Click to Print

C: Week 1

You can print this text-only version of this course for future reference.

If you wish to use the accessible version of our courses, which includes questions in text-only format, click **Text Only** on the log on page, and then enter your user ID and password from the Accessibility Log On page.

Unit 1. Day 1 Getting Started

Welcome! This unit shows you how to create a C program and introduces the basic elements of a simple program.

This unit starts you toward becoming a proficient C programmer. Today, you learn

- Why C is the best choice among programming languages.
- The steps in the program development cycle.
- How to write, compile, and run your first C program.
- About error messages generated by the compiler and linker.

Topic 1.1: A Brief History of the C Language

*** The Creation of C**
You may be wondering about the origin of the C language, and where it got its elegant name. C was created by Dennis Ritchie at the Bell Telephone Laboratories in 1972. The language was not created for the fun of it, but for a specific purpose: designing the UNIX operating system (which is used on many minicomputers). From the beginning, C was intended to be useful: to allow busy programmers to get things done.

*** ANSI Standard C**
Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs. Soon, however, different organizations began utilizing their own versions of C, and subtle differences between implementations started to cause programmers headaches. In response to this problem, the American National Standards Institute (ANSI) formed a committee in 1983 to establish a standard definition of C, which became known as *ANSI Standard C*. With few exceptions, every modern C compiler adheres to this standard.

*** The Name "C"**
Now, what about the name? The C language is so named because its predecessor was called B. The B language was developed by Ken Thompson, who was also at Bell Labs. You might guess easily why it was called B.

Topic 1.2: Why Use C?

*** Advantages of C**
In today's world of computer programming, there are many high-level languages to choose from, such as C, Pascal, BASIC, and Modula. These are all excellent languages suited for most programming tasks. Even so, there are several reasons why many computer professionals feel that C is on top of the list:

Reason	Details
C is a powerful and flexible language.	What you can accomplish with C is limited only by your imagination. The language itself places no constraints on you. C is used for projects as diverse as operating systems, word processors, graphics, spreadsheets, and even compilers for other languages.
C is a popular language.	C is a popular language, preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.
C is a portable language.	*Portable* means that a C program written for one computer system (an IBM PC, for example) can be compiled and run on another system (a DEC VAX system perhaps) with little or no modification. Portability is enhanced by the ANSI standard for C, the set of rules for C compilers discussed earlier.

C is a language of few words.	C is a language of few words, containing only a handful of terms, called *keywords*, which serve as a base on which the language's functionality is built. You might think that a language with more keywords (sometimes called reserved words) would be more powerful. This is not true. As you program with C, you will find it can be programmed to do any task.
C is modular.	C code can (and should) be written in routines called *functions*. These functions can be reused in other applications or programs. By passing pieces of information to the functions, you can create useful, reusable code.

*** What Is C++?**
As these features show, C is an excellent choice for your first programming language. What about this new language called C++ (pronounced C *plus plus*)? You may have heard already about C++ and a new programming technique called *object-oriented programming*.

*** C Versus C++**
Perhaps you're wondering what the differences are between C and C++ and whether you should be teaching yourself C++ instead of C. Not to worry! C++ is a *superset* of C, which means that C++ contains everything C does, plus new additions for object-oriented programming. If you do go on to learn C++, almost everything you learn about C will still apply to the C++ superset. In learning C, you are not only learning today's most powerful and popular programming language, but you also are preparing yourself for tomorrow's object-oriented programming.

Topic 1.3: Preparation for Programming

*** Steps for Solving a Problem**
Certain steps should be taken when a problem is being solved.

Step	Action
1	First, the problem must be defined. If you don't know what the problem is, you can't find a solution!
2	Once the problem is known, you can devise a plan to fix it.
3	Once there is a plan, usually you can implement it easily.
4	Finally, once the plan is implemented, the results must be tested to see whether the problem is solved.

This same logic can be applied to many other areas, too, including programming.

*** Steps for Creating a Program in C**
When creating a program in C (or for that matter, a computer program in any language), you should follow a similar sequence of steps:
1. Determine the objectives of the program.
Example 1
2. Determine the methods you want to use in writing the program.
Example 2

3. Create the program to solve the problem. (The Program Development Cycle)
4. Run the program to see the results. (The Program Development Cycle)

Example 3

An example of an objective might be to write a word processor or database program. A much simpler objective is to display your name on the screen. If you did not have an objective, you would not be writing a program, so you already have the first step done.

The second step is to determine the method you want to use in writing the program. Do you need a computer program to solve the problem? What information needs to be tracked? What formulas are going to be used? During this step, you should try to determine what you need to know and in what order the solution should be implemented.

As an example, assume that someone asks you to write a program to determine the area inside a circle.
Step one is complete because you know your objective: determine the area inside a circle.
Step two is to determine what you need to know to ascertain the area. In this example, assume that the user of the program will provide the radius of the circle. Knowing this, you can apply the formula pi r^2 to obtain the answer.
Now you have the pieces you need, so you can continue to steps three and four, which are called the Program Development Cycle.

Topic 1.4: The Program Development Cycle

*** Steps**

The Program Development Cycle has its own steps. You'll see more about these steps on subsequent screens..

