1 has a different meaning. For x, adding 1 performs integer addition, resulting in the value 5. For y, adding 1 requires converting the 1 into a floating point number (1.0) and then performing floating point addition, resulting in 5.0. For both integers and floating point

```c
int x = 4;
float y = 4.0;
int *ptr = &x;

x = x + 1;
y = y + 1;
ptr = ptr + 1;
```

Listing 16: Pointer Arithmetic

numbers, adding 1 has the basic semantics of “one larger”. For the integer pointer ptr (which initially points to x), adding 1 has the semantics of “one integer later in memory”. Incrementing the pointer should have it point to the “next” integer in memory. In order to do this, the compiler emits instructions which actually add the number of bytes that an integer occupies in memory (e.g., +1 means to change the numerical value of the pointer by +4). Likewise, when adding N to a pointer to any type T, the compiler generates instructions which add (N * the number of bytes for values of type T) to the numeric value of the pointer—causing it to point N Ts further in memory. This is why pointer arithmetic does not work with pointers to void; since the compiler has no idea how big a “thing” is, it does not know how to do the math to move by N “things”.

The code we have written here is legal as far as the compiler is concerned, however, our use of pointer arithmetic is rather nonsensical in this context. In particular, we have no idea what box ptr actually points at when this snippet of code finishes. If we had another line of code that did *ptr = 3;, the code would still compile, but would have undefined behavior — we could not execute it by hand and say with certainty what happens. Specifically, when ptr = &x, it is pointing at one box (for an integer) which is all by itself—it is not part of some sequence of multiple integer boxes (which we will see shortly). *Incrementing the pointer will point it at some location in memory, we just do not know what. It could be the box for y, the return address of the function, or even the box for ptr itself.*

The fact that we do not know what happens here is not simply a matter of the fact that we have not learned what happens— *it is a matter of the fact that the rules of C give the compiler a lot of freedom in how it lays out the variables in memory. One compiler may place one piece of data after x, while another may place some other data after x.* In fact, the same compiler may change how it arranges variables in memory when given different command line options changing its optimization levels.

We will consider all code with undefined behavior, such as this, to be erroneous. Accordingly, if you were to execute this code by hand, when you perform ptr = ptr + 1;, you should change the value of ptr to just be a note that it is &x+1, and not any valid arrow. If you then dereference a pointer which does not validly point at a box, you should stop, declare your code broken, and go fix it. **We will note that simply performing arithmetic on pointers such that they do not point to a valid box is not, by itself, an error—only dereferencing the pointer while we do not know what it points at is the problem.** We could perform ptr = ptr - 1; right after this code, and know with certainty that ptr points at x again. We might also just go past a valid sequence of boxes at the end of a loop, but not use the invalid pointer value. We will generally not do these things, and you should know the difference between what is and is not acceptable coding practice.

Use Memory Checker Tools

Now that you are starting to use pointers, it is crucial to use memory checker tools, such as valgrind and/or the compiler option -fsanitize=address. These will help you find erroneous behavior, and make fixing your program easier. Use them all throughout the testing process.

5.5 Array Basics

Array Declaration and Initialization

An array is a sequence of items of the same type (e.g., an array of ints is a sequence of ints). When an array is created, the programmer specifies its size (i.e., the number of items in the sequence)—so we might make an array of 26 ints for our frequency counting example. If we wanted to declare an array of 4 ints called myArray, we would write:

```
int myArray[4];
```

When we make this declaration, we end up with a slightly different semantics than we are used to. The variable name (in this case, myArray) is a pointer to the 4 boxes that make up the array. Unlike other variables, myArray is not an lvalue—we cannot change where it points. Instead, it just names a pointer to the first box in the array.

The Figure 23 depicts an array of 4 ints, resulting from the declaration of myArray. Notice that there are 4 boxes (which are not yet initialized), and that myArray does not have a box—its is just a name for an arrow pointing

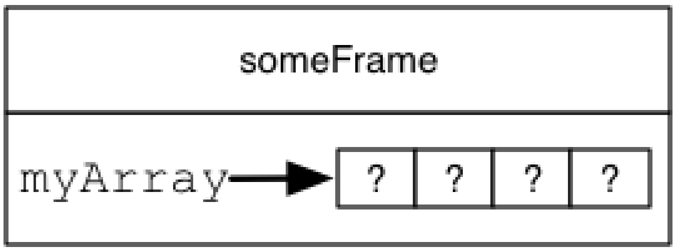


Figure 23: Array

at the first box.

he figure above illustrates the effects of this declaration. We have created the array (which has 4 uninitialized boxes for ints), and myArray names a pointer to it. Unlike other variables, we draw myArray without a box of its own, to underscore the fact that myArray is not an **lvalue**—we cannot change it. As with other variables, we can initialize an array in the same line that we declare it. However, as the array holds multiple values, we write the initialization data inside of curly braces.

```
int myArray[4] = {42, 39, 16, 7 };
```

If you write too few elements, the compiler will silently fill the remaining ones in with 0. We can write just a single 0 in the curly braces and the compiler will fill in as many zeros as there are elements of the array.

```
int myArray[4] = {0};
```

If we provide an initializer, we can also omit the array size, and instead write empty square brackets—the compiler will count the elements of the initializer, and fill in the array size:

```
int myArray[] = {42, 39, 1, 7 };
```

We'll also note that you can use a similar syntax to initialize structs. For example, the following will initialize the first field of p to 3 and the second field to 4:

```
point p = {3, 4};
```

However, for structs, this form of initialization is very brittle–if you add another field to the struct before or in between these two, you will no longer be initializing the fields the way you intend. In C99 (e.g., what you get

```
point myPoints[] = { {.x = 3, .y = 4},
                     {.x = 5, .y = 7},
                     {.x = 9, .y = 2} };
```

Listing 17: Composability

when you compile with -std=gnu99), you can designate which element you are initializing:

```
point p = { .x = 3, .y = 4};
```

Now, if another field is added to the struct, it will be zero-initialized, and the x and y fields will still be initialized properly.

If you are initializing an array of structs, these two techniques work particularly well together (an example of composability). For example, we could declare and initialize an array of three points as shown in Listing 17

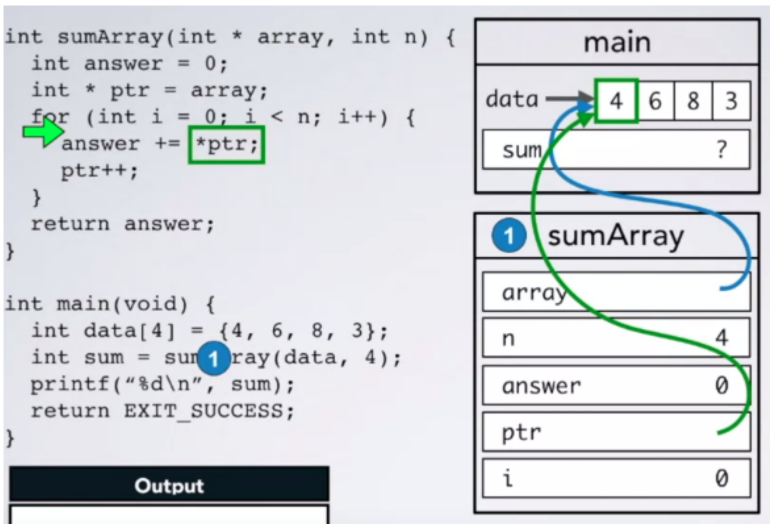
Accessing an Array

We can access the elements of an array in a couple different ways (which are actually the same under the hood!). We have already learned that the name of the array is a pointer to its first element, that we can do arithmetic on pointers, and that we can dereference pointers to access the data at the end of the arrow. We can put these concepts together to see one way that we could access elements in the array.

Accessing an array element using pointer arithmetic works fine, and sometimes is the natural way to access elements. However, sometimes we just want the nth element of an array, and it would be cumbersome to declare a pointer, add n to it, then dereference it. We can accomplish this goal more succinctly by indexing the array. When we index an array, we write the name of the array, followed by square brackets containing the number of the element we want to refer to: e.g., myArray[3]. Indexing an array names the specific box within the sequence, and can be used as either an lvalue or an rvalue.

We will note that accessing an array out of bounds (at any element that does not exist) is an error that the compiler cannot detect. If you write such code, your program will access some box, but you do not know what box it actually is. This behavior is much the same as the erroneous behavior we discussed when we talked about pointer arithmetic. In fact, pointer arithmetic and array indexing are exactly the same under the hood, the compiler turns myArray[i] into *(myArray + i) (This definition leads to a “stupid C trick” that you can use to perplex your friends: i[myArray] looks ridiculous, but is perfectly legal. Why? i[myArray] = *(i + myArray). Since addition is commutative, that is the same as *(myArray + i), which is the same as myArray[i]. This trivia is completely useless beyond amazing your friends at parties and winning bets with people who know a little C, but not this fact.).

Note that a consequence of this rule is that if we take &myArray[i], it is equivalent to &*(myArray +i), and the & and * cancel (as we learned previously, they are inverse operators), so it is just myArray + i. This result is fortunate, as it lines up with what we would hope for: &myArray[i] says “give me an arrow pointing at the ith box of myArray” while myArray + i says “give me a pointer i boxes after where myArray points”—these are two different ways to describe the same thing.



```
int sumArray(int * array, int n) {
  int answer = 0;
  int * ptr = array;
  for (int i = 0; i < n; i++) {
    answer += *ptr;
    ptr++;
  }
  return answer;
}

int main(void) {
  int data[4] = {4, 6, 8, 3};
  int sum = sumArray(data, 4);
  printf("%d\n", sum);
  return EXIT_SUCCESS;
}
```

Figure 24: Sum Array

In Figure 24 data is a pointer to the beggining of the array so sumArray has as input a pointer. This is the same in the line `int * ptr = array;`, we are declaring a pointer equals to a pointer.

When we do `ptr++;` we are doing pointer arithmetic to next element in the array. The problem here could be that after the last element in the array when we increment ptr we go beyond the array. This is a problem only if we dereference the pointer. Here is not a problem because we are going to get the answer in sum and destroy sumArray’s frame.

Array Access with Pointer Indexing

```c
int sumArray(int * array, int n){
  int answer = 0;
  for (int i = 0; i < n; i++){
    answer += array[i];
  }
  return answer;
}

int main(void){
  int data[4] = {4, 6, 8,10};
  int sum = sumArray(data, 4);
  printf("%d\n", sum);
  return EXIT_SUCCESS;
}
```

Listing 18: Array Access with Pointer Indexing

Another way to make this is by:

5.6 Arrays in Action

Passing Arrays as Parameters

In general, when we want to pass an array as a parameter, we will want to pass a pointer to the array, as well as an integer specifying how many elements are in the array.

There is no way to get the size of an array in C, other than passing along that information explicitly, and we often want to make functions which are generic in the size of the array they can operate on (i.e., we do not want to hardcode a function to only work on an array of a particular size). If we wanted, we could make a struct which puts the array and its size together, as one piece of data—then pass that struct around.

When we pass a pointer that actually points at an array, we can index it like an array (because it is an array—remember the name of an array variable is just a pointer), and perform pointer arithmetic on it. Such pointer arithmetic will be well defined, as long as the resulting pointer remains within the bounds of the array, **as we are guaranteed that the array elements are sequential in memory.**

We can also pass an array as a parameter with the square bracket syntax:

`int myFunction(int myArray[], int size){}`

This definition is functionally equivalent to `int myFunction(int * myArray; int size){}` with the pointer syntax. Some people prefer it, as it indicates more explicitly that myArray is intended to be an array. We can also write a size in the, however, the compiler will not make any attempt to check if that size is actually correct, thus it is easy to write something incorrect there (or something that becomes incorrect as you change your code), and may confuse a reader.

