freescale semiconductor

CodeWarriorU

Learn Programming with C

Welcome to the "Learn Programming with C" course from Freescale CodeWarriorU. This course consists of a collection of lessons that will introduce you to the fundamentals of programming using the C programming language

For additional courses, please visit the Embedded Learning Center on the Freescale Semiconductor website.

Lesson 1: Start at the Beginning

Lesson Description: A famous actress and singer noted that the beginning is a "very good place to start." Thus began the tune "Do Re Mi." So, let's start at the beginning by finding out why you're here and what your goals are.

Why Do You Want to Learn Programming?

Why do you want to learn programming? This question should be with you throughout this course. Essentially, there are two kinds of programmers. Those who do it for the pure love of programming, and those who do it as part of their job.

For the Love of Code

The first kind of programmer writes software for the sheer love and enjoyment of coding. These people are the true "hackers" of the world -- not those who break into systems, who are really "crackers."

Programmers in this category often write small programs to help others with a specific problem that they need to solve. These same programmers will often share ideas and even source code with others. Freeware and shareware authors generally fall into this category.

Working Programmers

Working programmers are those who program as part of their daily job. They may be contractors who write software to fulfill a specific need for a company, or they could be developers at a well-known (or maybe not so well-known) software development company.

Of course, just because programming is part of the job does not mean the programmer doesn't love or enjoy programming. However, these programmers have specific tasks in mind, which are often driven by hard deadlines, structure, and goals.

Which Are You?

Now we come back to my first question, "Why do you want to learn programming?" As you are only beginning to learn programming and have no experience, you probably don't fall into either category yet. However, that's not to say that you won't love programming at some point in the future. Chances are that you are seeking direction, trying to find out if you will like programming, or if you should seek another career choice. Programming is not always an easy task. It's sometimes full of frustration, upsets, and stress. But it is also full of joy, rewards, and a sense of pride from accomplishment. So you must give it a solid effort before you make your decision.

This course is very basic. I would encourage you to take at least one or two more programming courses (increasing in skill and knowledge) after this one. I believe that only then will you have the necessary experience and mindset to make a proper decision.

I personally fall into both categories. I program as part of my job, but I also love programming and try to do as much of it as I can in my spare time.

A Brief History of Programming

You've heard the phrase "necessity is the mother of invention." The idea is no different in programming. Strictly speaking, programs are developed out of necessity to solve a problem or to fill a need.

People have always had a desire to solve problems faster. For example, the abacus was developed in ancient China to perform mathematical calculations more quickly and easily. The abacus is a visual tool. The user eventually developed a skill not only for adding quickly but also for pattern matching, i.e., seeing a seven without counting the beads. Pattern matching is an extremely important skill in some high-level programming languages, particularly object-oriented languages such as C++ and Java.

Over time, the need to quickly solve increasingly complex equations became critical in the rapidly expanding fields of physics, chemistry, and engineering. But the flash point came during World War II in British efforts to design machines that could quickly break German codes. These machines had to perform in hours repetitive tasks that would take weeks for a group of people to complete. They were also enormous and broke down frequently. Fully electronic computers eventually evolved from these humble origins. So we've come from a "computer that can fit into a single room" to a computer that can perform a billion calculations per second and fits comfortably on your lap.

This high-velocity advancement requires programmers to continually create new instructions for the computer to follow. Remember, computers are incredibly fast but completely stupid. They only do the tasks you tell them to do. No more, no less.

Problem-Solving for Everyone

Computer programming is based on solving problems, such as "How can I draw a line from point A to point B?" or "How do I write an MP3 player?"

Let's take a practical example to illustrate further. Inventory has to be the most mundane task in any company. However, accurate inventory is vital to successful sales. Someone has to make sure that there is enough product in inventory to sell, order more if there isn't enough, and record all that information somewhere.

Problem: Repetition x Time

Taking inventory used to mean someone had to go around and physically count every item in stock. This task alone could often take several days, depending on the size of the business, the number and type of products being sold, and where those items were located. And this had to be done on a daily, weekly, or monthly basis. Here you see the repetition involved and the amount of time it took.

Next, the inventory would have to be reconciled with what had been sold to make sure everything matched up. This process could involve a little accounting practice as well, maintaining accounts payable and accounts receivable and matching them to inventory.

Finally, analysis of that data would determine how much product to purchase.

Solution: Computer and Software

In today's world, this whole process is virtually eliminated and replaced by a combination of computer hardware and software known as a Point of Sale (POS) system and takes seconds instead of days. The computer keeps track of how much inventory you have, how much you sold, and how much money you made (or didn't make). Additionally, some POS systems include features such as warning you when inventory is low and ordering more at that time electronically, showing you monthly averages of product moved so you can better gauge your business growth, and more. The possibilities really are endless.

Identity

The example above shows one situation and the accompanying solution. However, the entire computer industry is based around the principle and practice of problem solving. Solving problems requires that you first identify the problem (too much time and manpower for inventory). Once you've found the problem, you need to analyze the problem to find out where things can be simplified or automated. Finally, you write code to solve the problem.

Problem solving is a repetitive process that we'll discuss further in the next session.

Seven Steps of Computer Programming

Programming is a process that can be broken down into the following seven steps:

1. Define or refine the problem
2. Design the program
3. Write the code
4. Compile the code
5. Test and debug
6. Document
7. Modify and maintain

Let's take a closer look at each step.

Define the Problem

You've seen one example of this already. Defining the problem is called the problem domain. This step is often the easiest of the seven but that doesn't mean you can approach it haphazardly. This step determines the direction of what follows.

Defining the problem is essential to fully understanding the problem. The initial description of a problem is usually vague. You need to refine the problem so that it becomes precise. Programmers often work with users (the people who state the problem) in order to ensure that both parties understand the problem. From this work, the programmer creates specifications of the problem, including a precise definition of the input data, given data, and desired result.

Design the Program

Once you have the conceptual picture of what the program needs to do, you need to decide how the program will go about it. Questions need to be answered, such as what the interface (if any) should look like. How should the program be organized? Who is the target audience, that is, who will use the software? One important issue is what language will be used to solve the problem. Some programming languages are better at certain tasks than others. For example, the backend database engines used on many Web pages are written in Perl as opposed to C or even Java. On the other hand, you wouldn't use Perl to create a typical commercial application like Photoshop or Excel. For those types of applications you would use a language such as C or C++. Part of learning programming is learning which language is best suited for each type of problem you are trying to solve.

You also need to decide how to represent the data in the program and in files (if required) and what methods you'll use to process that data.

This step is not that specific. You don't need to think about exact lines of code. This step is more for organizing your thoughts and ideas; it helps you to produce algorithms and flowcharts for later use. We'll discuss algorithms and flowcharts in the next lesson.

Write the Code

Now that you have everything you need, you can start writing the code required to solve the problem. This means you translate your design, thoughts, and organization into lines of code that will be executed by the computer.

Compile the Code

Once the code is written, you need to compile it. Compiling is a process your development software goes through to translate your lines of code into a form that can be used by the computer, called machine code. The final product is called an executable, a program that can be run by the computer through user interaction.

The compiler checks and reports errors you've made while writing the code. Typos, unused variables, and forgotten commas or semicolons are all examples of errors the compiler picks up and reports. If there are any errors, you have to go back to step 3 (write the code) to fix them and try compiling again.

Test and Debug

This step can be one of the most tedious and lengthy parts of computer programming. This step involves actually running (executing) your program and trying it out. Input and manipulate data to see if the results match the data you gathered in step 1. This step will help you find any bugs (problems) in the program so you can go back to step 3 and fix them. (Remember, programming is a process.)

Document

Documentation is actually a twofold process. A good programmer needs to document the code he or she writes so that it can be understood many years later when you or someone else tries to modify the software. I strongly believe in documenting your code. This usually means using your language's comment features to describe what is happening in a particular section of code or describe the use of a variable or the like.

The second part of documenting comes near the end. This is the step where you (and/or a technical writer) create the manual on how to use the program you've just written. The technical writer needs to work closely with the programmer (more commonly called the developer) to ensure that all aspects of the program are well documented and can be easily understood by users.

Modify and Maintain

The last step is maintaining your code and modifying it for new features, bug fixes, or other enhancements.

All these steps together form the programming process.

Lesson 2: Algorithms and Flowcharts

Lesson Description: In this lesson, you embark on developing your problem-solving skills.

Algo What?

If you have never heard the term before, you may get tongue-tied just trying to say *algorithm* (pronounced Algo-rith-im).

Simply put, an algorithm is an outline of the steps your program needs to take in order to solve a specific problem. You can think of an algorithm as a recipe, but instead of cooking you are writing source code. Algorithms are written in pseudocode, a mix of English sentence structure with code notation. Pseudocode is completely independent of any programming language.

For example, let's say you need to create a payroll accounting program. Part of that program would be calculating the gross salary of each employee in the company, taking into account overtime (any work above and beyond 40 hours a week), as well as regular work hours. The pseudocode for that program might look like:

```
If Hours greater than 40
    Set Gross Pay to Rate times 40 plus 1.5 times the hours above 40.
Else
(%TAB%)Set gross pay to Rate times Hours
```

Using nothing more than this simple algorithm for calculating an employee's gross pay, we can quickly come up with the source code to handle that part of the problem. Now that you have the basic algorithm, you can replace some statements with their mathematical equivalents, giving you:

```
If Hours > 40
    Set Gross Pay to Rate X 40 + 1.5 X Rate X (Hours-40).
Else
(%TAB%)Set gross pay to Rate X Hours
```

Algorithms should never be used to solve a large problem, but they can be used to show where a problem breaks down into smaller bits; different algorithms can be applied to solve those bits. In the example above, computing gross salary is only one small portion of the overall problem of payroll accounting. There's no mention of tax deductions, employee benefits, and so on. These tasks can be computed elsewhere using different algorithms.

Algorithms to Solve Problems

Algorithms outline how to solve problems. Once again, consider an algorithm equivalent to a cooking recipe. If you want carrot cake, you pull out your favorite recipe. Similarly, in programming, if you want to solve a quadratic equation, you pull out an algorithm that does the job. Algorithms share two other similarities with recipes. The first is that you can find "canned" algorithms all over the place, in books, on the Internet, in computer archives, and elsewhere. The second is that there may be more than one algorithm to solve a given problem.

Canned Algorithms

One of the benefits of many years of computing history is that people have already solved many problems and written the algorithms down to share with colleagues and friends. Programmers face the same problems today and the same algorithms work just as well now as they did when computers first appeared.

There are a lot of good books on algorithms, particularly if you have access to a college or university bookstore. You can think of these books as recipe books; they make great reference material. Don't think that you have to memorize all the algorithms you encounter. While there may be a few "favorites" you will be able to recite on the spot, you generally look up what you need when you need it.

Plethora of Algorithms

Just as there are many recipes for carrot cake, each slightly different from the next, you will find many algorithms that solve the same problem in different ways. Sorting algorithms are a good example. There are many different sorting algorithms: quick sort, selection sort, and bubble sort, to name a few. Each of these algorithms approaches the same problem -- sorting data -- in different ways.

To choose a sorting algorithm (something you'll do in the final project in this course) you need to look at the amount of data to be sorted. While all of the sort algorithms will work regardless of the quantity of data that you're sorting, some are faster than others on large amounts of data. Conversely, some are better at sorting smaller amounts of data (e.g., quick sort).

Efficiency

There exists a whole area of computer science devoted to the study of algorithm efficiency. Efficiency is beyond the scope of this course. However, if you pursue your programming skills further, you can delve into the wonderful world of algorithm analysis.

Flowcharts and Data-Flow Diagrams

While algorithms are word outlines of a program, flowcharts are picture outlines. There are many different types of "visual" tools to help in program design. Flowcharts are among the most commonly used, as they can aid in creating algorithms. Why mention flowcharts after algorithms then? It's not necessary to create a flowchart to create an algorithm. I believe that understanding algorithms is more important than understanding flowcharts. Remember the algorithm books? You won't find any flowchart books and very few algorithm books use flowcharts to explain an algorithm.

Flowcharting a Program

Flowcharts are decision-based. Let's go back to our example of calculating an employee's gross pay. What decisions are involved? The first one is, what is the pay rate? Second, did the employee work less or more than forty hours? You can now use that information to create a flowchart for the problem. The flowchart might look like Figure 2-1 below.

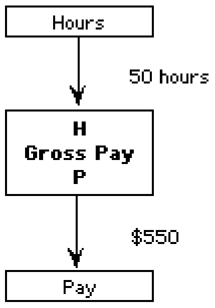


Figure 2-1: Pay flowchart

If you look back at the algorithm, you should see the wonderful relationship shared by algorithms and flowcharts. You should also see how easy it is to create the algorithm if you have a good flowchart.

Data-Flow Diagrams

A data-flow diagram is another visual tool to help you design good software. Data-flow diagrams, or DFDs, actually look very similar to flowcharts. The difference is that a data-flow diagram shows what is being done, whereas a flowchart shows how it is done. A DFD for our pay example might look like Figure 2-2, assuming a pay rate of 10 dollars per hour.