Step	Action
1	**Use an editor to write your *source code*.** By tradition, C source code files have the extension .C (for example, MYPROG.C, DATABASE.C, and so on).
2	**Compile the program using a *compiler*.** If the compiler does not find any errors in the program, it produces an object file. The compiler produces *object files* with the .OBJ extension and the same name as the source code file (for example, MYPROG.C compiles to MYPROG.OBJ). If the compiler finds errors, it reports them. You must return to step one to make corrections in your source code.
3	**Link the program using a *linker*.** If no errors occur, the linker produces an *executable program* located in a disk file with the .EXE extension and the same name as the object file (for example, MYPROG.OBJ is linked to create MYPROG.EXE).
4	**Execute the program.** You should test to determine whether it functions properly. If not, start again with step one and make modifications and additions to your

	source code.

*** Steps Presented Schematically**
Program development steps are presented schematically in Figure 1.1.
For all but the simplest programs, you may go through this sequence many times before finishing your program. Even the most experienced programmers can't sit down and write a complete, error-free program in just one step! Because you'll be running through the edit-compile-link-test cycle many times, it's important to become familiar with your tools: the editor, compiler, and linker.

Topic 1.4.1: Creating the Source Code

*** Source Code**
Source code is a series of statements or commands, used to instruct the computer to perform your desired tasks. As mentioned, the first step in the Program Development Cycle is to enter source code into an editor.
Example
*** Using an Editor**
Some compilers come with an editor that can be used to enter source code and some do not. Consult your compiler manuals to see whether your compiler came with an editor. If not, many editors are available.
Most computer systems include a program that can be used as an editor. If you are using a UNIX system, you can use such commands as `ed`, `ex`, `edit`, `emacs`, or `vi`. If you are using Microsoft Windows, Notepad is available. If you are using DOS 5.0, you can use edit. If you are using a version of DOS before 5.0, there is edlin.

For example, here is a line of C source code:

```
printf("Hello, Mom!");
```

This statement instructs the computer to display the message `Hello, Mom!` on the screen. (For now, don't worry about how this statement works.)

*** Using a Word Processor**
Most word processors use special codes to format their documents. These codes can't be read correctly by other programs. The American Standard Code for Information Interchange (ASCII) has specified a standard text format that nearly any program, including C, can use. Many word processors, such as WordPerfect and Microsoft Word, are capable of saving source files in ASCII form (as a text file rather than a document file). When you want to save a word processor's file as an ASCII file, select the ASCII or text option when saving.
If none of these editors is what you want to use, you can always buy a different editor. There are packages, both commercial and shareware, that have been designed specifically for entering source code.

*** Saving a Source File**
When you save a source file, you must give it a name. What should a source file be called? The name you give it should describe what the program does. In addition, when you save C program source files, name the file with a `.c` extension.

Topic 1.4.2: Compiling the Source Code

*** From Source Code to Object Code**
Although you may be able to understand C source code (at least, after taking this course, you will be able to!), your computer cannot. A computer requires digital, or binary, instructions in what is called *machine language*.

*** Role of the Compiler**
Before your C program can run on a computer, it must be translated from source code to machine language. This translation, the second step in program development, is performed by a program called a *compiler*.
The compiler takes your source code file as input and produces a disk file containing the machine language instructions that correspond to your source code statements. The machine language instructions created by the compiler are called object code, and the disk file containing them is called an *object file*.

If you want to try the coding examples in this course, you will need a C compiler. You may not have one since most computer systems don't include a C compiler with their standard configuration. If you need to get a C compiler, try using a search engine on the Internet to find a free one.

This course covers the ANSI Standard C. This means that it doesn't matter which C compiler you use as long as it follows the ANSI Standard.

*** Creating the Object Code**
Each compiler requires that its own command be used to create the object code. To compile, you typically use the command to run the compiler followed by the source filename.
Example
*** Naming the Object File**
After you compile, you have an object file. If you look at a list of the files in the directory in which you compiled, you should find a file with the same name as your source file, but with an .OBJ (rather than a .C) extension. The .OBJ extension is recognized as an object file and is used by the linker. On UNIX systems, the compiler creates object files with the .O extension instead of the .OBJ extension.

The following are examples of the commands issued to compile a source file called RADIUS.C using the command line on various DOS compilers:
Microsoft Visual C++ `cl radius.c`
Borland's C++Builder `bcc32 radius.c`
GNU minGW `g++ radius.c`
To compile RADIUS.C on a UNIX machine, you usually use
`cc radius.c`
Consult the compiler manual to determine the exact command for your compiler.

Topic 1.4.3: Linking to Create an Executable File

*** Function Library**
One more step is required before you can run your program. Part of the C language is a *function library* that contains object code (that is, code that has already been compiled) for *predefined functions*. A predefined function contains C code that has already been written and is supplied in a ready-to-use form with your compiler package. The `printf()` function used in the previous example is a library function.

*** Linking**
These library functions perform frequently needed tasks, such as displaying information on-screen and reading data from disk files. If your program uses any of these functions (and hardly a program exists that doesn't use at least one), the object file produced when your source code was compiled must be combined with object code from the function library to create the final executable program. (*Executable* means that the program can be run, or executed, on your computer.) This process is called *linking* and is performed by a program called (you guessed it!) a *linker*. Click the Note button.
The progression from source code to object code to executable program is diagrammed in Figure 1.2.