When you call a function that takes an array, you can pass any expression which evaluates to a pointer to a sequence of elements. Typically, this expression is just the name of the array (which is a pointer to the first element). However, we could perform pointer arithmetic on an array (to get a pointer to an element in the middle of it—which is a valid, but shorter array), or we might retrieve the pointer out of some other variable—it might be a field in a struct, or even an element in another array.

Writing Code with Arrays

When we write code with arrays, we need to look for patterns in where we access the array. A common pattern is to access each element of the array in order (the 0th, then the 1st, etc.). Such a pattern generally lends itself naturally to counting over the elements of the array with a for loop. However, we might have other patterns—as always, we should work the problem ourselves, write down what we did, and look for the patterns.

If we have complicated data (e.g., arrays of structs which have arrays inside them) it is very important to think carefully about how we can name the particular box we want to manipulate. For problems with complex structured data, drawing a diagram with the appropriate pointers and arrays can be an important aspect of working an example of the problem ourselves (Step 1).

If you have $[x] - - > [a] - - > [b]$ and we write

- x = ... : Then you change where x points
- *x = ... : Then you change where a points
- **x = ... : Then you change b

So, when we write *x=p; we're setting *x (in the example, a) to have the same value as p. Since p is ap ointer, its value is an arrow pointing somewhere. So *x will now point at the same place p is pointing. So again, if you have:

$[x] - - > [a] - - > [b]$ and $[p] - - > [c]$ and do *x = p; you end up with x's value being an arrow pointing at [c].

5.7 Array Caveats

Dangling Pointers

When you write code with arrays, you may be tempted to return an array from a function (after all, it is natural to solve problems where an array is your answer). However, we must be careful, because the storage space for the arrays we have created in this chapter live in the stack frame, and thus are deallocated after the function returns. The value of the expression that names the array is just a pointer to that space, so all that gets copied to the calling function is an arrow pointing at something that no longer exists. Whenever you have a pointer to something whose memory has been deallocated, it is called a **dangling pointer**.

Dereferencing a dangling pointer results in undefined behavior (and thus represents a serious problem with your code) because you have no idea what values are at the end of the arrow.

If we were to try to compile the code in the video, the compiler would warn us that our code is broken if we used this problematic behavior:

```
warning:  function returns address of
local variable [-Wreturn-local-addr]
return myArray;
```

However, you should understand why the compiler is giving this warning, and never write code which exhibits this bad behavior, rather than relying on the compiler to find it.

In Listing 19 gcc produces no warning, even though the code is equally broken. The returned pointer is to the first element of the array, not to the full array. But in the process of returning from this function, that frame goes away and the others elements in the array are not there in the stack. **So the pointer points to no box**

One thing to be particularly wary of with respect to dangling pointers is that sometimes you can get away with using them without observing any problematic effects—your code is still broken, it just does not look (or even behave) broken. Having the code seem fine is particularly dangerous for two

```c
// Still broken
int * initArray(int howLarge) {
  int myArray[howLarge];
  for (int i = 0; i < howLarge; i++) {
    myArray[i] = i;
  }
  int * p = myArray;
  return p;
}
```

Listing 19: Broken Code Returning Pointer

reasons. **First**, when there is a problem in our code, we want to find it. We want to fix it, so that our code is correct, and we do not have danger lurking inside. **Second**, the "but I did that before and it was fine" effect is dangerous to novices—if you learn that something is ok, you keep doing it. You do not want to form bad habits.

To understand why you only see problems sometimes, remember that a value stored in memory will remain there until changed by the program—and a deallocated memory location may not be reused immediately. Once a function returns and its frame is destroyed, the memory locations that were part of that frame will not be reused until more values must be placed on the stack. If we call another function, its frame will be placed in the next available stack slots, overwriting the recently deallocated memory. Only at this point will the values associated with the deallocated stack frame change (due to assignments to variables in the new frame). Now writing to memory through the dangling pointer will change variables in that frame in unpredictable ways. Note that is not safe to use a dangling pointer into a deallocated frame even when you have not called another function—even though most stack allocations will come from calling a function, there is nothing to guarantee that those are the only way that the stack is used.

Array Size

In various examples so far, we have represented the size and indices of an array with an int—a signed integer, meaning it can hold positive or negative values. If we think very carefully about this situation, we might wonder why we use a signed int—a negative array index would not be legal, as there is not myArray[-1] (this would be out of bounds). This analysis suggests that we should use an unsigned int. However, while we are thinking very carefully about what type we might use, we may as well ask "what is the most correct type to use?" In particular, there are a variety of sizes of

```c
point * closestPoint (point * s, size_t n, point p) {
  if (n == 0) {
    return NULL;
  }
  double bestDistance = computeDistance(s[0],p);
  point * bestChoice = &s[0];
  for (size_t i = 1; i < n ; i++) {
    double currentDistance = computeDistance(s[i],p);
    if (currentDistance < bestDistance) {
      bestChoice = &s[i];
      bestDistance = currentDistance;
    }
  }
  return bestChoice;
}
```

Listing 20: Array Size

unsigned integers—do we want 8 bits? 16 bits? 32 bits? 64 bits? For a four element array, any of these will work just fine. However, we might want to write a more general array manipulation function which works with any size of array. In that case, how large of an unsigned int would we want to use to describe the size of the array and/or index it?

In C, the number of bits we would need varies from one platform to another—on a 32-bit platform (meaning memory addresses are 32 bits), we would want a 32-bit unsigned int; on a 64-bit platform we would want a 64 bit unsigned int. Fortunately, the designers of C realized this possibility, and decided to make a type name for "unsigned integers that describe the size of things"—size_t. Whenever you see size_t, you should think "unsigned int with the right number of bits to describe the size or index of an array." For example, our closestPoint function would be slightly more correct if we wrote:

What does "slightly more correct" mean? Practically speaking, it would only make a difference for very large arrays—ones whose size can be represented by a size_t, but not an int (e.g., typically more than 2 billion elements). Such situations will not come up in any of the problems you will do in this book, nor many situations you are likely to encounter as a novice programmer.

While we are discussing the sizes of data, it is a good time to introduce the sizeof operator. Recall that you do not actually know how many bytes an int or a pointer is, as their actual sizes can vary from one platform to another. Instead of writing a numerical constant that represents the size on one platform, you should let the C compiler calculate the size of a type for you with the sizeof operator. The sizeof operator takes one operand, which can either be a type name (e.g., sizeof(double)) or an expression (e.g., sizeof(*p)). If the operand is an expression, the compiler figures out the type of that expression (remember expressions have types), and evaluates the size of that type.

5.8 Strings

5.8.1 String Literals

Now, that we understand pointers, we can understand their type: const char *, that is, a pointer to characters which cannot be modified (recall that here, the const modifies the chars that are pointed-to).

That is, if we wanted to have a variable which points to a string literal, we might write Figure 25:

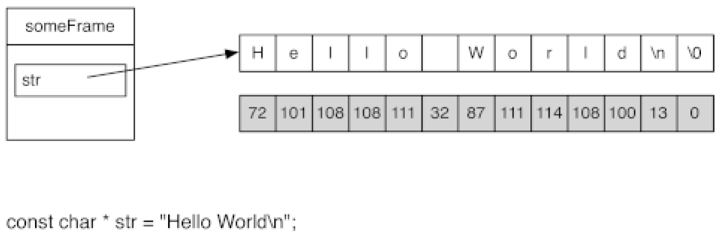

someFrame
str
H e l l o W o r l d \n \0
72 101 108 108 111 32 87 111 114 108 100 13 0
const char * str = "Hello World\n";

Figure 25: String Literal

5.9 Multidimentional Arrays

5.10 Function Pointers

Sorting Functions

Another example of using function pointers as parameters is a generic sorting function—one that can sort any type of data. Sorting an array is the process of arranging the elements of that array into increasing (or decreasing) order. Sorting an array is a common task in programs, as sorted data can be accessed more efficiently than unsorted data. We will formalize this notion

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Listing 21: qsort

of efficiency later, but for now, imagine trying to find a book in a library where the books are arranged alphabetically (i.e., sorted) versus in one where they are stored in no particular order.

As we will see when we learn more about sorting, there are many different sorting algorithms, but none of them care about the specific type of data, just whether one piece of data is "less than," "equal to," or "greater than" another piece of data. Correspondingly, we could make a generic sorting function—one that can sort an array of any type of data—by having it take a parameter which is a pointer to a function which compares two elements of the array. In fact, the C library provides such a function (which sorts in ascending order—smallest to largest):

The final parameter is the one we are most interested in for this discussion—compar is a pointer to a function which takes two `const void *s` and returns a `int` . Here, the const void *s point at the two elements to be compared (they are const since the comparison function should not modify the array). The function returns a positive number if the first pointer points at something greater than what the second pointer points at, 0 if they point at equal things, and a negative number for less than.

5.11 Recursion

1. Particular instance of the problem
2. Write down what we just did
3. Generalize
4. Test
5. Translate to code

Is Recursion Slow?

Another way to fix the performance problem without changing the underlying algorithm is **memoization**—keeping a table of values that we have already computed, and checking that table to see if we already know the answer before we recompute something. We will learn later how to efficiently store and lookup data, which would make a **memoized**version of the function quite fast.

5.12 Tail Recursion

The recursive functions we have seen so far use head recursion—they perform computation after they make a recursive call. In the factorial example, after the recursive call to factorial, the returned result is multiplied by n before that answer is returned. There is another form of recursion, called tail recursion. In tail recursion, the recursive call is the last thing the function does before returning. That is, for a tail recursive function f, the only recursive call will be found in the context of return f (...); —as soon as the recursive call to f returns, this function immediately returns its result without any further computation.

A generalization of this idea (separate from recursion) is a tail call—a function call is a tail call if the caller returns immediately after the called function returns, without further computation. Put another way: if function f calls function g, then the call to g is a tail call if the only thing f does after g returns is immediately return g's return value (or if f's return type is void, to just return). These two concepts tie together in that a recursive function is tail recursive if-and-only-if its recursive call is a tail call. To understand this definition, let us look at a tail recursive implementation of the factorial function:

Here we have a tail recursive helper function—a function that our primary function calls to do much of the work. Helper functions are quite common with tail recursion, but may appear in other contexts as well. The helper function takes an additional parameter for the answer that it builds up as it recurses. The primary function just calls this function passing in n and 1. Notice how the recursive call is a tail call. All that happens after that call returns is that its result is returned immediately. Note that the tail recursive version of factorial does no less work than the original recursive version. In the original recursive version, the factorial of a number is created by a series of multiplications that occur after each recursive call. In the tail recursive version, this multiplication takes place prior to the recursive call. The running product is stored in the second parameter to the tail recursive function.

The compiler will recognize that this is not an optimal implementation so it will perform **tail recursion elimination** where a stack will not be created

```c
int factorial_helper (int n, int ans) {
  //base case
  if (n <= 0) {
    return ans;
  }
  //recursive call is a tail call
  //after recursive call returns, just return its answer
  return factorial_helper (n - 1, ans * n);
}

int factorial (int n) {
  return factorial_helper (n, 1); //tail call
}
```

Listing 22: Tail Recursion

```c
int factorial (int n) {
  int ans = 1;
  while (n > 0) {
    ans = ans * n;
    n--;
  }
  return ans;
}
```

Listing 23: Iteration and Tail Recursion

for each recursive call to *fact_helper*

5.13 Equivalence of Tail Recursion and Iteration

Let us consider the following iterative implementation of the factorial function:

Executing both this iterative implementation, and the tail recursive implementation that you just saw by hand shows that they behave pretty much the same way. With the tail recursion elimination optimization that saw at the end of the previous video, the tail recursive implementation does not even create any new frames (the same as the iterative implementation)—it just re-uses the existing frame.

