Chapter 1
Information Representation and Storage

Norman Matloff
University of California at Davis

February 27, 2002

Contents

1 Introduction

A computer can store many types of information. A high-level language (HLL) will typically have several data types, such as the C language's **int**, **float**, and **char**. Yet a computer can not directly store any of these data types. Instead, a computer only stores 0s and 1s. Thus the question arises as to how one can represent the abstract data types of C or other HLLs in terms of 0s and 1s. What, for example, does a **char** variable look like when viewed from "under the hood"?

A related question is how we can use 0s and 1s to represent our program itself, meaning the machine language instructions that are generated when our C or other HLL program is compiled. In this chapter, we will discuss how to represent various types of information in terms of 0s and 1s. And, in addition to this question of *how* items are stored, we will also begin to address the question of *where* they are stored, i.e. where they are placed within the structure of a computer's main memory.

2 Bits and Bytes

2.1 "Binary Digits"

The 0s and 1s used to store information in a computer are called **bits**. The term comes from **b**inary dig**it**, i.e. a digit in the base-2 form of a number (though once again, keep in mind that not all kinds of items that a computer stores are numeric). The physical nature of bit storage, such as using a high voltage to represent a 1 and a low voltage to represent a 0, is beyond the scope of this book, but the point is that every piece of information must be expressed as a string of bits.

For most computers, it is customary to label individual bits within a bit string from right to left, starting with 0. For example, in the bit string 1101, we say Bit 0 = 1, Bit 1 = 0, Bit 2 = 1 and Bit 3 = 1.

If we happen to be using an n-bit string to represent a nonnegative integer, we say that Bit n-1, i.e. the leftmost bit, is the most significant bit (MSB). To see why this terminology makes sense, think of the base-10 case. Suppose the price of an item is $237. A mistake by a sales clerk in the digit 2 would be much more serious than a mistake in the digit 7, i.e. the 2 is the most significant of the three digits in this price. Similarly, in an n-bit string, Bit 0, the rightmost bit, is called the least significant bit (LSB).

A bit is said to be **set** if it is 1, and **cleared** if it is 0.

A string of eight bits is usually called a **byte**. Bit strings of eight bits are important for two reasons. First, in storing characters, we typically store each character as an 8-bit string. Second, computer storage cells are typically composed of an integral number of bytes, i.e. an even multiple of eight bits, with 16 bits and 32 bits being the most commonly encountered cell sizes.

The whimsical pioneers of the computer world extended the pun, "byte" to the term **nibble**, meaning a 4-bit string. So, each hex digit is called a nibble.

2.2 Hex Notation

We will need to define a "shorthand" notation to use for writing long bit strings. For example, imagine how cumbersome it would be for us humans to keep reading and writing a string such as 1001110010101110. So, let us agree to use **hexadecimal** notation, which consists of grouping a bit string into 4-bit substrings,

and then giving a single-character name to each substring.

For example, for the string 1001110010101110, the grouping would be

1001 1100 1010 1110

Next, we give a name to each 4-bit substring. To do this, we treat each 4-bit substring as if it were a base-2 number. For example, the leftmost substring above, 1001, is the base-2 representation for the number 9, since

$$1 \cdot (2^3) + 0 \cdot (2^2) + 0 \cdot (2^1) + 1 \cdot (2^0) = 9,$$

so, for convenience we will call that substring "9." The second substring, 1100, is the base-2 form for the number 12, so we will call it "12." However, we want to use a single-character name, so we will call it "c," because we will call 10 "a," 11 "b," 12 "c," and so on, until 15, which we will call "f."

In other words, we will refer to the string 1001110010101110 as 9cae. This is certainly much more convenient, since it involves writing only 4 characters, instead of 16 0s and 1s. However, keep in mind that we are doing this only as a quick shorthand form, for use by us humans. The computer is storing the string in its original form, 1001110010101110, *not* as 9cae.

We say 9cae is the hexadecimal, or "hex," form of the the bit string 1001110010101110. Often we will use the C-language notation, prepending "0x" to signify hex, in this case 0x9cae.

Recall that we use bit strings to represent many different types of information, with some types being numeric and others being nonnumeric. If we happen to be using a bit string as a nonnegative number, then the hex form of that bit string has an additional meaning, namely the base-16 representation of that number.

For example, the above string 1001110010101110, if representing a nonnegative base-2 number, is equal to

$$\begin{aligned} 1(2^{15}) &+ 0(2^{14}) + 0(2^{13}) + 1(2^{12}) + 1(2^{11}) + 1(2^{10}) + 0(2^9) + 0(2^8) \\ &+ 1(2^7) + 0(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 1(2^2) + 1(2^1) + 0(2^0) = 40,110. \end{aligned}$$

If the hex form of this bit string, 0x9cae, is treated as a base-16 number, its value is

$$9(16^3) + 12(16^2) + 10(16^1) + 14(16^0) = 40,110,$$

verifying that indeed the hex form is the base-16 version of the number. That is in fact the origin of the term "hexadecimal," which means "pertaining to 16." [But there is no relation of this name to the fact that in this particular example our bit string is 16 bits long; we will use hexadecimal notation for strings of any length.]

The fact that the hex version of a number is also the base-16 representation of that number comes in handy in converting a binary number to its base-10 form. We *could* do such conversion by expanding the powers of 2 as above, but it is much faster to group the binary form into hex, and then expand the powers of 16, as we did in the second equation.

The opposite conversion—from base-10 to binary—can be expedited in the same way, by first converting from base-10 to base-16, and then degrouping the hex into binary form. The conversion of decimal to base-16 is done by repeatedly dividing by 16 until we get a quotient less than 16; the hex digits then are obtained as the remainders and the very last quotient. To make this concrete, let's convert the decimal number 21602 to binary:

```
Divide 21602 by 16, yielding 1350, remainder 2.
Divide 1350 by 16, yielding 84, remainder 6.
Divide 84 by 16, yielding 5, remainder 4.
The hex form of 21602 is thus 5462.
The binary form is thus 0101 0100 0110 0010, i.e. 0101010001100010.
```

The main ingredient here is the repeated division by 16. By dividing by 16 again and again, we are building up powers of 16. For example, in the line

```
Divide 1350 by 16, yielding 84, remainder 6.
```

above, that is our second division by 16, so it is a *cumulative* division by 16^2. [Note that this is why we are dividing by 16, not because the number has 16 bits.]

2.3 There Is No Such Thing As “Hex” Storage at the Machine Level!

Remember, hex is merely a convenient notation **for us humans**. It is wrong to say something like “The machine stores the number in hex,” “The compiler converts the number to hex,” and so on. **It is crucial that you avoid this kind of thinking, as it will lead to major misunderstandings later on.**

3 Representing Information as Bit Strings

We may now address the questions raised at the beginning of the chapter. How can the various abstract data types used in HLLs, and also the computer’s machine instructions, be represented using 0s and 1s?

3.1 Representing Integer Data

Representing nonnegative integer values is straightforward: We just use the base-2 representation, such as 010 for the number +2. For example, the C language data type **unsigned int** (also called simply **unsigned**) uses this representation.

But what about integers which can be either positive or negative, i.e. which are signed? For example, what about the data type **int** in C?

Suppose for simplicity that we will be using 3-bit strings to store integer variables. [Note: We will assume this size for bit strings in the next few paragraphs.] Since each bit can take on either of two values, 0 or 1, there are $2^3 = 8$ possible 3-bit strings. So, we can represent eight different integer values. In other words, we could, for example, represent any integer from -4 to +3, or -2 to +5, or whatever. Most systems opt for a range in which about half the representable numbers are positive and about half are negative. The range -2 to +5, for example, has many more representable positive numbers than negative numbers. This might be useful in some applications, but since most computers are designed as general-purpose machines, they use integer representation schemes which are as symmetric around 0 as possible. The two major systems below use ranges of -3 to +3 and -4 to +3.

