13. *Bitwise Operators and Binary Numbers*

13.1. Introduction

Back in what this author still regards as "the good old days" it was easy to control individual bits in a computer's memory. Computers like the PDP-11/70 shown in Figure 13.1 actually had switches on the front for doing just that. In order to start the computer, the user would carefully set the switches to a particular pattern of ones and zeroes (usually written on a yellowed piece of paper taped to the front of the computer), perhaps repeating this process several times with different patterns, inching the computer along until it could continue on its own.

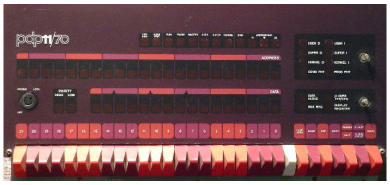


Figure 13.1: The front panel of a DEC PDP-11/70, showing the switches that were used to load a binary starting address into the computer's memory.
Image: Wikimedia Commons

Computers have changed a lot since then, but it's still possible, and sometimes necessary, to switch individual bits on and off. The C programming language provides us with a set of tools for doing that.

It's fun to think about what happens when your program changes the value of a single bit. Each memory cell in a modern computer is smaller than the wavelengths of visible light. When you change the value of a single bit, you're causing a precise physical change in an incredibly tiny object.

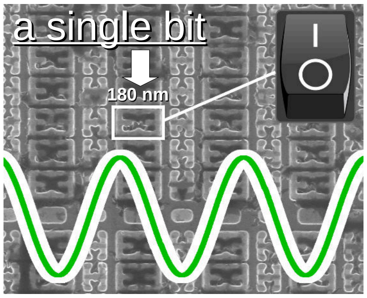


Figure 13.2: An electron microscope's view of a memory chip, showing individual memory cells. Each cell is essentially a little switch, approximately 180 nanometers wide. The wavelength of green light is shown to give an idea of the size.
Image: Wikimedia Commons

Why would you want to the ability to flip individual bits? First of all, bits are the smallest unit of data storage, and by efficiently setting bits you can minimize the amount of space required to store your data on a disk, and the amount of time required to transmit your data from one place to another. Secondly, the CPU in your computer understands how to flip bits on and off, and it can do these operations very quickly. If you can do your calculations by flipping bits instead of more complicated operations like multiplication and division, you can make your program run much faster.

13.2. Binary Numbers

Before we start talking about bits, we first need to understand binary numbers. In our society, we normally write numbers in what might be called “decimal positional” notation. This means that each digit of a number represents some multiple of a power of ten, and the position of the digit indicates which multiple. Take a look at Figure 13.3 for example.

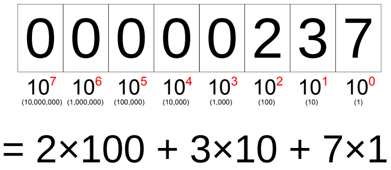


Figure 13.3: The number 237 in decimal positional notation.

The position of each digit tells us how “valuable” it is. You might think of the digits as being like the contents of the bins in a cash register. The rightmost slot is for one-dollar bills, the next is for tens, and the next is for 100s. If we had two $100 bills, three $10 bills, and seven $1 bills, we’d have $237. As we go from right to left, each slot has ten times the value of the preceding one. A number system based on powers of ten is called a “decimal” system from the Greek word *deka*, meaning ten. The `%d` we use when printing integers with `printf` stands for “decimal integer”.

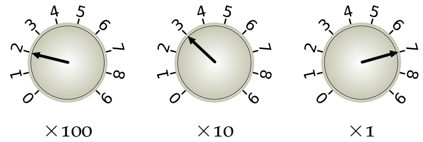


Figure 13.4: To represent a number in our usual decimal notation we need to be able to select from a set of ten digits at each position. We could think of each position as having a dial with ten settings, from zero to nine. The number 237 is shown here.

To make our cash register analogy accurate, we’d have to imagine that as soon as you get ten $1 bills you exchange them for a $10 bill and put that into the next slot to the right, and do a similar operation whenever we get to ten bills in any of the other slots. With this rule, each slot in our number can contain one of ten symbols — 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 — telling us how many “bills” are in that slot. If we go beyond nine, we need to move to the next slot to the right.

We don’t have to use powers of ten, though. We could base our system on any number we want. We probably started using the decimal system because we have ten fingers. If we’d had twelve fingers we might have used a system based on powers of twelve. There are even be some advantages to using such a “duodecimal” system[1]. In fact, vestiges of an old 12-based counting system show up in our daily lives whenever we buy a dozen doughnuts or look at a clock. If we used a 12-based positional system for writing numbers, we’d need twelve possible symbols for each slot.

What if we had to use only two symbols? Then we could write numbers in a “binary” positional notation. (The word binary comes from the Latin *bis*, meaning “twice”.) Each slot in a binary number indicates some number of multiples of two, and each digit is either `0` or `1`. (See Figures 13.5 and 13.6.)

Figure 13.5: In binary notation, each slot can contain only a zero or a one, so instead of the knob in Figure 13.4 you might think of each binary digit as a switch.

Why would we be interested in binary numbers? Because each digit

[1] Take a look at this Numberphile video: https://www.youtube.com/watch?v=U6xJfP7-HCc

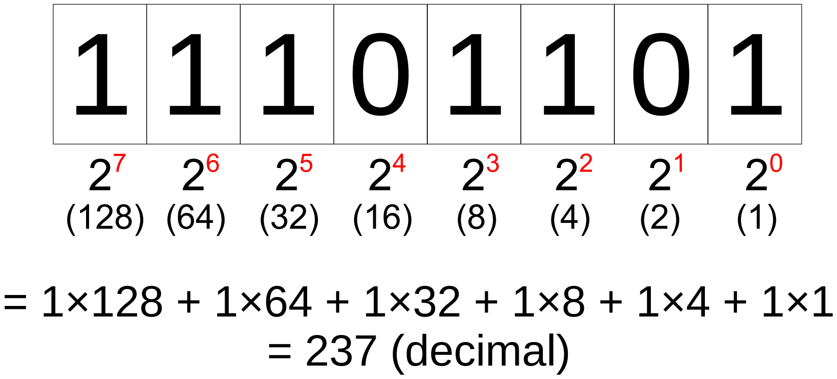


Figure 13.6: The number 237 (decimal) would be written like this in binary.

can be represented by a switch, and it's easy to make switches. We can make switches that are both very small (so that many of them can be packed into a small space) and very fast (meaning that each switch can be turned on or off very quickly)[2]. Once we have some switches, we can use them as the digits of binary numbers.

Each digit in a binary number is called a "bit". You can think of it as a switch that can be flipped to a value of zero or one. Computers usually deal with bits in groups of eight (or multiples of eight). A group of eight bits is called a "byte". (Half a byte, four bits, is sometimes called a "nybble".)

Figure 13.6 shows how the decimal number 237 would be written as a binary number. Each bit represents a power of 2, and can have a value of one or zero. Let's call the right-most bit "bit 0", the next one "bit 1" and so on, with each bit numbered according to the power of 2 that it represents.

Decimal	Binary
1	1
2	10
3	11
8	1000
10	1010
64	1000000
100	1100100
127	1111111
128	10000000
200	11001000
255	11111111

Figure 13.7: Decimal and binary representation of some numbers.

There are a couple of things we might notice right away with this system. First, the bit number increases toward the left. If we were given a bunch of bits, numbered zero through seven, and asked to write them down, we might be inclined to start with bit 0 on the left-hand side of the page, then write the others going left-to-right, as we usually arrange things in English. We write the digits of numbers in the opposite way, though, no matter which base (10, 12, 2, or something else) we use. We don't usually think about this, but it's important to keep it in mind as we start working with the digits of binary numbers.

Second, we might notice that this system can only represent positive integers. We haven't provided any way to represent non-integers or even negative integers. We'll address these concerns soon.

[2] Speed and size are correlated. As switches get smaller, they can also be turned on or off more quickly. That's one reason manufacturers put so much effort into making the already-microscopic components of modern CPUs even smaller.

Figure 13.7 shows the decimal and binary representations of some numbers. Notice that the largest number we can write with eight bits (one byte) is 255. This corresponds to all bits being set to 1. If we want to write larger numbers, we're going to need more bits.

13.3. Bits and Variables

When we write a statement like "number = 42;" in a C program, we're asking the computer to store the value 42 in a variable named number. But what really happens inside the computer? Each variable in our program is just a named section of the computer's memory. When we define a variable named number, the computer reserves a few bits of memory that can be used to store that variable's value.

We can use the sizeof statement[3] (see page 167) to find out how much space has been reserved for a given variable. The space is reported as a number of bytes (8-bit chunks). For example:

```
#include <stdio.h>
int main () {
  int i;
  double d;
  char c;

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) )
  printf ("Size of d is %d bytes.\n", (int)sizeof( d ) )
  printf ("Size of c is %d bytes.\n", (int)sizeof( c ) )
}
```

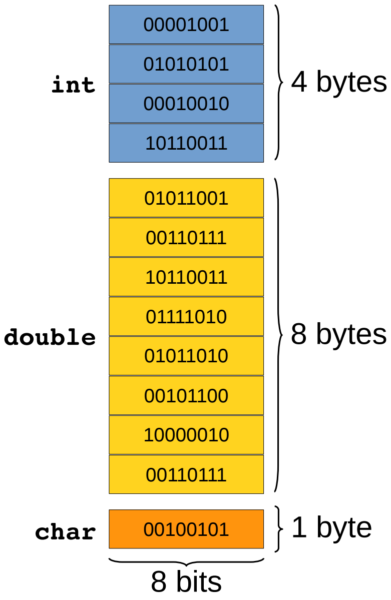


Figure 13.8: Different types of variable use different amounts of storage.

If we compiled and ran this program, we'd see something like this:

```
Size of i is 4 bytes.
Size of d is 8 bytes.
Size of c is 1 bytes.
```

As you can see, different types of variable will generally have different amounts of space. The C language standards don't specify exactly how big the storage space for each type of variable should be, so these numbers may vary from one C compiler to another, but the values shown above are typical.

If the program tells us that int variables are allocated 4 bytes (32 bits) of storage space, what's the biggest number we can store in an int? We might think it would be a binary number with 32 ones, like this:

[3] The value returned by sizeof isn't actually an int, so to keep printf from complaining we force the value to be an int by putting "(int)" in front of it.

```
11111111111111111111111111111111
```

which is 4,294,967,295 (a little over 4 billion) in decimal representation. But if we reached that conclusion we'd be forgetting that `int` variables can hold either positive or negative numbers. Somehow, we've got to reserve at least one bit to indicate whether the number is positive or negative.

It would be great if we could peek at the bits inside an actual variable, to see how they're arranged. Let's start by looking at the smallest kind of variable identified by the program above: the `char` variable, which has only one byte (8 bits) of storage.

13.4. Character/Number Equivalence

You might recall from Chapter 8 that each character is represented by a numerical ASCII code. Figure 13.9 shows the characters corresponding to the ASCII codes from zero to 127. You'll see that the table includes all of the letters and numbers you've come to know and love, and many of the other symbols on your keyboard. For example, the upper case letter "`A`" is character number 65.

There are also some non-printable characters. For example, the NUL character (which we sometimes write as `\0`) is there as character number zero. Character number 10 is the newline (also called linefeed) character, `\n`. Characters like newline, horizontal tab, formfeed, and space are used the move around on your screen, but don't actually print anything. Some of the non-printing characters are extremely weird things like `DLE`, `NAK`, `SYN`, and `ETB`. These are used to control the flow of ASCII data sent across a communications line.

One important fact is that, in C, a character and its ASCII number are almost entirely interchangeable. The following program prints the number 65 twice. The first time, it tells `printf` to print it as a number. The second time, it tells `printf` to print it as a character:

```
#include <stdio.h>
int main () {
  printf ( "As a number it's %d\n", 65 );
  printf ( "As a character it's %c\n", 65 );
}
```

The only substantial difference in the two `printf` lines is that one uses

ASCII		ASCII		ASCII		ASCII	
Number	Character	Number	Character	Number	Character	Number	Character
0	NUL '\0'	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL '\a' (bell)	39	'	71	G	103	g
8	BS '\b' (backspace)	40	(	72	H	104	h
9	HT '\t' (horizontal tab)	41	)	73	I	105	i
10	LF '\n' (new line)	42	*	74	J	106	j
11	VT '\v' (vertical tab)	43	+	75	K	107	k
12	FF '\f' (form feed)	44	,	76	L	108	l
13	CR '\r' (carriage ret)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative ack.)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. blk)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\ '\\'	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Figure 13.9: ASCII codes between zero and 127 and their corresponding characters.

`%d` as a placeholder and the other uses `%c`. The `%d` tells `printf` to treat 65 as a number, and `%c` says to treat it as a character. If we ran the program, we'd see:

```
As a number it's 65
As a character it's A
```

Perhaps even more surprisingly, we'd see the same results if we wrote the program this way:

```
#include <stdio.h>
int main () {
  printf ( "As a number it's %d\n", 'A');
  printf ( "As a character it's %c\n", 'A' );
}
```

As far as C is concerned, `'A'` is exactly equivalent to `65`.