We underscore this similarity to help you understand an important idea: tail recursion and iteration are equivalent. Any algorithm we can write with iteration, we can trivially transform into tail recursion, and vice-versa. A smart compiler will compile tail recursive code and iterative code into the exact same instructions.

The inverse is also true—we can convert tail recursion to a while loop by doing the reverse.

The equivalence of tail recursion and iteration is especially important for functional programming languages. In a purely functional language, you cannot actually modify a value once you create it (at least not from the standpoint of the language: the compiler is free to re-use the same "box" if it can conclude that you will never use it again). As such, there are not loops (which typically require modifying variables to change the conditions), but rather only recursive functions. What you would typically write as a loop, you instead just write as a tail recursive function.

Functional programming is a beautiful thing, and we highly recommend that any serious aspiring programmer become fluent in at least one functional language. One of the authors particularly loves SML, and highly recommends it. Even if your main line of programming work is in some other language (such as C, Java, C++, or Python), having experience in a functional language helps you think about problems in different ways and makes you a better programmer all around. Of course, you may also end up using a functional language for your primary work.

6 Interacting with the System and Managing Memory

6.1 The Operating System

Most interesting interactions with "the world"—reading input from the user, writing a file on disk, sending data over a network, etc.—require access to hardware devices (the keyboard, the disk drive, the network card), in ways that normal programs cannot perform themselves. One key aspect of this issue is that "normal" programs cannot be trusted to access hardware directly. If your program could read the disk directly, then it could ignore the permissions system in place to protect different users' files, and read or write any data it wanted. Furthermore, an error in a program that can access the disk directly could corrupt the entire file system, destroying all data on the system.

Instead, the program asks the operating system (OS)—low-level software responsible for managing all of the resources on the system for all programs—to access hardware on its behalf. The program makes this request via a system call—a special type of function call that transfers control from the program into the operating system. The OS checks that the program's request is within the bounds of its permissions before performing it, which is how the system enforces security rules, such as file permissions. If the request is not permissible, the OS can return an error to the program. If, on the other hand, the request is fine, then the OS performs the underlying hardware operations to make it happen, and then returns the result to the program.

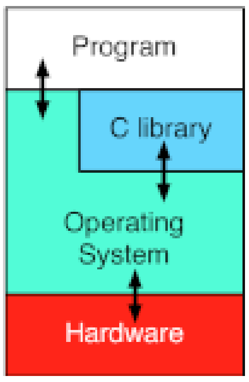


Figure 26: OS

The Figure 26 depicts the conceptual relationship between the program, C library, operating system, and hardware. While your C code can make system calls directly, it is more common to use functions in the C library. The C library functions then make system calls as they need. The OS then interacts with the hardware appropriately.

For example, suppose you call printf to write a string to standard output. The printf function is part of the C library, and involves significant code that does not require system calls: it must scan the format string for % signs, and perform the appropriate format conversions, building up the actual string to print. At some point, however, printf must actually write the resulting string out to standard output, which requires a system call. To perform this task, printf uses the write system call.

Novice programmers are often imprecise about the difference between a system call (which transfers control into the OS, requesting it to perform a task), and a library call (which calls a function found in a library, such as the C standard library). For example, calling printf a "system call" is technically incorrect, though most people will understand what you mean.

```c
int x = someSystemCall();
if (x != 0) {
  printf("someSystemCall() failed!\n");
  perror("The error was: ");
}
```

Listing 24: Broken code: errno

Being precise with this distinction is useful for two reasons. First, the more precise you are with your terminology, the more knowledgeable you come across during interviews (if you are interviewing for a programming-related job). Second, if you need to look a function up in the man pages, knowing whether it is a system call, or part of the C library tells you which section to look in—system calls are found in section 2, while C library functions are found in section 3.

6.1.1 Errors from System Calls

System calls can fail in a variety of ways. For example, you may try to read a file that does not exist or you do not have permissions for. You might also try to connect to a remote computer across the network at an address that is invalid or unreachable (the network or remote computer is down). Whenever these system calls fail in C, they (or technically their wrapper in the C library) set a global variable called errno (which stands for ”error number”).

In the particular case of **errno** , it is set by a failing call, and read if your program wants more information about why the call failed. If you want to check if a specific error occurred, you can compare it against the various constants which are defined in **errno.h**.

You may also wish to print out a message describing the error for the user. Just printing the numeric value of **errno** is not usually useful (Do you know what error 2 means?). Fortunately, the C library has a function **perror**, which prints a descriptive error message based on the current value of **errno**. The **perror**function takes one argument, a string that it prints before its descriptive message.

Note that since errno is global (there is one for the entire program), you must take care not to call anything that might change it before you test it or call perror.

Here, printf might change errno (it makes system calls), so we may not have a correct description of the error from perror.

6.2 Command Line Arguments

We can write our programs so that they can examine their command line arguments. To do so, we must write main with a slightly different signature:

```
int main(int argc, char ** argv){}
```

Here we see that main now takes two arguments. The first, int argc is the count of how many command line arguments were passed in (it stands for argument count). The second is an array of strings, which contains the arguments that were passed in (it stands for argument vector). The 0th element of argv (that is, argv[0]) contains the name of the program as it was invoked on the command line (so if you wrote ./a.out, then argv[0] is the sequence of characters "./a.out\0").

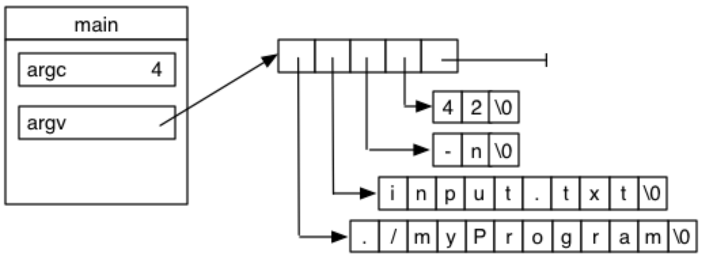


Figure 27: Command Line Arguments

The Environment Pointer

While much less commonly used than the command line arguments, main can potentially take a third argument: char ** envp, which is a pointer to an array of strings containing the values of environment variables. If your program needs to inspect its environment variables, you can include this third parameter, and access this array. If you do so, the elements of the array are strings of the form variable=value (e.g., PATH=/bin:/usr/bin). You can also access the environment variables with the functions getenv, setenv, putenv, and unsetenv. See their man pages for details.

Process Creation

While we do not want to delve into too many details of process creation, we do want to briefly mention a few points to assure you that there is no magic involved in making new programs, or getting the command line arguments to main. When the command shell (or any other program—the command shell is just a normal program) wants to run another program, it makes a couple of system calls. First, it makes a call to create a new process (fork). This new process (which is an identical copy of the original, distinguishable only by the return value of the fork system call) then makes another system call (execve) to replace its running program with the requested program. The execve system call takes an argument specifying the file with the binary (or script) to run, a second argument specifying the values to pass the new program as argv (which must end with a NULL), and a third argument specifying the values to pass for envp (even if main ignores envp, these are still passed to the new program so they can be accessed by the various environment manipulation functions mentioned in the previous subsection.)

When the OS executes the execve system call, it destroys the currently running program (the system call only returns on an error), and loads the specified executable binary into memory. It writes the values of argv and envp into memory in a pre-agreed upon format (part of the ABI—application binary interface: the contract between the OS and programs about how things work). The kernel then sets the execution arrow to a starting location specified in the executable binary (On Linux with gcc, the entry point is a symbol called _start—but the details are platform specific).

This startup code (which resides in an object file that is linked with any C program you compile—unless you request explicitly for it not to be) then calls various functions which initialize the C library. This startup code also counts the elements of argv to compute the value of argc and eventually calls main. Regardless of how main is declared, it always passes in argc, argv, and envp—if main is declared with fewer arguments, it still receives them but simply ignores them.

When main returns, it—like all other functions—returns to the function that called it. In the case of main, the caller is this startup code. This code then performs any cleanup required by the C library, and calls exit (which quits the program), passing in the return value of main as the argument of exit—which specifies the exit status of the program.

The shell (or other program that ran the program in question) can make a system call, which waits for its ”child” process(es) to exit, and then collects their return values.

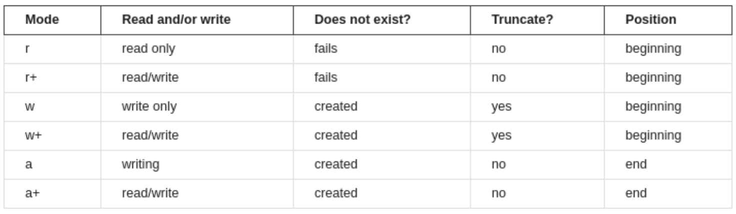
6.3 Opening and Reading Files

The first thing we must do to access a file (whether for reading or writing) is open it. Opening a file results in a stream associated with that file. A stream (which is represented by a FILE * in C) is a sequence of data (in this case chars), which can be read and/or written. The stream has a current position which may typically advances whenever a read or write operation is performed on the stream (and may or may not be arbitrarily movable by the program, depending on what the stream is associated with).

Typically, you will open a file with the fopen function, which has the following prototype:

```
FILE * fopen(const char * filename, const char * mode);
```

It is possible for fopen to fail—for example, the file you requested may not exist, or you may not have the permissions required to access it. Whenever fopen fails, it returns NULL (and sets errno appropriately). You should always check the return value of fopen to see if it is NULL or a valid stream before you attempt to use the stream.



Mode	Read and/or write	Does not exist?	Truncate?	Position
r	read only	fails	no	beginning
r+	read/write	fails	no	beginning
w	write only	created	yes	beginning
w+	read/write	created	yes	beginning
a	writing	created	no	end
a+	read/write	created	no	end

Figure 28: Possible modes for fopen

Reading a File

Once we have our file open, we might want to read input from it. We typically will use one of three functions to do so: **fgetc**, **fgets**, or **fread**. Of these, fgetc is useful when you want to read one character (e.g., letter) at a time. This function has the following prototype:

```
int fgetc(FILE * stream);
```

While you might expect this function to return a char, it returns an int so that it can return all possible chars, plus a distinct value to indicate that there are no more characters available in the stream—that the end of the file (EOF) has been reached. The value for end-of-file is defined as the constant EOF in stdio.h. Note that reading the character advances the current position in the stream.

```
//broken
FILE * f = fopen(inputfilename, "r");
if (f == NULL) { /* error handling code omitted */ }
while (fgetc(f) != EOF) {
  char c = fgetc(f);
  printf("%c",c);
}
//...other code...
```

Listing 25: Broken fgetc

```
//fixed
FILE * f = fopen(inputfilename, "r");
if (f == NULL) { /* error handling code omitted */ }
int c;
while ( (c=fgetc(f)) != EOF ) {
  printf("%c",c);
}
//...other code...
```

Listing 26: Fixed fgetc

The fact that function which read from streams advance the position of the stream poses a minor annoyance when writing a loop. Consider the following (broken) code which attempts to print every character from an input file:

This code will read one character, check if it is the end of the file, then read a different character, and print it. This code will actually print every other character from the input file, plus possibly something spurious at the end (if there are an odd number of characters in the file).