But this still leaves open the question as to which bit strings represent which numbers. The two major systems, **signed-magnitude** and **2's complement**, answer this question in different ways. Both systems store the nonnegative numbers in the same way, by storing the base-2 form of the number: 000 represents 0, 001 represents +1, 010 represents +2, and 011 represents +3. However, the two systems differ in the way they store the negative numbers, in the following way.

The signed-magnitude system stores a 3-bit negative number first as a 1 bit, followed by the base-2 representation of the magnitude, i.e. absolute value, of that number. For example, consider how the number -3 would be stored. The magnitude of this number is 3, whose base-2 representation is 11. So, the 3-bit, signed-magnitude representation of -3 is 1 followed by 11, i.e. 111. The number -2 would be stored as 1 followed by 10, i.e. 110, and so on. The reader should verify that the resulting range of numbers representable in three bits under this system would then be -3 to +3. The reader should also note that the number 0 actually has *two* representations, 000 and 100. The latter could be considered "-0," which of course has no meaning, and 000 and 100 should be considered to be identical. Note too that we see that 100, which in an unsigned system would represent +4, does *not* do so here; indeed, +4 is not representable at all, since our range is -3 to +3.

The 2's complement system handles the negative numbers differently. To explain how, first think of strings of three decimal digits, instead of three bits. For concreteness, think of a 3-digit odometer or trip meter in an automobile. Think about how we could store positive and negative numbers on this trip meter, if we had the desire to do so. Since there are 10 choices for each digit (0,1,...,9), and there are three digits, there are $10^3 = 1000$ possible patterns. So, we would be able to store numbers which are approximately in the range -500 to +500.

Suppose we can wind the odometer forward or backward with some manual control. Let us initially set the odometer to 000, i.e. set all three digits to 0. If we were to wind *forward* from 000 once, we would get 001; if we were to wind forward from 000 twice, we would get 002; and so on. So we would use the odometer pattern 000 to represent 0, 001 to represent +1, 002 to represent +2, ..., and 499 to represent +499. If we were to wind *backward* from 000 once, we would get 999; if we were to wind backward twice, we would get 998; and so on. So we would use the odometer pattern 999 to represent -1, use 998 to represent -2, ..., and use 500 to represent -500 (since the odometer would read 500 if we were to wind backward 500 times). This would give us a range -500 to +499 of representable numbers.

Getting back to strings of three binary digits instead of three decimal digits, we apply the same principle. If we wind backward once from 000, we get 111, so we use 111 to represent -1. If we wind backward twice from 000, we get 110, so 110 will be used to represent -2. Similarly, 101 will mean -3, and 100 will mean -4. If we wind backward one more time, we get 011, which we already reserved to represent +3, so -4 will be our most negative representable number. So, under the 2's complement system, 3-bit strings can represent any integer in the range -4 to +3.

This may at first seem to the reader like a strange system, but it has a very powerful advantage: We can do addition of two numbers without worrying about their signs; whether the two addends are both positive, both negative or of mixed signs, we will do addition in the same manner. For example, look at the base-10 case above, and suppose we wish to add +23 and -6. These have the "trip meter" representations 023 and 994. Adding 023 and 994 yields 1017, but since we are working with 3-digit quantities, the leading 1 in 1017 is lost, and we get 017. 017 is the "trip meter" representation of +17, so our answer is +17—exactly as it should be, since we wanted to add +23 and -6. The reason this works is that we have first wound forward 23 times (to 023) but then wound backward 6 times (the 994), for a net winding forward 17 times.

The importance of this is that in building a computer, the hardware to do addition is greatly simplified. The same hardware will work for all cases of signs of addends. For this reason, most modern computers are designed to use the 2's-complement system.

Although we have used the "winding backward" concept as our informal definition of 2's complement representation of negative integers, it should be noted that in actual computation—both by us humans and by the hardware—it is inconvenient to find representations this way. For example, suppose we are working with 8-bit strings, which allow numbers in the range -128 to +127. Suppose we wish to find the representation of -29. We *could* wind backward from 00000000 29 times, but this would be very tedious.

Fortunately, a "shortcut" method exists: To find the n-bit 2's complement representation of a negative number -x, do the following.

- (a) Find the n-bit base-2 representation of +x, making sure to include any leading 0s.
- (b) In the result of (a), replace 0s by 1s and 1s by 0s.
- (c) Add 1 to the result of (b), ignoring any carry coming out of the Most Significant Bit.

For instance, consider the example above, in which we want to find the representation of -29 in an 8-bit string. We first find the representation of +29, which is 00011101 [note that we remembered to include the three leading 0s, as specified in (a) above]. Applying Step (b) to this, we get 11100010. Adding 1, we get 11100011. So, the 8-bit two's complement representation of -29 is 11100011. We would get this same string if we wound back from 000000 29 times, but the method here is much quicker.

This transformation is its own inverse, i.e. if you take the 2's complement representation of a negative number -x and apply Steps (b) and (c) above, you will get +x. The reader should verify this in the example in the last paragraph: Apply Steps (b) and (c) to the bit string 11100011 representing -29, and verify that the resulting bit string does represent +29. In this way, one can find the base-10 representation of a negative number for which you have the 2's complement form.

By the way, the n-bit representation of a negative integer -x is equal to the base-2 representation of $2^n - x$. You can see this by looking at the base-10 example we used earlier. The corresponding statement would be that -x is represented as $10^n - x$, and since n = 3 in that example, the statement says that -x is represented as 1,000 - x. For example, recall that in that example, -2 was represented by 998, which is indeed 1,000 - 2. Similarly, the number -3 was represented by 997, which is 1,000 - 3, and so on.

Using this fact, you can now see why the "shortcut" above works: Let Q denote the n-bit string consisting of all 1s, and let A, B and C be the results of Steps (a), (b) and (c) applied to finding the n-bit 2's complement representation of a negative number -x. We then have the following:

- (1) A + B = Q [by definition of Step (b) and Q].
- (2) C = B + 1 [by definition of Step (c)].
- (3) C = Q - A + 1 [from (1) and (2)].
- (4) C = 2^n - A [since from definition of Q, $Q + 1 = 2^n$].

We found above that the right-hand side of 4. is the n-bit 2's complement representation of -x, which is indeed what we have been claiming for C.

The reader should also verify the following properties in the case of 4-bit strings. They are true for general n.

- (i) The range of integers which is supported by the n-bit, 2's complement representation is -2^{n-1} to $2^{n-1}-1$.
- (ii) The values -2^{n-1} and $2^{n-1}-1$ are represented by 10000...000 and 01111...111, respectively.
- (iii) All nonnegative numbers have a 0 in Bit n-1, and all negative numbers have a 1 in that bit position.

By the way, due to the slight asymmetry in the range in (i) above, you can see that we can not use the "shortcut" method if we need to find the 2's complement representation of the number -2^n-1; Step (a) of that method would be impossible, since the number 2^n-1 is not representable. Instead, we just use (ii).

We can now solve the mystery of the C statement

```
Sum = X + Y;
```

discussed in Chapter 0. It was asserted there that the value of Sum might become negative, even if both the values X and Y are positive. The reader should now be able to verify this: With 16-bit storage and X = 28,502 and Y = 12,344, the resulting value of Sum will be -24,690. Most machines have special bits which can be used to detect such situations, so that we do not use misleading information.