One final note on compiling and linking. Although compiling and linking are mentioned as two separate steps, many compilers, such as the DOS compilers mentioned earlier, do both as one step. Regardless of the method by which compiling and linking are accomplished, understand that these two processes, even when done with one command, are two separate actions.

Topic 1.4.4: Completing the Development Cycle

*** Execute the Program**
Once your program is compiled and linked to create an executable file, you can run it by entering its name at the system prompt, just like you would any other program. If you run the program and receive results different from what you believed you should, you need to go back to the first step. You must identify what caused the problem and correct it in the source code. When a change is made to the source code, you need to recompile and relink the program to create a corrected version of the executable file. You keep following this cycle until you get the program to execute exactly as you intended!

Topic 1.5: Your First C Program

*** Demonstration**
You are probably eager to try your first program in C! To help you become familiar with your compiler, here's a quick demonstration for you to work through. You may not understand everything at this point, but you should get a feel for the process of writing, compiling, and running a real C program.

*** Example Program**

*** Example Program**

This demonstration uses a program named HELLO.C, which does nothing more than display the words `Hello, World!` on your screen. This program, a traditional introduction to C programming, is a good one for you to learn. The source code for HELLO.C is in program Listing 1.1. Click the Listing button to see the listing.

Listing 1.1: HELLO.C

```
Code  1:  /* LIST0101.c: Day 1 Listing 1.1 */
      2:  /* HELLO.C: Displays the words Hello, World! */
      3:  /* on the screen. */
      4:
      5:  #include <stdio.h>
      6:
      7:  int main(void){
      8:    printf("Hello, World!");
      9:    return 0;
     10:  }
```

*** Entering and Compiling HELLO.C**

If you want to try entering and compiling this example program, be sure you have installed a C compiler as specified in the installation instructions provided with the software. Whether you are working with UNIX, DOS, or any other operating system, make sure you understand how to use the compiler and editor of your choice. Once your compiler and editor are ready, follow these steps to enter, compile, and execute HELLO.C.

*** Step 1**

Make active the directory your C programs are in and start your editor. As mentioned previously, any text editor can be used, but most newer C compilers (such as Borland's C++Builder and Microsoft's Visual C++) come with an integrated development environment (IDE) that allows you to enter, compile, and link your programs in one convenient setting. Check the manuals to see whether your compiler has an IDE available.

*** Step 2**

Use your keyboard to type the HELLO.C source code exactly as shown in Listing 1.1. Press Enter at the end of each line of the code.

*** Step 3**

Save the source code. You should name the file HELLO.C.

*** Step 4**

Verify that HELLO.C is on disk by listing the files in the directory. You should see HELLO.C within this listing.

*** Step 5**

Compile and link HELLO.C. Execute the appropriate command specified by your compiler's manuals. You should get a message stating that there were no errors or warnings.

*** Step 6**

Check the compiler messages. If you receive no errors or warnings, everything should be okay. But what if you made an error typing the program? The compiler catches it and displays an error message on your screen. For example, if you misspelled the word `printf` as `prntf`, the message

similar to the following is displayed `Error: undefined symbols:_prntf in hello.c (hello.OBJ)`

*** Step 7**
Go back to step two if this or any other error message is displayed. Open the HELLO.C file in your editor. Compare your file's contents carefully with this unit's Listing 1.1, make any necessary corrections, and then continue with step three, etc.
*** Step 8**
Your first C program should now be compiled and ready to run. If you display a directory listing of all files named HELLO (having any extension), you should see the following:

- HELLO.C (which is the source code file you created with your editor).
- HELLO.OBJ or HELLO.O (which contains the object code for HELLO.C).
- HELLO.EXE (which is the executable program created when you compiled and linked HELLO.C).

*** Step 9**
To run, or execute, HELLO.EXE, simply enter `hello`. The message `Hello, World!` is displayed on your screen.
*** Congratulations!**
Congratulations! You have just entered, compiled, and run your first C program. Admittedly, HELLO.C is a simple program that doesn't do anything useful, but it's a start. In fact, you ought to remember that most of today's expert C programmers started learning C in this same way — by compiling HELLO.C — so you're in good company.

Topic 1.6: Compilation Errors

*** When a Compilation Error Occurs**
A *compilation error* occurs when the compiler finds something in the source code that it can't compile. A misspelling, typographical error, or any of a dozen other things can cause the compiler to choke. Fortunately, modern compilers don't just choke, they tell what they're choking on and where it is! This makes it easier to find and correct errors in your source code.

*** Example of a Compilation Error**
This can be illustrated by introducing a deliberate error into HELLO.C. If you worked through that example (and you should have), you now have a copy of HELLO.C. Using your editor, move the cursor to the end of the line containing the call to printf(), and erase the terminating semicolon. HELLO.C should now look like Listing 1.2, except for the comments at the top. (Click the Listing button to view the listing.)