The cleanest way to re-structure the loop is to exploit the fact that an assignment is also an expression which evaluates to the value that is assigned. While that may sound like a technically complex mouthful, what it means is that x = 3 is not only an assignment of 3 to x, but also an expression which evaluates to 3. We could therefore write y = x = 3 to assign 3 to both x and y—however, we typically do not do so, as it makes the code less clear than writing two assignment statements. In this particular case, however, it is OK to exploit this property of assignments, and is in fact a common idiom. The following code correctly prints every character from the input file:

Observe how we assign the result of fgetc to the variable c in the while loop's conditional expression. We then wrap that assignment statement in parenthesis to ensure the correct order of operations, and compare the value which was assigned (whatever fgetc returned) to EOF.

You may have noticed that the type of c is int, not char. If we declared c as a char, our program would have a rather subtle bug. Can you spot it? Remember that we said fgetc returns an int so that it can return any possible character value read from the file, plus some distinct value for EOF. Assigning this return value to a char then inherently discards information—we are taking N + 1 possible values and assigning them to something that can hold N different bit patterns (in the case of char, N = 256). On most systems EOF is –1, so in this particular case, we would not be able to distinguish between reading character number 255 and the end of the file—if our input had character number 255 in it, our program would prematurely exit the loop, and ignore the rest of the input! You should aim to think of these sorts of corner cases when you write test cases.

6.3.1 Reading a file with fgets

The fgets function is useful when you want to read one line (with a maximum length) at a time. This function has the following prototype:

```
char * fgets(char * str, int size, FILE * stream);
```

This function takes three arguments. The first is a pointer to an array in which to store the characters read from the file. That is, fgets will write the data into str[0], str[1], str[2],... The second argument specifies how much space is available for it to write data into. That is, size specifies the size of the array str. The final argument specifies from what stream to read the data.

This function returns str if it succeeds (reads data without error), in which case, the data in str is null-terminated. It returns NULL if it fails—either if it encounters the end of the file before reading any data, or if it encounters some other error. If you need to distinguish between an error and end-of-file, you should use the feof and/or ferror functions, which specify whether something attempted to read past end-of-file, or whether some other error occured respectively (see their man pages for details).

Now is a good time to re-mention that you should never use the gets function. This function behaves somewhat similarly to fgets, but does not take an argument specifying the size of the array it reads into. This oversight means that it will continue to read data until it reaches a newline, even if it writes past the bounds of the array (it has no way to tell how big it is). The gets function therefore poses a significant security vulnerability, as it is susceptible to buffer overflows.

Look into mp4 fgets.

6.3.2 Reading a file with fread

We may also want to read non-textual data from a file. For example, we might have an image, video, sound, or other file where we write data in a binary format. In such a file, rather than writing the textual representation of an integer, the actual bytes for the integer are written to the file. The specific size of integer used for each piece of data is part of the file format specification. When we want to read data in this fashion, the most appropriate function is fread, which has the following prototype:

```
size_t fread (void * ptr, size_t size, size_t nitems, FILE * stream);
```

The first argument is a pointer to the data to write—it is a void *, since it could be any type of data. The next argument specifies the size of each item. That is, if we are writing an int or an array of ints, we would pass in sizeof(int). However, if we are reading and writing files, we probably want to work with the int types defined in stdint.h, which have their sizes fixed across systems (such as int32_t or uint64_t). The third argument specifies how many such items should be read from the stream, and the final argument specifies from which stream to read. The fread function returns how many items were successfully read. As with fgets, you should employ feof and/or ferror to obtain more information if fread returns fewer items than you requested.

We will note that there is also a function, fscanf, which reads formatted input and performs conversions in the reverse fashion of what printf does. If you need this functionality, it is often easier to deal with errors if you use fgets (or, as we will see later, getline) and then use sscanf on the resulting string. fgets (and later getline) will read a line at a time, and then you can attempt to convert the results out, which may or may not have the desired format. In such a case, you can easily continue reading the next line. By contrast, fscanf **will stop reading input as soon as it encounters something that does not match the requested format.** If you want to continue reading from the next line, you must then explicitly read out the rest of the line before proceeding. If you want to know more about fscanf or sscanf, see their man pages.

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void encrypt(FILE * f, int key) {
  int c;
  while ((c = fgetc(f)) != EOF) {
    if (isalpha(c)) {
      c = tolower(c);
      c -= 'a';
      c += key;
      c %= 26;
      c += 'a';
    }
    printf("%c", c);
  }
}

int main(int argc, char ** argv) {
  if (argc != 3) {
    fprintf(stderr,"Usage: encrypt key inputFileName\n");
    return EXIT_FAILURE;
  }
  int key = atoi(argv[1]);
  if (key == 0) {
    fprintf(stderr,"Invalid key (%s): must be a non-zero integer\n", argv[1]);
    return EXIT_FAILURE;
  }
  FILE * f = fopen(argv[2], "r");
  if (f == NULL) {
    perror("Could not open file");
    return EXIT_FAILURE;
  }
  encrypt(f,key);
  if (fclose(f) != 0) {
    perror("Failed to close the input file!");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Listing 27: Encription

6.4 Writing to and Closing Files

The other operation we may wish to perform is to write to a file. As with reading from a file, there are a variety of options to write to a file. One option—useful if we are printing formatted text—is the fprintf function. This function behaves the same as the familiar printf function, except that it takes an additional argument (before its other arguments), which is a FILE * specifying where to write the output.

You can also use **fputc**to write a single character at a time, or **fputs**to write a string without any format conversions. That is, if you do **fputs("%d")** it will just print "%d"to the file directly rather than attempting to convert an integer and print the result.

Finally, if you want to print non-textual data, your best choice is **fwrite**, which has the prototype:

```
size_t fwrite(const void * ptr, size_t size, size_t nitems, FILE
* stream);
```

The arguments are much the same as those given to **fread**, except that the data is read from the buffer pointed to by **ptr**and written to the stream (whereas **fread**reads from the stream and writes into the buffer pointed to by **ptr**).

All of these methods write at the current position in the file and advance it accordingly. Furthermore, any of these methods of writing to a file may fail, some of which are detected immediately, and others of which are detected later. See the relevant man pages for the functions you are using to see how they might fail and what values the return to indicate failures.

The reason that the failures may be detected later is that the C library functions may buffer the data, and not immediately request that the OS write it. Even once the application makes the requisite system calls to write the data, the OS may buffer it internally for a while before actually writing it out to the underlying hardware device. The motivation for not writing immediately is performance: making a system call is a bit of a slow process, and writing to a hard disk is quite slow. Furthermore, writing a disk becomes less efficient as you write smaller quantities of data to it—there are fixed overheads to find the location on the disk which can be amortized over large writes—so the OS tries to buffer up writes until there is significant data that can be written at once.

6.4.1 Closing Files

After you are finished with a file, you should close it with the fclose function, which has the following prototype:

```
int fclose(FILE * stream);
```

This function takes one argument, specifying which stream to close. Closing the stream sends any buffered write data to the OS, and then asks the OS to close the associated file descriptor. After calling fclose on a stream, you may no longer access it (a call to fgetc, fprintf, etc. is erroneous, though exactly what will happen is undefined).

Observe that fclose returns an int. This return value indicates the success (0 is returned) or failure (EOF is returned, and errno is set accordingly) of the fclose operation. The fclose operation can fail for a variety of reasons, the most serious of which arise in circumstances where the data cannot actually be written to the underlying hardware device (e.g., disk drive)—for example, the disk is full, or the file resides on a remote file system and network connectivity has been lost. Failure of fclose for a file you have been writing to is a serious situation—it means that the data your program has tried to write may be lost. You should therefore, always check the return value of fclose to see if it failed.

However, what you do in response to the failure of fclose is a bit of a difficult question. You cannot try again, as performing any operation on the stream you attempted to close—including another call to fclose—is erroneous (and results in undefined behavior). What you do, however, is highly situation-dependent.

In an interactive program, you may wish to inform the user of the problem, and she may be able to take corrective action before proceeding. For example, suppose you are writing an editor (like emacs). If the user attempts to save their work but the disk is full, she would much rather be told that the save failed (and why). The user could then proceed to free up disk space, and save again (which would involve fopening the file again, fwriteing all the data, the fcloseing that stream—not retrying to fclose the original stream). By contrast, if the editor ignored the return value of fclose and failed silently — not informing the user of the problem, she may quit, losing all of her work.

With `FILE * f = fopen(argv[3], "w");` we create the file if it does not exists or truncated to zero length if it does exists, then start writing to it from the beginning.

```c
//broken code: do not do
int * initArray(int howLarge) {
  int myArray[howLarge];
  for (int i = 0; i < howLarge; i++) {
    myArray[i] = i;
          }
  return myArray;
}
```

Listing 28: Broken Array Initializer

6.5 Motivation for Dynamic Allocation

Recall the task of writing a function that takes an integer, creates an array of the size specified by the integer, initializes each field, and returns the array back to the caller. Given the tools we had thus far, our code (which would not work!) would look like this:

The reason that this code will not work is that the array is created on the stack. Variables on the stack exist only until the function ends, at which point, the stack frame essentially disappears.

While it may not seem that important from this example to be able to create an array and return it, programmers often want to write functions which perform more complex creation tasks (and have the results persist beyond the function that created them). Suppose you needed to read information from a file (possibly with a complex format). You might want to write a function to read the information, allocating memory to hold the results in some combination of arrays and structs, and return the result to the caller. Fortunately, there is a way to do exactly this: dynamic memory allocation.

Dynamic memory allocation allows a programmer to request a specific amount memory to be allocated on the heap (highlighted in purple in the Figure 29)—not the stack. Because the memory is not in the stack frame, it is not freed when the function returns. Instead, **the programmer must explicitly free the memory when she is done using it.**

Dynamic memory allocation involves both allocating memory (with the **malloc**function), and freeing that memory when you no longer need it (with the **free** function). Programmers may also wish to reallocate a block of memory at a different size (for example, you may think an array of 8 elements is sufficient, then read some input and find that you need 16). The **realloc**function allows a programmer to do exactly this—asking the standard library to al-

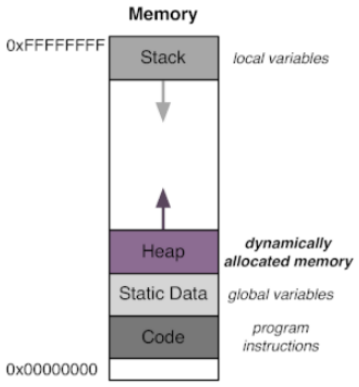


Figure 29: Dynamic memory allocation

locate a new (larger or smaller) block of memory, copy the contents of the original to the new one, and free the old one (although the library may optimize this procedure if it can expand the block in place). For all three of these functions, include ¡stdlib.h¿. We will also see a wonderful function, **getline**, for reading strings of arbitrary length using dynamic allocation.

6.6 Malloc

The most basic function for dynamic memory allocation is called malloc (which performs memory allocation). Calling this function is how you allocate memory dynamically. The malloc function, shown in the Figure 30 below takes one argument telling it how much memory is needed and it returns a pointer to that allocated memory in the form of a void *. Many beginning programmers are intimidated by the concept of a void *, but you should not be! Recall that a void * just means a pointer, but we do not know what type of thing it points to. If malloc instead returned something more specific (for example, an int *), we would need a new version of malloc for every data type. This would be both unwieldy and (in the context of user-defined data types) impossible. Just remember that you can assign a void * to any other pointer type—so just assign the return result of malloc to whatever pointer you want to initialize.