Again, most modern machines use the 2's complement system for storing signed integers. We will assume this system from this point on, except where stated otherwise.

3.2 Representing Real Number Data

The main idea here is to use **scientific notation**, familiar from physics or chemistry, say 3.2×10^{-4} for the number 0.00032. In this example, 3.2 is called the **mantissa** and -4 is called the **exponent**.

The same idea is used to store real numbers, i.e. numbers which are not necessarily integers (also called **floating-point** numbers), in a computer. The representation is essentially of the form

$$m \times 2^n,$$

with m and n being stored as individual bit strings. If for example we were to store real numbers as 16-bit strings, we might devote 10 bits, say Bits 15-6, to the mantissa m, and 6 bits, say Bits 5-0, to the exponent n. Then the number 1.25 might be represented as

$$5 \times 2^{-2},$$

that is, with m = 5 and n = -2. As a 10-bit 2's complement number, 5 is represented by the bit string 0000000101, while as a 6-bit 2's complement number, -2 is represented by 111110. Thus we would store the number 1.25 as the 16-bit string 0000000101 111110 i.e.

```
0000000101111110 = 0x017e
```

Note the design tradeoff here: The more bit sI devote to the exponent, the wider the range of numbers I can store. But the more bits I devote to the mantissa, the less roundoff error I will have during computations. Once I have decided on the string size for my machine, in this example 16 bits, the question of partitioning these bits into mantissa and exponent sections then becomes one of balancing accuracy and range.

The floating-point representation commonly used on today's machines is a standard of the Institute of Electrical and Electronic Engineers (IEEE). It uses 32-bit storage and follows the same basic principles, but has a couple of refinements to the simplest mantissa/exponent format. It consists of a Sign Bit, an 8-bit Exponent Field, and 23-bit Mantissa Field. These fields will now be explained. Keep in mind that there will be a distinction made between the terms *mantissa* and *Mantissa Field*, and between *exponent* and *Exponent Field*.

Recall that in base-10, digits to the right of the decimal point are associated with negative powers of 10. For example, 4.38 means

$$4(10^0) + 3(10^{-1}) + 8(10^{-2}).$$

It is the same principle in base-2, of course, with the base-2 number 1.101 meaning

$$1(2^0) + 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}),$$

that is, 1.625 in base-10.

Under the IEEE format, the mantissa must be in the form $\pm 1.x$, where 'x' is some bit string. In other words, the absolute value of the mantissa must be a number between 1 and 2. The number 1.625 is 1.101 in base-2, as seen above, so it already has this form. Thus we would take the exponent to be 0, that is, we would represent 1.625 as

$$1.101 \times 2^0.$$

What about the number 0.375? In base-2 this number is 0.011, so we *could* write 0.375 as

$$0.011 \times 2^0$$

but again, the IEEE format insists on a mantissa of the form $+-1.x$ So, we would write 0.375 instead as

$$1.1 \times 2^{-2},$$

which of course is equivalent, but the point is that it fits IEEE's convention.

Now since that convention requires that the leading bit of the mantissa be 1, there is no point in storing it! Thus the Mantissa Field only contains the bits to the right of that leading 1, so that the mantissa consists of $+-1.x$, where 'x' means the bits stored in the Mantissa field. The sign of the mantissa is given by the Sign Bit, 0 for positive, 1 for negative.

The Exponent Field actually does *not* directly contain the exponent; instead, it stores the exponent plus a **bias** of 127. The Exponent Field itself is considered as an 8-bit unsigned number, and thus has values ranging from 0 to 255. However, the values 0 and 255 are reserved for "special" quantities: 0 means that the floating-point number is 0, and 255 means that it is in a sense "infinity," the result of dividing by 0, for example. Thus the Exponent Field has a range of 1 to 254, which after accounting for the bias term mentioned above, means that the exponent is a number in the range -126 to +127 (1-127 = -126 and 254-127 = +127).

With all this in mind, let us find the representation for the example number 1.625 mentioned above. We found that the mantissa is 1.101 and the exponent is 0. Thus the Mantissa Field is

```
10100000000000000000000
```

(the mantissa is 1.101, but remember that we do not store the '1.' in the Mantissa Field). The Exponent Field is 0 + 127 = 127, or in bit form,

01111111

The Sign Bit is 0, since 1.625 is a positive number.

So, how are the three fields then stored altogether in one 32-bit string? Well, 32 bits fill four bytes, say at addresses n, n+1, n+2 and n+3. The format for storing the three fields is then as follows:[1]

- Byte n: least significant eight bits of the Mantissa Field
- Byte n+1: middle eight bits of the Mantissa Field
- Byte n+2: least significant bit of the Exponent Field, and most significant seven bits of the Mantissa Field
- Byte n+3: Sign Bit, and most significant seven bits of the Exponent Field

Suppose for example, we have a variable, say T, of type **float** in C, which the compiler has decided to store beginning at Byte 0x304a2. If the current value of T is 1.625, the bit pattern will be

```
Byte 0x304a2:  00; Byte 0x304a3:  00; Byte 0x304a4:  D0; Byte 0x304a5:  3F
```

As another check, the reader should verify that if the four bytes' contents are 0xe1 0x7a 0x60 0x42, then the number being represented is 56.12.

3.3 Representing Character Data

This is merely a matter of choosing which bit patterns will represent which characters. The two most famous systems are the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Information Code (EBCDIC). ASCII stores each character as the base-2 form of a

[1] This assumes a **little-endian** machine, which we will discuss later.

number between 0 and 127. For example, 'A' is stored as 65_{10} (01000001 = 0x41), '%' as 37_{10} (00100101 = 0x25), and so on.

A complete list of standard ASCII codes may be obtained by typing

```
man ascii
```

on most Unix systems. Note that even keys such as Carriage Return, Line Feed, and so on, are considered characters, and have ASCII codes.

Since ASCII codes are taken from numbers in the range 0 to $2^7 - 1 = 127$, each code consists of seven bits. The EBCDIC system consists of eight bits, and thus can code 256 different characters, as opposed to ASCII's 128. In either system, a character can be stored in one byte. The vast majority of machines today use the ASCII system.

What about characters in languages other than English? Codings exist for them too. Consider for example Chinese. Given that there are tens of thousands of characters, far more than 256, two bytes are used for each Chinese character. Since documents will often contain both Chinese and English text, there needs to be a way to distinguish the two. Big5 and Guobiao, two of the most widely-used coding systems used for Chinese, work as follows. The first of the two bytes in a Chinese character will have its most significant bit set to 1. This distinguishes it from ASCII (English) characters, whose most significant bits are 0s. The software which is being used to read (or write) the document, such as **celvis**, a Chinese version of the famous **elvis** text editor, will inspect the high bit of a byte in the file. If that bit is 0, then the byte will be interpreted as an ASCII character; if it is 1, then that byte and the one following it will be interpreted as a Chinese character.

3.4 Representing Machine Instructions

Each computer type has a set of binary codes used to specify various operations done by the computer's **Central Processing Unit** (CPU). For example, in the Intel CPU chip family, the code 0xc7070100, i.e.

```
11000111000001110000000100000000,
```

means to put the value 1 into a certain cell of the computer's memory. The circuitry in the computer is designed to recognize such patterns and act accordingly. You will learn how to generate these patterns in later chapters, but for now, the thing to keep in mind is that a computer's machine instructions consist of patterns of 0s and 1s.

Note that an instruction can get into the computer in one of two ways:

- (a) We write a program in machine language (or assembly language, which we will see is essentially the same), directly producing instructions such as the one above.
- (b) We write a program in a high-level language (HLL) such as C, and the compiler translates that program into instructions like the one above.