Listing 1.2: HELLO.C with an error

```
Code  1:  /* LIST0102.c: Day 1 Listing 1.2 */
      2:  /* HELLO.C with an error. */
      3:  /* Extra comment to keep lines right. */
      4:
```

```
4:
5:    #include <stdio.h>
6:
7:    int main(void){
8:      printf("Hello, World!")
9:      return 0;
      }
```

*** The Compiler Message**

Next, save the file. You're now ready to compile the file. Do so by entering the command for your compiler. Because of the error you introduced, the compilation is not completed. Rather, the compiler displays a message on the screen similar to the following:

```
hello.c(9) : Error: ';' expected
```

*** Three Parts**

Looking at this line, you can see that it has three parts.

Part	Description
`hello.c`	The name of the file where the error was found.
`(9)`	The line number where the error was found. (Note: Your line number may be different, depending on how many lines your comments take up at the top.)
`Error: ';' expected`	A description of the error.

*** Error: Line 9 or Line 8?**

This is quite informative, telling you that in line 9 of HELLO.C the compiler expected to find a semicolon but did not. Yes, you know the semicolon was actually omitted from line 8 in our listing, and that there is a discrepancy. You're faced with the puzzle of why the compiler reports an error in line 9 when in fact, a semicolon was omitted from the end of line 8. The answer lies in the fact that C doesn't "care" about things like breaks between lines. The semicolon that belongs after the `printf()` statement could have been placed on the next line (although doing so would be bad programming practice). Only after coming upon the brace in line 9 is the compiler sure that the semicolon is missing. Therefore, the compiler reports that the error is in line 9.

*** Interpreting the Message**

This points out an undeniable fact about C compilers and error messages. Although the compiler is very clever about detecting and localizing errors, it is no Einstein. You, using your knowledge of the C language, must interpret the compiler's messages and determine the actual location of any errors that are reported. They are often found on the line reported by the compiler and if not, they are almost always on the preceding line. At first, you may have a bit of trouble finding errors, but you should soon get better at it.

*** Another Example of a Compilation Error**

Before leaving this topic, take a look at another example of a compilation error. Load HELLO.C into your editor again and make the following changes:

1. Replace the semicolon at the end of the printf line.
2. Delete the double quotation mark just before the word `Hello`.

Save the file to disk and compile the program again.

Save the file to disk and compile the program again.

*** The Compiler Message**

This time, the compiler should display error messages similar to the following:

```
hello.c(8) : Error: undefined identifier 'Hello'
hello.c(9) : Lexical error: unterminated string
Lexical error: unterminated string
Lexical error: unterminated string
Fatal error: premature end of source file
```

*** Interpreting the Message**

The first error message finds the error correctly, locating it in line 8 at the word `Hello`. The error message `unidentified identifier` means that the compiler does not know what to make of the word `Hello`, because it is no longer enclosed in quotes. What, however, about the other four errors that are reported? These errors, the meaning of which you don't need to worry about now, illustrate the fact that a single error in a C program can sometimes cause multiple error messages.

The lesson to learn from all this is as follows: If the compiler reports multiple errors, and you can find only one, go ahead and fix that error and recompile. You may find that your single correction is all that's needed, and the program now compiles without errors.

Topic 1.7: Linker Error Messages

*** Linker Errors**

Linker errors occur when the linker is unable to find the library functions or files you specify. When you work with simple sample programs, such errors are relatively rare and usually result from mistyped file names, library names, or function names. In these cases, you'll see an `Error: undefined symbols:` error, followed by the misspelled name (preceded by an underscore). Once you correct the spelling, the problem should go away.

When you're working with multi-file programs, linker errors are more common. In these cases, linker errors can indicate missing files or incompatibility between files. Such issues are beyond the scope of this course.

Topic 1.8: Day 1 Q&A

*** Questions & Answers**

Take a look at some questions that are frequently asked by programmers new to C.

*** Question 1**

If I want to give someone a program I wrote, which files do I need to give them?

*** Answer**

One of the nice things about C is that it is a compiled language. This means that after the source code is compiled, you have an executable program. This executable program is a self-standing program. If you wanted to give HELLO to all your friends with computers, you could. All you need to give them is the executable program, HELLO.EXE. They don't need the source file, HELLO.C, or the object file, HELLO.OBJ. They don't need to own a C compiler either!

*** Question 2**

After I create an executable file, do I need to keep the source file (.C) or object file (.OBJ)?

After I create an executable file, do I need to keep the source file (.C) or object file (.OBJ)?
*** Answer**
If you get rid of the source file, you have no way to make changes to the program in the future, therefore this file should be kept! The object files are a different matter. There are reasons to keep object files; however, they are beyond the scope of what you are doing now. For now, you can get rid of your object files once you have your executable file. If you need the object file, you can recompile the source file.

*** Question 3**
If my compiler came with an editor, do I have to use it?
*** Answer**
Definitely not. You can use any editor as long as it saves the source code in text format. If the compiler came with an editor, you should try to use it. If you like a different editor better, use it. The editors that are coming with compilers are getting better. Some of them automatically format your C code. Others color-code different parts of your source file to make it easier to find errors.

*** Question 4**
Can I ignore warning messages?
*** Answer**
Some warning messages don't affect how the program runs. Some do. If the compiler gives you a warning message, it's a signal that something is not quite right. Most compilers let you set the warning level. By setting a warning level, you can get only the most serious warnings, or you can get all the warnings including the most minute. Some compilers even give various levels in between. In your programs, you should look at each warning and make a determination. It is always best to try to write all your programs with absolutely no warnings or errors. (With an error, your compiler won't create the executable file.)