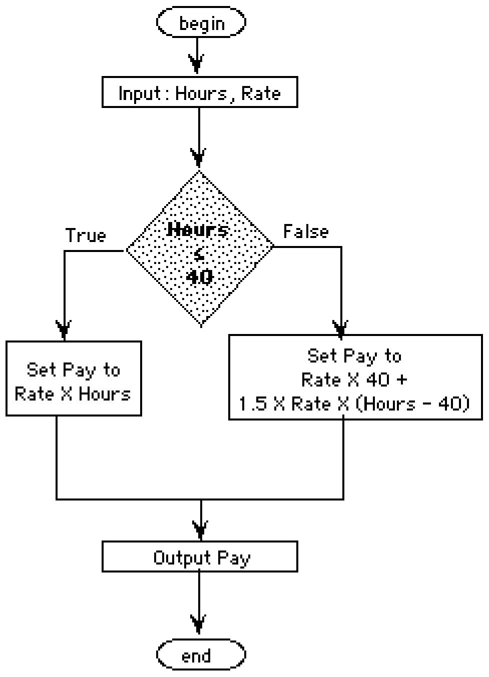


Figure 2-2: Pay data-flow diagram

Figure 2-2 shows the flow of data, from the input to the calculation to the final output. It does not take into account whether the employee worked overtime or not. So why use a data-flow diagram? DFDs are used to understand the movement of data. Let's use a real-world example to help you understand a little better.

In a government office, there is often a specific procedure and process in place to answer a request for funding. First you must fill out the required forms. You then submit those forms to the clerk at the front desk. That clerk would then take your form to the appropriate processing department. Your form is processed and then handed to a manager to sign off for authority. Finally, it is signed and your request is answered following a similar process. The movement of paper from one area to the next illustrates our data flow. You have no idea of what decisions were made at each point, just that the request was processed and you received an answer. This is data flow. Understanding this process goes a long way toward understanding the overall problem you need to solve.

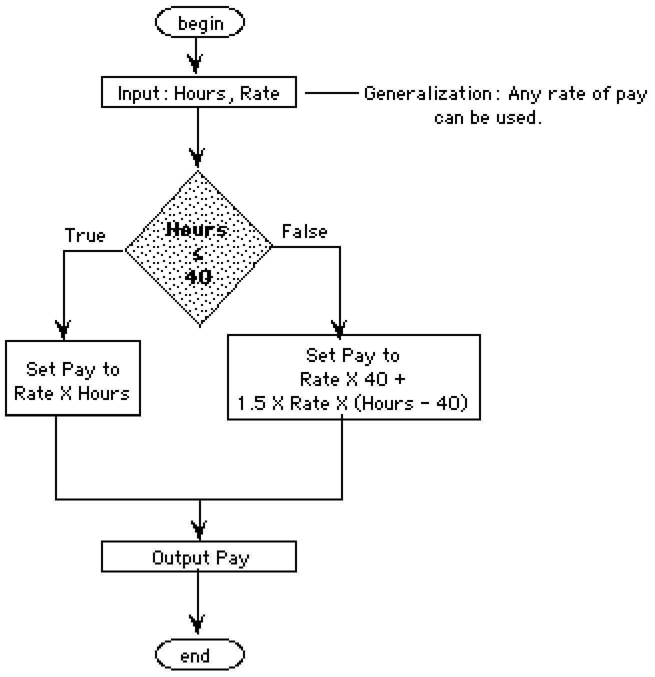
Other Visuals

Creating good algorithms and programs does not rest solely on flowcharts and DFDs. A good programmer will use any kind of visual to help him or her understand the problem. You might create tables representing data changing in response to other processes. Or you might create graphs and charts similar to those commonly found in mathematics and physics to help illustrate the full problem and understand all the data involved. If necessary, pull out that old math or physics textbook and reread the chapter on creating charts and graphs.

Connecting the Dots

Algorithm design is extremely important in computer programming. If you don't spend any time designing and implementing algorithms, you'll spend much more time than necessary on the project and may become frustrated when some of your code does not work.

Figure 2-3 shows our flowchart and algorithm side by side for a closer look.


begin
Input : Hours, Rate
Generalization : Any rate of pay can be used.
True
Hours ≤ 40
False
Set Pay to Rate X Hours
Set Pay to Rate X 40 + 1.5 X Rate X (Hours - 40)
Output Pay
end

```
If Hours greater than 40
        Set Gross Pay to Rate times 40 plus
                1.5 times the hours above 40.
Else
        Set gross pay to Rate times Hours
```

Figure 2-3 Flowchart analysis

The above algorithm does not state the input or the final result (the output). What it does show is the decision placed in line with the same area of the flowchart. The Hours > 40 block in the flowchart correlates to the "if" statement in the pseudocode. The branch in the flowchart corresponds to the calculations around the "else" part of the algorithm. Let's turn this small example into real code to see how it is done. It might help to keep the algorithm in sight while you look at the code examples.

Each programming language has specific syntax: rules that must be followed to write a program. Programming style, however, is not something cast in stone. An example of programming style would be defining how far to indent some statements (if, then, and else constructs, for example), or where to put the beginning and ending braces for a routine (C programmers tend to use varying styles here). As you learn to read source code, you'll quickly become aware of some of these styles and can choose one that suits you best and helps you understand more.

At this point in the course, we haven't yet covered variables. However, that should not hinder you in understanding what is going on in the following examples. If you feel confused after reading these code examples, come back to them after we cover variables in the next lesson.

BASIC

BASIC (Beginner's All-Purpose Symbolic Instruction Code) was developed by John Kemeny at Dartmouth College around 1967. It is a simple programming language, designed to be easy to learn and use. Here's how our algorithm would look in BASIC:

```
10 REM    SIMPLE PAY IN BASIC
20 REM    H IS THE NUMBER OF HOURS
30 REM    P IS THE GROSS PAY
40 INPUT H
50 IF H > 40 THEN 80
60 LET P = 10 * H
70 GOTO 90
80 LET P = 10 * 40 + 1.5 * (H-40)
90 PRINT "GROSS PAY IS ", P
100 END
```

In BASIC, the numbers at the beginning of each line are step references for the program. The keyword REM is used as a "remark," a comment to tell you and others what is happening and to help in understanding the program. Each programming language has different ways of marking comments. Some versions of BASIC limit the names of variables to a single letter. Here we use H for hours and P for pay. Also notice that the asterisk (*) is used as a symbol for multiplication instead of the X from mathematics. This use of the asterisk is common to most programming languages.

Pascal

Pascal, created by Niklaus Wirth in the 1970s, was based on an international language called ALGOL 60. It was designed as a teaching language for beginning students. Comments in Pascal are enclosed in braces.

```
PROGRAM SimplePay( INPUT, OUTPUT);
    { simple pay in pascal }
VAR
    hours, pay: INTEGER
```

```
BEGIN
    Read(hours);
    IF hours <= 40 THEN
        pay := 10 * hours
    ELSE
        pay := 10 * 40 + 1.5 * (hours-40)
    WRITELN( 'Gross pay is ', pay: 6);
END.
```

C

C is a programming language created at the Bell Laboratories in the early 1970s when low-level access to the machine was considered important. Comments in C start with /* and end with */.

```
/* Simple pay in C */
#include <stdio.h>
main()
{
    int hours, pay;

    scanf("%d", &hours);
    if (hours <= 40)
        pay = 10 * hours;
    else
        pay = 10 * 40 + 1.5 * (hours-40);
    printf("gross pay is %d", pay);
}
```

Again, don't worry if you don't understand everything that is happening in these code examples. The point here is to recognize the similarities between our algorithm and the resulting code. You will start writing real code in Lesson 5.

Lesson 3: Variables and Data Types

Lesson Description: In the last lesson, you caught a glimpse of a few different programming languages and how each would solve the problem of calculating gross pay. Each example used variables. In this lesson you will learn what a variable is and is not, be introduced to various types of variables, and look at constants.

What Is a Variable?

A variable is an "information holder" whose contents can be used or changed by the program. In programming, a variable is usually represented by an alphanumeric "name" like a single letter (H for hours in lesson 2) or an even longer name that is more representative of what the variable is used for. For example, "count" would be used for counting. If you've taken math or physics in school, you should be familiar with variables in equations such as v = dt (velocity = distance x time).

Variables in computer programming are no different. You can also think of a variable in programming as a box. Maybe you've moved recently. To move, you put things in boxes, label them so you know what's in them, move them to the new location, and then spend triple the time unpacking and arranging everything as you did packing. Programming is similar; variables are like boxes with labels, such as the one shown in Figure 3-1.

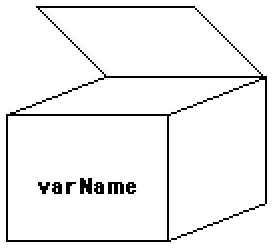


Figure 3-1: A variable

Naming Conventions

Our box has a label of "varName." When writing code, you refer to that variable by its label or name. Figure 3-1 also illustrates one of a few conventions for naming variables (at least in C). A variable is generally in lower-case text. If the variable name consists of more than one word, then the first letter of each successive word is capitalized.

With a few notable exceptions I will explain shortly, you should always use complete words to describe variables. The payroll examples from the previous lesson showed variables for hours and pay as both a single character (P and H) and as proper names (Pay and Hours). It's much easier to understand what the code does when variables have proper names.

The only time you should use single-character variables is for loops and counting, which we cover in more detail in Lesson 9. It's not wrong to use a variable that is only a single character name; however, modern programmers use variable names that describe their function (such as Hour, Rate, and Pay) as it makes the source code easier to understand.

Let's take another example: floor area in a room. The area of any rectangular shape is calculated as the length times the width. So if the living room in your house is 13 ft long by 10 ft wide, your total floor area is 130 square feet (see Figure 3-2). What are some of the variable names you could use in writing a program to calculate the area? HINT: Read the second sentence of this paragraph again.

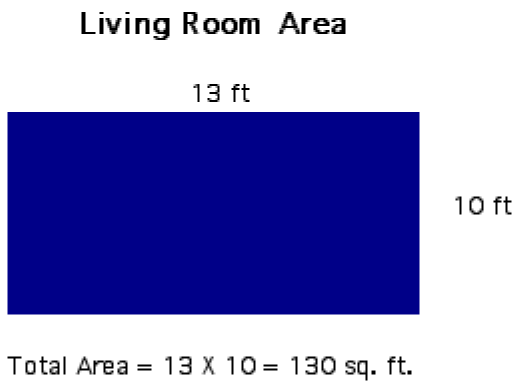


Figure 3-2: Area of a rectangle

Exceptions to the Rule

Some situations are suited for single character variable names that are not used as counting variables. For example, many mathematical equations have commonly accepted labels for some variables. If you're working with a known equation, it is easier to write it exactly as it would appear in context. Another programmer familiar with the equation would immediate recognize what the variables mean.

What Is a Constant?

Constants and variables share some similarities; nevertheless, they are used differently. Simply put, a constant is a variable whose value cannot be changed during the course of the program. Constants are usually given a value at the time they are declared (initially created). Familiar examples of constants include the mathematical value pi, the speed of light in a vacuum, the speed of sound, and the gravitational constant.

Like variables, constants have a label and can be referred to in calculations and equations in your program. The constant name is substituted for the actual value of the constant when doing calculations. Let's look at an example. The mathematical value for pi is generally considered to be approximately 3.14. This value can actually have more numbers after the decimal point, depending on how accurate you need to be in your equation. Now say you need to figure out the area of a circle. In mathematics, the equation is area = pi * r2, as shown in Figure 3-3.

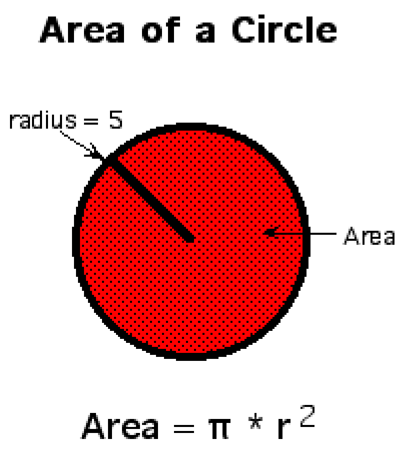


Figure 3-3: Area of a circle

The source code for our equation can look very similar to the equation itself. Here's an example written in C:

```
#define pi    3.14    /* define pi to be 3.14 */
main()
{
    float area, r;

    printf("Input the radius of the circle: \n");
    scanf("%f", r);
    area := pi * ( r * r );
    printf(" the area of a circle with radius %f is: 
%f\n", r,area);
}
```

Again, don't worry if you don't yet understand all the syntax of the example. What is important here is how we created the constant of pi. In C, constants are generally created with the **define** statement. You "define" a constant as having a certain value. You then use that constant in your equation as though it were the actual number. The final line of our short program prints out the area.

Different languages use different methods for defining constants. For example, Pascal uses the keyword CONST like this:

```
PROGRAM Area( INPUT, OUTPUT);
CONST pi    3.14;    /* define pi to be 3.14 */
VAR
    area, r : REAL;
```

```
BEGIN
    Write("Input the radius of the circle: ");
    readln(r);
    area = pi * ( r * r );
    Writeln("The area of a circle with radius 5 is: ",
area);
END.
```

There are a few programming languages that do not have or use named constants; the original BASIC is a good example. However, you can work around that problem by creating a variable, assigning it a value, and then not changing it in your program.