[By the way, the reader should keep in mind that the compilers themselves are programs. Thus they consist of machine language instructions, though of course these instructions might have themselves been generated from an HLL source too.]

3.5 What Type of Information is Stored Here?

A natural question to ask at this point would be how the computer "knows" what kind of information is being stored in a given bit string. For example, suppose we have the 16-bit string 0111010000101011, i.e. in hex form 0x742b, on a machine using an Intel CPU chip in 16-bit mode. Then

- (a) if this string is being used by the programmer to store a signed integer, then its value will be 29,739;
- (b) if this string is being by the programmer used to store characters, then its contents will be the characters 't' and '+';
- (c) if this string is being used by the programmer to store a machine instruction, then the instruction says to "jump" (like a **goto** in C) forward 43 bytes.

So, in this context the question raised above is,

> How does the computer "know" which of the above three kinds (or other kinds) of information is being stored in the bit string 0x742b? Is it 29,739? Is it 't' and '+'? Or is it a jump-ahead-43-bytes machine instruction?

The answer is, "The computer does *not* know!" As far as the computer is concerned, this is just a string of 16 0s and 1s, with *no* special meaning. So, the responsibility rests with the person who writes the program—he or she must remember what kind of information he or she stored in that bit string. If the programmer makes a mistake, the computer will not notice, and will carry out the programmer's instruction, no matter how ridiculous it is. For example, suppose the programmer had stored *characters* in each of two bit strings, but forgets this and mistakenly thinks that he/she had stored *integers* in those strings. If the programmer tells the computer to multiply those two "numbers," the computer will dutifully obey!

The discussion in the last paragraph refers to the case in which we program in machine language directly. What about the case in which we program in an HLL, say C, in which the *compiler* is producing this machine language from our HLL source? In this case, during the time the compiler is translating the HLL source to machine language, the compiler must "remember" the type of each variable, and react accordingly. In other words, the responsibility for handling various data types properly is now in the hands of the compiler, rather than directly in the hands of the programmer—but still not in the hands of the hardware, which as indicated above, remains ignorant of type.

Here are a number of examples of this:[2]

[2]The rather large number of examples should be a hint to the reader of the vast importance of this concept. Keep it in mind in all your programming activities.

3.5.1 Example

As an example, suppose in a C program X and Y are declared of types **char** and **int**, respectively, and the program includes the statement

```
X += Y;
```

The C language is not **strongly typed**, so a C compiler faced with the above statement will uncomplainingly go ahead and produce machine instructions which perform the indicated addition. By contrast, the C++ language *is* strongly typed, and a C++ compiler would respond to the above statement with a compile-time error message.[3]

The point here is that it is the software which is controlling this, not the hardware. The hardware will obey whichever machine instructions you give it, even if they are nonsense.

3.5.2 Example

So, the machine doesn't know whether we humans intend the bit string we have stored in a word to be interpreted as an integer or as a 4-element character string or whatever. To the machine, it is just a bit string, 32 bits long.

The place the notion of types arises is at the compiler/language level, not the machine level. The C language has its notion of types, e.g. **int** and **char**, and the compiler produces machine code accordingly.[4] But that machine code itself does not recognize type. Again, the machine cannot tell whether the contents of a given word are being thought of by the programmer as an integer or as a 4-character string or whatever else.

For example, consider this code:

```
...
int Y;  // local variable
...
strncpy(&Y,"abcd",4);
...
```

At first, you may believe that this code would not even compile successfully, let alone run correctly. After all, the first argument to **strncpy()** is supposed to be of type **char ***, yet we have the argument as type **int ***. But the C compiler, say **gcc**, will indeed compile this code without error,[5] and the machine code will indeed run correctly, placing "abcd" into X. The machine won't know about our argument type mismatch.

If we run the same code through a C++ compiler, say **g++**, then the compiler will give us an error message, since C++ is strongly typed. We will then be forced to use a cast:

[3]Note that the **gcc** compiler integrates a C compiler with a C++ compiler. This is possible since (ANSI standard) C is a subset of C++, except that C++ enforces strong typing. When you run **g++** on a C++ source file, **g++** actually is a script which calls **gcc**. But it does so with the option "-x c++", which tells **gcc** to consider this a C++ program. That is why a C file that has type mixing will cause no problem other than a warning if run through **gcc**, but will fail to compile if run through **g++**.

[4]The compiler will generate word-accessing machine instructions for **int**s and byte-accessing machine instructions for **char**s.

[5]It may give a warning message, though.

```
strncpy((char *) &Y,"abcd",4);
```

3.5.3 Example

Consider the following program:

```
main()

{  int X[5],Y[20];

   X[0] = 12;
   Y[20] = 15;
   printf("X[0] = %d\n",X[0]);  // prints out 15!
}
```

Say we compile this, say to **a.out**, and run it. What will happen?

There appears to be a glaring problem with Y here. We assign 15 to Y[20], yet to us humans there is no such thing as Y[20]; the last element of Y is Y[19]. But at the machine level there is no such thing as an array. Y is just a name for the first word of the 20 words we humans think of as being a package here.[6] When we write the C code Y[20], the compiler merely translates that to machine code which accesses "the word 20 words after Y." The C compiler will NOT consider such a reference to be illegal.

So, where is "Y[20]"? Recall that since these are local variables in main(), they will be stored in reverse order, i.e. Y first and then X. So, X[0] will follow immediately after Y[19]. Thus "Y[20]" is really X[0], and thus X[0] will become equal to 15!

3.5.4 Example

As another example, consider the C library function **printf()**, which is used to write the values of program variables to the screen. Consider the C code

```
int W;
...
W = -32697;
printf("%d %u %c\n",W,W,W);
```

again on a machine using an Intel CPU chip in 16-bit mode. We are printing the bit string in W to the screen three times, but are directing that this bit string be interpreted first as a decimal signed integer (%d); then as a decimal unsigned integer (%u); then as an ASCII character (%c); and then go to a new line. Here is the output that would appear on the screen:

[6] That's why, by the way, the expression "Y" by itself, without a subscript, means the address of Y[0].

```
-32697  32839  G
```

The bit string in W is 0x8047. Interpreted as a 16-bit 2's complement number, this string represents the number -32,697. Interpreted as an unsigned number, this string represents 32,839. If the least significant 8 bits of this string are interpreted as an ASCII character, they represent the character 'G'.

But remember, the key point is that the *hardware* is ignorant; it has no idea as to what type of data we intended to be stored in W's memory location. The interpretation of data types was solely in the software. As far as the hardware is concerned, the contents of a memory location is just a bit string, nothing more.

In fact, we can view that bit string without interpretation as some data type, by using the %x format in the call to **printf()**. This will result in the bit string itself being printed out (in hex notation). In other words, we are telling **printf()**, "Just tell me what the bit string is; don't do any interpretation."

A similar situation occurs with input. Say we have the statement

```
scanf("%x",&X);
```

and we input bbc0a168. Then we are saying, "Put 0xb, i.e. 1011, in the first 4 bits of X (i.e. the most-significant 4 bits of X), then put 1011 in the next 4 bits, then put 1100 in the next 4 bits, etc."

By contrast, if we have the statement

```
scanf("%d",&X);
```

and we input, say, 168, then we are saying, "Interpret the characters typed at the keyboard as describing the base-10 representation of an integer, then calculate that number (do $1 \times 100 + 6 \times 10 + 8$), and store it in X."