Topic 1.9: Day 1 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

Topic 1.10: Day 1 Try Its

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Use your text editor to look at the object file created by Listing 1.1. Does the object file look like the source file? (Don't save the file when you exit the editor.)
Answer

Listing 1.1: HELLO.C

```
Code   1: /* LIST0101.c: Day 1 Listing 1.1 */
       2: /* HELLO.C: Displays the words Hello, World! */
       3: /* on the screen. */
       4:
       5: #include <stdio.h>
       6:
       7: int main(void){
       8:   printf("Hello, World!");
       9:   return 0;
      10: }
```

When you look at the object file, you see a lot of control characters and other gibberish. Throughout the gibberish you also see pieces of the source file.

*** Exercise 2**
Enter the following program and compile it. What does this program do?

```
#include <stdio.h>
int radius, area;

int main(void){
```

```
int main(void){
  printf("Enter radius (i.e. 10): ");
  scanf("%d", &radius);
  area = 3.14159 * radius * radius;
  printf("\n\nArea = %d", area);
  return 0;
}
```

Answer

The program calculates the area of a circle. It prompts the user for the radius and then displays the area.
Note that the program may cause a compiler error that says "Warning, assignment to int from double." Basically the area of a circle is usually a number with a very fine decimal, and we are forcing it to be a regular number (int). Therefore, we are cutting off some of the precision of the number. You will learn more about this later.

*** Exercise 3**

Enter and compile the following program. What does this program do?

```
#include <stdio.h>
int x,y;
int main(void) {
  for (x = 0; x < 10; x++, printf("\n"))
    for (y=0; y<10; y++)
      printf("X");

  return 0;
}
```

Answer

This program uses a loop to print a 10-by-10 block made of the character X. A similar program is used and explained on Day 6, "Basic Program Control."

*** Exercise 4**

Make the following changes to the `printf` statements in the program in exercise three. Recompile and rerun this program. What does the program now do?

```
    printf( "%c", 1 );
```

Answer

Rather than a 10-by-10 block filled with the character X, the program now prints a 10-by-10 block of smiley faces.

*** Exercise 5**

Enter and compile the following program. This program can be used to print your source code

listings on the screen. If you get any errors, make sure you entered the program correctly

listings on the screen. If you get any errors, make sure you entered the program correctly.
The usage for this program is PRINT_IT filename.ext, where filename.ext is the source filename along with the extension. Note that this program adds line numbers to the listing. (Don't let this program's length worry you; you're not expected to understand it yet. It's included here to help you compare printouts of your programs with the ones given in the course.) It can also be used to print any text file with line numbers.
PRINT_IT.C

Answer

```
/* PRINT_IT.C -- This program prints a listing
 * with line numbers added.
 * MS-DOS Application Only
 * Non-ANSI Application */
#include <stdio.h>
#include <stdlib.h>
void do_heading(char *filename);
int line,page;
main(int argv, char *argc[]) {
  char buffer[256];
  FILE *fp;

  if(argv < 2) {
    fprintf(stderr, "\nProper Usage is: ");
    fprintf(stderr, "\n\nPRINT_IT filename.ext\n");
    exit(1);
  }

  if ((fp = fopen(argc[1], "r")) == NULL) {
    fprintf(stderr, "Error opening file, %s!", argc[1]);
    exit(1);
  }

  page = 0;
  line = 1;
  do_heading(argc[1]);

  while(fgets(buffer, 256, fp) != NULL) {
    if(line % 55 == 0)
      do_heading(argc[1]);

    fprintf(stdout, "%4d:\t%s", line++, buffer);
  }

  fprintf(stdout, "\f");
  fclose(fp);
  return 0;
}
void do_heading(char *filename) {
  page++;

  if (page > 1)
      fprintf(stdout, "\f");

  fprintf(stdout, "Page: %d, %s\n\n", page, filename);
}
```

This exercise had you enter a program that can be used to print listings. There is no applicable answer.

Topic 1.11: Day 1 Summary

Why Use C?
After taking this unit, you should feel confident that selecting C as your programming language is a wise choice. C offers an unparalleled combination of power, popularity, and portability. These factors, together with C's close relationship to the new C++ object-oriented language, make C unbeatable.
The Program Development Cycle
This unit has explained the various steps involved in writing a C program—the process known as program development. You should have a clear grasp of the edit-compile-link-test cycle, as well as the tools to use for each step.

Compilation Errors
Errors are an unavoidable part of program development. Your C compiler detects errors in your source code and displays an error message, giving both the nature and the location of the error. Using this information, you can edit your source code to correct the error. Remember, however, that the compiler cannot always report accurately the nature and location of an error. Sometimes you need to use your knowledge of C to track down exactly what is causing a given error message.

Unit 2. Day 2 The Components of a C Program

[**Skip Unit 2's navigation links**]
2. Day 2 The Components of a C Program
- 2.1 A Short C Program
- 2.2 The Parts of a Program: Review
- 2.3 Day 2 Q&A
- 2.4 Day 2 Think About Its
- 2.5 Day 2 Try Its
- 2.6 Day 2 Summary

Every C program consists of several components combined in a certain way. Most of this course is devoted to explaining these various program components and how you use them. To get the overall picture, however, you should begin by seeing a complete (though small) C program with all its components identified.