More Exceptions

Technically speaking, a constant does not have to be defined or declared as constant. If you create a variable in your program and assign that variable a known value (for example, radius = 5) and never change it, that variable can be considered a constant. However, it is considered poor programming practice. Constants should be clearly set up to avoid confusion and allow greater flexibility in programming.

Numerical Data Types

In order for a variable to work correctly, it must have a data type associated with it. A data type is a classification for the variable that expresses what kind of number it is. Most programming languages include data types for the two basic number formats, whole numbers (also called integers) and decimals (also called real or floating-point numbers).

Integer Types

Integer types are numbers that do not contain a decimal point and can have positive or negative values. -50; 200; -1,023; 0; 10; and 256 are all examples of integers. C and Java offer a variety of integer types. They vary in the range of values they can hold and in whether negative numbers can be used. The common integer data types are:

- int
- short
- long

All of these data types are signed data types, meaning they can have a positive or negative value. If you specifically need a number to only be positive, you add the keyword "unsigned" to the data type:

- unsigned int
- unsigned short

- unsigned long

Additionally, you can combine data types as follows:

- long int
- short int
- long long

Why the different types? Each integer type is only useful for a certain range of numbers.

Data types in C and Java		
Data Type	**Range in C**	**Range in Java**
Int	-32,767 to 32,767 (2 byte int)	-2,147,483,648 to 2,147,483,647(4 byte int)
Short	-32,768 to 32,768	-32,768 to 32,678
Long	-2,147,483,647 to 2,147,483,647	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned int	0 to 65,535	n/a
unsigned short	0 to 65,535	n/a
unsigned long	0 to 4,294,967,295	n/a

Unlike C and C++, Java does not have any "unsigned" data types: you cannot combine data types as in C and C++. We will look at these data types in more detail in Lesson 5.

Floating Point Types

Like integer types, there are a few different decimal types. Each type serves a different purpose and has a different range associated with it. Mathematical programs often require a high degree of precision for calculations. These floating-point types differ in the degree of precision that can be achieved. There are two main floating-point types:

- float
- double

A float has to represent at least six significant figures and allow a range of 10-37 and 1037. These values are often represented in exponential notation, the same notation used in math and physics. For example, the number 1,000,000,000 would be written as 1.0e9, where "e" represents "10 to the power of" and the number after "e" is the exponent.

Double is for double-precision floating-point type. The double type is similar to a float but extends the minimum number of significant figures to ten. Additionally, C and C++ allow for a third floating-point type: long double, which provides even greater precision.

Data Types as Boxes

Remember our box example from the first section? Let's take the concept a little further and apply it to data types. Not all boxes are the same size. Small boxes are useful for storing small items and large boxes for large or many items. Similarly, you may think of data types as boxes of varying size. Thus, a box of type "short" is a small box, a box of type "int" is a medium box, and a box of type "long" is a large box. Float and Double boxes are still larger.

Size of an int

One of the problems in writing C code to work on a variety of different systems concerns the "size of an int." Some processors can actually use a larger memory space (larger box) for the int data type. A program that uses a lot of int data types may work fine on an Intel-based PC but would function incorrectly on a Macintosh or Sun workstation. Be aware that there may be a different size for an int on your system of choice than is mentioned here, though the basic principles still apply. Consult the documentation for your system for more information.

Variable Assignment

This lesson would not be complete without a discussion of variable assignment -- how you give variables a value. Actually, you've already seen examples of this in the source examples shown earlier. To provide a value to a variable, you say it is "equal" to the number you want to assign. Remember our variable box from Figure 3-1? Now, let's "pack" the box with a number.

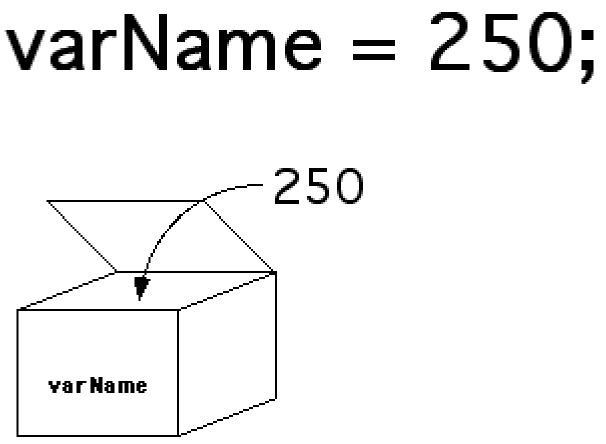


Figure 3-4: Variable assignment

Here are some more examples of variable assignment:

```
hours = 80
rate = 10
pay = rate * hours
```

Here we calculate pay again and provide values for the number of hours worked (80) and the rate of pay (10). In the last example, we assign pay to be the value of the calculation.

Lesson 4: More Data Types

Lesson Description: Numbers are not the only type of data that can be represented by variables. Many programming languages offer data types to handle letters, either singularly (a single character) or in a sequence of different characters (strings). Additionally, many languages also handle logic data.

Character Type

Variables that are defined as a character type can hold letters as well as numbers and even punctuation marks. In this case, numbers are not treated like integer and decimal values but are treated as characters, the same as the letters A or Z. In C and Java, character types are defined with the keyword **char.** Variables of the char type can only hold a single letter or number. However, the Java implementation of the char data type is 16 bits in size (larger box), which allows it to handle foreign (also called double-byte) characters. The C implementation of char is only 8 bits and thus limited to handling ASCII values (described below).

That said, char is actually an integer data type. This has to do with the way in which the computer stores data. Computers use a numerical code to store characters in which certain integers represent certain characters. The numerical code is called the American Standard Code for Information Interchange code, or ASCII (pronounced "ask-key"). There are a few others, but ASCII is the most common and is what we will use throughout this course. Using the ASCII code, the integer value 65 represents the letter 'A' (uppercase A). The standard ASCII code runs numerically from 0 to 127. When the range of values is small enough to fit into 7-bits and the char data type has a size of 8 bits (or 1 byte), there is enough space to handle standard ASCII code.

The Standard ASCII Codes

Table 4-3: Standard ASCII Codes

Decimal Value	Character	ASCII Name
0	^@	NUL
1	^A	SOH
2	^B	STX
3	^C	ETX
4	^D	EOT
5	^E	ENQ
6	^F	ACK
7	^G	BEL
8	^H	BS (backspace)

9	^I, tab	HT
10	^j	LF (line feed)
11	^K	VT
12	^L	FF (form feed)
13	^M	CR (carriage return)
14	^N	SO
15	^O	SI
16	^P	DLE
17	^Q	DC1
18	^R	DC2
19	^S	DC3
20	^T	DC4
21	^U	NAK
22	^V	SYN
23	^W	ETB
24	^X	CAN
25	^Y	EM
26	^Z	SUB
27	^[, esc	ESC (escape)
28	^\	FS
29	^]	GS
30	^^	RS
31	^_	US
32	space	SP
33	!	
34	"	
35	#	
36	$	
37	%	
38	&	
39	'	(single quote)
40	(	
41	)	
42	*	
43	+	

44	,	(comma)
45	-	
46	.	(period)
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	

79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	
92	\	
93	]	
94	^	(caret symbol)
95	_	(underscore)
96	`	(back tick)
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	

114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124	|	
125	}	
126	~	(tilde)
127	del, rubout	DEL

The first 31 characters are considered control characters, represented by the ^ symbol. These are also called non-printing characters, as they cannot be printed out on a printer. Some of these control characters -- ^L, for example -- are used to send commands to a device such as a line printer or modem. ^L represents the command FF, or Form Feed, which causes the printer to advance one page before continuing to print.

Character Variable Assignment

Let's look at an example. In school, many tests have to be taken, especially at the end of the year. Most tests are given a letter grade based on an actual number value, but in general, letter grades are given based on a number range. So, if you scored between 95 and 100 you received an "A," but if you scored 20 you received an "F." That letter grade is a character type.

One thing that separates character types from numerical types is the way in which you assign a value to the variable. When you assign a character value, you place the value between single quotation marks. Let's look at our letter grade example in a short C code snippet:

```
. . .
char grade;
short score;
. . .
if (score >= 95)
```

```
         grade = 'A';
     if (score <= 94) and (score >= 90)
         grade = 'B';
     . . .
```

Since the char data type can only hold a single character, you cannot have grades like A+ or C-.

Strings

Strings are another character-based data type. You can think of a string as a series of characters. Strings are usually longer than a single character, but do not have to be. A string can be a single word such as "text" or a small phrase such as "type a number" or even a combination of characters and numbers such as "#14 -- 1519 Nowhere Lane".

Some languages, such as Pascal and Java, have an official string data type. C and C++ do not have a formal string type. In these languages, strings are represented as an array, a series of values of the same data type (in this case, char). You will learn more about arrays in Lesson 6.

Like numbers and characters, you can assign a string to a variable. For example:

```
     char    fullName[26] = "Joe Programmer";
```

The [26] after the variable name identifies fullName as an array. In this case, the array is a character array (type char) that is a maximum of 26 characters long. A space in a string is treated as one character (ASCII 32).

While fullName can hold a maximum of 26 characters, "Joe Programmer" is not 26 characters long. How then does the computer know where the string stops? How does it know not to print out everything past the last character in the name? A special character, called the null terminator, is automatically added by the compiler to the end of the string. The null terminator is represented by \0. Strings in C are always stored with this terminating null character. The presence of the null character means that the array must have at least one more cell than the number of characters to be stored. For example, if your program accepts a name that is 25 characters long, your variable must be declared as an array of 26 characters so the last one can be used for the null terminator.

```
     char    fullName[26];
```

In memory, the variable looks like Figure 4-1.

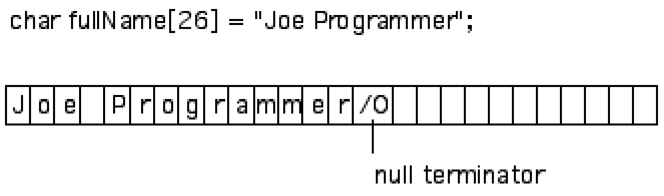

char fullName[26] = "Joe Programmer";
J o e P r o g r a m m e r /O
null terminator

Figure 4-1: String variable in memory

Notice that while the name "Joe Programmer" only occupies a portion of the array, the entire array space is allocated. The "boxes" after the null terminator are empty.

Null terminator

The Null terminator is a single character; however, it is typed as two characters: / followed by 0 (zero). The C compiler knows whenever it encounters /0 to treat it as a single character.

True or False?

The last main data type to look at is called a logic, or Boolean, type. A Boolean type can only be one of two values, true or false. That value is usually represented by a 1 (true) or a 0 (false). These values are actually carryovers from the early computer days where switches were used to turn a bit on or off. A switch in the "on" position was 1 and a switch in the "off" position had the value of 0. Thus you can think of Boolean as being true or false, 1 or 0, on or off.

Some languages such as Pascal and Java have a native Boolean data type. In C, however, such a type does not exist. In fact, the Boolean data type in C is actually defined in a series of constants:

```
#define BOOLEAN int    /* define BOOLEAN to be the
same data type as int */

#define TRUE 1

#define FALSE 0
```

Here we see that Boolean is defined to be an integer type, int. So wherever you declare a Boolean in your code, it's the same as declaring an int data type. For example, look over the following code example.

```
BOOLEAN    result;

int    anotherResult;

result = FALSE;

anotherResult = result;
```

Because BOOLEAN is the same as int, we can assign them the same values. Both have a value of 0 in the end.

Non-Zero Value

Technically, any non-zero value is the same as TRUE. A few of the modern languages such as Java and C++ handle the Boolean data type properly.

Size of Data Types

We've covered a lot of information in the last two lessons. However, understanding the basic data types is important in good program design. One last important point to consider is the size of the data type, or how much memory it uses.

You may recall from the last lesson that the numeric data types were limited to values in a specific range. The smallest measurement of data on any computer is the bit. You need 8 bits to make one byte, the next unit up from a bit. The size of a data type is calculated in bytes. Each data type fits completely into one or more bytes of data. The following table shows each data type and its size in bytes on different processors and operating systems.

	Motorola 68000 series processor	**Motorola PowerPC**	**IBM PC (DOS and Windows 3.1)**	**IBM PC (Windows 98/NT)**
Char	1	1	1	1
Int	4	2 or 4	2	4
Short	2	2	2	2
Long	4	4	4	4

Table 4-2: Data types are sized differently, depending on the processor and operating system.

In early computing, the size of the data was extremely important. Memory was limited and expensive. In the table above, take special note of the int data type. The size of an int is dependent on the type of processor and the operating system used. This size issue frequently causes problems when porting (or translating) a program from one type of computer to a different one. This is also why many programmers use short and long instead of int, because the size of the type is known and defined by a standards committee known as the American National Standards Institute (ANSI).

Lesson 5: Computer Arithmetic

Lesson Description: In this lesson, you will learn about the various number systems computers use, how to perform mathematical operations using these systems, and how to convert between different systems. This lesson also introduces your first programming lab.

Counting Systems

Computers support four primary number systems for programmers to use: decimal, binary, hexadecimal, and octal (also called base 8). In the end, however, computers see numbers as a series of ones and zeros, a system known as binary.