Unlike variables that resides in stack frames, this box that you will create in the heap does not have its own na.e The return value of malloc is a pointer to this newly allocated memory.

Because the memory that is allocated by malloc is in heap, this memory will not be freed automatically.

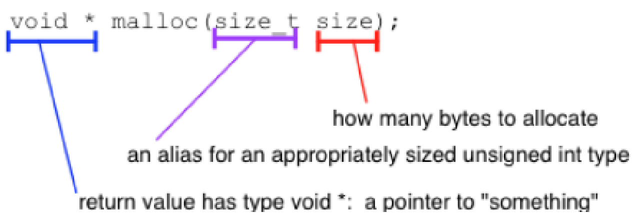

void * malloc(size_t size);
how many bytes to allocate
an alias for an appropriately sized unsigned int type
return value has type void *: a pointer to "something"

Figure 30: malloc

```c
// this code does work!
int * initArray(int howLarge) {
  int *array = malloc (howLarge * sizeof(*array));
  if (array != NULL) {
    for (int i =0; i < howLarge; i++) {
      array[i] = i;
          }
        }
        return array;
}
```

Listing 29: Fixing initArray

In the case of `int * p;`

and `p = malloc(6 * sizeof(*p));`

We have 6 times the size of *p, which means that it would be an array of six elements. As p is an int pointer, those elements would be ints.

Fixing initArray

With `malloc` at our disposal, we are finally able to correctly implement the task introduced at the beginning of this chapter:

Note that in this function, if malloc fails (i.e., returns NULL), then the function returns NULL—this pushes the task of handling the error up to whoever called this function. Whenever you write a function where you would not know how the error should be handled, making the caller handle the error is a good strategy.

More Complex Structures

```c
struct point_tag {
  int x;
  int y;
};
typedef struct point_tag point_t;

struct polygon_tag {
  size_t num_points;
  point_t * points;
};
typedef struct polygon_tag polygon_t;

polygon_t * makeRectangle(point_t c1, point_t c2) {
  polygon_t * answer = malloc(sizeof(*answer));
  answer->num_points = 4;
  answer->points = malloc(answer->num_points * sizeof(*answer->points));
  answer->points[0]   = c1;
  answer->points[1].x = c1.x;
  answer->points[1].y = c2.y;
  answer->points[2]   = c2;
  answer->points[3].x = c2.x;
  answer->points[3].y = c1.y;
  return answer;
}
```

Listing 30: More Complex Structrures - Malloc

Even though our examples so far have shown mallocing arrays of ints, we can, of course, malloc any type that we want. We can malloc one of a thing if we need (rather than an array) or any number of things (as memory permits). We can form complex data structure in the heap by mallocing structs which have pointers, and then setting those to point at other locations in the heap (which themselves point at blocks of memory allocated by malloc). We state this explicitly because it is important; however, you could also realize that all of this is possible due to the principle of composability —we can put all these different pieces together to make larger things.

For example, we might write:

Here, we have a function that mallocs space for a polygon (one struct), which itself has a pointer to an array of points. This particular function makes a rectangle, so it mallocs space for 4 points and fills them in before returning its answer to the caller.

6.6.1 Shallow vs Deep Copying

Suppose we had a **polygon_t * p1** pointing at a polygon we created (e.g., by calling **makeRectangle**), and we wanted to make a copy of the polygon it points to. If we just wrote the following, we would only copy the pointer—we would not actually copy the object it points to:

```
polygon_t * p2 = p1;
```

The figure below illustrates the (hopefully now familiar) effects of this statement. After assigning the pointer p1 to the pointer p2, both point at the exact same memory location. The only new box that was created was the box for the pointer p2, and if we change anything about the polygon through one pointer, we will see the change if we examine the values through the other pointer—since they point at the same values.

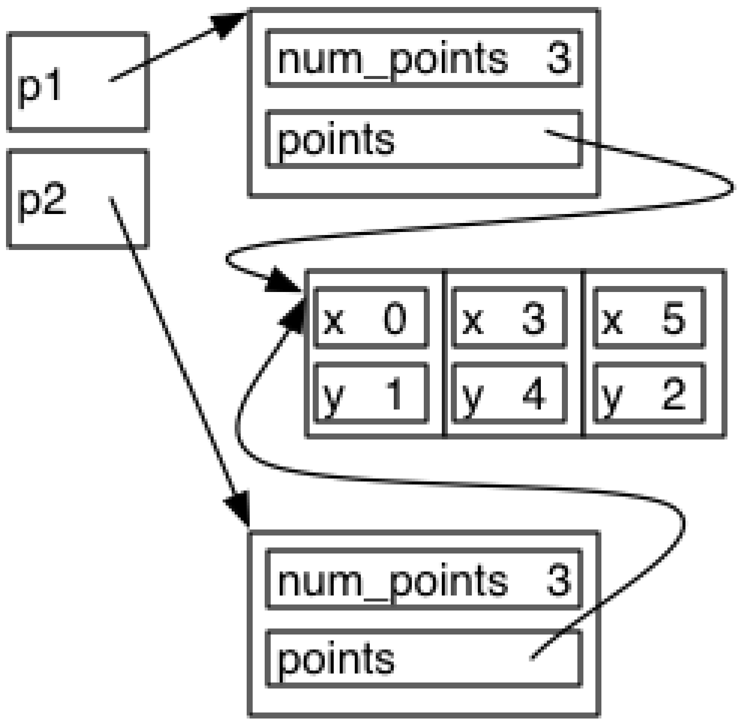


Figure 31: Pointer copy

If we were to use malloc, we could create a copy. For example, we might write:

```
polygon_t * p2 = malloc(sizeof(*p2));
*p2 = *p;
```

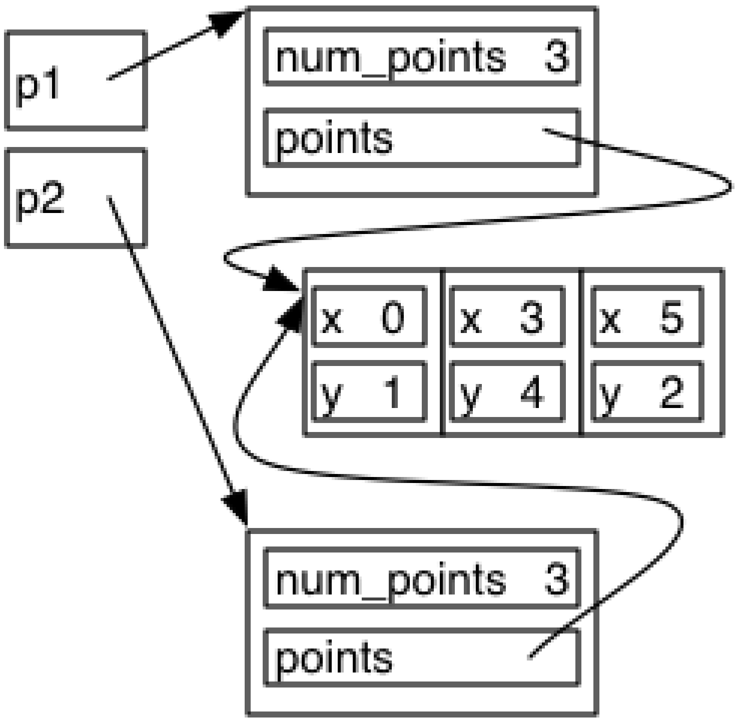


Figure 32: Shallow copy

Now, we have two polygons, but only one array of points. We have created a shallow copy of the polygon—we have made a copy of the polygon, by exactly copying the fields inside of it. Each pointer in the shallow copy points at exactly the same location in memory as the corresponding pointer in the original. In some cases, a shallow copy may be what we want. However, if we did **p1-¿points[0].x = 42;** we would change the x of p2's 0 point, since **p1-¿points** points at the same memory as **p2-¿points.** Notice that we must also be careful when freeing an object which has had a shallow copy made of it—we need to take care to only free the array of points when we are done with both shallow copies. As they share the memory, if we free the points array when we are done with one, then try to use the other copy, it will have a dangling pointer.

If we want two completely distinct polygon objects, we want to make a deep copy—in which we do not just copy pointers, but instead, allocate new memory for and make deep copies of whatever the pointers point to. To

```c
polygon_t * p2 = malloc(sizeof(*p2));
p2->num_points = p1->num_points;
p2->points = malloc(p2->num_points * sizeof(*p2->points));
for (size_t i = 0; i < p2->num_points; i++) {
  p2->points[i] = p1->points[i];
}
```

Listing 31: Deep Copy

make a deep copy of our polygon_t, we would write:

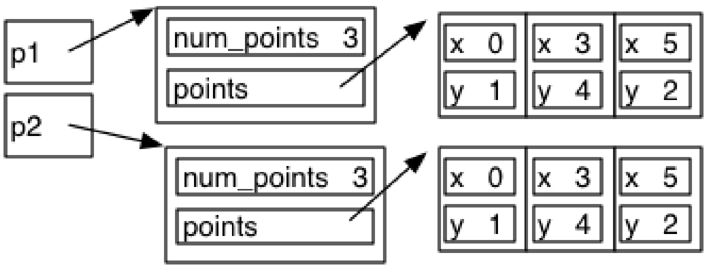


Figure 33: Deep Copy

Here we have created two completely distinct polygons which have the same values for their points, but their own memory. If we change the x or y of one polygon's points, the other remains unaffected, as they are two completely distinct data structures. Similarly, we can now completely free one polygon when we are done with it without worrying about anything else sharing its internal structures.

6.7 Free

Unlike memory allocated on the stack (which is freed as soon as the function associated with that stack frame returns), memory on the heap must be explicitly freed by the programmer. For this, the C Standard Library provides the function **free**.

The free function, whose signature is shown in the figure above takes one argument: the starting address of the memory that needs to be freed. This starting address should match the address that was returned by malloc. As a good rule of thumb, every block of memory allocated by malloc should be freed by a corresponding call to free somewhere later in the execution of the

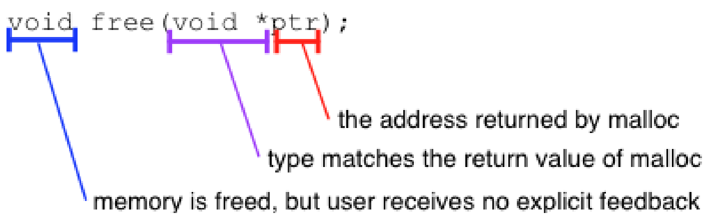

void free(void *ptr);
the address returned by malloc
type matches the return value of malloc
memory is freed, but user receives no explicit feedback

Figure 34: free

program.

When you free memory, the block of memory that you freed is deallocated, and you may no longer use it. Any attempt to dereference pointers that point into that block of memory is an error—those pointers are dangling. Of course, as with all dangling pointers, exactly what will happen is undefined. When you execute code by hand, and you need to free memory, you should erase the block of memory that is being freed. Specifically, if the code says free(p), you should follow the arrow for p (which should reach a block of memory in the heap, or be NULL), and then erase that entire block of memory. If p is NULL, nothing happens. If p points at something other than the start of a block in the heap, it is an error and bad things will happen (your program will likely crash, but maybe something worse happens).