So, in summary, in the first of the two **scanf()** calls above we are simply giving the machine specific bits to store in X, while in the second one we are asking the machine to convert what we input into some bit string and place that in X.

By the way, **printf()** is only a function within the C library, and thus does not itself directly do I/O. Instead, **printf()** calls another function, **write()**, which is in the operating system (and thus the call is referred to as a **system call**).[7] All **printf()** does is convert what we want to write to the proper sequence of bytes, and then **write()** does the actually writing to the screen.

Similarly, when we call **scanf()**, it in turn calls the system function **read()**, which does the actual reading from the keyboard.

This idea of "who is responsible for what?"—e.g. asking whether a particular concept is in the domain of hardware versus software—will be one of the themes of this book.

Let us illustrate this with one more point on the example above, in which "-32697" is printed. The machine instructions in **write()** print out the character string "-32697" to the screen. In other words, those instructions

[7] The name of the function is write() on UNIX; the name is different from other OSs.

will first send 0x2d, the ASCII code for the character '-', to the screen, 0x33 for '3', 0x32 for '2', and so on. It is easy to take this for granted, but there is quite a bit involved here.

First of all, keep in mind the vast difference between the *number* -32697 and the *character string* "-32697". In our context, it is the former which is stored in W, as 0x8047, i.e.

```
1000000001000111
```

(Verify this!) In order to convert the part after the negative sign, i.e. convert 32,697, to the characters '3', '2', '6', '9' and '7', the code in **printf()** must do repeated division: It first divides by 10,000, yielding 3 with a remainder of 2,697. The 3, i.e. 11, is changed to '3', i.e. the ASCII 00110011. The 2,697 is then divided by 1,000, yielding 2 with a remainder of 697. The 2 is converted to '2', i.e. 00110010, and so on. (We will then divide by 100 and 10.)

The second question is, how will these characters then be displayed on the screen?

You may already know that a computer monitor consists of many dots called **pixels**. There may be, for instance, 1028 rows of 768 pixels per row. Any image, including characters, are formed from these pixels.

On a nongraphics screen (often called a **plain – or "dumb" – ASCII terminal**), the hardware directly forms the dots for whichever ASCII character it receives from the computer. Software has no control over individual pixels; it only controls which ASCII character is printed. In our example above, for instance, the software sends 0x33 along the cable from the computer to the screen, and the screen then forms the proper dots to draw the character '3', called the **font** for '3'.

By contrast, on a graphics screen, software has control over individual pixels. For concreteness, consider an **xterm** window on Unix systems. (You need not have used **xterm** to follow this discussion. If you have used Microsoft Windows, you can relate the discussion to that context if you wish.) The **printf()** call we saw above will send the value 0x33 not directly to the screen, but rather to the **xterm** program. The latter will look up the font for '3' and then send it to the screen, and the screen hardware then displays it.

A Chinese version of **xterm**, named **cxterm**, will do the same, except it will first sense whether the character is ASCII or Chinese, and send the appropriate font to the screen.

Programs which display images then work on individual pixels at an even finer level.

4 Main Memory Organization

During the time a program is executing, both the program's data and the program itself, i.e. the machine instructions, are stored in main memory. In this section, we will introduce main memory structure. (We will usually refer to main memory as simply "memory.")

4.1 Bytes, Words and Addresses

4.1.1 The Basics

Memory can be viewed as a long string of consecutive bytes. Each byte has an identification number, called an **address**. Again, an address is just an "i.d. number," like a Social Security Number identifies a person,

a license number identifies a car, and an account number identifies a bank account. Byte addresses are consecutive integers, so that the memory consists of Byte 0, Byte 1, Byte 2, and so on.

On each machine, a certain number of consecutive bytes is called a **word**. The number of bytes or bits (there are eight times as many bits as bytes, since a byte consists of eight bits) in a word in a given machine is called the machine's **word size**. This is usually defined in terms of the size of number which the CPU addition circuitry can handle, which for most machines today is 32 bits. In other words, the CPU's adder inputs two 32-bit numbers, and outputs a 32-bit sum, so we say the word size is 32 bits.

Most MIPS, SPARC and other CPUs popular today have 32-bit words, although the really powerful machines now typically have 64-bit words.

Early members of the Intel CPU family had 16-bit words, while the later ones were extended to their current 32-bit size. In order to ensure that programs written for the early chips would run on the later ones, Intel designed the later CPUs to be capable of running in two modes, one with 16-bit words and the other being a 32-bit mode.

Note carefully that most machines do not allow overlapping words. That means, for example, that on a 32-bit machine, Bytes 0-3 will form a word and Bytes 4-7 will form a word, but Bytes 1-4 do NOT form a word. If your program tries to access the "word" consisting of Bytes 1-4, it will cause an execution error. On Unix systems, for instance, you will get the error message "bus error."

Just as a bit string has its most significant and least significant bits, a word will have its most significant and least significant bytes. To illustrate this, suppose word size is 32 bits and consider storage of the integer 25, which is

```
00000000000000000000000000011001
```

in bit form and 0x00000019 as hex. Three bytes will each contain 0x00 and the fourth 0x19, with the 0x19 byte being the least significant and the first 0x00 byte being most significant.

4.1.2 "Endian-ness"

Not only does each byte have an address, but also each word has one too. The address of a word will be the address of its lowest-address byte. A question, though, is whether that lowest-address byte is the most or least significant byte.

The Intel family handles this in a **little-endian** manner, meaning that the least significant byte within a word has the lowest address. For instance, consider the above example of the integer 25. Suppose it is stored in Word 204, which on any 32-bit machine will consist of Bytes 204, 205, 206 and 207. On an Intel machine (or any other 32-bit little-endian machine), Byte 204 will be the least significant byte, and thus in this example will contain 0x19.

Note carefully that when we say that Byte 204 contains the least significant byte, what this really means is that the arithmetic hardware in our machine will treat it as such. If for example we tell the hardware to add the contents of Word 204 and the contents of Word 520, the hardware will start at Bytes 204 and 520, not at Bytes 207 and 523. First Byte 204 will be added to Byte 520, recording the carry, if any. Then Byte 205 will be added to Byte 521, plus the carry if any from the preceding byte addition, and so on, through Bytes 207 and 523.

SPARC chips, on the other hand, assign the least significant byte to the highest address, a **big-endian** scheme. This is the case for IBM mainframes too, as well as for the Java Virtual Machine.

Some chips, such as MIPS and PowerPC, even give the operating system a choice as to which rules the CPU will follow; when the OS is booted, it establishes which "endian-ness" will be used.

The endian-ness can certainly make a difference. As an example, suppose we are writing a program to sort character strings in alphabetical order. The word "card" should come after "car" but before "den", for instance. Suppose our machine has 32-bit word size, variables X and Y are of type **int**, and we have the code

```
strncat(&X,"card",4);
strncat(&Y,"den ",4);
```

Say X and Y are in Words 240 and 804, respectively. This means that Byte 240 contains 0x63, the ASCII code for 'c', Byte 241 contains 0x61, the code for 'a', byte 242 contains 0x72, the code for 'r' and byte 243 contains 0x64. Similarly, bytes 804-807 will contain 0x64, 0x65, 0x6e and 0x20. All of this will be true regardless of whether the machine is big- or little-endian.

But on a big-endian machine, we can actually use word subtraction to determine which of the strings "card" and "den" should precede the other alphabetically, by subtracting Word 240 from Word 804. To see that this works, note that the contents of the two words will be 0x63617264 (1667330660 decimal) and 0x64656e20 (1684368928 decimal), so the result of the subtraction will be negative, and thus we will decide that "card" is less than (i.e. alphabetically precedes) "den"—exactly what we want to happen.