Exercise 60: Character Building

Write a program named `charnum.cpp` that uses a `for` loop to print out the numbers from 33 to 126, inclusive, and the ASCII character that corresponds to each number. The program's output should be two columns, with the first column being the number and the second column its corresponding ASCII character. Note that, because of the equivalence of characters and numbers in C, the loop can either go from `33` to `126` or from `'!'` to `'~'` (see Figure 13.9).

If we could look directly at the 8 bits that store a character variable's value, we'd see that they're just a binary representation of a character's ASCII number. For example, ’A’ is character number 65, which is 01000001 when expressed as an 8-bit binary number.

13.5. A Simple Encryption Scheme (`rot13`)

Imagine that we took all of the lower-case ASCII letters and arranged them in a circle, as in Figure 13.10. Below each letter is shown its ASCII character number. There are 26 letters, so if we start at any letter then move 13 spaces around the circle we'll find ourselves at a different letter that's exactly on the opposite side of the circle.

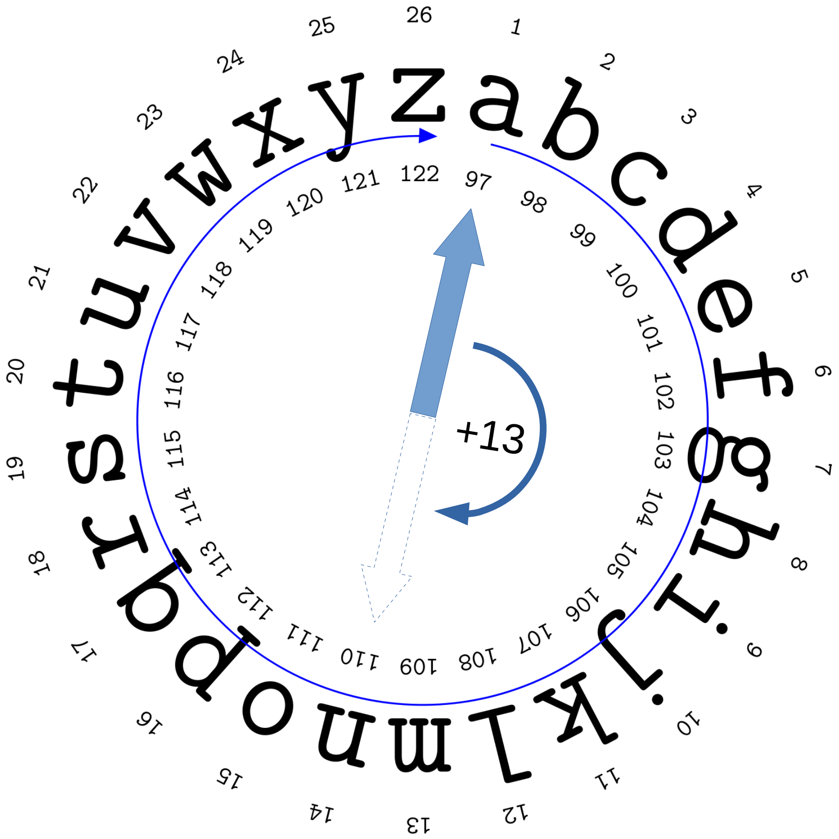


Figure 13.10: Adding 13 to the value `'a'` moves halfway around the circle to `'n'`. Adding 13 again would bring us back to `'a'`.

You could use this as a simple way of "encrypting" a message. Start out by writing down your message, then replace each letter with a different one, halfway around the circle. The person receiving your encrypted message could easily decode it (assuming he or she knows the code!) by just going 13 more spaces around the circle, to get back to the original letter.

This simple encryption scheme is called `rot13`, since it picks a replacement letter by *rotating* 13 spaces around the circle. In the early days of the Internet, `rot13` was often used to obscure text. For example, if you posted a movie review that contained spoilers, you might use `rot13` to encrypt those parts. Anyone who really wanted to read them could decrypt the text, but if you didn't want to know how the movie ended, you'd be in no danger of having it accidentally spoiled for you.

Fortunately, it's easy to write a program that can understand `rot13`. Since moving by 13 spaces can be used to either encrypt or decrypt a message, you can write one program that will work for either task. Give it some plain text, and it will encrypt it. Give it some encrypted text, and it will give you back the original message.

Writing such a program is particularly easy in C, since we're free to use characters and their numbers interchangeably. Program 13.1 shows one way to do it.

Program 13.1: rot13.cpp

```
#include <stdio.h>
int main () {
  char letter, position, newposition;
  while ( scanf("%c", &letter) != EOF ) {
    if ( letter >= 'a' && letter <= 'z' ) {
      position = letter - 'a';
      newposition = (position + 13)%26;
      letter = 'a' + newposition;
    }
    printf ("%c",letter);
  }
}
```

Ony change lower-case letters.

Shift by 13 letters.

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

The program uses a "`while`" loop to read characters, one at a time. Each time it reads a character, it checks to see whether this is one of the lower-case characters `'a'` through `'z'`. These are the only letters that are part of the circle that we're using for encryption (see Figure 13.10). Any other characters will be left alone. Notice that we don't have to switch between the character's name (like `'a'`) and its ASCII number (like `97`). C takes care of this for us automatically.

Whenever we find a lower-case letter, we then identify the letter that's opposite it on our letter circle. This circle of letters is a lot like a clock. Remember that in Chapter 4 we talked about clocks, and said that they're an example of modular arithmetic. When a clock's hand goes past twelve, it starts over again at one. We say that the *modulus* of the clock is twelve. If we set a clock's hour hand at 3 and wait 16 hours we'll find that the hand now points to 7. In terms of modular arithmetic, we'd say that `(3 + 16)%12 = 7`, since `7` is the remainder obtained after dividing `3+16` by `12`.

Our circle of letters has 26 positions instead of 12, so it has a modulus of 26. But there's also another difference: our letters don't start at 1. Instead, they start with 'a' (or, equivalently, 97 in C). When we look at a clock, the "12" position is both the beginning and the end of the circle of numbers. When we do modular arithmetic on a clock, we assume that the clock numbers tell us how many hours away from 12 we are. We could imagine that, when the clock's hour hand gets to 12 it's briefly twelve hours away from where it started, then as is passes twelve it's instantaneously back at zero. In our clock's modulo-12 counting system, zero is just the flip side of 12.

So, for example, if we were given the letter 'y' and wanted to find out which letter was on the opposite side of the circle, we first find how far we are from 'a'. This is just 'y'- 'a'. Then we add 13 to this distance and find the remainder after dividing by 26:

```
( 'y' - 'a' + 13 ) % 26
```

The remainder tells us how far away from 'a' we'll be when we move to the letter on the opposite side of the circle. To find out this letter's ASCII number, we just add 'a' to it. That's what Program 13.1 does.

Exercise 61: A Lot of Rot

Create and compile Program 13.1. Run the program and type some text. When you press Enter or Return, the program should print the rot13-encrypted version of your text. For example, if you type *"this is a test"* the program will tell you that the encrypted version of this is *"guvf vf n grfg"*. Press Ctrl-D to exit the program. If you run the program again and type *"guvf vf n grfg"*, the program will translate it back into *"this is a test"*.

If you have a whole file full of text you want to encrypt (a file named secretmessage.txt, for example), you can rot13-encrypt the whole thing by typing this command:

```
cat secretmessage.txt | ./rot13
```

Amaze your friends! Confuse your enemies!

Now that we have some understanding of how character variables are stored, it's natural to wonder how other kinds of variables stored. It would be nice if we could examine them bit by bit to find out. We can do this, but first we'll need to learn a little about C's "bitwise operators". In particular, we'll need to learn about "bitwise shift" and "bitwise and".

13.6. The Shift Operators

It should be obvious that the following program will print "1". (Try it yourself if you don't believe me!)

```
#include <stdio.h>
int main () {
        printf ( "%d\n", 1 );
}
```

but what would the following do?:

```
#include <stdio.h>
int main () {
        printf ( "%d\n", 1<<3 );
}
```

You might be surprised to find that it prints "8". What's going on here? What does that "<<3" do?

Figure 13.11: If you've never used a typewriter, you might not know that the shift key originally shifted part of the typewriter up or down, to get access to upper-case letters. This reduced the number of keys that were needed. Before the shift key was invented typewriters either had separate keys for each upper and lower case letter, or they could only type in upper (or lower) case.

Image: Wikimedia Commons

Let's think about binary numbers again, and imagine that we have an 8-bit binary number representing the value "1", like this:

0	0	0	0	0	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)

Figure 13.12: The number 1 written as an 8-bit binary number. As with decimal numbers, extra zeros on the left-hand side don't matter. We can write 1 or 01 or 00001, and they all mean the same thing.

Referring to Figures 13.12 and 13.13, we see that we could write any power of 2 in binary form by just writing down a lot of zeros and putting a one in the slot corresponding to the desired power. So, 2 (decimal) is written as 10 (binary), 4 is written as 100, 8 is 1000, and so on. If we started with a one in the first slot, we could imagine generating all of the other powers of 2 by just shifting the one to the left by some number of slots. (See Figure 13.13.)

Power	Decimal	Binary
2^0	1	00000001
2^1	2	00000010
2^2	4	00000100
2^3	8	00001000
2^4	16	00010000
2^5	32	00100000
2^6	64	01000000
2^7	128	10000000

Figure 13.13: Powers of 2 written as 8-bit binary numbers.

That's exactly what C's << operator does. It shifts all of the bits in a number toward the left by a given amount. Bits shifted past the left-hand edge are lost, and empty slots on the right-hand side are filled in with zeros. In the program above, 1<<3 means "Start with the number 1 in binary, then shift all of the bits to the left by three spaces." As you can see from Figure 13.14, that would give you 8, and that's what the program above prints out.

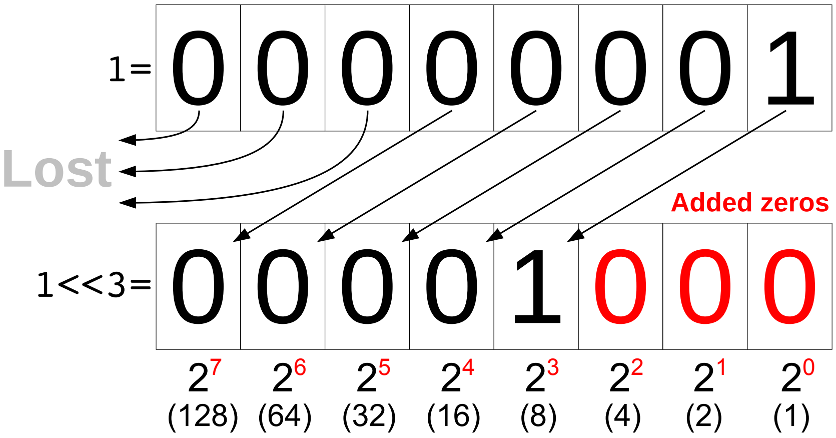


Figure 13.14: Starting with the 8-bit binary representationn of the number `1`, we can shift all of the digits three spaces to the left to get 8 by saying "`1<<3`".

It's interesting to think about what happens to a number when we shift its digits to the left like this. Shifting the digits of a *decimal* number to the left is equivalent to multiplying that number by some power of ten. Ten times 237 is 2370. One hundred times 237 is 23700. Similarly, if we shift the digits of a *binary* number to the left, we multiply it by a power of *two*. For example, `<<1` multiplies the number by two, `<<2` multiplies by four, and `<<3` multiplies by eight.

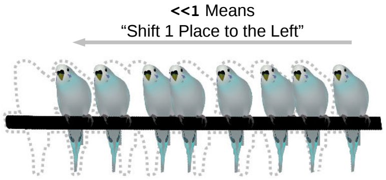


Figure 13.15: The "left-shift" operator, `<<`, shifts each bit leftward by a given amount. Like birds on a perch, bits that get shifted too far "fall off".

Image: OpenClipart

What do we mean when we say that bits shifted "past the edge" are lost? Where is the edge? As we saw above when we were playing with the `sizeof` statement, each variable in a program has some amount of storage space allocated to it. The bits that represent that variable's value are stored in that space. We can shift those bits around, but the space is finite, and if we shift too far we lose some information[4]. For simplicity, many of the figures in this chapter will assume that we only have eight bits available (one byte), but in reality we'll usualy have more space (four or eight bytes) for each of our numerical variables.

Figure 13.17 shows some examples that start with a different 8-bit binary number (237 in decimal notation). Each time we shift left, some bits drop off the edge and are irretrievably lost. If we shift far enough, as in the bottom case, all of the original bits are lost, and we're left with only zeros. If you only have eight bits to store your number in, you're in trouble if you shove things over by eight spaces.

Note that it's perfectly OK to shift the bits by 0 spaces, even though that doesn't change anything. The expression `1<<0` is just the same as `1`. As we'll see, this is sometimes convenient.