Note that free only frees the memory block you ask it to. If that memory block contains other pointers to other blocks in the heap, and you are done with that memory too, **you should free those memory blocks before you free the memory block containing those pointers**. In our polygon example, we would free a polygon like this (although better would be to write a freePolygon function with the two calls to free in it)

```
polygon_t * p = makeRectangle(c1, c2);
free(p->points);
free(p); ;
```

Note that doing free(p) followed by free(p-¿points) would be wrong: p would be dangling at the second call, and thus we should not dereference it.

`free(p)` does not actually affects p, but rather the memory that p points to. So, if we had `int * q = p;` when we free p, q is also a dangling pointer.

Check free.mp4

6.7.1 Memory Leaks

When you lose all references to a block of memory (that is, no pointers point at it), and the memory is still allocated, you have leaked memory (or you might say your program has a memory leak). For small programs that you write in this book, a memory leak may not seem like a big deal—all of the memory allocated to your program will be released by the operating system when your program exits, so who cares if you have a few hundred bytes lying around?

In small programs that run for at most a few seconds, a memory leak may not have much impact. However, in real programs, memory leaks present significant performance issues. Do you ever find that certain programs get slower and slower as they are open longer? Maybe you have a program which, if open for a day or two, you have to restart because it is annoyingly sluggish. A good guess would be that the programmers who wrote it were sloppy, and it is leaking memory.

You, however, are not going to be a sloppy programmer. You are going to free all your memory. When you write a program, you should run it in valgrind, and be sure you get the following message at the end:

```
All heap blocks were freed -- no leaks are possible
```

The next video shows code that allocates memory, but does not free it—leaking the memory. Notice how, when p goes out of scope, there are no more references to that block of memory, but it remains allocated. The memory cannot be reused for future allocation requests (as it has not been freed), but is not needed by the program any longer (so it should have been freed). The video concludes by fixing the code by adding a call to free.

Watch memory_leak video

A Dynamic Memory Allocation Analogy

malloc and **free** can be confusing to novice programmers, but they are crucial to producing correct, efficient, and clean code. To help you understand them, consider the following analogy: You are at the bus depot and there are 50 lockers available for rent. To request a locker, you simply tell the worker at the window how many lockers you need. You will be given a contiguous stretch of lockers, and you will be told which is the first locker that is yours. For example, you might say "5 lockers, please," and you might be told "Your starting locker is number 37," at which point, you have been given lockers 37 through 41. You might also be told "No, sorry." because there are not 5 contiguous lockers available for rent. In response to your successful request, the worker at the window records that 5 lockers are in use, starting at locker 37.

When you are done using your lockers, you return to the window and tell the worker "I'm done with the lockers starting at 37," and those 5 lockers now become available to someone else behind you in line. When you return your lockers, you free all or none of them. You can't return a subset of your request. Also, you need to keep track of your starting locker (37), because that is how the worker has recorded your booking. **In technical terms, you may call free only with a pointer that was returned by malloc.** Furthermore, you cannot return your lockers twice. Even if you and your partner are both using the lockers, only one of you should return them. Again, in technical speak, **you may call free only once**, even if there are multiple pointers to that location in memory. Freeing the same location in memory multiple times is illegal.

Common problems with free

Our locker analogy made references to two common errors that programmers make when using malloc and free. Trying to "free" the same lockers twice is a problem. In the case of memory allocation, trying to free the same block of memory more than one time is called double freeing. Generally, your program will crash (segfault), although, other more sinister behaviors can occur. A segfault on the line where the double free happened is nice: it makes debugging easier. However, you may get stranger symptoms—including your program crashing the next time you call malloc. In general, if malloc crashes, an earlier error in your code has corrupted its bookkeeping structures, and you have just now exposed the problem. Run your code in valgrind, and it is quite likely to help you expose the error sooner.

Another common problem, also alluded to in the locker analogy, is freeing something that is not at the start of the block returned by malloc. If the locker attendant gave you lockers 37–41, you cannot go back and say "I'm done with 38." This may seem silly: why can't the locker attendant just figure out that when you say you are done with 38, you mean the block from 37–41? For a human tracking lockers, this may seem like a silly rule; however, it makes much more sense for malloc and free.

Neither of these functions is magical (nothing in your computer is—as you should have learned by now). They need to do their own bookkeeping to track which parts of memory are free and which are in use, as well as how big each block that is in use is. Bookkeeping requires memory: they must store their own data structures to track the information—but where do they get the memory to track what memory is in use? The answer is that malloc actually allocates more memory than you ask for, and keeps a bit for itself, right before the start of what it gives you. You might ask for 16 bytes, and malloc gives you 32—the first 16 contain its information about the block, and the next 16 it gives you to use. When you free the block, the free function calculates the address of the metadata from the pointer you give it (e.g., subtract 16). If you give it a pointer in the middle of the block, it looks for the metadata in the wrong place.

Going back to the locker analogy, this would be as if the locker attendant gives you lockers 37–41, but then puts a note in locker 36 that says "this block of lockers is 5 long." When you return locker 37, he looks in locker 36 and finds the note. If you instead tried to give back locker 38, he would look in locker 37 and become very confused.

A third common mistake is freeing memory that is not on the heap. If you try to free a variable that is on the stack (or global), something bad will happen—most likely, your program will crash.

Watch problems with free.mp4

1. Trying to free again is illegal and it will return in undefined behavior.
2. Try to free memory that's not actually in the heap. If we want to free data in the stack is illegal and it will result in early termination of our program.
3. Free the middle of the block. p++ so instead of pointing to the 0th element of the arrya, it points to the first element of the array before free(p). Illegal and it will result in early termination of the program.

6.8 Realloc

Suppose a program initially asks for n bytes on the heap, but later discovers that it needs more than n bytes. In this case, it is not acceptable to simply reference past the size of the initial malloc request. For example, if an array has 4 elements, which are indexed via array[0] to array[3], it would not be acceptable to simply write into array[4] just because the program's space requirements for the array have changed. In locker-speak, this is the equivalent to realizing you need a 6th locker and taking locker 42, even though it was not given to you to use. (Locker 42 might be in use by someone else, or it might be given to another person at a later time.) The proper way to respond to this increased space needs is to use **realloc**.

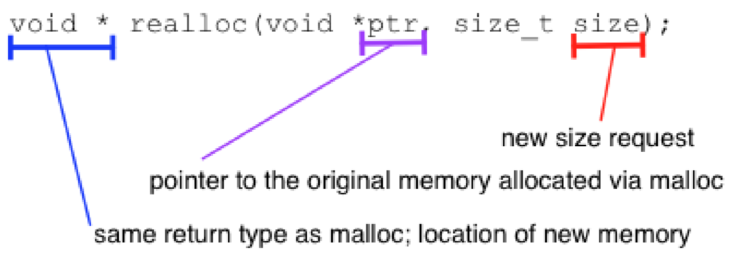

void * realloc(void *ptr, size_t size);
new size request
pointer to the original memory allocated via malloc
same return type as malloc; location of new memory

Figure 35: Realloc

realloc effectively resizes a malloced region of memory. Its signature is shown in the figure above. The arguments to realloc are the pointer to the region in memory that you wish to resize and the new size you wish the region in memory to have. If successful, realloc will return a pointer to the new, larger location in the heap that is now at your disposal. If no area in the heap of the requested size is available, realloc returns NULL.

Keep in mind that the new location in memory does not need to be anywhere near the original location in the heap. In terms of our locker analogy, if you return to the window and say "I have the lockers starting at locker 37, but I need 6 lockers now," the worker may not be able to give you locker 42. Instead the worker may respond "Okay, your new starting locker is 12." Conveniently, the worker will move the contents of lockers 37–41 into lockers 12–16 for you.

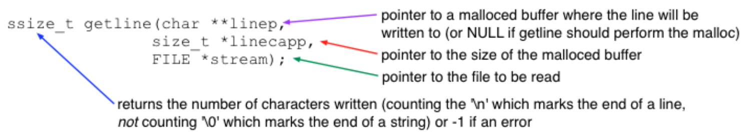
Beyond malloc, free, and realloc, there are a few more standard functions available for memory allocation, such as calloc (which zeroes out the region in memory for you—by contrast, malloc does nothing to initialize the memory). The man pages for all of these functions are, as always, great reference.

6.9 getline

You have already seen fgets, which lets you read a string into a buffer that you have preallocated, specifying the maximum size for the string to read (i.e., how much space is in the buffer). However, how could you write code that would read a string of any length? Before this chapter, we have not had the tools to think about such a function—it clearly requires dynamic allocation as we would need that function to allocate memory which lasts after the function returns. Now, we can learn about **getline**

getline is a C function available in the C Standard I/O Library (**#include**

¡stdio.h¿). Its signature (shown in Figure 36) looks a bit intimidating, but it is worth understanding.



```
ssize_t getline(char **linep,
                size_t *linecapp,
                FILE *stream);
```

pointer to a malloced buffer where the line will be written to (or NULL if getline should perform the malloc)
pointer to the size of the malloced buffer
pointer to the file to be read
returns the number of characters written (counting the '\n' which marks the end of a line, *not* counting '\0' which marks the end of a string) or -1 if an error

Figure 36: getline

getline reads a single line from the file specified in its third argument, stream. It does this by reading characters from the file repeatedly until it sees the character '\n', which indicates the end of a line. As it reads each character, it copies the characters into a buffer in memory. After reading the newline character, getline places a '\0' character, which indicates the end of the string.

So far, this behavior sounds much like **fgets**; however, the difference is in the fact that getline allocates space for the string as needed. The difference arises from the fact that getline uses **malloc**and **realloc**to allocate/enlarge the buffer. The **linep**parameter points at a pointer. If ***linep** is NULL, getline mallocs a new buffer. If ***linep** is not NULL, getline uses that buffer (which is ***n** bytes long) to start with. If the initial buffer is not long enough, getline reallocs the buffer as needed. Whenever getline mallocs or reallocs the buffer, it updates ***n** to reflect the number of bytes allocated. It also updates ***linep** to point at the new buffer. When the getline function returns, ***linep** is a pointer to the string read from the file.

The getline function returns -1 on an error (including end of file), and the number of bytes read (not counting the null terminator byte) on success. Note that the return type is ssize_t, which stands for "signed size_t"—that is, the signed integer type which is the same number of bytes as size_t (so it can return -1).

The getline function can be used in two ways. First, the user can provide getline with a pointer to a malloced buffer ***linep**, whose size is pointed to by **linecapp**. If the line being read is larger than ***linecapp,** getline will perform a realloc for the user and modify the values of ***linep** (to point to the newly realloced region) and ***linecapp** (to be the size of the newly reallocated buffer) accordingly. Second, the user can provide getline no buffer, indicated by **linep**containing a pointer to NULL (**linep** cannot be NULL, but ***linep** can be NULL). In this case, getline will perform a malloc for the user and modify the values of ***linep** (to point to the newly malloced region) and ***linecapp** (to be the size of the newly allocated buffer) accordingly. These two ways can be used together—i.e., one can write a loop where no

```c
1 size_t sz = 4;
2 char * line = NULL;
3 getline(&line, &sz, stdin);
```

Listing 32: getline

```c
1 size_t sz= 4;
2 char * line = mallocS(42 * sizeof(*line));
3 getline(&line, &sz, stdin);
```

Listing 33: getline2

buffer is provided the first time, and the same buffer is reused on subsequent iterations.

The next video shows an example of using getline to read lines from a file and print them out. The final video shows a slightly larger example, which combines getline and realloc to read all the lines from a file into an array. The example then sorts them (using **qsort**, which you saw in Course 3), prints out the results, and frees the memory appropriately.