Today, you learn

- A short C program with its components identified.
- The purpose of each program component.

The purpose of each program component.

- To compile and run a sample program.

Topic 2.1: A Short C Program

*** A Sample Program**

Listing 2.1 presents the source code for MULTIPLY.C. This is a very simple program that inputs two numbers from the keyboard and calculates their product. For now, don't worry about understanding the program's details. Follow the links in the Description section of the listing in order to gain some familiarity with the parts of a C program. These descriptions identify the function and purpose of each element of the sample program. Click the Listing button below to see the program.

Listing 2.1: MULTIPLY.C

Code

```
 1: /* LIST0201.c: Day 2 Listing 2.1
 2:  * MULTIPLY.C: Program to calculate the product
 3:  * of two numbers. */
 4: #include <stdio.h>
 5: int a, b, c;
 6: int product(int x, int y);
 7: int main(void) {
 8:   /* Input the first number. */
 9:   printf("Enter a number between 1 and 100: ");
10:   scanf("%d", &a);
11:
12:   /* Input the second number. */
13:   printf("Enter another number"
14:          " between 1 and 100: ");
15:   scanf("%d", &b);
16:
17:   /* Calculate and display the product. */
18:   c = product(a, b);
19:   printf("\n%d times %d = %d", a, b, c);
20:   return 0;
21: }
22: /* Function returns the product of its two
23:  * arguments. */
24: int product(int x, int y) {
25:   return (x * y);
26: }
27:
28:
29:
30:
```

Output

```
Enter a number between 1 and 100: 35
Enter another number between 1 and 100: 23
35 times 23 = 805
```

Description

The following paragraphs describe the various components of Listing 2.1.

Line numbers are included, so you can easily identify the program parts

	being discussed. The main() Function (Lines 10 – 24) #include Directive (Line 5) Variable Definition (Line 7) Function Prototype (Line 8) Program Statements (Lines 12-13, 16-18, 21-23, 29) Function Definition (Lines 28 – 30) Program Comments (Lines 1-3, 11, 15, 20, 26-27) Braces (Lines 10, 24, 28, 30)

*** Functions**
Before you'll fully understand the sample program, you need to know what a function is, because functions are central to C programming. A *function* is an independent section of program code that performs a certain task and has been assigned a name. By referencing a function's name, your program can execute the code in the function.
*** Arguments**
The program also can send information, called *arguments,* to the function, and the function can return information to the program.

*** Two Types of C Functions**
The two types of C functions are *library functions,* which are a part of the C compiler package, and *user-defined functions,* which you, the programmer, create. You'll learn about both types of functions.

DO add abundant comments to your program's source code, especially near statements or functions that could be unclear to you or to someone who might have to modify it later.
DON'T add unnecessary comments to statements that are already clear. For example, typing

```
/* The following prints Hello World! on the screen */
printf("Hello World!);
```

may be going a little too far, at least once you're completely comfortable with the printf() function and how it works.
DO learn to develop a style that will be helpful. A style that's too lean or cryptic doesn't help, nor does one that's so verbose you're spending more time commenting than programming!

*** Running the Program**
Take the time to enter, compile, and run MULTIPLY.C now. It provides additional practice using your editor and compiler. Recall these steps from Day 1, "Getting Started."
Steps

Step	Action
1	Make your programming directory current.
2	Start your editor.

3	Enter the source code for MULTIPLY.C exactly as shown in Listing 2.1, but be sure to omit the line numbers.
4	Save the program file.
5	Compile and link the program by entering the appropriate command(s) for your compiler. If no error messages are displayed, you can run the program by entering multiply at the command prompt.
6	If one or more error messages are displayed, return to step two and correct the errors.

*** A Note on Accuracy**

A computer is fast and accurate, but it is also completely literal. It doesn't know enough to correct your simplest mistake; it takes everything you enter exactly as you entered it, and not as you meant it!

This goes for your C source code as well. A simple typographical error in your program can cause the C compiler to choke, gag, and collapse. Fortunately, although the compiler is not smart enough to correct your errors (and you'll make errors—everyone does!), it's smart enough to recognize them as errors and report them to you. How the compiler reports error messages and how you interpret them is covered on Day 1, "Getting Started."

You must also be aware of runtime errors. Runtime errors are those which will still compile the code (the syntax is correct), but when run, don't perform as expected. These are usually due to errors in program logic, but can also be due to unfortunate typos.

Example:

Should

```
if (a = b)
...
```

really be

```
if (a == b)
...
```

?

Both will compile but they can give totally different results.

Topic 2.2: The Parts of a Program: Review

*** Another Sample Program**

Now that all the parts of a program have been described, you should be able to look at any program and find some similarities. Look at Listing 2.2, LIST_IT.C. and see whether you can identify the different parts.