[4] As we'll see, this also means that there's a maximum number that can be stored in any variable. What that number is will depend on the variable's type.

Not surprisingly, there's also a "right-shift" operator, >>. Figure 13.18 shows some examples. It works just like the left-shift operator, but moves things in the opposite direction. Don't think that you can use a right-shift to recover bits that have dropped off the edge due to a left-shift, though. That doesn't work[5]. Any bits that are dropped are gone forever.

Exercise 62: Bit Drill

Let's get some practice with the bit-shift operators. Write a program named `bitdrill.cpp` that loops through values of `i` from 0 through 31 and prints `i` and `1<<i` for each value. You might see a surprise for `1<<31` !

When you printed the value of `1<<i` you probably used "`%d`" in your `printf` statement. Try changing this to "`%u`", then recompile your program and run it again. Does the value of `1<<31` change? We'll explain why this happens a little later.

One final note about the exercise above: if you look at Figure 13.13 you can see that the binary representation of each of the numbers you generated (each `1<<i`) would be mostly zeros except for a single `1` in bit number `i`. Apparently, we can use `1<<i` to create a number that just has one particular bit turned "on". This fact will come in handy in the next section.

Okay, so we see that it's possible to shift bits left and right. What good does that do us? Remember that our goal was to be able to see how the bits are really arranged when we store a number in a variable. Bit-shifting is one of the tools we'll need to do that, but we'll also need another tool: the "bitwise *and*". We'll get to that in a later section, but first we need to learn a little more about how a computer stores numbers.

Figure 13.16: Bit drill, meet drill bits.

Image: Wikimedia Commons

[5] More accurately, you can't depend on it.

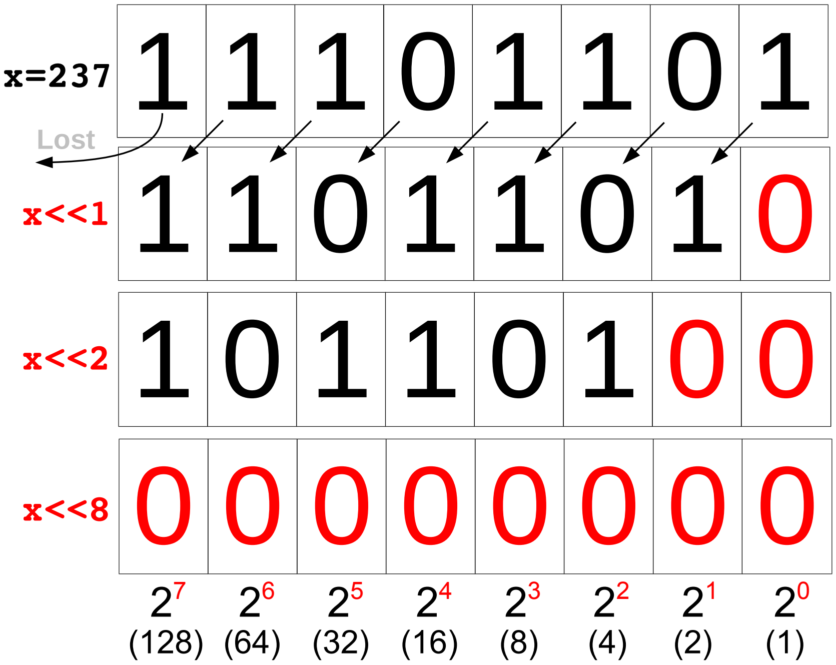


Figure 13.17: If you shift far enough, all of the original bits are lost.

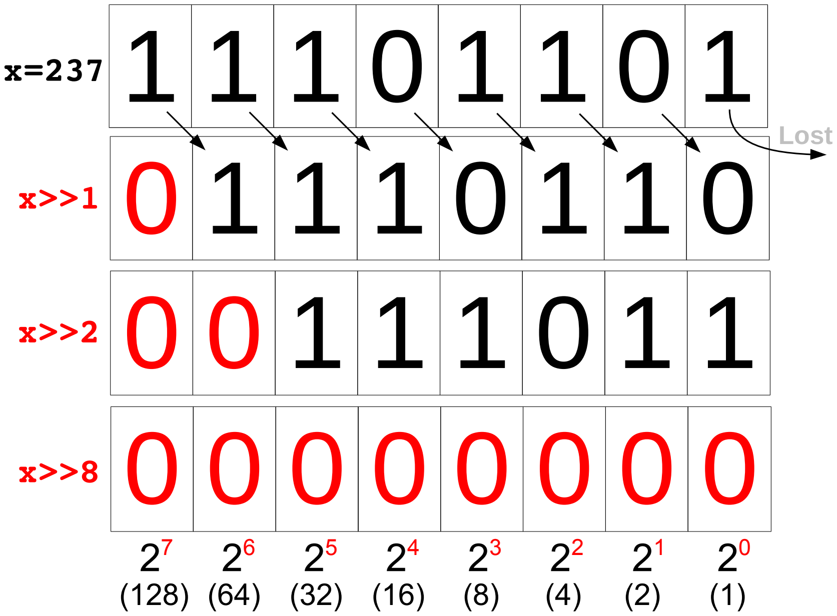


Figure 13.18: The right-shift operator, >>, shifts bits rightward by a given number of slots.

13.7. Signed and Unsigned Integers

In Section 13.6 you might have been surprised by the output of your `bitdrill.cpp` program. If you used "`%d`" when printing the values you probably saw that the program's output looked like Figure 13.19.

Why is the last number negative? To figure it out, let's start by considering what this number's bits look like. Figure 13.20 shows the left-most few bits of this 32-bit number (all of the other bits are zero). We might expect this number to be equal to 2^{31}, which is a little over two billion.

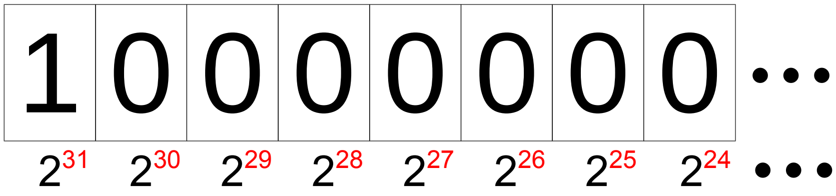


Figure 13.20: The left-most bits of the 32-bit number `1<<31`.

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
11 2048
12 4096
13 8192
14 16384
15 32768
16 65536
17 131072
18 262144
19 524288
20 1048576
21 2097152
22 4194304
23 8388608
24 16777216
25 33554432
26 67108864
27 134217728
28 268435456
29 536870912
30 1073741824
31 -2147483648
```

Figure 13.19: The output of the `bitdrill.cpp` program from Section 13.6, using `%d` to print the numbers.

The explanation has to do with the way the computer stores negative numbers. In order to store both negative and positive numbers we need to reserve at least one bit that will indicate the number's sign. That's part of the explanation for what we see in our program's output, but it's clearly not the whole story. If the top-most bit just indicated the sign, then our last number would be "-0", not $-2,147,483,648$.

To make computations faster, computers actually use a slightly more complicated way of storing negative integers called "two's complement" notation. The two's complement of a binary number can be formed by:

1. Flipping every `1` to a `0`, and every `0` to a `1`, and

2. Adding `1` to the result.

This might seem pointless, but it has a distinct advantage: it lets the computer add numbers together without needing to check their signs. Adding the two's complement of a number turns out to work just the same as *subtracting* that number.

If we just reserved one bit as a "sign bit", we'd always need to check the sign when adding numbers, and then decide whether to add or subtract. This would add extra steps to our calculation, slowing things down. By using two's complement notation we avoid this. As it turns out, the bit pattern we produced by doing `1«31` is the two's complement of $2,147,483,648$, so to the computer it represents the negative of that number.

Using two's complement notation for negative numbers, a 32-bit integer can hold any number between $-2,147,483,648$ and $2,147,483,647$. Any number that has `1` as its left-most bit is assumed to be negative, and interpreted as the two's complement of the value. Figure 13.21 shows the binary representation of some typical numbers.

Decimal	Binary, 32 bits, Signed
0	00000000.00000000.00000000.00000000
1	00000000.00000000.00000000.00000001
32	00000000.00000000.00000000.00100000
256	00000000.00000000.00000001.00000000
1 billion	00111011.10011010.11001010.00000000
2 billion	01110111.00110101.10010100.00000000
2,147,483,647	01111111.11111111.11111111.11111111
-2,147,483,648	10000000.00000000.00000000.00000000
-256	11111111.11111111.11111111.00000000
-32	11111111.11111111.11111111.11100000
-1	11111111.11111111.11111111.11111111

Figure 13.21: Some representative signed 32-bit integers. Groups of eight bits (1 byte) have been separated by dots for clarity.

The table is arranged in order of increasing binary numbers, from all bits "off" to all bits "on". Notice that when we go past the biggest positive number (a little over 2 billion) the value jumps immediately to the smallest negative number. The value when all bits are "on" is -1.

What if we know that all of our values are going to be positive? Are we still limited to a maximum value of 2,147,483,647? It seems a shame to reserve part of the available range for negative numbers when we know we won't have any.

If you try changing "`%d`" into "`%u`" in your `bitdrill.cpp` program, you'll see that the last thing it prints is now a positive number:

```
31 2147483648
```

The "`%u`" format specifier tells the program to interpret the binary data as an "unsigned integer". Unsigned integers don't wrap around to negative values halfway through their range. Instead, they start at zero and just keep getting bigger. The biggest value that can be stored in a 32-bit unsigned integer is $4,294,967,295$ (a little over 4 billion), which in this case is represented by a 32 "on" bits. Figure 13.23 shows the bit patterns that correspond to some representative unsigned integers.

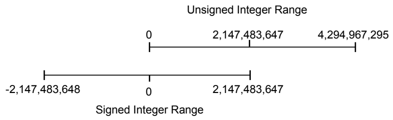


Figure 13.22: A visual comparision of the range of `int` and `unsigned int` variables.

Decimal	Binary, 32 bits, Unsigned
0	`00000000.00000000.00000000.00000000`
1	`00000000.00000000.00000000.00000001`
32	`00000000.00000000.00000000.00100000`
256	`00000000.00000000.00000001.00000000`
1 billion	`00111011.10011010.11001010.00000000`
2 billion	`01110111.00110101.10010100.00000000`
2,147,483,647	`01111111.11111111.11111111.11111111`
2,147,483,648	`10000000.00000000.00000000.00000000`
3 billion	`10110010.11010000.01011110.00000000`
4 billion	`11101110.01101011.00101000.00000000`
4,294,967,295	`11111111.11111111.11111111.11111111`

Figure 13.23: Some representative *un*-signed 32-bit integers. Groups of eight bits (1 byte) have been separated by dots for clarity.

The `int` variables we've used so far are for holding signed integers. If you know you won't need negative numbers, you can define a variable as "`unsigned int`", and use `%u` as a placeholder when reading or writing its value.

Notice that an *unsigned* integer uses the same pattern of bits to represent $4,294,967,295$ as the pattern that's used to represent -1 for *signed* integers. It's worth pausing to think about what this means. If we see 32 "on" bits in the computer's memory, we don't know whether it represents -1 or $4,294,967,295$. It could even represent other values. 32 bits is the same size as four 8-bit `char` variables, so these bits could represent an array of four characters. It's not enough to know what binary data is stored in a variable's memory location. We also need to know the variable's *type*. The type tells us how to interpret the data we see.

Program 13.2 can be used to illustrate the difference between `int` variables and `unsigned int` variables.

Program 13.2: unsigned.cpp

```
#include <stdio.h>
int main () {
  int i;
  unsigned int j;

  printf ( "Enter an integer: " );
  scanf ( "%d", &i );
  printf ( "You entered %d\n", i );

  printf ( "Enter an integer: " );
  scanf ( "%u", &j );
  printf ( "You entered %u\n", j );
}
```

If you ran this program and gave it `4000000000` (4 billion) each time it asked you for a number, you'd see the following:

```
Enter an integer: 4000000000
You entered -294967296
Enter an integer: 4000000000
You entered 4000000000
```

Sometimes we can choose which way we want to display the same data by just changing the placeholder in our `printf` statement, as we did when we changed `%d` to `%u` in the `bitdrill.cpp` program. There are subtle rules that control the way C converts data from one type to another, though, so be careful, especially when comparing variables of different types. Take a look at the following program, for example[6]:

Program 13.3: plusminus.cpp