Check out getline.mp4 and getline_free.mp4

Questions

1. Suppose you have the code in Listing 32:

The **getline** call on line 3 will **malloc** an implementation-specific amount of space.

2. Suppose you have the code in Listing 33

What will the call to **getline** on line 3 will assume about the amount of space it has available to read characters from **stdin** into: it will assume that there are at least 4 bytes.

6.10 Valgrind

The C compiler will give you warnings and errors for things it can tell are problematic. However, what the compiler can do is limited to static analysis of the code i.e., things it can do without running the code or knowing the inputs to the program. Consequently, there are many types of problems the compiler cannot detect. These types of problems are typically found by running the code on a test case where the problem occurs.

However, just because the problem occurs does not mean that it produces a useful symptom for the tester. Sometimes a problem can occur with no observable result, with an observable result that only manifests much later, or with an observable result that is hard to trace to the actual cause. All of these possibilities make testing and debugging more difficult.

Ideally, we would like to have any problem be immediately detected and reported with useful information about what happened. When we just run a C program directly, such things do not occur, as the compiler does not insert any extra checking or reporting for us. However, there are tools that can perform additional checking as we run our program to help us test and debug the program. One such tool is Valgrind, in particular, its Memcheck tool.

Valgrind is actually a collection of tools, which are designed so that more can be added if desired. However, we are primarily interested in Memcheck, which is the default tool and will do the checking we require. Whenever you are testing and debugging your code, you should run it in Valgrind (called ”valgrinding your program”). While valgrinding your program is much slower than running the program directly, it will help your testing and debugging immensely. You should fix any errors that valgrind reports, even if they do not seem to be problems.

To ”valgrind your program,” run the valgrind command, and give it your program's name as an argument (Memcheck is the default tool). If your program takes command line arguments, simply pass them as additional arguments after your program's name. For example, if you would normally run ./myProgram hello 42, instead run valgrind ./myProgram hello 42. For the most benefits, you should compile with debugging information in your program (pass the -g or -ggdb3 options to gcc).

When you run Valgrind, you will get some output that looks generally like this:

```
==11907== Memcheck, a memory error detector
==11907== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==11907== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==11907== Command: ./myProgram hello 42
==11907==
==11907==
==11907== HEAP SUMMARY:
==11907==     in use at exit: 0 bytes in 0 blocks
==11907==   total heap usage: 2 allocs, 2 frees, 128 bytes allocated
==11907==
==11907== All heap blocks were freed -- no leaks are possible
==11907==
==11907== For counts of detected and suppressed errors, rerun with: -v
==11907== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Each line that starts with ==11907== here is part of the output of Valgrind. Note that when you run it, the number you get will vary (it is the process ID, so it will change each time). Valgrind prints a message telling you what it is doing (including the command it is running), then it runs your program. In this case, the program did not produce any output on stdout; however, if it did, that output would be interspersed with Valgrind's output. There are also no Valgrind errors—if there were, they would be printed as they happen. At the end, Valgrind gives a summary of our dynamic allocations and errors. The last line shows that we "valgrinded cleanly"—we had no errors that Valgrind could detect. If we did not valgrind cleanly, we should fix our program even if the output appears to be correct.

Note that if our program has errors, Valgrind will report them and keep running (until a certain error limit is reached). Whenever you have multiple errors, you should start with the first one, fix it, and then move to the next one. Much like compiler errors, later problems may be the result of an earlier error.

One common misconception novice programmers often get is that they should run Valgrind only after they have debugged their program and otherwise think it works. However, you will find the debugging process much easier if you use Valgrind's Memcheck throughout your testing and debugging process. You may find that the odd or confusing bug you have not been able to figure out for hours is actually caused by a subtle problem you did not notice earlier, which Valgrind could have found for you. Note that Valgrind does not "play nice" with code that has been compiled with -fsanitize=address, so you should compile without that option to valgrind your code. Ensuring that both tools cannot find any problems with your code is likely a great idea, as they can detect different problems.

We will introduce the basics here but recommend further reading in the Valgrind user's manual: http://valgrind.org/docs/manual/manual.html for further information.

6.10.1 Uninitialized Values

When we execute code by hand, we put a '?' in a box if that location has not been initialized. Similarly, when we execute by hand, we consider it an error to use an uninitialized value. It is indicative of a problem in our program, and we have no idea what will actually happen when we run our code.

However, when you run your program, there is always some value there, we just do not know what it is. Therefore, when we run our program, we will get a value (it is just whatever happened to be in that memory location before), and there is not any checking that the value is initialized. We will not know that we have used an uninitialized value, and if we get unlucky, our program will print the correct output anyways.

Use of uninitialized values are a great example of why "silent" bugs are dangerous and need to be fixed. We might think our program is right when we test and debug it because the value in that location is coincidentally correct. However, some change in the execution of the program can cause that "luck" to end, resulting in the program exhibiting an error.

The change that disrupts your "luck" could be due to a variety of circumstances. For example, changing the input to the program could cause a different execution path leading up to the use of the uninitialized value, thus causing it to have a different value in a variety of different possible ways. Another possibility is if you compile your code with optimizations versus for debugging. When the compiler optimizes your code, it may place different variables in different locations relative to the debugging version. Such a bug can therefore lead to the annoying situation where the debug version of your program appears to work, but the optimized version does not (even on the same inputs). We have even seen students experience problems with differences between the values read for uninitialized locations between redirecting the output to a file and printing it directly to the terminal.

Valgrind's Memcheck tool explicitly tracks the validity of every bit in the program, and can tell us about the use of uninitialized values. By default, Memcheck will tell you when you use of an uninitialized value; however, these errors are limited to certain uses. If x is uninitialized, and you do y = x, Memcheck will not report an error, but rather just note that y now holds an uninitialized value. In fact, even if we compute y = x + 3, we will still not get an error immediately.

However, if we use an uninitialized location (or one holding a value computed from an uninitialized location) in certain ways that Memcheck considers to affect the behavior of the program, it will give us an error. One such case is when the control flow of the program depends on the uninitialized value—when it appears in the conditional expression of an if statement, or a loop, or in the selection expression of a switch statement. In such a situation, Memcheck will produce the error:

```
Conditional jump or move depends on uninitialized value(s)
```

and produce a call-stack trace showing where that use occurred. For example, if we write the function:

```
void f(int x) {
  int y;
  int z = x + y;
  printf("%d\n", z);
}
```

Valgrind's Memcheck will report the following error (this code is inside of uninit.c, which has other lines before and after those shown above):

```
==12241== Conditional jump or move depends on uninitialised value(s)
==12241==    at 0x4E8158E: vfprintf (vfprintf.c:1660)
==12241==    by 0x4E8B498: printf (printf.c:33)
==12241==    by 0x400556: f (uninit.c:7)
==12241==    by 0x400580: main (uninit.c:15)
```

This error indicates that the uninitialized value was used inside of vfprintf, which was called by printf, which was called by f (on line 7), which was called by main (on line 15). We may be able to fix the program directly by observing the call to printf on line 7 and seeing what we did wrong.

However, if we do not see the problem right off (which could be likely if the uninitialized value has been passed through a variety of function parameters and data structures before Memcheck reports the error), we need more help from Memcheck. In fact, when Memcheck reports such errors, it will helpfully suggest the option we need to get more information from it at the end of its output:

```
==12241== Use --track-origins=yes to see where uninitialised values come from
```

If we run Valgrind again passing in this option (**valgrind –track-origins=yes ./myProgram**), it will report where the uninitialized value was created when it reports the error:

```
==12260== Conditional jump or move depends on uninitialised value(s)
==12260==    at 0x4E8158E: vfprintf (vfprintf.c:1660)
==12260==    by 0x4E8B498: printf (printf.c:33)
==12260==    by 0x400556: f (uninit.c:7)
==12260==    by 0x400580: main (uninit.c:15)
==12260==  Uninitialised value was created by a stack allocation
==12260==    at 0x40052D: f (uninit.c:4)
```

We can now see that the value was created by stack allocation (meaning allocating a frame for a function), and we have the particular line of code that caused that creation (int y; inside of f).

There are other cases where a use of an uninitialized value will result in a message such as:

```
==12235== Use of uninitialised value of size 8
```

Here Memcheck is telling us that we used an uninitialized value in a way it considered problematic, and that the value we used was 8 bytes in size (that is, how many bytes of memory it was accessing). If our uninitialized value is passed to a system call, we will get an error message that looks like this:

```
==12362== Syscall param write(fd) contains uninitialised byte(s)
```

All of these indicate the same fundamental problem: we have a value that we did not initialize. We need to find it (probably by using –track-origins=yes) and properly initialize it.

6.10.2 Invalid Reads and Writes

Valgrind's Memcheck tool will also perform stricter checking of memory accesses (e.g. via pointers) than normally occurs when you run your program. In particular, Memcheck will track whether or not every address is valid at any particular time (as well as whether or not that address contains initialized data). Any access to an invalid address will result in an error for Memcheck.

```c
//horribly broken--returns a dangling pointer!
int * makeArray(size_t sz) {
  int data[sz];
  for (size_t i = 0; i < sz; i++) {
    data[i] = i;
        }
        return data;
}
```

Listing 34: Invalid Reads and Writes

For example, suppose we wrote the following (broken) code:

Depending on what we do with the return result of this function, our code might either appear to work, or give us rather strange errors. If we read the array in the calling function (e.g., main), then we might not immediately observe anything bad (however, if we call other functions, the values in the array might "mysteriously" change). If we ran such code in Valgrind, we would might get an error such as this:

```
==24640== Invalid read of size 4
==24640==    at 0x40060C: main (dangling.c:16)
==24640==  Address 0xfff000340 is just below the stack ptr.
```

Here, Memcheck is telling us that we tried to read 4 bytes from an invalid (currently unallocated) memory location. It gives us a call stack trace for where the invalid read occurred (in this case, it was in on line 16 of the code, which is not shown here). Memcheck tells us what address experienced the problem and gives us the most information it can about where that address is relative to valid regions of memory. In this particular case, the address is just below the stack pointer, meaning that it is in the frame of a function that recently returned. If we had written to the address instead, we would get a message about an "Invalid write of size X."

Note that Memcheck cannot detect all memory-related errors (even though it can detect many that will slip through otherwise). If another function were called (which would allocate a frame in the same address range), the memory would again become valid, and Memcheck would be unable to tell that accesses to it through this pointer are not correct. Likewise, Memcheck may not be able to detect an array-out-of-bounds error because the memory location that is improperly accessed may still be a valid address for the program to access (e.g., part of some other variable).

Note that using **-fsanitize=address** can find a lot of problems of this type that Memcheck cannot. The reason is that **-fsanitize=address** forces extra unused locations between variables and marks them unreadable with the validity bits it uses. Because there is now space invalid space between the variables, the checks inserted by **-fsanitize=address** will detect accesses in between them, such as going out of the bounds of one array.

6.10.3 Valgrind with GDB

We may have a problem that appears on a particular line of code, but only under specific circumstances that do not manifest the first several times that line of code is encountered. When such a situation occurs, we may find it more difficult to debug the code, as simply placing a breakpoint on the offending line may not give us enough information—we might not know when we reach that breakpoint under the right conditions. What we would like is the ability to run GDB and Valgrind together, and have Valgrind tell GDB when it encounters an error, giving control to GDB.

Fortunately, we can do exactly that. If we run Valgrind with the options **–vgdb=full –vgdb-error=0**, then Valgrind will stop on the first error it encounters and give control to GDB. Some coordination is required to get GDB connected to Valgrind (they run as separate processes); however, when run with those options, Valgrind will give us the information we need to pass to GDB to make this happen:

```
==24099== (action at startup) vgdb me ...
==24099==
==24099== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==24099==   /path/to/gdb ./a.out
==24099== and then give GDB the following command
==24099==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=24099
==24099== --pid is optional if only one valgrind process is running
```

At this point, Valgrind has started the program, but not yet entered main—it is waiting for you to start GDB and connect it to Valgrind. We can do so by running GDB (in a separate terminal, or Emacs buffer) and then copying and pasting the target command that Valgrind gave us into GDB's command prompt:

```
(gdb) target remote | /usr/lib/valgrind/../../bin/vgdb --pid=24099
Remote debugging using | /usr/lib/valgrind/../../bin/vgdb --pid=24099
```

```
relaying data between gdb and process 24099
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from ...
done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00000000040012d0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb)
```

At this point, we can give GDB the commands we want. Most often, we will want to give GDB the continue command, which will let it run until Valgrind encounters an error. At this point, Valgrind will interrupt the program and return control to GDB. You can now give GDB whatever commands you want to, so that you can investigate the state of your program.

The combination of Valgrind and GDB is quite powerful and gives you the ability to run some new commands, via the monitor command. For example, if we are trying to debug pointer-related errors and want to know what variables still point at a particular memory location, we can do so using the **monitor who_points_at** command:

```
gdb) monitor who_points_at 0x51fc040
==24303== Searching for pointers to 0x51fc040
==24303== *0xfff000450 points at 0x51fc040
==24303==  Location 0xfff000450 is 0 bytes inside local var "p"
==24303==  declared at example.c:6, in frame #0 of thread 1
```

There are many other monitor commands available for Memcheck. See `http://valgrind.org/docs/manual/mc-manual.html#mc-manual.monitor-commands` for more information about available monitor commands and their arguments.

6.10.4 Dyamic Allocation Issues

Valgrind's Memcheck tool is quite useful for finding problems related to dynamic allocation—whether malloc and free in C, or, if you go on to learn about C++, new/new[], and delete/delete[]. As with other regions of memory, Memcheck will explicitly track which addresses are valid and which are not. It also tracks exactly what pointers were returned by malloc, new, and new[], and places some invalid space on each side of the allocated block.

From all this information, Memcheck can report a wide variety of problems. First, if an access goes just past the end of a dynamically allocated array, Memcheck can detect this problem. Second, Memcheck can detect double freeing pointers, freeing the incorrect pointer, and mismatches between allocations and deallocations (e.g., deleteing memory allocated with malloc, or mixing up delete[] with delete, which would be an error for a C++ programmer). For example, suppose we wrote the following (obviously buggy) code:

```
int * ptr = malloc(sizeof(int));
ptr[1] = 3;
```

If we run this inside of Memcheck, it reports the following error:

```
==5465== Invalid write of size 4
==5465==    at 0x40054B: main (outOfBounds.c:8)
==5465==  Address 0x51fc044 is 0 bytes after a block of size 4 alloc'd
==5465==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-lin
==5465==    by 0x40053E: main (outOfBounds.c:7)
```

The error first tells us the problem (we made an invalid write of 4 bytes), with a stack trace indicating where that invalid write happened (in main, on line 8 of the file outOfBounds.c). The second part tells us what invalid address our program tried to access and the nearest valid location. In this case, it reports it as "0 bytes after" (meaning in the first invalid byte past a valid region) "a block of size 4" (meaning how much space was allocated into that valid region). Memcheck then reports where that valid region of memory was allocated (in main on line 7, by calling malloc).

Note that if we recently freed a block of memory, Memcheck will report proximity to that block of memory, even though it is no longer valid. For example, if we write:

```
int * ptr = malloc(sizeof(int));
free(ptr);
ptr[0] = 3;
```

Then Memcheck will report the invalid write as being inside of the freed block (and tell us where we freed the block):

```
==5486== Invalid write of size 4
==5486==    at 0x4005A3: main (outOfBounds2.c:9)
==5486==  Address 0x51fc040 is 0 bytes inside a block of size 4 free'd
```

```
==5486==    at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==5486==    by 0x40059E: main (outOfBounds2.c:8)
```

Valgrind's Memcheck will also check for memory leaks. However, by default it only reports a summary of the leaks, which is not useful for finding and fixing the problems. If you have memory leaks, you will want to run with the **–leak-check=full** option. When you do so, Memcheck will report the location of each allocation which was not freed. You can then use this information to figure out where you should free that memory.

Note that when running Valgrind's Memcheck with GDB, you can run the leak checker at any time with the monitor command **monitor leak_check full reachable any**.

6.10.5 Memcheck

Sometimes we may want to interact with Valgrind's tools directly in our program. For example, we might want to explicitly check if a value is initialized at a certain point in the program (e.g., as part of debugging an error about uninitialized values). Valgrind provides header files, such as memcheck.h, which contains a variety of macros for exactly this purpose. For example, we could change the function we were using earlier as an example of uninitialized values to

```
void f(int x) {
  int y;
  int z = x + y;
  VALGRIND_CHECK_MEM_IS_DEFINED(&z,sizeof(z));
  printf("%d\n", z);
}
```

Now, when we run this program in valgrind, we get the error message more immediately:

```
==12425== Uninitialised byte(s) found during client check request
==12425==    at 0x4007C9: f (uninit4.c:8)
==12425==    by 0x400811: main (uninit4.c:17)
==12425==  Address 0xfff000410 is on thread 1's stack
==12425==  Uninitialised value was created by a stack allocation
==12425==    at 0x400765: f (uninit4.c:5)
```

Many of Memcheck's features are available through these macros. Most other tools have similar header files for programs to interact directly with them. See `http://valgrind.org/docs/manual/mc-manual.html#mc-manual.clientreqs` for more details.

6.10.6 Other Valgrind Tools

Memcheck is not the only tool in Valgrind—although it is one of the most commonly used ones, and is what many people think of when they hear "Valgrind." As you become a more advanced programmer, you may find it useful to put some of the other tools to use. We will not delve into them too deeply, but we will note a couple that exist, so that you can know to explore them further as you need to.

There are a variety of issues that arise in multi-threaded programming, such as deadlocks due to improper ordering of acquiring locks, data races, or improper use of synchronization primitives. You should also know that these can be difficult to find due to the non-deterministic behavior of multi-threaded executions—a bug may manifest one time, but then not show up in the next hundred times you run the program. The Valgrind tool **Helgrind**is designed to check for a variety of errors related to multi-threaded programming. See `http://valgrind.org/docs/manual/hg-manual.html` for more details on Helgrind.

Valgrind's tools are not limited to helping your find correctness bugs in your program, but also can be helpful in understanding performance memory usage issues. For example, the **Callgrind** tool gives information about the performance characteristics of a program based on Valgrind's simulation of hardware resources as it executes the program. Another tool is **Massif**, which profiles the dynamic memory allocations in the heap, and gives information about how much memory is allocated at any given time and where in the code the memory was allocated.

See `http://valgrind.org/info/tools.html` for an overview of the tools. Also note that many of these other tools have header files that you can include with macros that allow your program to interact directly with Valgrind (similarly to the functionality in **memcheck.h**).

7 Software Engineering

Very large software projects and/or being an expert in programming in the large.

7.1 Abstraction

One of the key techniques for designing any large system is abstraction. As we previously discussed, abstraction is the separation of interface from implementation. The interface of something is what it does, while the implementation is how it does it.

Consider this example of abstraction: driving a car versus knowing how it works under the hood. The car provides a simple interface (turning the steering wheel turns the car, pushing the accelerator makes the car go faster, pushing the brake slows it down, ...). However, the implementation (how everything works under the hood) is quite complex. Of course, you do not actually have to know how the car works to drive it—the implementation is hidden from you. You only need to know the interface.

A similar principle applies in designing programs— **you should be able to use a function if you know what it does**, without having to know how it does it. In fact, in your programming so far, you have routinely used a wide variety of C library functions without knowing their implementation details (even ones that you may have written an equivalent for as a practice exercise may be implemented in a much more optimized manner in the real C library).

7.1.1 Hierarchical Abstraction

Designing software with hierarchical abstraction can primarily happen in two ways: *bottom-up*, or *top-down.* In a bottom-up design, you start with the smallest building blocks first, and build successively larger components from them. This approach lends itself well to incremental testing (build a piece, test it, build a piece, test it...). However, **the downside is that you have to be sure that you are building the right blocks, and that they all fit together in the end.**

The other design philosophy is **top-down**. In top-down, you start by designing the highest-level algorithm and determine what other functions you need in support of it. You then proceed to design these functions, until you reach small enough functions that you do not need to abstract out any more pieces. This design approach should sound rather familiar, as it is exactly what we have described to you when discussing how to translate your generalized steps into code. The advantage here is that you know exactly what pieces you need at every step (they are required by the higher-level algorithms that you have already designed), and how they fit together.

The downside to top-down design can arise in testing. If you try to write the whole thing, then test it, you are asking for trouble. However, if you implement your algorithm in a top-down fashion, you may have high-level algorithms that rely on lower-level pieces that do not exist. This problem can be overcome in a couple of ways.

First, you can still test what you have every time you build a complete piece. That is, when you finish a "small block," you can test it. Then once you have built (and tested) all the "small blocks" for a medium sized piece, you can test it. Effectively, you are building and testing your code in a bottom-up fashion, even though you have done the design in a top-down fashion (and may have some partially implemented/untested higher-level algorithms).

The second way you can address this problem is to write test stubs—simple implementations of the smaller pieces which do not actually work in general, but exhibit the right behaviors for the test cases you will use to test your higher-level algorithms. Such test stubs may be as simple as hard coded responses (e.g., `if(input==3) {return 42;}`). You can then test your higher-level functions assuming the lower-level pieces work correctly, before proceeding to implement those pieces.

7.1.2 Naming

The names that you use for identifiers can contribute significantly to (or detract significantly from) the readability of your code. Name your variables, functions, and types to indicate what they mean and/or do. If you can tell what something does from reading its name, you do not have to work (as hard) to figure it out. Of course, at the same time, you should not name your variables in overly long ways that become cumbersome to type.

A good rule of thumb here is that the length of a variable's name should be proportional to the size of its scope and the complexity of its use. It is reasonable to name the counter variable of a for loop i because it has a relatively small scope (one loop) and a simple use (it just counts: you can tell that from reading the for loop where it is declared). Functions and types should therefore generally have relatively descriptive names. They (usually) exist at quite large scopes (all of the functions we have seen so far have had a scope of the entire program), and perform complex tasks.

Some people like naming conventions. We have seen a few naming conventions so far. One is placing a **_t** suffix on the type name (e.g., color_t). Another is writing the names of constants in all capitals. Some programmers like the Hungarian notation scheme, where variable names are prefixed with a sequence of letters indicating their types (e.g., chInput starts with a "ch" indicating its type is char, and while iLength starts with an "i" indicating its type is int).

Another set of conventions arise in how you "glue together" multiple words, as spaces are not allowed in variable names. The two major ways are to use underscores (_) wherever you would want spaces (e.g., num_letters_skipped). The other is to capitalize the first letter of words other than the first (e.g., numLettersSkipped)—this approach is called "inner caps" (also called "camel case"). Either of these methods is fine, though many programmers have a strong preference for one over the other. We note that "inner caps" can also be applied to names that start with a capital letter on the first word by convention—such as class names, which you will see if you take a subsequent class in C++.

8 Questions

1. Difference between returning pointer from func and returning array from function? Where do I need to malloc.