Depending on your needs, you can write code that uses any or all of these number systems. For example, when trying to access a specific area in a computer's memory, it's easiest to use the hexadecimal system. Similarly, when writing a lot of mathematical code, it's easiest to use the decimal system.

The octal number system isn't used much any more and thus won't be discussed in this lesson.

Let's take a look at each number system in more detail.

Decimal Numbers

You should already be familiar with the decimal number system. Decimal numbers encompass all numbers from negative infinity to positive infinity. The decimal number system includes 5, -26, 8.34, and so on. This is the number system you use most often in calculations when writing programs.

The decimal number system is also called the base 10 system. For example, 3,421 has a 3 in the thousands place, a 4 in the hundreds place, a 2 in the tens place, and a 1 in the ones place. This means you can think of 3,421 like this:

```
    3 x 1000 + 4 x 100 + 2 x 10 + 1 x 1
```

Remember, though, that the number 1000 is really 10 cubed, 100 is 10 squared, 10 is 10 to the first power, and by convention 1 is 10 to the zero power. We can then write 3,421 like this:

$3 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$

We say that 3,421 is written in base 10 because our system of writing numbers is based on powers of ten.

Binary Numbers

The binary number system is the lowest level system and ultimately what everything gets converted to. The binary number system is often referred to as base 2.

One byte is a set of 8 bits that is processed as a single unit. You can think of these 8 bits as being numbered from 7 to 0, left to right. Bit 7 is called the high-order bit, and bit 0 is the low-order bit in the byte. Each bit number corresponds to a power of 2. So 128 is 2 to the 7th power, and so on, as shown in Figure 5-1.

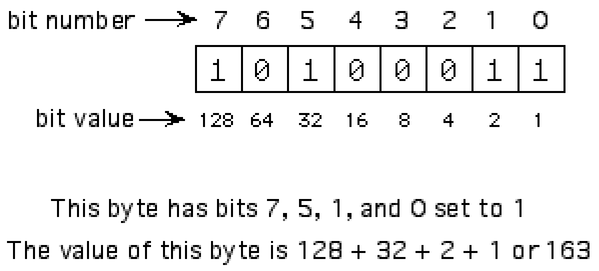

bit number
7 6 5 4 3 2 1 0
1 0 1 0 0 0 1 1
bit value
128 64 32 16 8 4 2 1
This byte has bits 7, 5, 1, and 0 set to 1
The value of this byte is 128 + 32 + 2 + 1 or 163

Figure 5-1: byte value

The largest value represented by a byte would be with all bits set to 1: 11111111, or 255. The smallest number is 00000000, or just 0. So the value range for a byte is from 0 to 255, for a total of 256 values. If we allow for negative numbers, the range is -128 to 127, again, for a total of 256 values.

Signed Binary Values

Representation of signed numbers is determined by the hardware, not the programming language. In binary, the high-order bit is used as the sign bit, if the bit is 1 the number is negative. So for a 1-byte value, the last 7 bits are the actual number. Therefore, -1 is 10000001 and 00000001 is 1. The total range then is -127 to 127.

The problem with this method is that you can end up with a -0 and a +0, which is confusing. The most common method used to avoid this problem is called two's-complement. Numbers in two's-complement are represented in the same way as mentioned above. However, to determine the value of a negative number in two's-complement, you subtract the bit pattern from the 9-bit pattern 100000000 (256 in binary), the result is the magnitude of the value.

Let's look at an example. 128 in binary form is 10000000. As a signed value, it is negative (bit 7 is 1) and has a value of 100000000 -- 10000000, or 10000000, which is 128. Therefore the number is --128 (it would be --0 if two's-complement is not used). So now we have our full binary range of --128 to 127 as stated above. Here are a few more examples:

3 in two's-complement is the same as 3 in normal binary: 00000011

2 in two's-complement is the same as 2 in normal binary: 00000010

1 in two's-complement is the same as 1 in normal binary: 00000001

0 in binary is: 00000000

--1 in two's-complement is: 10000001

--2 in two's-complement is: 10000010

--3 in two's-complement is: 10000011

Hexadecimal Numbers

Hexadecimal (referred to simply as hex) is a base 16 number system. Can you figure out what this means? If you said, "because it uses powers of 16," you are correct. In addition, hex uses 16 digits ranging from 0 to 15. There are no single digits to represent numbers 10 through 15, so letters A through F are used instead. Thus, the hex number A4D (written as 0xA4D in C) converts to:

$10 \times 16^{2} + 4 \times 16^{1} + 13 \times 16^{=} = 2637$ in base 10

In C, you can use upper or lowercase letters for the hex values. So you can write 2637 as 0xa4d.

Converting Between Systems

Converting from one number system to another is a relatively easy task. Let's look at converting from decimal numbers to each number system first. Then we'll discuss how to convert between binary and hexadecimal.

Decimal to Binary

Converting a decimal number to binary is a simple matter of continual division. Let's look at the algorithm first.

```
Set N to the number to convert
While N > 0
    Divide N by 2 yielding Quotient and Remainder
    Output Remainder
    Set N to Quotient
Reverse the order of outputs
```

Now let's look at a practical example. Figure 5-2 demonstrates what happens when we apply this algorithm to a number like 13.

Applying decimal to binary convertion algorithm

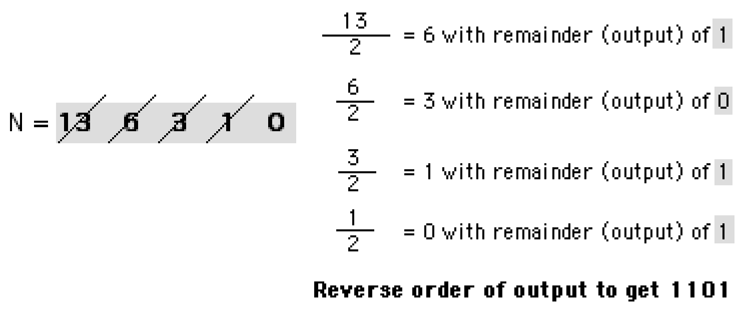

N = 13 6 3 1 0
13/2 = 6 with remainder (output) of 1
6/2 = 3 with remainder (output) of 0
3/2 = 1 with remainder (output) of 1
1/2 = 0 with remainder (output) of 1
Reverse order of output to get 1101

Figure 5-2: Applying binary conversion algorithm

Decimal to Hexadecimal

To convert decimal numbers into hex, you can use the same algorithm as the binary conversion, but instead of dividing by 2, you divide by 16. You then substitute each output for the appropriate hex value. Let's use our earlier example and convert 2637 to hex.

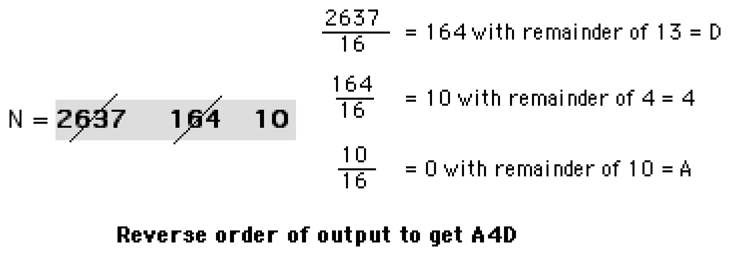

$\frac{2637}{16}$ = 164 with remainder of 13 = D
$\frac{164}{16}$ = 10 with remainder of 4 = 4
N = ~~2637~~ ~~164~~ 10
$\frac{10}{16}$ = 0 with remainder of 10 = A
Reverse order of output to get A4D

Figure 5-3: Converting decimal to hex

Binary to Hexadecimal

One of the easiest conversions is going from binary to hex and back again. Working in binary is difficult as it's easy to misread ones and zeros. It is much easier to work with the hex equivalents, which translate to binary (and back again) easily when you need to examine a specific bit.

Each set of four binary digits corresponds to a single hex digit. The largest set of four binary digits, 1111, or 15 in decimal, is F in hex. There are two hex digits in a byte (8 bits): the first digit represents the upper 4 bits, the last digit represents the lower 4 bits.

The following table shows decimal numbers up to 15 and their binary and hexadecimal equivalents.

Decimal Value	Binary Value	Hexadecimal Value
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8

9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 5-1: Decimal numbers up to 15 and their binary and hexadecimal equivalent

Using Number Systems

Regardless of the number system you choose to use for your programming needs, all numbers are stored in binary form. So the largest 16-bit value that can be represented is:

```
0xFFFF = 1111111111111111 = 65,535
```

Note that this is for an unsigned number.

With these things in mind, the following statements are legal C code:

```
short    base10 = 255;
short    base16 = 0xff;
```

The above code initializes each variable to a value of 255. Obviously, if you needed a larger number you would use the corresponding data type (long, for example) to allow enough room to store the data.

Why Learn Binary?

C programmers do not use binary variables. In fact, C has no standard for writing binary variables, as it does for hex or octal. All operations are done in decimal, octal, or hexadecimal representation instead. Why learn binary then? Binary and converting between hex and binary become important for many advanced operations in C known as bit-fiddling. Bit-fiddling is beyond the scope of this course.

Mathematics and Problem Solving

Using math in programming is the same as using math normally. You just have to be aware of the data types needed for the operation. While you can assign a smaller data type to a larger one and have it keep its value, you cannot do the reverse. For example, say you wanted to write an equation that would cube any given value. You could write something like this:

```
short    input, result /* this is shorthand for saying
that both variables are of type short */
```

```
scanf("input a number %d", input);  /* get a number */

result = input * input * input;  /* do the math */

printf("The cube of %d is %d\n", input, result); /*
print result */
```

This code sample will work for numbers that fall within the range of the short data type. However, if the user inputs a large value, the result may not be in the same number range. For example, if the user inputs 255 as the initial number, the answer is 16,582,375, which is outside of the range for a two-byte data type such as short. Instead, the program should be:

```
short    input;

long    result;

scanf("input a number %d", input);  /* get a number */

result = input * input * input;  /* do the math */

printf("The cube of %d is %ld\n", input, result); /*
long result */
```

You can change this program even further so that it accepts long values (4 bytes) instead of short ones.

You should learn about the mathematical operations allowed by your language of choice. Find out how to use the various math operators (+, -, *, / (divide)) and in what order they are handled. Then find out what special operators and functions are provided, such as % (modulus), sqrt (square root), and so on. Learn how to use formatting characters, for example, so that you get exactly two digits after the decimal place for money calculations. All these are important to know and to learn.

Lesson 6: Handling Complex Data

Lesson Description: This lesson teaches you how to use the native data types to create your own custom types and handle more complex data.

Arrays

We've already seen an example of arrays in Lesson 4 as a string data type. You can create arrays of any data type, including any custom data type you choose. This makes arrays one of the most flexible methods of handling large amounts of data. There are two main types of arrays, linear and tabular.

Linear Arrays

Linear arrays consist of one data type written in sequential fashion. Strings are a good example of a linear array. The sequence of characters continues "along a line" and stops at the end. You can think of this type of array as having one dimension.

Linear arrays are declared as follows:

```
dataType    variableName[number];
```

dataType can be any of the data types we've looked at up to this point, or any custom data type you create. (More on this later). The name of the variable and number representing the size of the array is the variableName. So, if you needed an array to hold 100 numbers, you would declare it as follows:

```
int    numArray[100];
```

The 100 is called the subscript. In C, arrays are 0-based which means that the array actually starts at position 0 instead of position 1. In our example above, we would have a range of numArray[0] to numArrray[99].

Tabular Arrays

Tabular arrays have more than one subscript, but can still only hold one kind of data. Tabular arrays are most commonly called multi-dimensional arrays. The easiest to visualize is a two-dimensional array, which looks like a table (thus the name). Consider the following example:

```
Boolean    hotDays[7][52];
```

In this example, meetings is the name of an array that allows you to keep track of how many hot days you had in the year by storing "true" or "false" in the appropriate location. The 7 is the number of rows (days) and the 52 is the number of columns (weeks). Conceptually, the array looks like this:

hotDays [0][0]	hotDays [0][1]	hotDays [0][2]	hotDays [0][3]

hotDays [1][0]	hotDays [1][1]	hotDays [1][2]	hotDays [1][3]
hotDays [2][0]	hotDays [2][1]	hotDays [2][2]	hotDays [2][3]

And so on. By changing the first subscript, you move from day to day. Changing the second subscript moves you from week to week. So, for your average daily temperature for the second day of the 8th week of the year, you would perform the following assignment:

```
    meetings[1][7] = true;
```

Remember that arrays are 0-based in C, so the second day is 1 instead of 2 and the 8th week is 7 instead of 8.

More Dimensions

You are not limited to two-dimensional arrays. You can have as many dimensions as you require. You are limited by memory space, however (remember the size of your data types). To create a three-dimensional array, the above example can be further changed to include months of the year as follows:

```
    Boolean    hotDays [7][5][12];
```

You can think of a thee-dimensional array as having width, height, and depth. While it's more difficult to visualize an array with more than 3 dimensions, you can definitely have as many dimensions as you need.