The reader should pause to consider the speed advantage of such a comparison. If we did not use word subtraction, we would have to do a character-by-character comparison, that is four subtractions. (Note that these are word-based subtractions, even though we are only working with single bytes.) The arithmetic hardware works on a word basis.) Suppose for example that we are dealing with 12-character strings. We can base our sort program on comparing (up to) three pairs of words if we use word subtraction, and thus gain roughly a fourfold speed increase over a more straightforward sort which compares up to 12 pairs of characters.

We could do the same thing on a little-endian machine if we were to store character strings backwards. However, this may make programming inconvenient.

The endian-ness problem also arises on the Internet. If someone is running a Web browser on a little-endian machine but the Web site's server is big-endian, they won't be able to communicate. Thus as a standard, the Internet uses big-endian order. There is a Unix system call, htons(), which takes a byte string and does the conversion, if necessary.

Here is a function that can be used to test the endian-ness of the machine on which it is run:

```
int Endian()  // returns 1 if the machine is little-endian, else 0

{  int X;
   char *PC;
```

```
   X = 1;
   PC = (char *) &X;
   return *PC;
}
```

(Make SURE that you understand why this works.)

4.1.3 Other Issues

Many machines insist that words be **aligned**. In a 32-bit machine, this would mean that words only begin at addresses which are multiples of 4 (32 bits is 4 bytes). Thus Bytes 568-571 would form Word 568, but Bytes 569-572 would not be considered a word.

As we saw above, the address of a word is generally taken to be the address of its lowest-numbered byte. This presents a problem: How can we specify that we want to access, say, Byte 52 instead of Word 52? The answer is that for machine instruction types which allow both byte and word access (some instructions do, others do not), the instruction itself will indicate whether we want to access Byte x or Word x.

For example, we mentioned earlier that the Intel instruction 0xc7070100 in 16-bit mode puts the value 1 into a certain "cell" of memory. Since we now have the terms *word* and *byte* to work with, we can be more specific than simply using the word *cell*: The instruction 0xc7070100 puts the value 1 into a certain *word* of memory; by contrast, the instruction 0xc60701 puts the value 1 into a certain *byte* of memory. You will see the details in later chapters, but for now you can see that differentiating between byte access and word access *is* possible, and is indicated in the bit pattern of the instruction itself.

Note that the word size determines capacity, depending on what type of information we wish to store. For example:

- (a) Suppose we are using an n-bit word to store a nonnegative integer. Then the range of numbers that we can store will be 0 to $2^n - 1$, which for n = 16 will be 0 to 65,535, and for n = 32 will be 0 to 4,294,967,295.
- (b) If we are storing a signed integer in an n-bit word, then the range will be -2^{n-1} to $2^{n-1} - 1$, which will be -32,768 to +32,767 for 16-bit words, and -2,147,483,648 to +2,147,483,647 for 32-bit words.
- (c) Suppose we wish to store characters. Recall that an ASCII character will take up seven bits, not eight. But it is typical that the seven is "rounded off" to eight, with 1 bit being left unused (or used for some other purpose, such as a technique called **parity**, which is used to help detect errors). In that case, machines with 16-bit words can store two characters per word, while 32-bit machines can store four characters per word.
- (d) Suppose we are storing machine instructions. Most RISC machines tend to use a fixed instruction length, equal to the word size. On the other hand, most CISC machines instructions are of variable lengths. On earlier Intel machines, for instance, instructions were of lengths one to six bytes (and the range has grown further since then). Since the word size on those machines was 16 bits, i.e. two

bytes, we see that a memory word might contain two instructions in some cases, while in some other cases an instruction would be spread out over several words. The instruction 0xc7070100 mentioned earlier, for example, takes up four bytes (count them!), thus two words of memory.

It is helpful to make an analogy of memory cells (bytes or words) to bank accounts, as mentioned above. Each individual bank account has an account number and a balance. Similarly, each memory has its address and its contents.

As with anything else in a computer, an address is given in terms of 0s and 1s, i.e. as a base-2 representation of an unsigned integer. The number of bits in an address is called the **address size**. Among earlier Intel machines, the address size grew from 20 bits on the models based on the 8086 CPU, to 24 bits on the 80286 model, and then to 32 bits for the 80386, 80486 and Pentium. Today, most machines, both CISC and RISC, have 32-bit addresses (though several advanced RISC processors have 64-bit addresses).

The address size is crucial, since it puts an upper bound on how many memory bytes our system can have. If the address size is n, then addresses will range from 0 to $2^n - 1$, so we can have at most 2^n bytes of memory in our system. It is similar to the case of automobile license plates. If for example, license plates in a certain state consist of three letters and three digits, then there will be only $26^3 10^3 = 17,560,000$ possible plates. That would mean we could have only 17,560,000 cars and trucks in the state.

Keep in mind that an address is considered as unsigned integer. For example, suppose our address size is, to keep the example simple, four bits. Then the address 1111 is considered to be +15, not -1.

We will use the notation c( ) to mean "contents of," e.g. c(0x2b410) means the contents of memory word 0x2b410.

5 Storage of Variables in HLL Programs

5.1 Basic Principles

When you execute a program, both its instructions and its data are stored in memory. Keep in mind, the word *instructions* here means machine language instructions. Again, these machine language instructions were either written by the programmer directly, or they were produced by a compiler from a source file written by the programmer in C or some other HLL. Let us now look at the storage of the *data* in the C case.[8]

In an HLL program, we specify our data via names. The compiler will assign each variable a location in memory. Generally (but not always, depending on the compiler), these locations will be consecutive.

C compilers tend to store local variables in the reverse of the order in which they are declared. For example, consider the following program:

```
PrintAddr(char *VarName,int Addr)

{  printf("the address of %s is %x\n",VarName,Addr);  }
```

[8]The C language became popular in the early 1980s. Later it was extended to C++, by adding class structures (it was originally called "C with classes"). If you are wondering whether a certain C++ construct is also part of C, the answer is basically that if it is not something involving classes (or the **new** operator), it is C.

```
main()

{  int X,Y,W[4];
   char U,V;

   PrintAddr("X",&X);
   PrintAddr("Y",&Y);
   PrintAddr("W[3]",&W[3]);
   PrintAddr("W[0]",&W[0]);
   PrintAddr("U",&U);
   PrintAddr("V",&V);

}
```

I ran this on a Pentium machine running the Linux operating system, which runs in 32-bit mode. The output of the program was

```
the address of X is bffffb84
the address of Y is bffffb80
the address of W[3] is bffffb7c
the address of W[0] is bffffb70
the address of U is bffffb6f
the address of V is bffffb6e
```

On the other hand, there is wide variation among C compilers as to how they store global variables.

Virtually all C compilers store variables of type **int** in one word, as we have seen above, and we will assume this from now on unless otherwise stated. The same is true for variables of type **float**, as long as the word size is 32 bits, the size of the IEEE floating-point standard; for 16-bit machines, this means that **float** variables must be stored in a pair of words. Typically **char** variables are stored as one byte each.

In the C language, the **sizeof** operator tells us how many bytes of storage the compiler will allocate for any variable type. For instance, the expression

```
sizeof(int)
```

will be equal to the number of bytes that the compiler allocates to a variable of type **int**. An example in which this is useful is the **malloc** function, which is used in C to allocate extra memory **dynamically**, i.e. at the time the program is running rather than at compilation time. Suppose we have a C program which needs to dynamically allocate memory space for 100 variables of type **int**. On a 16-bit machine, we would do this with the C statement

```
P = malloc(200);
```

since each **int** variable needs two bytes of memory. (In executing this statement, the system will allocate the 200 bytes of memory, and then set the variable P to contain the address of that memory space, so as to inform us where that space is.) On the other hand, on a 32-bit machine, our statement would need to be

```
P = malloc(400);
```

It would be very inconvenient to need to have a different version of our program for each different machine. The **sizeof** operator eliminates this need; we can use the single statement

```
P = malloc(100*sizeof(int));
```

on *any* machine.