```
#include <stdio.h>
int main() {
  unsigned int plus_one = 1;
  int minus_one = -1;

  if( plus_one < minus_one ) {
    printf("1 < -1 \n");
  } else {
    printf("Math isn't broken.\n");
  }
}
```

If we compiled this program, *g++* would give us a warning about comparing signed and unsigned numbers, but it would still create a program we could run. If we ran the program, it would erroneously tell us that 1 is less than -1. That's because in this situation *g++* assumes we want to compare the two numbers as though they were both unsigned integers. As we saw above, the signed integer -1 is the same as the unsigned integer $4,294,967,295$.

Notice that $4,294,967,295$ is $2^{32} - 1$. In general, if we have n bits for storing an unsigned integer, the biggest number we can store will be $2^n - 1$.

One final note: If you wanted to, you could explicitly define `int` variables as `signed int`, to make clear that they're different from `unsigned int`. You don't have to, though. If you don't specify whether the variable is signed or unsigned, the default is `signed`.

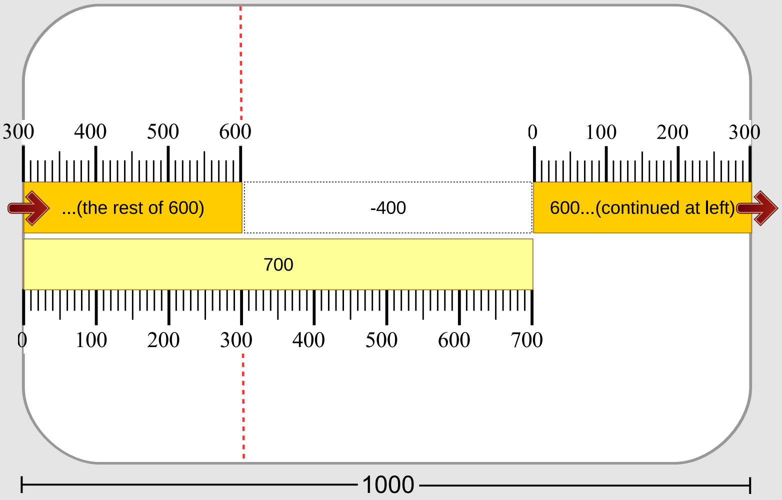
[6] This example is adapted from an excellent, but rather technical, explanation by Ozgur Ozcitak at StackOverflow.

But what about...?

What does "two's complement" mean, anyway? And how can you possibly subtract by adding? The answers depend on the fact that numbers in a computer always have a limited number of digits.

For example, an `unsigned int` variable might have 32 bits — 32 binary digits – in the computer's memory. If we try to put too large a number into that space, the upper digits of the number will be lost. If the biggest number we can store is 4,294,967,295 (expressed in binary, of course) but we try to put in 4,294,967,296, we'll see that our variable ends up containing the value zero! 4,294,967,297 would give us 1, 4,294,967,298 would give us 2, and so forth. It's like the numbers get to the maximum value and then wrap around to the beginning again.

This is analogous to an old-fashioned arcade game like Asteroids, where characters that went off the right side of the screen reappeared on the left side.



It might be easier to understand if we look at a "ten's complement" example, where we work in the more-familiar base 10. Imagine that we want to subtract 400 from 700. Let's start by putting a 700-pixel-long bar on the screen of a 1,000-pixel-wide arcade console, as in the figure above.

If we wanted to subtract 400 pixels from the bar's length, we could

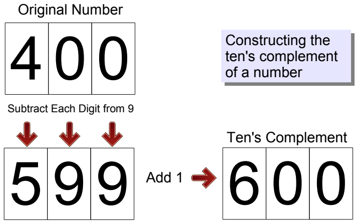

Original Number
400
Constructing the ten's complement of a number
Subtract Each Digit from 9
599
Add 1
Ten's Complement
600

Figure 13.24: The "ten's complement" of a number can be formed by subtracting each digit from 9 and then adding 1 to the resulting number.

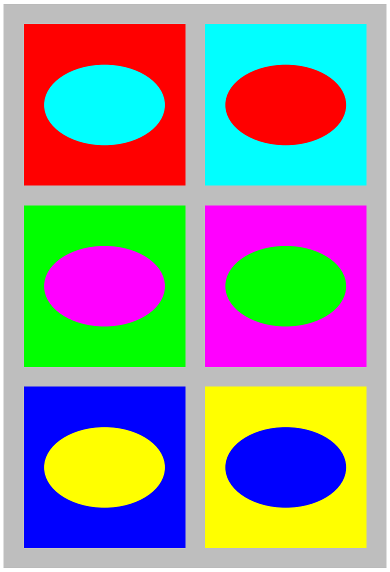


Figure 13.25: "Complementary colors" are pairs of colors that combine to form white (or black in some color systems). Each is the "complement" of the other, meaning that each supplies what the other lacks. Similarly, the ten's complement (or two's complement) of a number supplies what the number lacks to become a power of ten (or two).

just move to the left by that amount and chop the bar off at the vertical dashed line. This is the same as subtracting 400 pixels from the bar's length.

Alternatively, though, we could find the ten's complement of 400 and *add* that much to the bar's length, causing it to wrap around when it bumps into the edge of the screen. The ten's complement of the number is just the difference between the number and the total width of the screen. That difference turns out to be 600 pixels in this example. As you can see, going forward a distance of 600 pixels (which wraps us around to the other side of the screen) leaves us at the same dashed vertical line. We end up at the same place as if we'd subtracted 400 pixels.

For 3-digit numbers the ten's complement is the amount you need to add to get to 1,000, because you can only hold numbers up to 999 in three digits. After that, the numbers roll over like an odometer to 000. For 4-digit numbers the limit would be 9,999, and the ten's complement would be the amount you need to add to get 10,000. In general, for n digits the ten's complement of a number x is $10^n - x$. Similarly, the n-digit two's complement of x would be $2^n - x$.

One way to find the ten's complement of a number is to subtract each of the number's digits from 9, and then add 1 to the result, as shown in Figure 13.24. This might seem roundabout when we could just subtract the number from 1,000 (for 3 digits) and be done with it, but it's useful when working with binary numbers. In base 2, instead of subtracting from 9, you just flip the value of each bit. This is something the computer can do very quickly. It turns out that flipping the bits and adding 1 to the result is much faster than finding the twos complement any other way.

Figure 13.26: Many cartoon characters also have four digits. This is Felix the Cat, one of the author's favorites. He first appeared in *Feline Follies* in 1919. You can watch it at archive.org.

Image: Wikimedia Commons

13.8. Bitwise Logic

Way back in Chapter 3 we learned about the "and" and "or" (`&&` and `||`) logical operators that we often use inside "`if`" statements. For example, the statement:

```
if ( a<3 && b>4 )
```

can be read as "if `a` is less than three **and** `b` is greater than four", whereas the statement:

```
if ( c==1 || d<7 )
```

means "if `c` equals one **or** `d` is less than seven".

The `&&` operator compares two expressions and tells us whether *both* expressions are true. The `||` operator compares two expressions and tells us whether *at least one* of them is true.

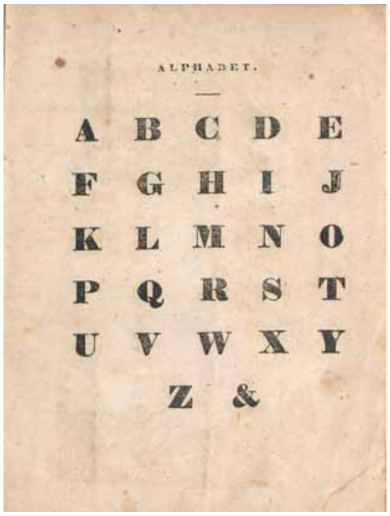
It turns out that C has a set of similar operators for comparing individual bits of binary numbers. These operators are `&` and `|`. (Note that, unlike the logical operators we've used before, these new operators aren't doubled. Each is just a single character.) These operators treat "`1`" as *true* and "`0`" as *false*.

Consider the example in Figure 13.29, which sets `z` equal to `x&y`. As you can see, the `&` operator compares two numbers, bit by bit, looking for places where the bits of both numbers are set to "`1`". If both bits are "`1`", then the resulting bit is "`1`", otherwise, it's "`0`".

We can summarize the behavior of the `&` operator with a *truth table*, as in Figure 13.30. This shows the value that a bit of `x&y` will have if the corresponding bits of `x` and `y` have the values given in the shaded squares. A bit of `x&y` is only true (has a value of "`1`") if the corresponding bits of `x` and `y` are both true.

Figure 13.27: The mathematics of logic is called "Boolean Algebra" after its inventor, George Boole, a 19th Century British mathematician. Boole studied how true and false assertions could be chained together using "ands" and "ors" to trace a mathematically rigorous path leading to a specific conclusion. His work was the foundation of modern computer science.

Image: Wikimedia Commons



ALPHABET.

A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
Z &

Figure 13.28: The ampersand, `&`, once had the distinction of being a member of the alphabet, as seen in this page from the 1863 "Dixie Primer, for the Little Folks". When reciting the alphabet the "little folks" would end by saying "X, Y, Z, and *per se* and", the slurring of which gave rise to the character's name. The character itself is a combination of the letters *Et*, the Latin word for "and".

Image: Wikimedia Commons

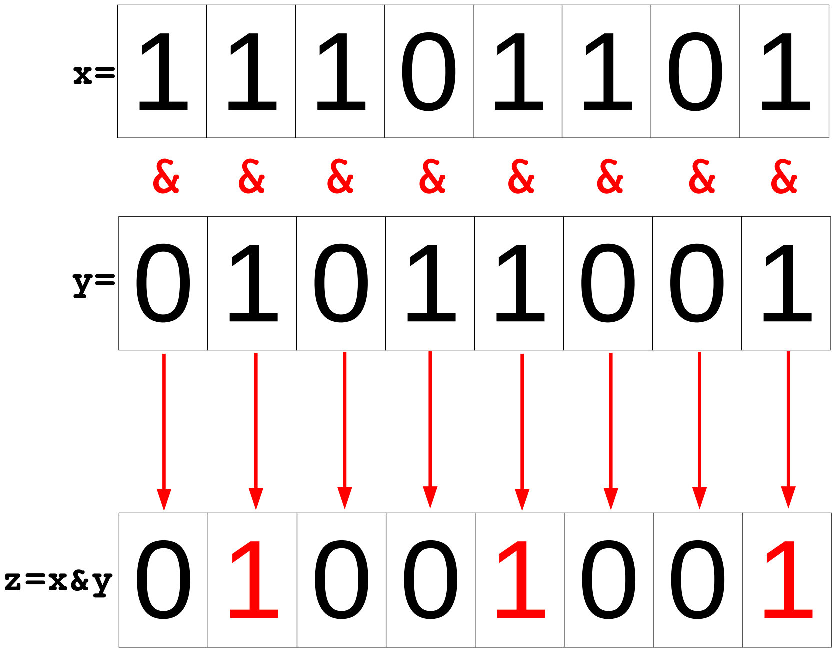
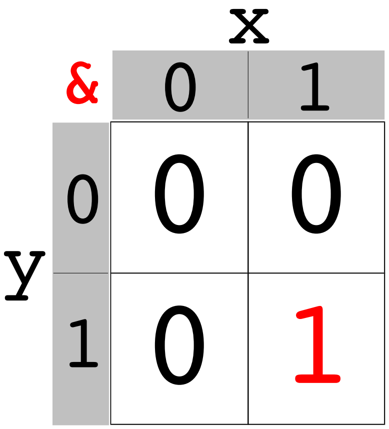


Figure 13.29: The result of a bitwise "and" of two binary numbers is another number that only has a `1` in the spots where *both* of the original numbers had a `1`.

The `&` operator is valuable because it can be used to find out whether a particular bit of a given number has a value of `0` or `1`. Let's try it out by examining the bits in the number 237. You can see this number's binary representation at the top of Figure 13.31.



&	x = 0	x = 1
y = 0	0	0
y = 1	0	1

Figure 13.30: Truth table for `&`, the "bitwise and" operator. The result is only true if both x and y are true.

Now let's constuct another binary number. Recall that we can make a binary number with a single `1` in any slot we choose by starting with `1` and shifting with the `<<` operator. The middle line of Figure 13.31 shows a number that has a single `1` in slot number 3. The number is `1<<3`. That's equal to 8 in decimal notation, but all we really care about is the fact that it's all zeros except for a `1` in bit number 3. We might call this number a "mask" because (as we'll see) we're going to use it to hide all but one bit of the first number.

If we "bitwise and" these two numbers together, the result is what's shown in the bottom row of Figure 13.31. The `1` in this row is telling us that bit number 3 is "on" in the number we're testing (the top row).

We can use a other masks to test other bits. For example, Figure 13.32 shows how we could use `1<<4` as a mask to test bit number 4 of our number. In this case, the bottom row shows all zeros, indicating that bit number 4 is "off".

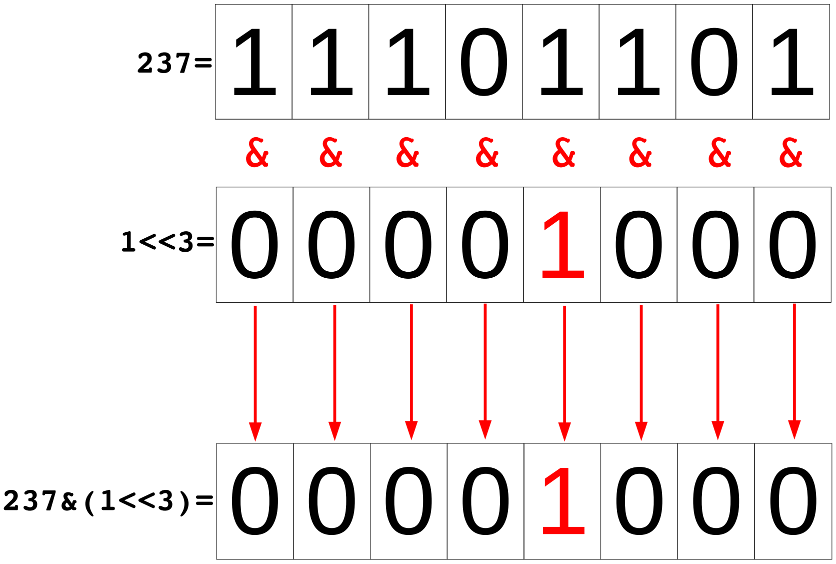


Figure 13.31: Testing bit 3 of the number 237. The result shows that this bit is "on".

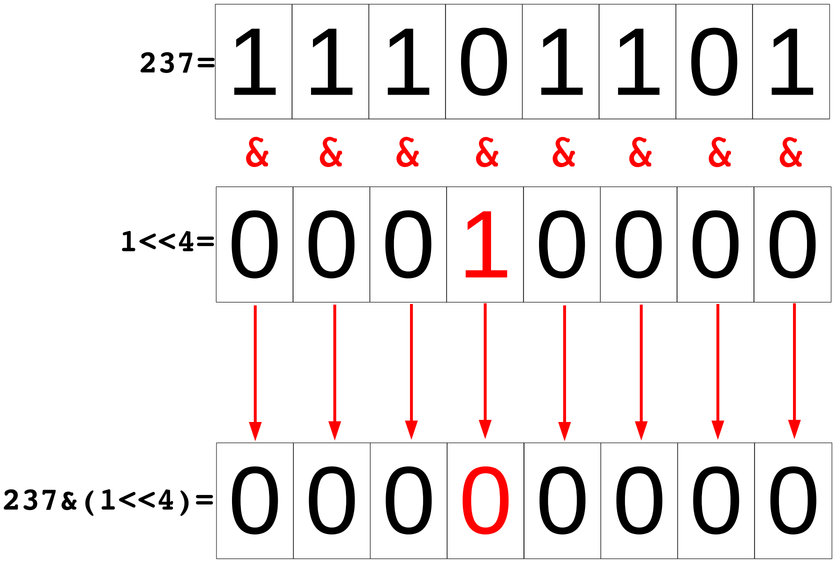


Figure 13.32: Testing bit 4 of the number 237. The result shows that this bit is "off".

All of this leads us to the general rule that we can test bit number `i` of a number, `n`, by checking the value of `n&(1<<i)`. If this value is zero, then the bit is "off". If the value is non-zero, then the bit is "on". We'll use this fact in the next section, when we start examining the inner workings of variables.

Before we go on, though, let's look at another useful bitwise logic operator: the "bitwise or" operator, `|`. Figure 13.33 shows how this operator works. If we have two values, `x` and `y`, and combine them with the `|`operator to get a new value, `z = x|y`, the result has a `1` in any slot where either `x` or `y` had a `1`. The truth table for this operator is shown in Figure 13.34.

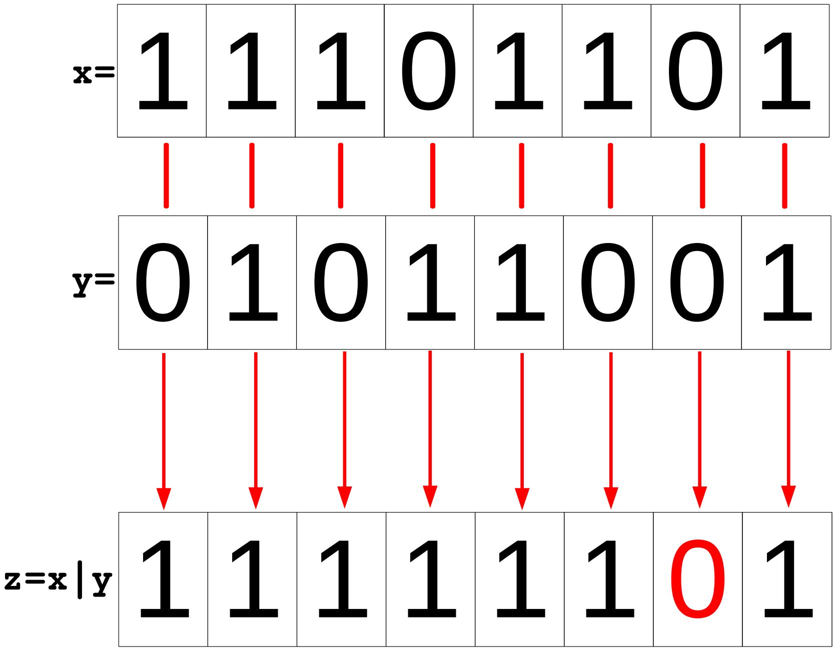
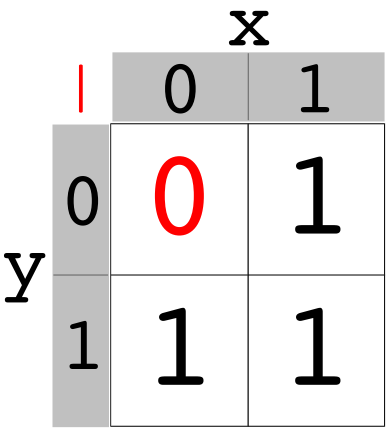


Figure 13.33: The result of a bitwise "or" of two binary numbers is another number that has a `1` in the spots where *either* of the original numbers had a `1`.



|	x = 0	x = 1
y = 0	0	1
y = 1	1	1

Figure 13.34: Truth table for `|`, the "bitwise or" operator. The result is true wherever either x or y is true.

Another bitwise operator we should talk about is the "exclusive or" (often called "xor") operator, ^. Unlike the "or" operator, `z = x^y` gives a `1` in any position where one *and only one* of the original numbers has a `1`. Figure 13.35 illustrates this. As you can see in the bottom row, there's a zero wherever both bits are "off", but there are also zeros whenever both bits are "on". The truth table for the ^operator is shown in Figure 13.36.

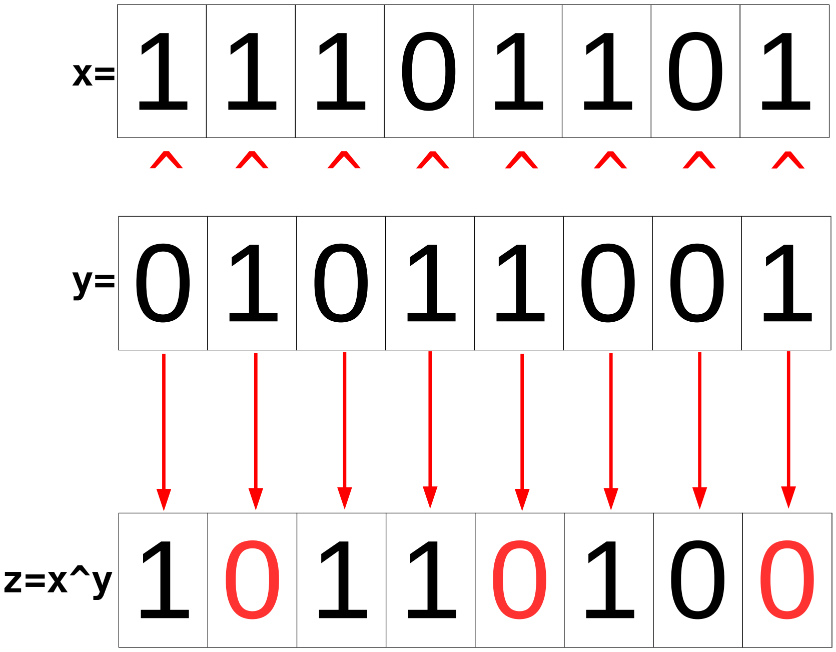
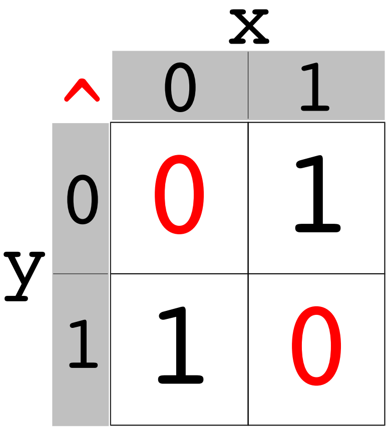


Figure 13.35: The result of a bitwise "exclusive or" of two binary numbers is another number that has a `1` in the spots where *only one* of the original numbers had a `1`.



^	x = 0	x = 1
y = 0	0	1
y = 1	1	0

Figure 13.36: Truth table for ^, the "exclusive or" operator. The result is true wherever only one of x or y is true.

Finally, to complete our toolkit of bitwise operators there's the "bitwise not", `~`. This operator changes every `0` to `1` and every `1` to `0`. For example, if `x` contains the bits `11101101`, then `~x` will be `00010010`. If you think of each `1` or `0` as "true" or "false", then the "not" operator changes each "true" into "not true", and each "false" into "not false".

The following table summarizes the bitwise logical operators we've talked about in this section:

Operator	Symbol	Usage	Description
Bitwise **and**	`&`	`z = x&y`	Bits of `z` are `1` only where bits of both `x` and `y` are `1`.
Bitwise **or**	`|`	`z = x|y`	Bits of `z` are `1` where bits of either `x` or `y` are `1`.
Bitwise **xor** ("Exclusive or")	`^`	`z = x^y`	Bits of `z` are `1` where bits of either `x` or `y` are `1`, but not where both are `1`.
Bitwise **not**	`~`	`z = ~x`	Bits of `z` are the opposite of the corresponding bits in `x`.

13.9. Examining Bits

Now that we know about bit-shifting and the bitwise “and” operator we’re ready to write a program that examines individual bits in the computer’s memory. Remember that we can check the value of `n&(1<<i)` to find out whether bit number `i` of the number `n` is “on” or “off”, as we saw in Section 13.8.

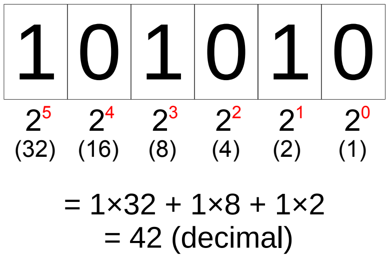


Figure 13.37: The number 42 expressed in binary.

Program 13.4 puts the value 42 into a variable named `n` and then examine’s the variable’s bits one by one.

Program 13.4: printbits.cpp

```
#include <stdio.h>
int main () {
  unsigned int n = 42;
  int i, on, nbits;

  nbits = 8*sizeof( n );

  for ( i=nbits-1; i>=0; i-- ) {

    on = n & (1<<i);

    if ( on != 0 ) {
      printf ( "1" );
    } else {
      printf ( "0" );
    }

  }

  printf ("\n");
}
```

`sizeof` gives bytes. 1 byte = 8 bits.

Start at highest bit and work down.

If this value is zero, the bit is “off”.

If we ran this program, it would print:

```
00000000000000000000000000101010
```

Is this really equal to 42 in decimal? Let’s take a look. We can ignore the long line of zeros on the left-hand side, so we only need to look at the right-most 6 bits. As you can see from Figure 13.37, this is indeed the binary representation of the decimal number 42.

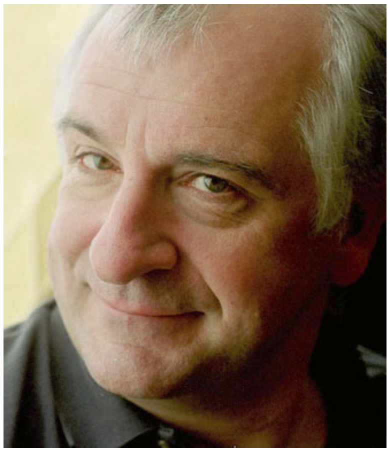


Figure 13.38: In Douglas Adams’ *Hitchhiker’s Guide to the Galaxy* books, 42 is the answer to the “ultimate question of life, the universe and everything”.

Image: Wikimedia Commons

Exercise 63: Bit by Bit

Create, compile and run Program 13.4. It should print the binary version of the decimal number 42. Try changing the value of n in the program, recompiling it and running it again. What happens if you set n to a power of 2 (like 2, 4, 8, 16, and so forth)? What happens if you set it to a value that's one less than a power of 2 (like 3, 7, 15, 31, ...)?

Try a few even numbers, paying attention to the right-most digit of the output, and then try a few odd numbers doing the same. Do you see a pattern?

If you're tired of re-compiling the program, modify it so that it asks you for the number instead of having the number written into the program. Pay attention to the kind of format specifier (placeholder) you use in your `scanf` statement. Make sure it matches the type of the variable you're reading the number into.

Try giving the program the number 4294967295 (the biggest number that an `unsigned int` can hold. What does the output look like? What happens when you give it even bigger numbers?

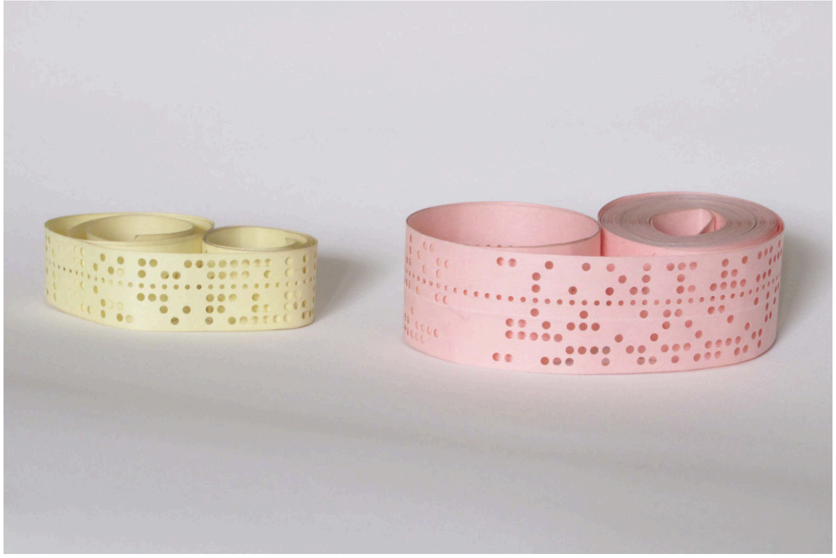


Figure 13.39: In the past, data was sometimes saved on paper tapes like this. Each line (vertical column in this picture) represents a binary number. A large punched-out hole represents a `1` and the un-punched spaces are zeros. (Ignore the line of small holes. That's for moving the tape.) The right-hand tape in this picture is written using 7-bit ASCII characters. The bottom-most position in each line is a special "parity bit" wich isn't part of the character but is used for error-checking. The other bits can be read from bottom to top as a binary number representing an ASCII character (see the table in Figure 13.9). The visible part of the right-hand tape says: `10.1 TYPE"DO YOU LOVE ME?"` Let's hope the poor programmer wasn't disappointed.

Image: Wikimedia Commons

13.10. Using xor for Encryption

You and your best friend want to exchange secret messages. Fortunately, you know about the bitwise "exclusive or" (xor) operator, ^, and that's all you need for writing a simple encryption program.

Take a look at Program 13.5 (`secretletter.cpp`) below.

Figure 13.40: *Les Deux Soers* (1889), by Pierre-Auguste Renoir.

Image: Wikimedia Commons

Program 13.5: secretletter.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  char letter;
  char key;

  if ( argc != 2 ) {
    fprintf ( stderr, "Usage: %s key\n", argv[0] );
    exit(1);
  }

  key = argv[1][0];

  while ( scanf( "%c", &letter ) != EOF ) {
    printf ( "%c", letter^key );
  }
}
```

Make sure user has supplied a key.

The key is the first (and only) letter of the first argument.

Combine each letter with the key, using xor.

This program takes one command-line argument: a single letter that forms the "key" for your encrypted message. Only someone who knows the key will be able to unscramble the message.

The program works by taking each letter you type and "xor-ing" its bits with the bits of the key. The resulting encrypted letter is then printed out. The program will keep reading letters until it sees an "End of File" signal, which you can give it by typing Ctrl-D. The encrypted characters the program generates might not be viewable on your screen, and some of them might even cause your display to misbehave in weird ways. Because of this, it would be a good idea to redirect the program's output into a file, like this:

```
./secretletter b > secretstuff.dat
```

In this example I've used the letter "b" as my secret key, but you can use any character you like. Just don't tell anyone except your friend.

Once you've made the encrypted file (`secretstuff.dat` in the example above), you can e-mail it to your friend, secure in the knowledge that nobody else will be able to read it.

But how will your friend decode the message? That's where the xor operation really comes in handy. It turns out that if you have three binary numbers, `plain`, `key`, and `encrypted`, and you do this to them:

```
encrypted = plain ^ key
```

then the following is *also* true:

```
plain = encrypted ^ key
```

so all your friend needs to do is run the encrypted message back through the same program, using the same key. Once your friend receives your message, he or she can decrypt it by typing:

```
cat secretstuff.dat | ./secretletter b
```

This is like what we did earlier to "decrypt" files created with Program 13.1 (`rot13.cpp`).

One obvious weakness of this program is that a Bad Guy could decrypt our message by just trying all the possible letters we might have used as our secret key. Even if we allow any letter (upper or lower case) or number as the key, that's still only 62 possibilities, and it wouldn't take that long to try them all.

We could improve our security by using a whole word as our key – a password! Each time the program encrypts a character it could use the next letter in the word, until it gets to the end and then starts over at the beginning of the word. With that change, the number of possible keys (still assuming only letters and numbers) becomes 62^n, where n is the maximum length of our password. For 8-letter passwords, that's $62^8 = 218,340,105,584,896$ possiblities! It would be very hard for a Bad Guy to try all of these.

Program 13.6 shows how you might modify the `secretletter.cpp` program to make it use a whole word as the key. Then necessary changes are shown in bold. Notice that we use the `strlen` function to find the length of the "key word" (or "password"), and we use the modulo operator (%) and the number of characters read so far (`nchars`) to keep cycling through the letters of the key word.

Figure 13.41: Julia Child — author of *The Art of French Cooking* and beloved host of *The French Chef* — had an earlier career as a spy. During World War II she worked for the OSS, stationed in Sri Lanka and China, where she passed encrypted intelligence back to the US.

Image: Wikimedia Commons

Program 13.6: secretword.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ( int argc, char *argv[] ) {
  char letter;
  char key;
  int keynumber, keylength, nchars=0;

  if ( argc != 2 ) {
    fprintf ( stderr, "Usage: %s key\n", argv[0] );
    exit(1);
  }

  keylength = strlen( argv[1] );

  while ( scanf( "%c", &letter ) != EOF ) {
    keynumber = nchars%keylength;
    key = argv[1][keynumber];
    printf ( "%c", letter^key );
    nchars++;
  }
}
```

Figure 13.42: *"Say the secret word and win a hundred dollars!"* Groucho Marx was the host of a quiz show named *You Bet Your Life*. At the beginning of the show the audience was shown a secret word. If a contestant used the word during the quiz, a rubber duck holding a $100 bill descended on a string.

Image: Wikimedia Commons

This program works just the same as the earlier version. Start typing your message after entering a command like this:

```
./secretword groucho > secretstuff.dat
```

where "`groucho`" is whatever you choose to use as your password and `secretstuff.dat` is the encrypted version of your message. Again, type Ctrl-D when you're finished typing the message. Your friend can then decrypt the message by typing:

```
cat secretstuff.dat | ./secretword groucho
```

The simplicity of xor encryption comes with a price. Another relation between the key, the plain message, and its encrypted version is this:

```
key = encrypted ^ plain
```

meaning that, if an enemy ever obtains both the encrypted and decrypted versions of one of your messages, they can find the key you've used! Even with a multi-letter key, if bad guys ever get snippets of encrypted and unencrypted data that are longer than our key, they'll be able to calculate all the letters in the key. Because of this weakness, the same password shouldn't be used more than once when doing this kind of encryption.

In the days of the cold war, Soviet spies came to the US armed with a pad full of encryption keys. Their associates back in the USSR had identical pads. Whenever a spy needed to send back some information, he'd use one of the keys to encrypt it, then throw away that key. When his compatriot received the message, he'd decrypt it using the first key on his pad, and then discard that key. This type of encryption key is called a "one-time pad".

Figure 13.43: A Soviet poster: *"In order to have more, it is necessary to produce more. In order to produce more, it is necessary to know more."*

Image: Wikimedia Commons

Exercise 64: Spies Like Us

Create, compile, and run Program 13.6 (`secretword.cpp`). Try encrypting a message, writing the encrypted output into a file named "`encrypted.dat`".

Look at `encrypted.dat` with *nano*. It should look like nonsense.

Now try decrypting the message. Does the text displayed on your screen match your original message? What happens if you use the wrong password when attempting to decrypt the message?

What happens if you only use *the first part* of the password when decrypting the message? For example, if you used "charlottesville" as the password when encrypting the message, what happens if you use "charlotte" when decrypting it? (Make sure your message is longer than the password.)

Note that you can use spaces in the password, but if you do you'll need to enclose it in quotes, like this:

```
./secretword "a long password" > encrypted.dat
```

and do the same when decrypting the message.

13.11. long and long long Variables

There are almost 8 billion people on Earth. Imagine that you had to give each person an ID number. You wouldn't be able to store that number in an int or even an unsigned int. They can only store numbers up to about 2 and 4 billion, respectively.

What if we want to store an integer that's bigger than the biggest thing that will fit into an unsigned int? C offers some other variable types that might accommodate your needs. Two of them are "long int" and "long long int".

The C standard doesn't specify how many bits each of these types has. It only requires that long int be *at least as* large as int, and that long long int be *at least as* large as long int. In some cases, two (or even all three) of these types of variables will have the same number of bits. Typically, though, you'll find that at long long int can hold significantly larger numbers than int.

Figure 13.44: Long, long hair! An average human has about 100,000 scalp hairs, a number that could easily be stored in an int.

Image: Wikimedia Commons

We can again use sizeof to find out how many bits each of these types uses.

```
#include <stdio.h>
int main () {
  int i;
  long int ilong;
  long long int ilonglong;

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) );
  printf ("Size of ilong is %d bytes.\n", (int)sizeof( ilong ) );
  printf ("Size of ilonglong is %d bytes.\n", (int)sizeof( ilonglong ) );
}
```

If we ran this program on a typical computer, we might see something like this:

```
Size of i is 4 bytes.
Size of ilong is 8 bytes.
Size of ilonglong is 8 bytes.
```

Since a byte is 8 bits, this means that an int has $4 \times 8 = 32$ bits, a long int has $8 \times 8 = 64$ bits, and a long long int also has 64 bits. If we stored the number 42 in an int variable on this computer, its bits would look like this:

00000000000000000000000000**101010**

but the same value stored in a `long int` would look like this:

0000000000000000000000000000000000000000000000000000000000**101010**

Just as with plain `int` variables, there are signed and unsigned versions of `long` and `long long int` variables: `unsigned long int` and `unsigned long long int`.

Each of these types has an appropriate placeholder to use in `printf` and `scanf` statements. The table below lists them.

Type	Format Specifier
`int`	`%d`
`unsigned int`	`%u`
`long int`	`%ld`
`unsigned long int`	`%lu`
`long long int`	`%lld`
`unsigned long long int`	`%llu`

What's the biggest number that will fit into each of these types of integer? With the information from `sizeof` and what we now know about how integers are stored we could figure it out, but there's an easier way. Your programs can use the file `limits.h` to find the biggest or smallest numbers we can put into a given type of variable. `limits.h` defines symbols like `INT_MAX` and `INT_MIN`, which tell you the maximum and minimum valuse an `int` can hold. Here are some useful values to look at:

Symbol	Meaning
`INT_MIN`	The smallest negative value that can be held in an `int`.
`INT_MAX`	The largest positive value that can be held in an `int`.
`LONG_MIN`	The smallest negative value that can be held in a `long int`.
`LONG_MAX`	The largest positive value that can be held in a `long int`.
`LLONG_MIN`	The smallest negative value that can be held in a `long long int`.
`LLONG_MAX`	The largest positive value that can be held in a `long long int`.
`UINT_MAX`	The largest value that can be held in an `unsigned int`.
`ULONG_MAX`	The largest value that can be held in an `unsigned long int`.
`ULLONG_MAX`	The largest value that can be held in an `unsigned long long int`.

Notice that there are no symbols for the minimum values of the unsigned types. For those, the minimum is always zero.

With all that in mind, we could write a little program to tell us the limits on our various integer types. Program 13.7 does that.

Program 13.7: printsizes.cpp

```
#include <stdio.h>
#include <limits.h>
int main () {
  printf ("INT_MAX is %d\n", INT_MAX );
  printf ("LONG_MAX is %ld\n", LONG_MAX );
  printf ("LLONG_MAX is %lld\n", LLONG_MAX );

  printf ("INT_MIN is %d\n", INT_MIN );
  printf ("LONG_MIN is %ld\n", LONG_MIN );
  printf ("LLONG_MIN is %lld\n", LLONG_MIN );

  printf ("UINT_MAX is %u\n", UINT_MAX );
  printf ("ULONG_MAX is %lu\n", ULONG_MAX );
  printf ("ULLONG_MAX is %llu\n", ULLONG_MAX );
}
```

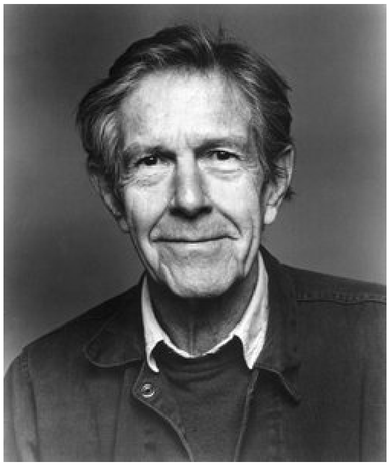


Figure 13.45: Composer John Cage. The longest piece of music I'm aware of is John Cage's *Organ^2/ASLSP (As Slow as Possible)*, which is currently being performed on an organ in Halberstadt, Germany. The performance will end on September 5, 2640, after 639 years! If we created a timer that counted how many seconds the performance has lasted, it would only need to count to a little over 20 billion. This wouldn't fit into an `int`, but it would easily fit into a 64-bit `long long int`.

Image: Wikimedia Commons

If we ran this program on a typical computer we might see something like the following:

```
INT_MAX is 2147483647
LONG_MAX is 9223372036854775807
LLONG_MAX is 9223372036854775807
INT_MIN is -2147483648
LONG_MIN is -9223372036854775808
LLONG_MIN is -9223372036854775808
UINT_MAX is 4294967295
ULONG_MAX is 18446744073709551615
ULLONG_MAX is 18446744073709551615
```

This tells us that, on this computer, the biggest number we can store in any of these integer types is $2^{64} - 1$, or $18,446,744,073,709,551,615$ (about 1.8×10^{19}, or 18 quintillion). This is the maximum value of an `unsigned long int` or an `unsigned long long int` here.

How big is 18 quintillion? That's more than twice the estimated number of grains of sand on earth! We could give each sand grain a serial number if we wanted to.

Figure 13.46: Mathematicians at the University of Hawaii have estimated that there are about 7.5 quintillion grains of sand on Earth.

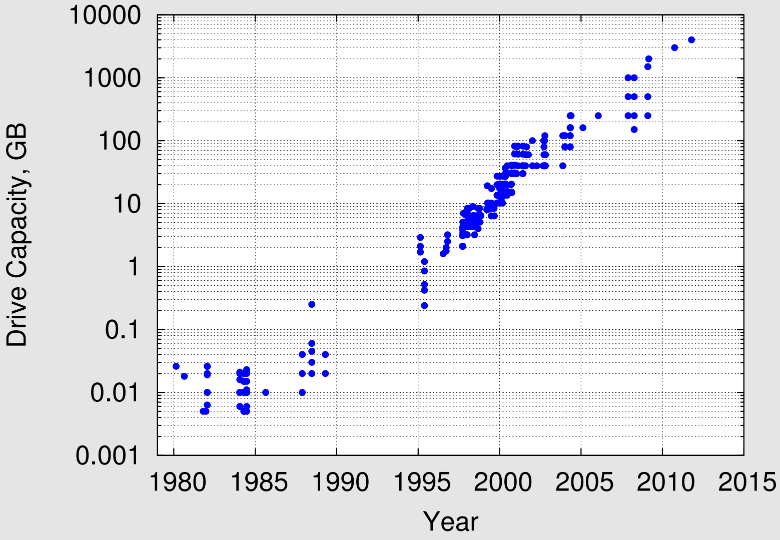
Image: Wikimedia Commons

But what about. . . ?

Operating systems like Microsoft Windows and computer processors like those made by Intel often say they're "64-bit" or "32-bit". What does that mean?

64-bit processors are CPUs that can read and process data in 64-bit chunks. This effectively lets them do more work in less time than a 32-bit processor. Most CPUs you'll find in desktop and laptop computers today are 64-bit devices.

A 64-bit operating system takes advantage of a 64-bit CPU's abilities. One key advantage of 64-bit operating systems is that they can store the addresses of memory or disk locations in 64-bit-wide variables. For example, a 32-bit operating system has trouble with files larger than 2 Gigabytes or amounts of memory larger than 4 Gigabytes — the maximum numbers that can be stored in signed and unsigned 32-bit integers. As we've seen above, the limits on 64-bit integers are astronomically higher, allowing 64-bit operating systems to use much larger files and amounts of memory.



Data adapted from Wikimedia Commons

As you can see from the graph above, disks have continued to grow rapidly in size over the last 40 years. Disks holding tens of thousands of Gigabytes are now available. The move to 64-bit operating systems was necessary to accommodate this growth.

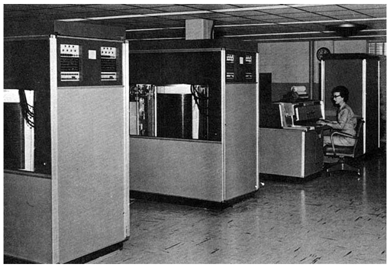


Figure 13.47: The two closet-sized boxes in the foreground are IBM 350 disk drives, introduced in 1956. Each drive can hold a whopping 3.75 Megabytes of data. That's about the size of a single photograph from a modern digital camera.

Image: Wikimedia Commons

13.12. int Variables with Specific Widths

As we noted above, the C standards don't specify the exact number of bits that an `int`, `long`, or `long long` variable should have. The standards just say that each has at least as many bits as the preceding type.

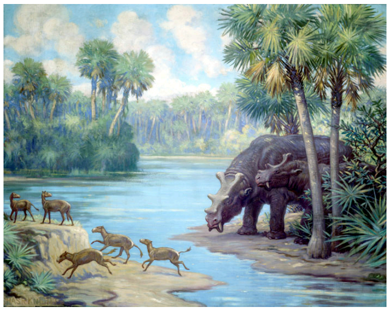


Figure 13.48: Two uintatheres pause for a drink in this beautiful painting by Charles R. Knight.
Image: Field Museum

Sometimes, though, we want to have an integer variable with a specific number of bits. For example, if we wanted our program to read binary data in 4-byte chunks, it would be nice to have a 4-byte-long variable to store each chunk.

The header file `stdint.h` defines some new types that we can use in situations like this:

Type	Size	Format for `printf`	Format for `scanf`
`uint8_t`	8 bits (1 byte)	`PRIu8`	`SCNu8`
`uint16_t`	16 bits (2 bytes)	`PRIu16`	`SCNu16`
`uint32_t`	32 bits (4 bytes)	`PRIu32`	`SCNu32`
`uint64_t`	64 bits (8 bytes)	`PRIu64`	`SCNu64`

These variable types each hold unsigned integers. There are corresponding signed types with names like `int8_t`, but you'll probably find that the unsigned types are more useful.

There are new placeholders ("format specifiers") for each of the new types, and that these placeholders look different from the ones we've used before. In order to use them, you'll first need to include `inttypes.h`[7]. Notice that there are different placeholders for `printf` and `scanf`.

These placeholders are used a little differently, too. Take a look at the following example, which reads a number into a `uint8_t` variable and then prints the value back out:

Program 13.8: uintathere.cpp

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
int main () {
  uint8_t n;
  printf ("Enter a number: ");
  scanf( "%"SCNu8, &n );
  printf ("You entered %"PRIu8"\n", n);
}
```

[7] Some older C compilers will complain about these formats unless you also add this arcane line at the top of your program:
`#define __STDC_FORMAT_MACROS`

Instead of saying something like `"%d"`, where the placeholder is inside the quotes, these new placeholders need to go *outside* the quotes. Notice, for example, how the template we give `printf` in the program above has three sections: a beginning quoted section, followed by `PRIu8`, then finally another quoted section. Whenever you want to use one of these new-fangled placeholders, you need to exclude it from the quotes like this, but *leave the % inside!*

Here's another example, where we're printing two `uint8_t` variables:

```
printf ( "The numbers are %"PRIu8" and %"PRIu8"\n", n, m );
```

Why all these complications? It's because these types were added to the C standards long after the original types like `int`. The new types add new functionality without breaking anything that's already there. This required a little fancy footwork.

13.13. The Size of Literal Numbers

When a program does a calculation like "`n = 2*5-3`" it stores the numbers `2` and `5` somewhere in the computer's memory, then multiplies `2*5` and stores the result somewhere, then adds `3` to the result. What amount of memory does the program reserve for these numbers and intermediate results while it's working?

In general, the program will look at each number and try to find an appropriately-sized type of storage to put it in. For example, the current version of *g++* will first try to treat the number as an `int`. If it won't fit into the number of bits allocated for an `int`, it will move up to a `long int` or a `long long int`. The numbers 2, 5, and 3 in the example above would all fit into an `int`, which has 32 bits (4 bytes) on most computers. If we looked at the computer's memory while the program was running, we'd see something like this:

```
00000000.00000000.00000000.00000010
00000000.00000000.00000000.00000101
00000000.00000000.00000000.00000011
```

representing (from top to bottom) 2, 5, and 3. (I've inserted dots in the numbers above to separate them into byte-sized chunks for clarity.)

If we used the number "`8000000000`" (8 billion) in our program, it might be stored in memory like this:

```
00000000.00000000.00000000.00000001.11011100.11010110.01010000.00000000
```

since that number is too large to fit into only 32 bits.

Figure 13.49: The scientist and author Carl Sagan was famous for talking about the "billions and billons" of stars out there. The phrase was used so often that an informal unit called the "sagan" has been defined. It's equal to at least 2 billion plus 2 billion ("billions and billions"), or 4 billion.

Image: Wikimedia Commons

You can use the sizeof statement to find out exactly how many bytes a program will use to store a given number. For example, this program:

```
#include <stdio.h>
int main () {
  printf ( "%d bytes\n", (int)sizeof(2) );
  printf ( "%d bytes\n", (int)sizeof(2000000000) ); // 2 billion.
  printf ( "%d bytes\n", (int)sizeof(4000000000) ); // 4 billion.
  printf ( "%d bytes\n", (int)sizeof(8000000000) ); // 8 billion.
}
```

might print:

```
4 bytes
4 bytes
8 bytes
8 bytes
```

That's all interesting, but do we need to worry about it? Yes, it turns out that we do sometimes. Imagine what would happen if we had a statement like:

```
n = 2000000000*4+7;
```

which starts by multiplying 2 billion (a number that will fit in 4 bytes) by 4.

When the computer sees an expression like 2000000000*4 it guesses how much space to allocate for the result by looking at the sizes of the numbers being multiplied. Since each of the numbers in this example would fit into 4 bytes, the computer allocates 4 bytes for the result. But, as we've seen above, the result here (8 billion) won't fit into 4 bytes.

Most modern compilers are smart enough to anticipate this problem, and they'll give you a warning message like:

```
warning: integer overflow in expression
n = 2000000000*4+7;
               ^
```

But the compiler can only catch the most obvious variations on this problem. Imagine what would happen in a slightly more complicated situation:

Program 13.9: literal.cpp

```
#include <stdio.h>
int main () {
  int x;
  long int n;

  printf ( "Enter multiplier: " );
  scanf ( "%d", &x);

  n = 2000000000*x+7;

  printf ( "%ld\n", n );
}
```

Here, instead of multiplying by 4, we ask the user to enter a multiplier when we run the program. The compiler can't know in advance what the user will type, so the compiler would give you no warning or error messages, but if you ran the program and entered 4 as the multiplier, bad things could happen.

The variable `n` is a `long int`, which is large enough[8] to hold the expected result of our calculation (8,000,000,007), but the program might incorrectly tell us that the answer is -589,934,585!

What's happening in this case? Let's follow the process step by step:

1. The program multiples 2,000,000,000,000 × 4, which should equal 8 billion. If we had 8 bytes (64 bits) to store the number, it would look like this:

   ```
   00000000.00000000.00000000.00000001.11011100.11010110.01010000.00000000
   ```

2. Since the computer has only allocated 4 bytes, the left-most `1` gets chopped off:

   ```
           Chop this off
    00000000.00000000.00000000.00000001. 11011100.11010110.01010000.00000000
   ```

   leaving us with:

   ```
   11011100.11010110.01010000.00000000
   ```

   This isn't equal to 8 billion now. Interpreted as an `int`, it's equal to -589,934,592.

3. We now add 7 to this, to get -589,934,585, which is exactly what the program told us.

[8] See Section 13.11 above. Note that the details will vary, depending what operating system and compiler you use, but the principles are the same everywhere.

How could we have avoided this problem? It turns out that you can tell the compiler how much space to reserve for a number. In the example above, we could fix the problem by adding one letter:

```
n = 2000000000L*x+7;
```

The L after the number tells the compiler that we want to reserve as many bits as a long int variable[9]. With that change, the program would correctly tell us that the answer is 8,000,000,007.

We can use other suffixes on numbers to select other types. The table below shows some of the possibilities:

Type	Suffix
long int	L
long long int	LL
unsigned int	U
unsigned long int	UL
unsigned long long int	ULL

If you want to use the specific-width types like uint32_t, defined in stdint.h[10], the syntax is a little different. For those types, you can use a statement like:

```
n = UINT64_C(2000000000)*x+7;
```

which tells the compiler that you specifically want to reserve 64 bits for storing the first number. Some other similar options are listed in the table below:

Type	Syntax
uint8_t	UINT8_C()
uint32_t	UINT32_C()
uint64_t	UINT64_C()

As you can see, numbers inside a computer aren't as simple as the numbers we use in math class. The complications arise because the computer has a limited amount of space to store each number. You should think about this whenever you're working near the limit of the largest numbers your variables can contain.