You may have noticed that the above example isn't a real world situation as days and weeks in a month vary by month and year. The array shown here has too many places in it. However, you could still use this as a base array and then write code to handle a specific situation. Additionally, you may change the data type to a numerical type such as short and store the actual average temperature. You could then store an unreasonable value in one array position (-999, for example) that would signal your code to switch months.

Array Limitations

Arrays do have limitations in their use. When you declare an array, the program allocates exactly enough space for the number of items of the type declared. If you need to store more data than you initially declared in your array, you have to change the program accordingly. You can't just add another place in the array while the program is running.

As mentioned previously, you can still only store one type of data in an array. You cannot create an array that can store both numbers and characters, for example, at least not without creating a custom data type. (More on that below.)

Arrays also take up a large amount of memory. When you declare an array to hold 100 int values for example, all the memory needed to store those int values is allocated immediately in memory, even though you are not using them yet. That equates to 100 *

2bytes = 200 bytes of memory. Not a lot you say, but what about if you're using larger data types like doubles or floats? Or a custom data type you create? Suddenly, memory becomes an issue, especially if you don't really need all of it. You learn more about memory management later in this lesson.

Array Initialization

Like other variables, you can initialize an array to specific values on creation instead of filling the array with values later on. To initialize an array when it's declared, you use the braces ({ and }) that contain the values for each array position separated by a comma. This is similar to set notation in mathematics. For example:

```
short    anArray[3] = {12, 5, 150};
```

is the same as

```
short    anArray[3];
anArray[0] = 12;
anArray[1] = 5;
anArray[2] = 150;
```

Initializing an array like this allows for a little shorthand in code and is great for setting up tabular data that will be referenced in another part of your program.

Initializing arrays is not limited to single-dimensional arrays. To initialize a multi-dimensional array, you do the same task but with additional sets, also separated by commas. Consider the following example:

```
short    junk[2][4] = {
    {2, 4, 6, 7},
    {3, 8, 10, 6}
    };
```

Here, junk is an array consisting of two rows of four values each. You define a "set" for each row and then enclose that set in braces to mark the beginning and ending of the data. Notice that the last set did not include a comma (,) after it.

Structures

One of the easiest methods of creating custom data types, particularly for more complex data, is to use structures (called records in Pascal). Not all programming languages have the capability to create structures, but most modern ones do.

Simply put, a structure consists of one or more elements of data (variables) wrapped in a single container variable for easy access and retrieval. Each data element in the structure can be a different data type. You can think of structures as a mini-database. Let's look at a practical example.

Many companies keep various pieces of information about their customers: name, age, income, gender, etc. All these data elements taken together form the data. To create this structure programmatically, you use the keyword **struct** followed by the definition of that structure, like this:

```
struct customers

{

char name[40];

short age;

float income;

char gender;

Boolean hasComputer;

}
```

Customers is the name of the data type. Note that it is not the name of the variable; we'll look at that in a second. The braces { and } define the bounds of the structure -- everything inside the braces is contained in the structure definition. All the variables defined inside the structure are called fields. In our example, all the fields are different data types.

Now that we have defined our address structure, we can create a variable with it:

```
struct customers customerBase;
```

Now the variable customerBase is defined to be a customers structure (custom type). We can now fill each field (assign each field a value) with the proper data. You assign values to the files by first naming the variable (customerBase) and then using the field name preceded by a period (.). That period tells the compiler that you are accessing a field in the structure as opposed to the structure itself. For example, if our vital status is:

```
Name: Joe Programmer

Age: 28

Income: $3,000,000

gender: M

Owns Computer: True
```

We do the following:

```
customerBase.name = "Joe Programmer";

customerBase.age = 28;

customerBase.income = 3000000;

customerBase.gender = "M"

customerBase.hasComputer = TRUE;
```

Suppose you defined a structure that contained people's addresses. The structure contained fields for name, street address, city, postal code, and country. It hardly seems appropriate to store only one address. Most people want to store many addresses: friends, family, co-workers, and so on. Well, you could declare a new variable for each person like this:

```
struct address myAddress;
struct address bill;
struct address work;
```

But a more efficient method would be to put all those addresses in a single place: an array, for example. So you could then do the following to store the addresses of ten people:

```
struct address everybodyIKnow[10];
```

To access friend number 4, you put the array subscript after the array name, not after the structure field.

```
everybodyIKnow[3].name
```

Using typedef

While we have created this new data type to handle our addresses, creating variables that use the new data type is a little cumbersome. You must declare variables as follows:

```
struct address someVar;
```

It seems like it would be simpler if you could avoid the extra typing by using something like:

```
ADDRESS addressList;
```

And indeed it would. Enter typedef. C's typedef function allows you to name your data types. Using addresses as an example, you'd do the following:

```
typedef struct address
{
    char name[40];
    char street[100];
    char apt[5];
    char city[25];
    char state[5];
    char zip[9];
    char country[3];
} address;
```

This defines address to be the same as "struct address." The use of all lowercase is arbitrary. You could just as easily use all uppercase or a combination of uppercase and lowercase. Most programmers tend to name the type the same as the struct (including case).

The typedef function can be used for much more than described here, but that is left as an exercise for you.

Stick With One Method

Once you choose a method of defining your data types, you should stick with it. It's a good idea to look at how others handle the same situation and follow their example. It is then easier to understand other people's code when or if you use the same programming style as they do.

Pointer Variables

Probably one of the most powerful features of most modern programming languages is the use of pointers. Pointers are a very powerful feature and allow you to perform tasks using less code than would otherwise be required. Not only that, but pointers allow you to create new custom types, some of which you'll see shortly.

Simply put, a pointer "points" to data. Specifically, pointers are memory locations that store addresses. Whenever you see the word "pointer" your mind should immediately say, "store the address of." For example, if you see:

```
aPtr is a long pointer."
```

Your mind says, Aha! What is really meant is that . . .

```
aPtr stores the address of a long."
```

One thing to remember is that pointers are not a data type. You can retrieve the value of pointer variable holds in a manner similar to retrieving information from other data types, but pointers themselves are not a data type. The main differences between a data type and a pointer are their purposes. The purpose of a variable with a char data type is to store a character (or string). The purpose of a character pointer is to store the address of the pointer value as well as to point to the data type.

The basic format for declaring pointers is to place an asterisk (*) in front of the variable name. For example:

```
char(%TAB%)*charPtr;

int(%TAB%)*intPtr;
```

When assigning a value to a pointer, the golden rule is that you only assign addresses. This matches nicely with the fact that pointers store addresses. To assign an address to a pointer variable, you use the assignment operator (=) with the & character.

In the context of pointers, the & character means "the address of." If "foo" is the name of a variable, then &foo is the address of the variable. When you assign a pointer variable, you are actually assigning an address in memory in which that variable will reside.

Consider the following code example:

```
int myInt;

int *intPtr;

/* assign the value of 10 to myInt */

myInt = 10;

/* Addign myInt's address to intPtr */

intPtr = &myInt;
```

To look at intPtr to find its value, you "dereference" the pointer. You can do this to assign another variable the same value that intPtr points to, like this:

```
int yourInt;

/*assign the value of myInt to yourInt by
dereferencing intPtr */

yourInt = *intPtr;

/*Print the results */

printf("myInt is %d. \n", myInt);

printf("yourInt is %d.\n", yourInt);
```

Whenever you see *inPtr in code, should always read it as "the int pointed at by the address contained in intPtr."

Pointer Uses

While pointers look cool, you may be wondering, "Why use them? What is their purpose?" You can perform some operations using pointers that are more difficult using conventional means. For example, you can use pointers to step through an array you may not know the exact size of. Also, you can design some complex data types that would not be possible without the use of pointers.

One classification of custom data types is called Abstract Data Types. These data types are designed to mimic real-world objects or concepts. For example, a long line of people waiting to get into a movie is called a queue and it operates on the principle of first come, first served. A pile of dishes at the local cafeteria is called a stack. There are many others. One such popular one is called a linked list. A linked list consists of a structure of one or more data types, plus a pointer to a similar structure, and it looks like this:

```
struct aList

{
    int    count;
    struct aList *next;
};
```

The variable next is a pointer to another aList structure. Thus the next variable links one struct to another, forming a chain or list. You can use this to form a simple database of objects.

More detail on pointers and abstract data types is beyond the scope of this course. However, you will run into things like this when reading other people's code, so you should be aware of them.

Lesson 7: Modularity

Lesson Description: Modularity is probably the single most important part of good program design. A modular program is easier to write, easier to maintain, easier to fix, and easier to understand. Actually, a modular program can be much easier to design as well.

What Is Modular Programming?

Modular programming is the process of breaking down a large problem into smaller, simpler, and more manageable chunks that can then be coded with greater ease.

It wasn't that long ago that BASIC programs were classified as "spaghetti BASIC" because of the nature of the language and style of written code. Source code for these programs appeared to be strung together like spaghetti making the code hard to read and understand. Shortly thereafter, BASIC programmers tried harder to make their code a little more manageable by using "subroutines." Subroutines are small segments of code that perform a specific task. Even using these subroutines, however, the code was still very difficult to manage and follow. I still see code that is spaghetti-like in nature, even in modular languages such as C, C++, and Pascal. This, in my opinion, is unacceptable.

The use of subroutines allowed programs to become more structured, removing the use of the explicit (and often dreaded) GOTO command used by most BASIC programs. The move towards structured programming spawned a new language called Pascal. Pascal used "procedures" and "functions" that are similar in concept to subroutines. A procedure was called to perform a specific task. Functions also performed a specific task, but in addition returned a result, allowing programmers to use function calls as part of an equation. Generically, you can think of procedures and functions as "routines."

Even though C was around at the time, it was not in common use. Now however, C, C++, and Java dominate the academic, hobby, and professional programming fields. In C, you write everything as a function. That function may or may not return a value, but it is still written following the syntax of a function.

The idea of modular programming then spawned object-oriented languages such as C++, Java, and SmallTalk. Writing source code as concise modules gives programmers a lot of flexibility. For example, say you have written a source module that computes a person's taxes in a payroll program. Later, you find that the calculation for those taxes changed. You could modify the modules without changing how the rest of the program works. It still computes taxes but does it differently, or maybe even more efficiently.

Procedures and functions can be thought of as modules in languages such as C and Pascal. The concept changes slightly for object languages like C++, Java, and objective-C. In these languages, a module might be an object class. The finer details are not important at this time. What is important is to understand that modules represent small tasks that can be called and performed at any point in your program.

Modularity is often linked with conditionals. You might call a procedure or function as a result of a specific condition in your code. You will learn more about conditionals in the next lesson. For now, let's focus on the modules themselves.

Using Functions

I'll describe functions first, as it will be easier to understand how procedures are written in C, C++, and Java if you know how functions are written.

Functions are most commonly used when you need to know the result of an operation or series of calculations before the program continues. Some languages, notably Pascal, use the keyword "function" to define a function. C, C++, and Java do not have this feature. Instead, everything in these languages is written as a function. In C, functions are written as follows:

```
ReturnType    FunctionName(parameters)
```

ReturnType is the data type the function returns. This could be a native data type such as long or char, or it can be a custom data type like our address structure from the previous lesson.

FunctionName is the name you give the function. Common practice is to name functions in all lowercase with the first letter of each word in uppercase. Function names should be named according to the task they are to perform. For example:

```
    IsFinished();

    GotAddress();

    FinalDeductions();
```

The parameters are a list of one or more variables a function needs to complete the task. Giving a function a parameter list is called "passing parameters" and helps eliminate the use of global variables (variables that can be accessed from any point in the program). We'll cover parameters in more detail a little later in this lesson.

Often, functions are used for calculations or Boolean (true or false) statements. You can then use functions in a test condition as follows:

```
if (ItIsTimeToLeave())

...
```

In this example, we're making the call to the function directly in the test, so we are relying on the result of the function to continue. If the function returns true we do one thing, and if false we do another. The function is called and the result returned before the "if" statement completes.

Additionally, we can assign the result of a function to a variable as follows:

```
float    result;

result = CalculateTaxes();
```

Here, we are essentially saying that the result of the function CalculateTaxes is assigned to the variable result. Because result is a variable of type float, CalculateTaxes must be defined to return a float type or a type that is compatible with float. Here's what the function definition might look like for CalculateTaxes given these conditions:

```
float CalculateTaxes( void )
{
    float(%TAB%)taxes;
    /* code to calculate taxes goes here */
    return taxes;
}
```

In this example, we declare a local variable called taxes to use for the calculation. Once the calculation is finished, we "return" that variable. Notice that the data type for the taxes variable is the same as the data type the function returns.

When you write a function that uses parameters, you give the variable names and the data type. When calling a function that uses parameters, you only pass the variable(s) of the appropriate type or a compatible type.

Using Procedures

Procedures are called when you want to perform a task or series of tasks that doesn't require the result to be known before the program continues. Some languages, notably Pascal, use the keyword "procedure" to define a procedure. C, C++, and Java do not have this distinction. In C, procedures are written as follows:

```
ReturnType    ProcedureName(parameters)
```

At first glance, you may ask, "but isn't that a function?" Yes it is. Remember, every routine you write in C is a function. The difference between the two becomes clearer when we use a real example.

```
void    AddDatabaseItem( itemType theItem )
```

The use of the keyword "void" as the return type tells us that this routine doesn't return a value, therefore it's considered to be a procedure, not a function.