In general, the C language is very liberal in allowing type-mixing. It sometimes requires the programmer to confirm that he/she is deliberately mixing types; this is done with a **cast**.

As seen in an example above, array variables are generally implemented by compilers as contiguous blocks of memory. For example, the array declared as

```
int X[100];
```

would be allocated to some set of 100 consecutive words in memory.

What about two-dimensional arrays? For example, consider the four-row, six-column array

```
int G[4][6];
```

Again, this array will be implemented in a block of consecutive words of memory, more specifically 24 consecutive words, since these arrays here consist of $4 \times 6 = 24$ elements. But in what order will those 24 elements be arranged? Most compilers use either **row-major** or **column-major** ordering. To illustrate this, let us assume for the moment that this is a global array and that the compiler stores in non-reverse order, i.e. G[0][0] first and G[3][5] last.

In row-major order, all elements of the first row are stored at the beginning of the block, then all the elements of the second row are stored, then the third row, and so on. In column-major order, it is just the opposite: All of the first column is stored first, then all of the second column, and so on.

In the example above, consider G[1][4]. With row-major order, this element would be stored as the 11th word in the block of words allocated to G. If column-major order were used, the it would be stored in the 18th word. (The reader should verify both of these statements, to make sure to understand the concept.) C compilers use the row-major system.

Advanced data types are handled similarly, again in contiguous memory locations. For instance, consider the **struct** type in C, say

```
struct S {
   int X;
   char A,B;
};
```

would on a 16-bit machine be stored in four consecutive bytes, first two for X and then one each for A and B. On a 32-bit machine, it would be stored in six consecutive bytes, since X would take up a word and thus four bytes instead of two.

Note, though, that most compilers will store complex objects like this to be **aligned on word boundaries**. Consider the above **struct** S on a 32-bit machine, and suppose we have the local declaration

```
struct S A,B;
```

A and B collectively occupy 12 bytes, thus seemingly three words. But most compilers will now start an instance of S in the middle of a word. Thus although A and B will be stored contiguously in the sense that no other variables will be stored between them, they will be stored in four consecutive words, not three, with their addresses differing by 8, not 6. There will be two bytes of unused space between them.

In C++, a **class** is stored like a **struct**, except that member functions are stored merely as pointers to the actual functions. This way if we have multiple objects of the same class, we do not waste space by storing multiple copies of a given function.

We have not yet discussed how **pointer** variables are stored. *A pointer is an address.* In C, for example, suppose we have the declaration

```
int *P,X;
```

We are telling the compiler to allocate memory for two variables, one named P and the other named X. We are also telling the compiler what types these variables will have: X will be of integer type, while the ‘’ says that P will be of pointer-to-integer type—meaning that P will store the address of some integer. For instance, in our program, we might have the statement

```
P = &X;
```

which would place the address of X in P. (More precisely, we should define this as the lowest-address byte of X.) If one then executed the statement

```
printf("%x\n",P);
```

the address of X would be printed.

Another aspect of C pointers which explicitly exposes the nature of storage of program objects in memory is **pointer arithmetic**. For example, suppose P is of pointer type. Then P+1, for example, points to the next consecutive object of the type to which P points. If for instance we have the declaration

```
float A,B,C,*Q;
```

and we execute the statement

```
Q = &B;
```

Then Q+1 points to A, if A and B are stored contiguously and in "reverse" order.

Note by the way that in the **float** declaration above, Q may be stored contiguously with A, B and C, even though it is a pointer variable. If so, Q will have the lowest address, followed by C, then B, then A. In fact, if we execute the statement

```
Q = &B;
```

then Q-2 will point to Q!

5.2 Use of the Stack

An executing program will typically have an area of memory reserved for use as a **stack**.[9] Most CPUs have a special register called the **stack pointer** (SP). Whatever word of memory is pointed to by the SP is called the **top of the stack**. Stacks typically grow "downward": If an item is added — **pushed** — onto the stack, the SP is decremented to point to the word preceding the word it originally pointed to, i.e. the word with the next-lower address than that of the word SP had pointed to beforehand. Similarly SP is incremented if the item at the top of the stack is **popped**, i.e. removed. Thus the words following the top of the stack, i.e. with numerically larger addresses, are considered the interior of the stack.

In most modern machine/compiler combinations, function calls are handled via the stack. A machine CALL instruction saves a "return address" (the point to return execution to when the function is finished), and also the compiler produces code to push function parameters on the stack, as well as producing code to allocate stack space for local variables in the called function.

For example, suppose we have a C function whose first few lines are

```
x(int a,int b)
```

```
{  int c,d,e;
```

and that there is a call to the function like this:

```
x(m,n);
```

[9] It is important to always remember that the stack is in memory.

Let's assume a generic machine which has PUSH and CALL instructions, and whose successive word addresses differ by 4.

the compiler will translate the call to something like

```
PUSH N
PUSH M
CALL X
```

Also the compiler will translate the beginning of the function itself to something like

```
SUB SP,12
```

in order to allocate three words on the stack for x's local variables c, d and e.

In other words, for a given function call, the portion of the stack will be devoted to that call will look like this:

```
local variables
return address (1 word past the point where the call was made)
parameters
```

This is called the **stack frame** for that call.

5.3 Variable Names and Types Are Imaginary

When you compile a program from an HLL source file, the compiler first assigns memory locations for each declared variable.[10] For its own reference purposes, the compiler will set up a **symbol table** which shows which addresses the variables are assigned to. However, it is very important to note is that *the variable names in a C or other HLL program are just for the convenience of us humans.* In the machine language program produced from our HLL file by the compiler, all references to the variables are through their addresses in memory, with no references whatsoever to the original variables names. Those names are discarded when the compiler finishes its work.

For example, suppose we have the statement

```
X = Y + 4;
```

When the compiler produces some machine-language instructions to implement this action, the key point is that these instructions will *not* refer to X and Y. Say X and Y are stored in Words 504 and 1712, respectively. Then the machine instructions the compiler generates from the above statement will only refer to Words 504 and 1712, the locations which the compiler chose for the C variables X and Y.

For instance, the compiler might produce the following sequence of three machine language instructions:

[10] These locations will not actually be used at that time. They will only be used later, when the program is actually run.

```
copy Word 1712 to a cell in the CPU
add 4 to the contents of that cell
copy that cell to Word 504
```

There is no mention of X and Y at all! The names X and Y were just *temporary* entities, for communication between you and the compiler. The compiler chose locations for each variable, and then translated all of your C references to those variables to references to those memory locations, as you see here. Again, when the compiler is done compiling your program, it will simply discard the symbol table entirely. If say on a Unix system you type

```
gcc x.c
```

which creates the executable file a.out, that file will not contain the symbol table.

There is one exception to this, though, which occurs when you are debugging a program, using a debugging tool such as **gdb**.[11] Within the debugging tool you will want to be able to refer to variables by name, rather than by memory location, so in this case you do need to tell the compiler to retain the symbol table in a.out. You do so with the -g option, i.e.

```
gcc -g x.c
```

But even in this case, the actually machine-language portion (as opposed to the add-on section containing the symbol table) of the executable file a.out will not make any references to variable names.