Listing 2.2: LIST_IT.C

```
Code  1: /* LIST0202.c: Day 2 Listing 2.2
      2:  * LIST_IT.C -- This program displays a listing
      3:  * with line numbers! */
      4:
      5: #include <stdio.h>
```

```
 6: #include <stdlib.h>
 7:
 8: void display_usage(void);
 9:
10: int line;
11:
12: main(int argv, char *argc[]) {
13:   char buffer[256];
14:   FILE *fp;
15:
16:   if(argv < 2) {
17:     display_usage();
18:     exit(1);
19:   }
20:
21:   if ((fp = fopen(argc[1], "r")) == NULL) {
22:     fprintf(stderr, "Error opening file, %s!",
23:             argc[1]);
24:     exit(1);
25:   }
26:
27:   line = 1;
28:
29:   while(fgets(buffer, 256, fp) != NULL)
30:     fprintf(stdout, "%4d:\t%s", line++, buffer);
31:
32:   fclose(fp);
33:   return 0;
34: }
35:
36: void display_usage(void) {
37:     fprintf(stderr, "\nProper Usage is: ");
38:     fprintf(stderr, "\n\nLIST_IT filename.ext\n");
39: }
```

Output

```
1:/* LIST0202.c: Day 2 Listing 2.2
2: * LIST_IT.C - This program displays a listing
3: * with line numbers! */
4:
5:#include <stdio.h>
6:#include <stdlib.h>
7:
8:void display_usage(void);
9:
10:int line;
11:
12:main(int argv, char *argc[]) {
13:  char buffer[256];
14:  FILE *fp;
15:
16:  if(argv < 2) {
17:    display_usage();
18:    exit(1);
19:  }
20:
21:  if ((fp = fopen(argc[1], "r")) == NULL) {
22:    fprintf(stderr, "Error opening file, %s!",
23:            argc[1]);
24:    exit(1);
25:  }
26:
```

```
27:  line = 1;
28:
29:  while(fgets(buffer, 256, fp) != NULL)
30:    fprintf(stdout, "%4d:\t%s", line++, buffer);
31:
32:  fclose(fp);
33:  return 0;
34:}
35:
36:void display_usage(void) {
37:  fprintf(stderr, "\nProper Usage is: ");
38:  fprintf(stderr, "\n\nLIST_IT filename.ext\n");
39:}
```

Description

LIST_IT.C is very similar to PRINT_IT.C, which you entered in exercise five of Day 1, "Getting Started."

Looking at the listing, you can summarize where the different parts are.

The required main() function is lines 12 – 34.

In lines 5 and 6, you have #include directives.

Lines 10, 13, and 14 have variable definitions.

A function prototype, `void display_usage(void)`, is in line 8.

This program has many statements (lines 16 – 18, 21 – 24, 27, 29 – 33, 37, and 38).

A function definition for display_usage() fills lines 36 – 39.

Braces enclose blocks throughout the program.

Finally, only lines 1 – 3 have comments. In most programs, you should probably include more comment lines!

LIST_IT.C calls many functions. It calls only one user-defined function, display_usage(). The library functions that it uses are:

exit() in lines 18 and 24
fopen() in line 21
printf() in lines 22, 30, 37, and 38
fgets() in line 29
fclose() in line 32.
These library functions are covered in more detail throughout this course.

Topic 2.3: Day 2 Q&A

Topic 2.3: Day 2 Q&A

*** Questions & Answers**
Take a look at some questions that are frequently asked by programmers new to C.

*** Question 1**
What effect do comments have on a program?
*** Answer**
Comments are for the programmer. When the compiler converts the source code to object code, it throws the comments and the whitespace away. This means that they have no effect on the executable program. Comments do make your source file bigger, but this is usually of little concern. To summarize, you should use comments and whitespace to make your source code as easy to understand and to maintain as possible.

*** Question 2**
What is the difference between a statement and a block?
*** Answer**
A block is a group of statements enclosed within braces (`{}`). A block can be used in most places that a statement can be used.

*** Question 3**
How can I find out what library functions are available?
*** Answer**
Many compilers come with a manual dedicated specifically to documenting the library functions. They are usually in alphabetical order. Another way to find out what library functions there are is to buy a book that lists them. After you begin to understand more of C, it is a good idea to review this material so that you don't waste energy writing a function that performs the same task as an existing library function. (No use in reinventing the wheel!)

Topic 2.4: Day 2 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
1:  /* EXER0202.c: Day 2 Exercise 2.2 */
2:
3:  #include <stdio.h>
4:
5:  void display_line(void);
6:
7:  int main(void) {
8:    display_line();
9:    printf("\n C User Guide\n");
10:   display_line();
11:   return 0;
12: }
13:
14: /* Print asterisk line. */
15: void display_line(void) {
16:   int counter;
17:
18:   for (counter = 0; counter < 21; counter++)
19:     printf("*");
20: }
```

```
1:  /* EXER0202.c: Day 2 Exercise 2.2 */
2:
3:  #include <stdio.h>
4:
5:  void display_line(void);
6:
7:  int main(void) {
8:    display_line();
9:    printf("\n C User Guide\n");
10:   display_line();
11:   return 0;
12: }
13:
14: /* Print asterisk line. */
15: void display_line(void) {
16:   int counter;
17:
18:   for (counter = 0; counter < 21; counter++)
19:     printf("*");
20: }
```