Procedures are not directly used in conditional expressions such as the if statement from the previous section. Generally speaking, procedures are called linearly like this:

```
procedure1();
procedure2();
...
```

And also after the result of a condition is met (more on that in the next lesson).

Let's look at a real example of a procedure:

```
void    DeleteItem( long itemNum )
{
...
}
```

Note there is no return statement at the end of the procedure. Whatever needs to be accomplished is done in the procedure and that's it. Other than the differences listed here, everything you know about functions also applies to procedures.

Using Parameters

An important part of good program design is using parameters to give a routine data to work on instead of using a global variable. You should try to minimize the use of global variables as much as possible. You can use parameters with procedures and functions. A procedure or function that does not use parameters is said to be "void" of parameters. In C, the keyword "void" is used when defining the routine, but not used when calling the routine. For example, remember the CalculateTaxes routine from above:

```
float CalculateTaxes( void )
{
    float    taxes;
    ...
    return taxes;
}
```

There is nothing wrong with this routine; it's perfectly functional. However, this routine would need to rely on reading information that can be used when calculating taxes either from global variables, or from a data file, or from some other means. To make the routine a little more efficient, it's best to pass in the data the function needs as parameters, like this:

```
float CalculateTaxes( float income, float taxRange )
{
    float    taxes;
    ...
    return taxes;
}
```

Then, you would call the function like this:

```
float grossIncome, taxPercentage;
...
```

```
scanf("Input employee"s gross income %f",
grossIncome);

scanf("Input employee"s tax bracket %f",
taxPercentage);

...

float netTaxes = CalculateTaxes( grossIncome,
taxPercentage);
```

There are a few things to notice about how CalculateTaxes is defined and how it is called. The most important is the parameters given when defining the function. The variables do NOT have to be the same name, but they do need to be the same type or of a compatible type. Inside the function, the variable names in the parameter list are used to perform the calculation. The next thing to note is that CalculateTaxes is called using an assignment statement. The variable, netTaxes, is the same data type that CalculateTaxes returns.

Type Matching

Technically speaking, the parameter data types don"t always have to match, but they do have to be compatible. The data type used when calling the function must be able to fit in the data space used to define the function. For example, a short can fit in an int type and a float can fit in a double type. However, a double cannot fit into a float type and an int cannot (on some systems anyway) fit into a short type.

Lesson 8: Conditionals and Branching

Lesson Description: In this lesson, you will delve deeper into the different types of conditional statements and learn what branching helps you to accomplish.

Conditional Statements

You are probably already aware of some the conditional statements you make when talking or making decisions. You've seen some examples of conditions in the previous lessons using an "if statement" to perform a test. However, the "if" is not the end of the condition. There are two main types of conditional statements used in C, C++, and Java. The first is the "if . . . then . . . else" condition. The second is the "switch . . . case" condition. Let's look at each one in more detail.

If . . . then . . . else

The most common conditional statement used in programming is called the if . . . then . . . else statement. It looks like this:

```
if (some condition is met) then
do something
else
do something else
```

The first line is the test condition. If the condition inside the parentheses evaluates as TRUE, then the line below is executed. The "else" part of the statement handles the case if the test condition is false. There can be any number of lines for the respective "do something" lines in our pseudocode. However, general practice is that if there are more than two or three lines, that code is put into a separate routine and called explicitly on its own (a process known as branching).

Some languages, such as Pascal, use "then" as a keyword and part of the statement. Other languages such as C, C++, and Java assume "then," thus you never see the word in this type of conditional statement. Let's use the example of calculating overtime pay from an earlier lesson. If Tom, an employee, worked over 40 hours in one week, his salary is 1.5 times the regular salary of $10 per hour for each hour over 40 he worked and $10 for every hour up to 40 hours. That statement written in C might look like this:

```
#define kNormalPay    10
#define kNormalWeek    40
float overtimePay;
...
if (hours > 40)
overtimePay = (hours - kNormalWeek) * 1.5;
```

```
else
overtimePay = 0.0;
. . .
```

The test condition is if the number of hours is greater than 40 hours. If that condition is true, we calculate the overtime pay rate. However, if that condition is false, the program executes the "else" part of the statement and assigns overtimePay to be zero.

Let's look at a different example. What happens if there is no "else" statement? The answer is, the program just keeps working. When you do not put an explicit "else" statement in the condition, the effective result is that if the test condition fails, do nothing special, just carry on. For example:

```
if (hours <= 40)
grossPay = kNormalPay * hours;
...
DoSomethingElse();
```

In the example above, there is only a test for hours less than or equal to 40. If that test fails, the program does not calculate grossPay, but skips that line and continues chugging along. This particular bit of example code is actually a coding bug. We'll talk more about bugs in Lesson 12.

Let's take a second to talk a little about coding style for "if" statements. Common practice is to use braces ({ and }, representing begin and end) when there is more than one task you want to perform in either the "if . . . then" part of the condition or the "else" part. If, however, you only use one statement, braces are not required (as shown in the examples above). Consider the following example:

```
#define kNormalPay    10
#define kNormalWeek    40
float overtimePay, grossPay;
...
if (hours > 40)
{
overtimePay = (hours - kNormalWeek) * 1.5;
grossPay = (kNormalWeek * kNormalPay) + overtimePay;
}
else
grossPay = hours * kNormalPay;
. . .
```

The braces keep the two calculations together. Both calculations must be done before the program can continue. Notice that we don't use braces for the "else" because there is only one statement that needs to be executed.

What would happen if the braces were removed from the "if"? When you compile the program, the compiler would return an error. Why? The compiler would see the first assignment in the condition and assume no "else" statement. The grossPay assignment is then executed regardless of the test condition. Suddenly, the "else" statement appears, but there's no corresponding "if" statement, so the compiler returns a syntax error.

Expanding the Tests

You can create more elaborate "if" statements than those shown here. For example, suppose the employee worked more than 40 hours, but less than 60 hours. The "if" statement would look like this:

```
if ((hours > 40) && (hours <= 60))
{
rate = 1.5;
pay = rate * hours;
}
```

Use of the double ampersand symbols "&&" creates a logical relationship between the test conditions that says if hours less than 40 AND hours is less than or equal to 60. Using a multiple test condition in this way makes it easy to test range values.

What about a situation that required a logical OR condition? You can do that too. For example, if the value of hours was greater than 40 OR it was a holiday, you might code it like this:

```
if ((hours > 40) || (holiday == TRUE))
{
rate = 1.5;
pay = rate * hours;
}
```

In this case, the double bar character "||," often referred to as the "pipe" character, is used to make the logical OR condition.

"Case" Statements

The other main condition statement is the "case" statement. "Case" statements are most often used when there are more than two possible answers for a condition, and each answer requires a separate action. The "case" statement is used in conjunction with a "switch" statement to tell the compiler what variable you want to test. Let's look at an example.

Suppose you are writing a program that computes sales commissions. Commissions are paid at particular milestones. For sales that total $1,000 the commission is 2 percent. For $5,000 the commission is 3 percent. For $10,000 the commission is 5 percent. For everything over $10,000 the commission is 10 percent. You could use a series of "if . . . then . . . else" statements, which work fine but which tend to render a solution that is less elegant and harder to read. It's easier to use a case statement like this:

```
/* code to calculate the milestone, not shown */
switch (milestone)
{
case 0:
commission = 0.00;
break;
case 1000:
commission = 0.02;
break;
case 5000:
commission = 0.05;
break;
default:
commission = 0.10;
break;
}
```

The "switch" statement isolates the variable you want to examine, in this case sales. The "case" statement will not work with any other variable. Each successive case tests one of the possible values and takes appropriate action if the test condition (milestone hit) is true. The "break" statement tells the compiler to break out of the switch/case statement and continue following code. The "default" condition at the end satisfies the test of sales over 10,000. Why? Because the program will have gone through each condition and failed. The last condition therefore is met and means "for anything else, do the following."

If no "break" statement is encountered, the program will also execute the code in the next case (called falling through). There are times when this can be useful, but in the example above the wrong code is executed and you end up with errors and bugs.

Additionally, you can combine cases to form an OR condition. For example, say that a commission of 5 percent is paid on milestones of $5,000, $6,000, and $7,000. The code might look like this:

```
/* code to calculate the milestone, not shown */
switch (milestone)
{
case 0:
commission = 0.00;
break;
case 1000:
commission = 0.02;
break;
case 5000:
case 6000:
case 7000:
commission = 0.05;
break;
default:
commission = 0.10;
break;
}
```

Notice how the three test conditions are grouped together. This tells the program that if any of these milestones are met, the commission rate is 5 percent.

Unlike the "if" statement, if you have more than one line of code that needs to be executed when a test condition is met, you do NOT need to place the code between braces. Some programmers do and some do not. There is nothing wrong with either situation. However, if you do decide to use the braces, the "break" statement should come after the last brace for each case, like so:

```
switch( someVar )
{
case 1:
        {
            ...
        }
break;
case 2:
```

```
            {
                ...
            }
    break;
        ...
    }
```

Case statements are great when you need to test many possible results where you know the result can only one of a few values. Using "case" statements for only a couple tests is rare, but I've seen it done to facilitate readability and coding.

What Is Branching?

There are a couple of different techniques that are commonly referred to as branching. Strictly speaking, branching is using a test to decide between alternative sequences. Those sequences may be calling a different procedure or function, or setting variables to different values (like in the examples you've seen above). In fact, you've seen examples of branching all through this course. Let's look again at the flowchart for payroll from Lesson 2.

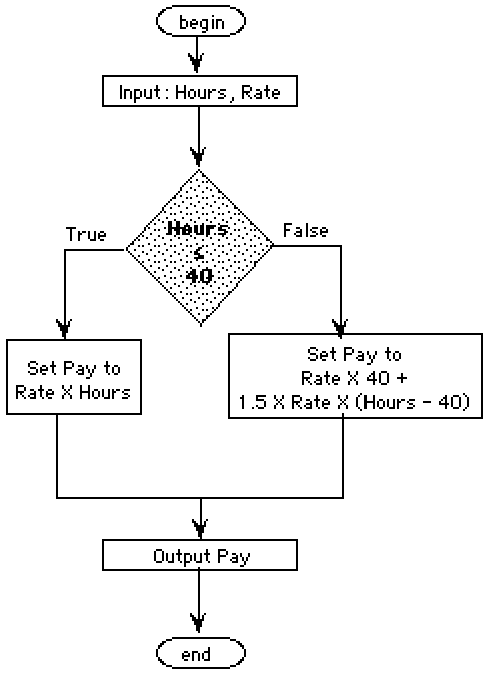


Figure 8-1: Branching Flowchart

Notice the test condition for hours. If hours is less than or equal to 40, one sequence of statements is implemented. However, if hours is greater than 40, another sequence of statements is executed. Using flowcharts can help you determine where branches occur in your program.

Let's look at a slightly different example. The most common type of branching uses a test condition to determine a course of action (as per our definition), but then calls a different procedure or function to perform that sequence of commands. Thus, you "branch" out of the current routine and into another. For example:

hour up to 40 hours. That statement written in C, might look like the following:

```
#define kNormalPay    10

#define kNormalWeek    40

float overtimePay;

...

if (hours > 40)

overtimePay = CalculateOvertime(hours, kNormalPay *
1.5);

else

overtimePay = 0.0;

...
```

In this example, if an employee worked overtime (more than 40 hours) the code "branches" into a function to calculate the overtime pay. Often, if you need to perform many commands as a result of a branch, you use a procedure or function. If, however, you only need to perform a few commands (such as a variable assignment) you can place those commands in the branch instead of in a separate routine.

Using Expressions as Parameters

Notice how the pay rate is passed as a parameter to the CalculateOvertime function. You can use a mathematical expression as a parameter. The expression is evaluated first and the result is passed to the procedure or function. Using expressions as parameters in this way can save time and coding; however, it's not recommended to use complicated expressions. Use short well-defined expressions that are easily understood by reading the code.

Lesson 9: Going Loopy

Lesson Description: In this lesson, you learn about different kinds of loops, when to use loops, how to use loops to access data, and more. Once you finish the lesson, you will have all the skills necessary to complete the final project in Lessons 10-12.

What Is a Loop?

There's an old joke about a programmer who couldn't get out of the shower. The reason? He took the instructions on the shampoo bottle literally. They read, "lather, rinse, repeat." The programmer effectively got stuck in an "infinite loop." He read each line, performing the action described, and then hit repeat -- at which point he began the whole process again. This joke illustrates the concept of simple looping.

A loop is created when you repeat the same set of actions more than once in a single section of code. To exit a loop, you require a test condition that evaluates to false. The test condition is usually part of the loop structure and can be either at the beginning or the end of the loop. A loop with the test at the beginning is called an entry-condition loop. A loop with the test at the end is called an exit-condition loop. Some programmers consider an entry-condition loop superior to the exit-condition variety because it's easier to test before you loop. In some cases, you may need to skip the loop entirely if the test condition fails, which cannot be done with an exit-condition loop.

There are many kinds of loops. You'll find that some types of loops are common between programming languages, although they may have a slightly different structure. Other loops may be unique to your programming language of choice. If a particular loop is not covered here, you should research the kinds of loops your programming language offers and how to use them.