[9] We could have used either an upper- or lower-case L. I've used an upper-case letter here to avoid mistaking it for the number 1.

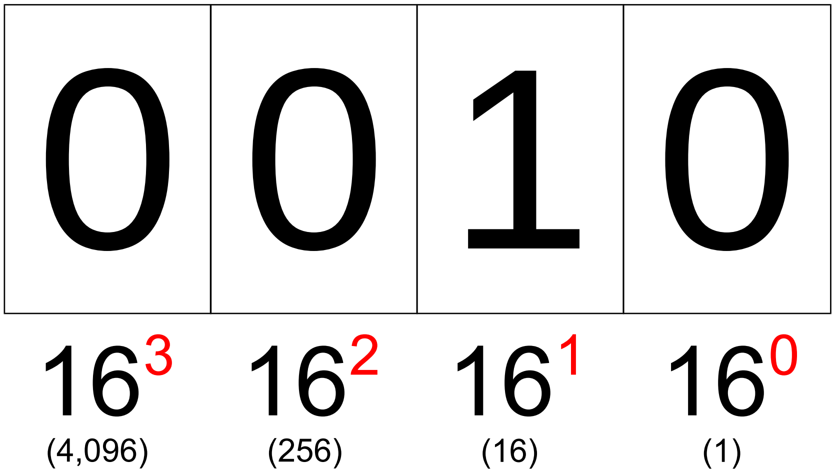
[10] See Section 13.12.

13.14. Hexadecimal Numbers

When we write a number like `1729` we assume that it's expressed in base 10 (decimal) notation. In the preceding sections we've seen that it's also possible to write numbers in base 2 (binary) notation. There are a couple of other useful notations that you should be aware of. One of them is "hexadecimal" (or "hex"), which uses 16 as its base.

We know that in base 10 we have ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) and in base 2 we have two digits (0 and 1). Apparently we'll need sixteen digits for base 16! Since we only have ten number symbols on our keyboards, what symbols do we use for the extra six digits? The convention is to use the letters 'A' through 'F'. The table in Figure 13.51 shows some decimal numbers with their hex equivalents. It also shows the binary version of each number.

As you can see, this is how we'd count to sixteen in hexadecimal: 1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10. In hex, the number "10" is equal to $1 \times 16 + 0 \times 1$:



Hexadecimal numbers are useful whenever you can divide a set of bits into 4-bit groups. Notice that in table in Figure 13.51 we've split the binary version of each number into two 4-bit chunks. You might recall that 4 bits (half a byte) is called a "nybble". The minimum number that can be stored in 4 bits is, of course, zero, and the maximum number is 15. That gives 16 possible values that we can represent with 4 bits, just like the 16 possible digits of hexadecimal numbers.

Since there are two nybbles in each byte, that means that the value stored in a byte can be represented by two hexadecimal digits. That value can be anything from `00` through `FF`, which corresponds to a

Figure 13.50: Srinivasa Ramanujan (1887-1920) was a brilliant Indian mathematician. While visiting Ramanujan, the English mathematician G.H. Hardy remarked that the number on a taxicab, 1729, was rather uninteresting. Ramanujan replied that this number was, in fact, *very* interesting, being the smallest number that's the sum of two cubes in two different ways:$1^3 + 12^3$ and $9^3 + 10^3$. Since then, 1729 has been known as "Ramanujan's taxicab number".

Image: Wikimedia Commons

Decimal	Hex	Binary
0	0	0000 0000
1	1	0000 0001
2	2	0000 0010
3	3	0000 0011
4	4	0000 0100
5	5	0000 0101
6	6	0000 0110
7	7	0000 0111
8	8	0000 1000
9	9	0000 1001
10	A	0000 1010
11	B	0000 1011
12	C	0000 1100
13	D	0000 1101
14	E	0000 1110
15	F	0000 1111
16	10	0001 0000
17	11	0001 0001
18	12	0001 0010
31	1F	0001 1111
32	20	0010 0000
63	3F	0011 1111
64	40	0100 0000
100	64	0110 0100
128	80	1000 0000
255	FF	1111 1111

Figure 13.51: Some decimal numbers with their hexadecimal and binary (8-bit) equivalents. A space splits each binary number into two 4-bit "nybbles".

range of decimal values from zero to 255.

Let's look at the binary representation of the number 1729:

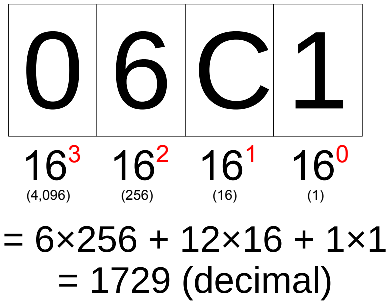
```
00000000.00000000.00000110.11000001
```

We could use spaces to break this into 4-bit nybbles:

```
0000 0000 0000 0000 0000 0110 1100 0001
```

Converting each nybble into its hex equivalent, we'd get:

```
0 0 0 0 0 6 C 1
```



These eight hex digits (`000006C1`) represent the same value as the 32 bits of the number's binary representation. Since hex notation is much more compact than binary, and since it's easy to convert between binary and hex, we often use hexadecimal numbers in computing.

Hex numbers are used so often in computing that the C language has some built-in facilities for dealing with them. For example, we can write hex numbers directly into our program without needing to convert them into decimal. If we wanted to give a variable the value `FF` in hex (which is 255 in decimal), we could say:

```
n = 0xFF;
```

When we start a number with "`0x`", the compiler assumes that the number is written in hexadecimal notation. This might take a little getting used to, since it looks like you're writing "zero times `FF`", but you'll get the hang of it. The zero at the beginning tells the compiler that this is a number, and not a variable name (since variable names can't start with digits), and the `x` means that the number is in he**X**adecimal. (Note that it doesn't matter whether you use an upper-case or lower-case `x`, but programmers usually stick to lower-case.)

Figure 13.52: Symbols like these, often found on the sides of Pennsylvania barns, are called "Hex signs".

Image: Wikimedia Commons

We can also read and write numbers in hex notation. The placeholder "`%x`" means "read or write this number as hex". For example, these statements:

```
n = 1729;
printf ( "In hex the number is %x.\n", n );
```

would print:

```
In hex the number is 6c1.
```

Notice that this printed a lower-case “`c`”. If we wanted to print upper-case letters, we could have used “`%X`” instead, to get `6C1`. If we wanted to print a `0x` at the beginning of the number, as we would if we used the number in a program, we could use “`%#x`”, like this:

```
printf ( "In hex the number is %#x.\n", n );
```

which would print “`In hex the number is 0x6c1.`” As before, using an upper-case `X` would cause the printed letters to be upper-case.

When using “`%x`” with `scanf`, case doesn’t matter. For example, these statements:

```
  printf ("Enter number: ");
  scanf("%x",&n);
```

would accept a number written as `6c1` or `6C1`. It would also be OK if you entered `0x6c1` or `0x6C1`. As always, `scanf` tries to be forgiving.

If you’ve ever created web pages, you’ve probably already encountered hex numbers. In that context, these numbers are often used to specify colors. For example, if you see “`#FFA500`” that means 100% red, about 65% green, and no blue, which would mix together on your screen to give you a nice orange color. (The color `#0006C1`” is a nice dark blue.)

Figure 13.53: The three pairs of hex digits in this kind of color specification tell the computer how much red, green, and blue to mix together. Each pair of digits is a number between `00` and `FF`.

Color specifications like these consist of three pairs of hex digits, telling the computer how much red, green, and blue to use. Each pair of digits represents one 8-bit byte. The number stored in this byte says how much of that color to use, on a scale from `00` (none of it) to `FF` (all of it). Altogether, the color specification takes up 3 bytes of storage, or 24 bits. You’ll often see this referred to as “24-bit color”. This many bits can store any of $2^{24} = 16,777,216$ different values, so that’s how many colors it’s possible to specify this way.

Let’s try writing a little program that uses hex numbers to play with color. Program 13.10 loops through some of the values for R, G, and B and prints the resulting hexadecimal color identifier. Since (as we noted above) there are over 16 million possible R,G,B combinations, the program won’t print them all. Instead, it uses only eight out of the possible 256 values for R, G, or B. That means the program will print $8 \times 8 \times 8 = 512$ different colors.

The program has three nested loops. Each loop gives one of the primary colors (R, G, or B) values between `00` and `FF` in steps of 32. That gives eight values for each of the primary colors (since $256/32 = 8$).

Program 13.10: colorcube.cpp

```
#include <stdio.h>
int main () {
  int r,g,b;
  int step=32;

  for ( r=0; r<=0xFF; r+=step ) {
    for ( g=0; g<=0xFF; g+=step ) {
      for ( b=0; b<=0xFF; b+=step ) {
        printf ( "%d %d %d ", r,g,b );
        printf ( "0x" );
        printf ( "%02x", r );
        printf ( "%02x", g );
        printf ( "%02x", b );
        printf ( "\n");
      }
    }
  }
}
```

The program's output has four columns, with each line looking something like this:

```
16 176 48 0x10b030
```

The first three numbers are the R, G, and B values expressed as decimal numbers between 0 and 255. The fourth colum is a single hexadecimal number corresponding to these RGB values.

Note that we've written the number in the format C uses for hex numbers, by starting it with "0x". This makes it easy to use the output with other programs that understand this way of writing hex numbers. After the 0x we write the R, G, and B values as hex numbers. Since we want each of these numbers to have two digits, we can't just use %x as the placeholder. We want to force printf to print two digits, even if the left-hand one is zero. We can make this happen by adding "02" between % and x. The 2 tells printf to always leave room for 2 digits, and the 0 says to put a zero on the left if there would otherwise not be a digit there[11].

Figure 13.54: *Hexe* means "witch" in German. Here's W.W. Denslow's illustration of the Wicked Witch of the West, from L. Frank Baum's *The Wonderful Wizard of Oz*.

Image: Wikimedia Commons

[11] See Appendix E for other printf tricks.

Exercise 65: Color Cube

Create and compile Program 13.10. Run the program and redirect its output into a file, like this:

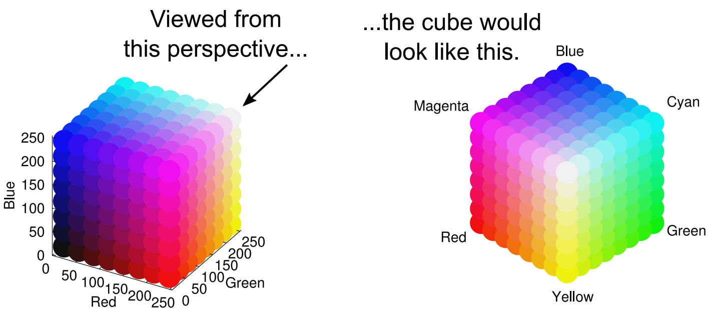
```
./colorcube > colorcube.dat
```

We can use *gnuplot* to visualize the output of our program. We'll use *gnuplot*'s `splot` command to plot points in 3-dimensional space, where the x, y, and z coordinates of each point are the R, G, and B values from our program. The fourth column will determine each point's color. Fortunately, *gnuplot* also uses a leading `0x` to identify hex numbers, just as C does.

Start up *gnuplot* and give it the following commands:

```
set hidden3d
set xyplane 0
set view equal xyz
splot "colorcube.dat" using 1:2:3:4 pt 7 ps 6 lc rgb variable
```

The result should look something like the left-hand figure below:



Try grabbing the cube with your mouse and rotating it around!

How does this *gnuplot* magic work? The first three commands have the following effects:

- `set hidden3d` causes *gnuplot* to "hide" objects that are "behind" others in 3-d plots like this. Without this setting, the stacking of objects depends on the order in which *gnuplot* draws them, which might not have anything to do with which object is "closer" to the viewer.

- `set xyplane 0` causes *gnuplot* to set the cube on the x-y plane. Without this, `splot` will leave some space below the cube.

- `set view equal xyz` causes all of the axes to have equal scales. This makes the plot a cube, rather than a shoebox.

While still in *gnuplot*, try turning each of these options off, one at a time, by using `unset` instead of `set`, and then typing `replot` after unsetting each one. This will show you the effect of each setting.

The last *gnuplot* command (`splot`...) says that we want to use columns 1, 2, 3, and 4 of the file. The "`lc rgb variable`" at the end of the line tells *gnuplot* that the color of each point will be specified in RGB format and will be given by the last column of our data file. The "`pt 7`" and "`ps 6`" control the point type and point size. These choices cause the points to be plotted as large circles.

Figure 13.55: A "hex" can also be a curse. Here's a Roman curse written on a lead tablet. The writing says, in part, *"I curse Tretia Maria and her life and mind and memory and liver and lungs mixed up together. . . "* Yow! The tablet is in the British Museum.

Image: Wikimedia Commons