Similarly, there is no such thing as a type of a variable at the machine level, as discussed in Section 3.5.

5.4 Segmentation Faults and Bus Errors

Most machines today which are at the desktop/laptop level or higher (i.e. excluding handheld PDAs, chips inside cars and other machines, etc.) have **virtual memory** (VM) hardware. We will discuss this in another unit in detail, but the relevance here is that it can be used to detect situations in which a program tries to access a section of memory which was not allocated to it.

In a VM system, memory is (conceptually) broken into chunks called **pages**. When the operating system loads a program, say **a.out**, into memory, the OS also sets up a **page table**, which shows exactly which parts of memory the program is allowed to access. Then when the program is actually run, the hardware monitors each memory access, checking the page table. If the program accesses a part of memory which was not allocated to it, the hardware notices this, and switches control to the OS, which announces an error and kills in the program. This is called a **segmentation fault** in UNIX and a **general protection fault** in VM Windows systems.

For example, consider the following program, which we saw in Section 3.5:

[11] tool when you are debugging a program. *Don't debug by just inserting a lot of **printf()** calls!* Use of a debugging tool will save you large amounts of time and effort. Professional software engineers use debugging tools as a matter of course. There is a lot of information on debugging, and debugging tools, on my debug Web page, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`

```
main()

{  int X[5],Y[20];

   X[0] = 12;
   Y[20] = 15;
   printf("X[0] = %d\n",X[0]);  // prints out 15!
}
```

Say we compile this, say to **a.out**, and run it. What will happen?

As noted earlier, Y[20] is a perfectly legal expression here, which will turn out to be identical to X[0]. That's how X[0] becomes equal to 12.

Even though that is clearly not what the programmer intended, the VM hardware will not detect this problem, and the program will execute without any announcement of error. The reason is that the program does not access memory that is not allocated to it.

On the other hand, if Y[20] above were Y[20000], then the location of "Y[20000]" would be far outside the memory allocated to the program. The hardware would detect this, then jump to the OS, which would (in UNIX) shout out to us, "Segmentation fault."

So, remember:

> A "seg fault" error message means that your program has tried to access a portion of memory which is outside the memory allocated to it. This usually is due to a buggy pointer variable.

The first thing you should do when a seg fault occurs is to find out where in the program it occurred. **Use a debugging tool, say DDD or GDB, to do this!**

Note again that seg faults require that our machine have VM hardware, and that the OS is set up to make use of that hardware. Without both of these, an error like "Y[20000]" will NOT be detected.

Many CPU types require that word accesses be **aligned**. On Intel machines, for example, words must begin at addresses which are multiples of 4. Word 200, for instance, consists of Bytes 200, 201, 202 and 203, and Word 204 consists of Bytes 204-207, but there is no such thing as "Word 201," consisting of Bytes 201-204. Suppose your program has code like

```
int X,*P;
...
P = (int *) 201;
X = *P;
```

The compiler will translate the line

```
X = *P;
```

into some word-accessing machine instruction. When that instruction is executed, the CPU hardware will see that P's value is not a multiple of 4.[12] The CPU will then cause a jump to the operating system, which will (for example in UNIX) announce "Bus error."

As is the case with seg faults, you should use a debugging tool to determine where a bus error has occurred.

A "Binary" Files

You will encounter the term *binary file* quite often in the computer world, so it is important to understand it well.

A.1 Disk Geometry

Files are stored on disks. A disk is a rotating round platter with magnetized spots on its surface.[13] Each magnetized spot records one bit of the file.

The magnetized spots are located on concentric rings called **tracks**. Each track is further divided in **sectors**, consisting of, say 512 bytes (4096 bits) each.

When a file is created, the operating system finds unused sectors on the disk in which to place the bytes of the file. The OS then records the locations (track number, sector number within track) of the sectors of the file.

A.2 Definition of "Binary File"

First, keep in mind that the term **binary file** is a misnomer. After all, ANY file is "binary," in the sense that it consists of bits. (Each bit is a magnetized spot on the disk surface.)

What the term **binary file** really means is that bytes in the file are not intended to be interpreted as ASCII. The term **text file** means that the bytes are to be interpreted as ASCII. That's all there is to it, nothing more than that.

Say for example I use a text editor, say the **vim** extension of **vi**, to create a file named z, whose contents are

```
The quick brown fox
jumped over the fence.
```

Then **vim** will write the ASCII codes for the characters 'T', 'h', 'e' and so on (including the ASCII code for newline in the OS we are using) onto the disk.

If on the other hand I have a C source file x.c and type

```
gcc x.c
```

[12] Note that this does not require VM capability.

[13] Colloquially people refer to the disk as a **disk drive**. However, that term should refer only to the motor which turns the disk.

the compiler will write the machine-language file a.out to the disk. Many of the bytes in that a.out file will be ASCII code, but this is just a coincidence; they are intended to be interpreted as machine language, not characters.

A.3 What Does "Interpreted" Mean Here?

Note the word *interpreted* above. What does that really mean? Suppose for example we now display the file z on our screen by typing

```
cat z
```

Your screen will then indeed show the following:

```
The quick brown fox
jumped over the fence.
```

The reason this occurred is that the **cat** program did interpret the contents of the file z to be ASCII codes. What "interpret" means here is the following:

Consider what happens when **cat** reads the first byte of the file, 'T'. The ASCII code for 'T' is 84, i.e. 0x54 = 01010100. The program **cat** contains **printf()** calls which use the %c format. This format sends the byte, in this case to the screen hardware. The latter looks up the font corresponding to the number 84, which is the font for 'T', and that is why you see the 'T' on the screen.

By contrast, consider what would happen if you were to type

```
cat a.out
```

for that machine-language file discussed above. The **cat** program will NOT know that a.out is not a text file; on the contrary, **cat** assumes that any file given to it will be a text file. Thus **cat** will use %c format and the screen hardware will look up fonts, etc., even though it is all meaningless.

A.4 An Example

Consider the difference between the two writes to disk here:

```
int i = 38;

FILE *f1;
```

```
f1 = fopen("abc","w");
int f2 = open("definitelyNLY|O_CREAT,0700);

...

printf("%d",i);
fprintf(f1,"%d",i);
write(f2,&i,sizeof(int));
```

Recall that **printf()** and **fprintf()** actually produce calls to **write()**, so we actually have three calls to the latter in the code fragment above. The difference is that **printf()** and **fprintf()** may do some modification to the data before passing it to **write()**. The system call **write()** simply passes the bytes given to it to whatever destination file is specified (including the screen, which is file number 1, i.e. called via write(1,...)).

The call to **printf()** uses %d format, which is used to display integer values. Since we only understand what we see on the screen, that means that the *characters* '3' and '8', rather than the number 38, must be sent to the screen hardware. If we were to send 38 to the screen hardware, it would print the font associated with ASCII 38, i.e. an ampersand, '&', which is certainly not what we want. The %d format tells **printf()** to prepare the character version of 38, i.e. '3' and '8', and pass those characters — actually, their ASCII codes, 51 and 56 – to **write()**, which will in turn pass them to the screen hardware.

Then **fprintf()** will do the same thing, except that it will direct **write()** to write the characters to the disk instead of to the screen. So, it will result in the following magnetized bits being placed on the disk:

```
0011 0011 0011 1000
```

The direct call to **write()** which we see above would simply pass the bytes given to it to the disk. Since the variable i is of type **int** on a 32-bit machine, its 32 bits are written to the disk. Since 38 = 0x26 =0x00000026, what is written to disk is

```
0000 0000 0000 0000 0000 0000 0010 0110
```

Make SURE you understand all this.