In C, there are three fundamental kinds of loops. They are the "for" loop, while . . . do, and do . . . while. Let's look at each one in more detail.

"For" Loop

The "for" loop is an incremental loop, meaning that the loop is performed a fixed number of times incrementally. All the parameters (conditions) for the loop are given at the beginning. The parameters are starting value, ending value, and increment value. The parameters for the "for" loop perform the following three tasks:

- Initialize the counter variable
- Compare the counter with the stopping value
- Increment the counter each time the loop iterates

Let's look at a practical example. Suppose you want to print the numbers 1 through 10 on the screen. You could write ten lines of code, or you can write just a few lines of code like this:

```
#include <stdio.h>

int main(void)
{
    short    count;
    for (count = 1; count <= 10; count++)
        printf("%d\n", count);
}
```

The first part of the loop sets the variable "count" to 1. The semicolons separate each section of the loop. The next section is an expression of the test condition. If this expression becomes false, we exit the loop. In this case, we continue looping as long as our count variable is less than or equal to 10. Finally, count++ is C shorthand for saying count = count + 1. This last section incrementally increases the variable count by 1. Lastly, the "printf" statement writes out the value of the variable "count" on a new line for each iteration of the loop.

With all this in mind, what would happen if the test condition was "count < 10" instead of "count <= 10"? The program would write out "count" until 10 but would not write 10 on the last loop because "count = 10" and the expression would be false.

Like "if" and "case" statements, you can write a loop that requires more than one line of code to execute. Consider the following pseudo example:

```
short count;
for (count = 0; count <= 500; count+=2)
    {
        DoSomething();
        DoSomethingElse();
        BringItOn();
    }
```

Whenever you use multiple statements in a loop, you need to group them together using braces as shown above. There are many varied opinions on placement of the braces and on whether or not using braces for even a single line in the "for" loop makes the loop more readable or not. After you read other people's source code for a while, you'll develop your own opinion and adopt a style you feel comfortable using.

One of the most common uses of a "for" loop is to step through an array variable to read or write information to that array position. Consider the following example:

```
Boolean vote[50]; /* remember, Boolean is really an
int */
```

```
     /* code to fill vote array not shown*/
     short count, numVotes;
     numVotes=0;
     for (count = 0; count < 50; count++)
         {
             if (vote[count] == 1)
                 numVotes++;
             printf("Woohoo! You have %d votes!",
     numVotes);
         }
```

This code snippet looks at each position of the array vote looking for a value of 1, or true. If the value at position "count" is 1, then numVotes is increased by 1. The total number of true votes is then written out in the "printf" statement. Normally, the total votes would only be printed once the loop is finished; you could then eliminate the braces in the above example. However, the goal here is to show you that the code in the loop is indeed being run through the loop each time.

One other thing to note about this loop is the comparison used as the test case: count < 50. Remember that arrays in C are 0 based. That is, they start at zero and continue to the size of the array minus one (50 -1). By using the < (less than) operator as opposed to the <= (less than or equal to) operator, we avoid going "out of bounds" by counting past the end of the array. Alternatively, you could use the <= operator; just remember to use the size of the array minus 1 as the comparison.

"While" Loop

"While" loops are another form of loop common in many modern programming languages. Some languages call "while" loops "while-do" loops because they have the following structure:

```
     while (something is true)
         do something
```

The test condition for a "while" loop is at the beginning, similar to the "for" loop. Thus, if the test condition fails or evaluates to false, then the loop is skipped. Let's change our "for" loop from above to a "while" loop.

```
     Boolean vote[50]; /* remember, Boolean is really an
     int */
     /* code to fill vote array not shown */
     short count, numVotes;
```

```
count = 0;
while (count < 50)
{
    if (vote[count] == 1)
        numVotes++;
    printf("Whohoo! you have %d votes!", numVotes);
    count++;
}
```

A few things to note in this example. This code is essentially the same as for the "for" loop, with the following exceptions:

- The variable "count" was initialized (given a starting value) outside the loop structure.
- The variable "count" is incremented near the end of the loop.

In general, many programmers do not set a variable immediately before a "while" loop. In reality though, it depends on what you need to do.

All test expressions must evaluate to true or false. You can even use a regular Boolean expression, like so:

```
while (someBool == TRUE)
{
...
}
```

Do . . . While Loop

The "do . . . while" loop is an exit condition loop. This means that the loop is executed at least one time regardless if the test condition fails or not. This quality gives a "do . . . while" loop certain advantages and disadvantages. On the disadvantage side, if the test condition is met before going into the loop, the loop still executes at least one time until it tests at the end of the loop. On the advantage side, if you know for sure that you need to perform a series of actions at least one time (regardless of any prior condition being met), and then may need to execute that same series of actions again after that, then a "do . . . while" loop fits the bill quite nicely.

Let's use our vote counting example once again:

```
Boolean vote[50]; /* remember, Boolean is really an
int */
/* code to fill vote array not shown */
```

```
short count, numVotes;
count = 0;
do
{
    if (vote[count] == 1)
        numVotes++;
    printf("Whohoo! you have %d votes!", numVotes);
    count++;
} while (count < 50)
```

While and Do . . . While Caveats

This particular example is somewhat impractical as a "do . . . while" loop, or as a "while" loop, as shown previously. Consider the situation in which the variable "count" is not initialized. What happens? Because "count" would not be explicitly set to zero, it could be any arbitrary value. Some compilers catch this and automatically initialize all variables to zero until you change their value. However, you cannot rely on a compiler to do that for you. We may forget to set "count" to zero and inadvertently attempt to access an arbitrary array position, 125 for example, which does not exist, causing the program to access invalid memory or even crash.

Let's modify the example slightly. In an election, votes are counted until the last ballot is reached. You know you will count at least one ballot and may count more. Suppose we have a function that tests if we have counted the last ballot, and there are three parties we're counting ballots for. The code might look like this:

```
long partyA, partyB, partyC;
short vote; /* the value of vote can be 1, 2 or 3 */
partyA = partyB = partyC = 0;
/* a little shorthand to initialize all variables to 0
*/
do
{
    vote = GetNextBallot();
    switch (vote)
{
case 1:
    partyA++; /* increment partyA's number of votes */
    break;
case 2:
    partyB++;
    break;
case 3:
    partyC++;
    break;
    }
printf("Running totals-Party A: %d, Party B: %d, Party
C: %d",
partyA, partyB, partyC);
} while ( !LastBallot() )
```

In this example, we have a function at the beginning of the loop to get the next ballot. We use a "case" statement to sort out the possibilities of the vote. Use of the "!" character in the test condition is the same as saying NOT. So the test condition reads: while NOT LastBallot. LastBallot() is a Boolean function that peeks to see if there are more ballots. If so, the loop begins again by grabbing the next ballot in the pile.

Lesson 10: Real World Problems

Lesson Description: In this lesson, you will complete your first project. You will gain all the background knowledge you need to start the project by developing flowcharts, algorithms, and program specifications.

Problem Description

The problem you will solve over the next three lessons is keeping track of a bookstore inventory. The program you write does not need to be elaborate; in fact, we'll keep the program requirements manageable to ensure simplicity. The bookstore in question is a secondhand bookstore.

The Inventory Process

The bookstore -- let's call it Books With Style -- receives books by donation or purchase. These books are handed off to the inventory clerk for entry into their database. Each book has an International Standard Book Number (ISBN) associated with it. Since a fallible human is entering the data, it's quite possible a mistake could be made and the wrong ISBN input into the database. The program needs to check to ensure the correct ISBN is typed in (more on that a little later). In addition to the ISBN, the book's title, author, publication date, category, and whether the book is hard or soft cover are also stored in the database. However, the book cannot be stored until the correct ISBN is input.

Warning the user of incorrect input like this is called "error checking." Checking to see if certain conditions are satisfied before continuing, checking the result after calling a function that may return an error result, and checking a value to see if it's out of range are all examples of error checking. Error checking can be a time-consuming process since you are performing an action that may not need to be done 100 percent of the time, thus slowing down your program a bit. However, error checking is an extremely important part of programming and needs to be done to ensure that some errors and bugs can be prevented during the course of running your program.

Database

Once we are sure that the ISBN for the book entered is correct, we can allow the user to add other important data to the store's database. The program must allow the following actions to be performed on the database:

- Add new book
- Delete a book
- Show all books in the database

For our purposes, we don't really want anything that is overly complex. We'll keep the database small and simple to use and access. More information on the database will be provided in the next lesson.

ISBN System

Almost all books have printed somewhere on the back cover a ten-digit number called the International Standard Book Number, or ISBN. The ISBN is broken into four parts, each of which has a special meaning. For example, the ISBN for the book *C Primer Plus*, used to supplement this course, is this:

```
1-571690-161-8
```

The first section is the group identifier, the second is the publisher prefix, and the third is the title identifier. These first three sections vary in size. The number of digits in the group number and in the publisher prefix is determined by the quantity of titles planned to be produced by the publisher or publisher group. Publishers or publisher groups with large title outputs are represented by fewer digits.

In our example above, the first digit, 1, means that the book was published in an English-speaking country. The second set of digits, the publisher's prefix, means that it was published by a section of SAMS publishing. The third group is unique to the book itself. You may find other books like this published by SAMS that will have the same publisher's prefix. The last digit is called a check digit. (Remember our discussion of error checking? Error checking is built into the ISBN system.)

The Check Digit

The check digit is calculated using a modulus of 11 and a "weighted sum." Each digit of the ISBN, including the check digit, is multiplied by a number from 10 to 1 and the sums of the products are added together. That value is then divided by 11 to see if there is any remainder. Many books use an X as the check digit in lieu of using the number 10. X is 10 in the Roman numeral system. Let's look at a couple of examples.

	Group ID	**Publisher Prefix**					**Title ID**			**Check Digit**
ISBN	1	5	7	1	6	9	1	6	1	8
Weight	10	9	8	7	6	5	4	3	2	1
	10	+45	+56	+7	+36	+45	+4	+18	+2	+8
Total	231	Checksum: 231 / 11 = 21 with no remainder. So the ISBN is valid.								

Another way you can do this is in reverse. Multiply the first 9 digits by their position (1 to 9) and add the sum. Then divide that sum by 11 and see if there is a remainder. The remainder should be the check digit. Using our example above, we get:

	Group ID	**Publisher Prefix**					**Title ID**			**Check Digit**
ISBN	1	5	7	1	6	9	1	6	1	8
Weight	1	2	3	4	5	6	7	8	9	
	1	+10	+21	+4	+30	+54	+7	+78	+9	
Total	184	Checksum: 184 / 11 = 16 with a remainder of 8 which is the check digit. So the ISBN is valid.								

You can find out more information about the ISBN system and how it works from the International ISBN Agency Web site at:
http://www.isbn.spk-berlin.de

Let's look at an example that has an error in it. We'll use both methods for demonstration purposes.

	Group ID	**Publisher Prefix**					**Title ID**			**Check Digit**
ISBN	5	7	6	2	6	1	1	6	3	2
Weight	10	9	8	7	6	5	4	3	2	1
	50	+49	+48	+14	+36	+5	+4	+15	+6	+2
Total	229	Checksum: 229 / 11 = 20 with a remainder of 9. So the ISBN is not valid.								

Using the reverse method, we get:

	Group ID	**Publisher Prefix**					**Title ID**			**Check Digit**
ISBN	5	7	6	2	6	1	1	6	3	2
Weight	1	2	3	4	5	6	7	8	9	
	5	+14	+18	+8	+30	+5	+7	+48	+18	
Total	153	Checksum: 153 / 11 = 13 with a remainder of 10 (or X) which is not the check digit. So the ISBN is not valid.								

Lesson 11: Coding Up A Storm

Course Slug: 100000117

Lesson Description: Let's talk a little more about program design, features, and documentation.

Design and Features

One of the hardest parts of a programmer's job is getting input for program features that a user or users want implemented and then taking those and coming up with a workable program design to accommodate the request. Why is this so hard? Think for a moment about one of your favorite software programs. Think now about all the features that program has. Now, how many of those features do you really use? Probably not all of them, right? The argument in the software industry is that some users somewhere asked for a feature to be included in the program. While this may indeed be true, it seems that some of those features are implemented without much thought to their real design and implementation.

The program discussed in the last lesson does not need to be sophisticated. It does, however, need to accomplish the following set of goals:

- Add new books.
- Show all books in the inventory (print a list).
- Keep track of how many books are in the inventory.
- Check for errors of input of the ISBN (check code, as well as length).
- Limit the inventory to 50 books (for now).
- No file processing -- the program does not need to be able to save the inventory to file and read it back in. A full implementation would have that feature, but we're not concerned with that for now.
- Be easy to use.
- Be easy to change and update.

You only need to implement one of the ISBN check digit algorithms discussed in the previous chapter. Choose whichever version you feel will be easiest for you to write and understand.

In order to get input from the user, there are two approaches you can use. You can write a procedure that uses a few printf() statements to print a textual menu like this:

```
void PrintMenu( void )
{
    /* print a bunch of empty lines first */
    printf("\n\n\n\n\n\n\n\n\n");
    /* the \t is a special character that means tab */
    printf("\t\t Type the first letter of the option
you want\n");
    printf("\t\t\t [A]dd a book\n");
    printf("\t\t\t [P]rint list of books\n");
    /* add a couple more blank lines to make the menu
look nice */
    printf("\n\n");
}
```

Then, in your main routine, you would have something like this:

```
void main( void )
{
    char ch;
    PrintMenu();
    printf("Type the letter you want and hit return:
");
    scanf("%c", &ch);
    printf("\n\n");
    /* other code to call routines based on the user's
choice */
}
```

You don't have to use letters; you could use numbers instead. For example, 1 is add and 2 is print. The order of the listing is somewhat arbitrary. However, common practice is to put actions that are performed frequently (add) near the top of the menu and actions performed less frequently near the bottom (print).

You do not need to be as elaborate as in the above examples. You could just use printf() and scanf() to request the user input an action in a more simplified manner, like this:

```
void main( void )
{
    char ch;
    printf("type the first letter of the action to
perform\n");
    scanf("[A]dd a book, [P]rint book list: %c", &ch);
```

```
        }
```

Each operation (add and print) should be a separate procedure or function. This keeps the code "modular" and easier to maintain and understand. You should name the procedure or function according to the operation it is to perform. For example: AddBook, PrintInventory.

One thing to remember in programming: if the program works -- it does what it is designed to do -- the implementation (how the program was written) is not wrong. The code might be inefficient and might benefit from streamlining, but it is NOT wrong. Streamlining and optimizing a program come with experience. I do not expect you to have that experience and you should not expect it of yourself. What you should expect is to learn by doing. As you continue programming, you'll learn new and better ways to accomplish certain tasks. You can then use that knowledge to go back and refine software you've previously written, or just apply it to new programs you write.

Design Hints

You do not need to read this section if you already have ideas on how to accomplish the above goals. If you are unsure, or just need a little pointer to head you off in the right direction, read on.

The easiest way to accomplish all the goals listed above is to use a combination array and structure for the book inventory. Since we're dealing with a finite amount of information, an array will work perfectly. Using an array makes it easy to step through the inventory to print it and easy to add items if you keep track of the last array position. It can also be easily changed later on without significantly affecting the rest of the program.

The ISBN should be entered as an array with one digit of the ISBN occupying one position in the array. The size of this array then would be 10 (0 to 9). One way is to accomplish this is to try and separate out each number as it is entered and place the value into the array. You need to test for errors during input and convert the X to a 10 for the check digit where appropriate. For a further hint on how you might do this, look over Chapter 7 in your book, especially pages 211-215 and 232-236.

For convenience purposes, and because it's easier to code, use global variables for the book structure, the book array, the ISBN array, and an index variable into the book array so you know how many books you have just by looking at that variable. As a bonus, once you've validated an ISBN number, store it as a string variable in the book information structure. This is easier to do than it sounds. Remember: arrays and strings are essentially equal except for a null terminator "\0" at the end of a string, which means your string variable will need to be one character larger than your ISBN array.

Program Documentation

Let's talk a bit about documentation. Documenting a program occurs on two levels, the source level and the user level. User-level documentation is the written manual or document that you give to the people who will use your program. A dedicated technical writer who works closely with the programmer (you) to help users understand the program usually does user-level documentation. Occasionally, however, the programmer does that job.

The documentation you must write is the source documentation. By source documentation, I mean using comments throughout the source code to explain what is happening for a section of code. This does two things. First, it helps anyone who may read your code at a later date to understand the code better. Programmers can be notoriously vague when it comes to their own code. Do your best not to succumb to that temptation. The second thing it does is that it helps YOU, the programmer, understand what is happening in your code if you go back to it at a later date to make changes or even simply review what you have written.

You've seen examples of source code comments in some of the examples throughout this course. Here are some pointers on writing well-documented source code.

2. 1;Write a comment block before a procedure or function definition to describe what the function is supposed to do.

8. Write comments throughout a function or procedure to describe areas that may not be immediately understandable.

In C, a comment block may look like this:

```
/*****************************************************
******

*

* FunctionName - This function does something really
cool.

*

*****************************************************
*****/
```

A comment in C continues until it is explicitly terminated, even spanning multiple lines. A comment starts with the characters /* and ends with the characters */. You can design a comment block however you like. You may also add information to the comment block, such as who wrote the procedure or function (good if working on a program with a group of people), date it was written, and maybe a description of the parameters.

Bad Comments

Like many other good things, comments can be abused or overused when there is no need. Consider the following:

```
short count; /* a count variable /*

...

test = 0; /* set test to 0 */
```

```
...
addresses myAddresses[5000] /* an array to keep track
of addresses */
for (count = 0; count < 5000; count ++) /* a for loop
*/
    PrintAddresses();
```

These comments state the obvious. The programmner has choosen to use variables that clearly define their use and make comments unnecessary. Howerever, commenting on loops and assignments is always a good practice. The basic idea is to use your judgment.

Lesson 12: Debugging Technique

Lesson Description: The last thing to learn before you go on your merry way is a little about debugging technique; that is, how to track down and correct problems in your programs.

Debugging Methods

There are many different methods programmers use to "debug" their code. Debugging is the process of going through your program to find errors and problems, and then eliminating them. Let's look at a couple different methods.

Strategic Statements

One of the easiest techniques to use in debugging a program is the use of strategically placed printf statements. Using a printf call to print where you are in the source code, or to print the value of a variable at a given point in time, allows you to actively track the progress of your program.

For example, let's say I write a routine that calls a particular function. However, the results from that function are either erroneous or the program "locks up" and doesn't return from the function call. I can place a printf statement just before I call the function, and somewhere else in the function to know my program actually got to a certain point.

```
void main( void )
{
    long num = 0;
    /* some other code may go here */
    printf("Calling function CalculateSomething()\n");
    num = CalculateSomething();
    printf("The value of num after
CalculateSomething() is %d",num);
}
```

In this example, if you don't see the text from the second printf call on the screen, you know something messed up during the call to CalculateSomething() and you need to dig further.

You can do other strategic things with printf as well. For example, you can print the value of parameters before calling a routine and print what the values are once you arrive in that routine.

Some languages and operating systems allow other types of strategic statements. For example, calling the operating system to make a beep sound or pause the program to wait for input from the user. It depends on your choice of languages and the OS-specific features you want to use. Printing information to the screen is the most common type of strategic statement debugging.

Hard-Copy Debugging

Hard-copy debugging is the second most common form of debugging. In fact, the process of hard-copy debugging is what led to the source-level debugger that we'll discuss in the next section.

Simply put, hard-copy debugging involves printing out the source code to your program and, using a pencil and maybe some extra paper, tracing through your program as though you were the computer. You look at each line of code carefully to try to figure out what it is doing. If there are any mathematical calculations involved, you perform them and write down the result. If a variable changes value, you write down the new value. When you step through a loop, you write down the result of each pass through the loop so you can track what you think is happening and compare it to what may really be happening.

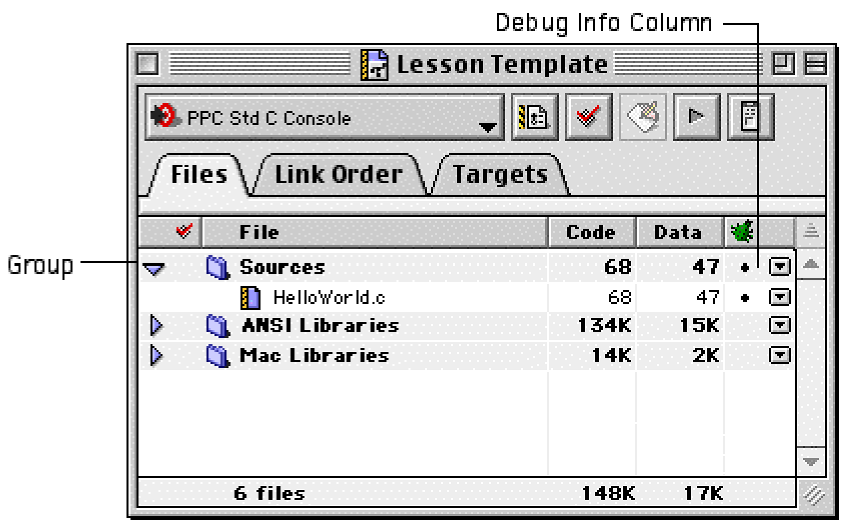
Obviously, hard-copy debugging can be a time-consuming process. However, it's not always necessary to print out the source for an entire program. Often, all you need to print is just a few key files (in the case of a multi-file project) or routines and just trace through those.

While debugging source code this way can be extremely tedious, it can also be very rewarding. You learn a little more about your program by thinking like the computer. You can often find mistakes much easier this way, especially mistakes in logic. You may also want to ask someone to help you with the process. Often a fresh perspective on your code finds bugs much quicker. I don't know how many times I've looked at a line of source code trying to find the error and someone (often my wife) looks at it and says, "I think you forgot a comma here." Needless to say, she's usually right.

Source-Level Debuggers

Now that we've covered a few of the traditional debugging techniques, let's talk about how changes in compilers and the introduction of Integrated Development Environments (IDEs) make your life that much easier. Since this course uses the Metrowerks *CodeWarrior* IDE as the basis, I'll focus on it. However, this discussion can apply to any IDE and source level debugging tool or IDE.

Integrated Development Environments are so called because they combine a compiler, editor, project management tool, and debugger into a single package. The project window in *CodeWarrior,* for example, allows you to see all the files and libraries associated with your project and even keep them in groups.

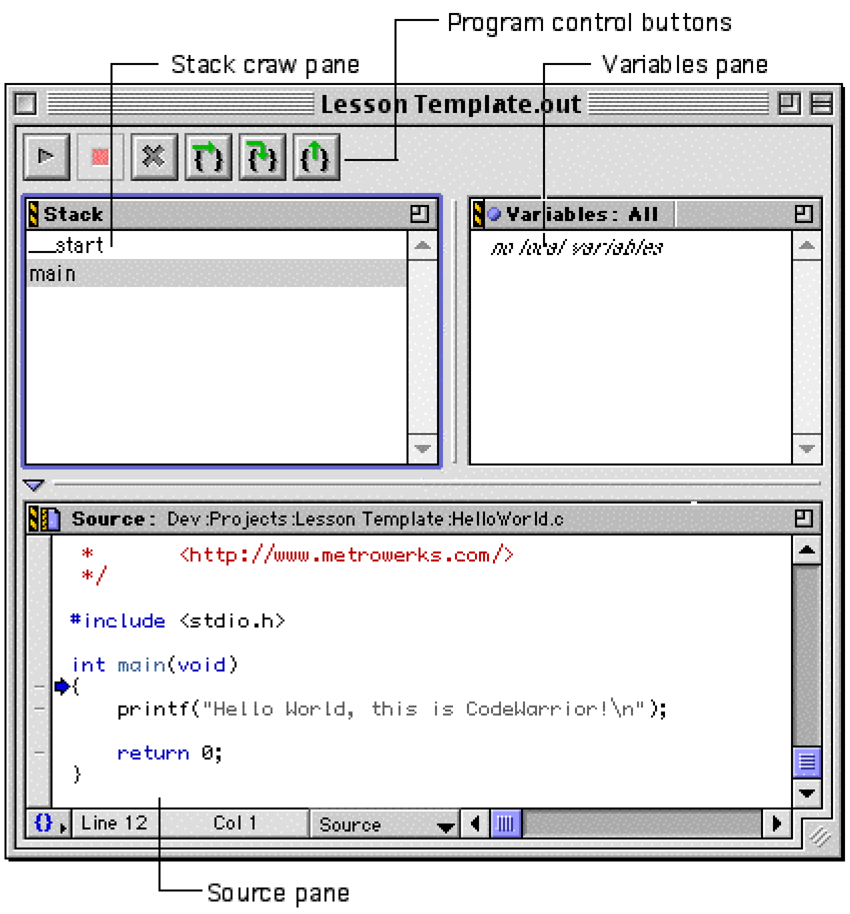


File	Code	Data
Sources	68	47
HelloWorld.c	68	47
ANSI Libraries	134K	15K
Mac Libraries	14K	2K
6 files	148K	17K

Figure 12-1: CodeWarrior Project Window

CodeWarrior also allows you to turn debugging information on and off for a particular file right in the project window. You can easily add and remove files using drag and drop, as well as get a "snapshot" look at your project.

The source-level debugger included with *CodeWarrior* is one of the most powerful of any IDE. You can step through your source code line by line using the program control buttons (or keyboard shortcuts), thus allowing you the same capabilities of hard-copy debugging via electronic means. You can see variables and watch them as they change in the variable pane. You can see what functions you have called in the stack crawl pane, allowing you to trace the path your program took to get to that point. Additionally, you can jump between files and functions and display source code in source form, assembler (also called machine language), or a mixed version of the two.



```
 *       <http://www.metrowerks.com/>
 */

#include <stdio.h>

int main(void)
{
    printf("Hello World, this is CodeWarrior!\n");

    return 0;
}
```

Figure 12-2: CodeWarrior Debugger Window

There is much more the *CodeWarrior* IDE can do for you than can be explained here in a short lesson. I encourage you to go through the tutorials as well as the printed and electronic documentation that come with *CodeWarrior* to learn more about its power and flexibility.