```
1:  /* EXER0202.c: Day 2 Exercise 2.2 */
2:
3:  #include <stdio.h>
4:
5:  void display_line(void);
6:
7:  int main(void) {
8:    display_line();
9:    printf("\n C User Guide\n");

10:   display_line();
```

```
11:   return 0;
12: }
13:
14: /* Print asterisk line. */
15: void display_line(void) {
16:   int counter;
17:
18:   for (counter = 0; counter < 21; counter++)
19:     printf("*");
20: }
```

```
1:  /* EXER0202.c: Day 2 Exercise 2.2 */
2:
3:  #include <stdio.h>
4:
5:  void display_line(void);
6:
7:  int main(void) {
8:    display_line();
9:    printf("\n C User Guide\n");
10:   display_line();
11:   return 0;
12: }
13:
14: /* Print asterisk line. */
15: void display_line(void) {
16:   int counter;
17:
18:   for (counter = 0; counter < 21; counter++)
19:     printf("*");
20: }
```

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

Topic 2.5: Day 2 Try Its

*** Exercise 1**
Write the smallest program possible.
Answer

Remember, only the main() function is required in C programs. The following is the smallest possible program, but it doesn't do anything.

```
void main(){}
```

*** Exercise 2**

Write an example of a comment.

Answer

A comment is any text included between `/*` and `*/`. Examples include

```
/* This is a comment */

/*???*/

/*
 *This is a
 *third comment
 */
```

*** Exercise 3**

What does the following program do? (Enter, compile, and run it.)

```
/* EXER0204.c: Day 2 Exercise 2.4 */
#include <stdio.h>
int main(void) {
  int ctr;

  for (ctr = 65; ctr < 91; ctr++)
    printf("%c", ctr);
  return 0;
}
/* End of program. */
```

Answer

Exercise three is a program that prints the alphabet in all capital letters. You should understand this program better when you finish Day 10, "Characters and Strings."

The output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

*** Exercise 4**

What does the following program do? (Enter, compile, and run it.)

```
/* EXER0205.c: Day 2 Exercise 2.5 */
#include <stdio.h>
#include <string.h>
int main(void) {
  char buffer[256];

  printf("Enter your name and press <Enter>:\n");
  gets(buffer);

  printf("\nYour name has %d characters and spaces!",
          strlen(buffer));
  return 0;
}
```

Answer

This program counts and prints the number of characters and spaces that you enter. This program also will be clearer after you finish Day 10.

Program Components

This unit is short but it's important, because it introduces you to the major components of a C program. You learned that the single required part of every C program is the `main()` function. You also learned that the program's real work is done by program statements that instruct the computer to perform your desired actions. This unit also introduced you to variables and variable definitions, and it showed you how to use comments in your source code.

Functions

In addition to the `main()` function, a C program can make use of two types of subsidiary functions: library functions supplied as part of the compiler package and user-defined functions created by the programmer.

Unit 3. Day 3 Numeric Variables and Constants

[**Skip Unit 3's navigation links**]

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C has two ways of storing number values — *variables* and *constants* — with many options for each. A variable is a data storage location that has a value which can change during program execution. In contrast, a constant has a fixed value that cannot change.

Today, you learn

- How to create variable names in C.

- The use of different types of numeric variables.
- The differences and similarities between character and numeric values.
- How to declare and initialize numeric variables.
- C's two types of numeric constants.

Topic 3.1: Computer Memory

*** Optional Material**
If you already know how a computer's memory operates, you can skip this section. If you're not sure, however, please continue. This information will help you better understand certain aspects of C programming.

*** RAM**
A computer uses *random-access memory* (RAM) to store information while it is operating. RAM is located in integrated circuits, or *chips,* inside your computer. RAM is *volatile,* which means it's erased and replaced with new information as often as needed. Being volatile also means that RAM "remembers" only while the computer is turned on and loses its information when you turn off the computer.

*** Amount of RAM**
Each computer has a certain amount of RAM installed. The amount of RAM in a system is usually specified in kilobytes (K), or megabytes (MB). One kilobyte of memory consists of 1,024 bytes. Thus, a system with 640K of memory actually has 640 times 1,024, or 655,360 bytes of RAM. One megabyte is 1,024 kilobytes.

*** Bytes**
A *byte* is the fundamental unit of computer data storage. Day 20, "Odds and Ends," has more information about bytes. For now, though, to get an idea of how many bytes it takes to store certain kinds of data, you can click here.

Table 3.1: Memory Space Required to Store Data

Data	Bytes Required
The letter *x*	1
The number *100*	2
The number *120.145*	4
The phrase *Teach Yourself C*	17
One typewritten page	3,000 (approximately)

*** Addresses**
The RAM in your computer is organized sequentially, one byte following another. Each byte of

memory has a unique *address* by which it is identified, an address that also distinguishes it from all other bytes in memory. Addresses are assigned to memory locations in order, starting at 0 and increasing to the system limit. For now, you needn't worry about addresses; it's all handled automatically for you by the C compiler.

*** Usage of RAM**

What is your computer's RAM used for? It has several uses, but only one, data storage, need concern you as a programmer. *Data* means the information with which your C program works. Whether your program is maintaining an address list, monitoring the stock market, keeping a household budget, or tracking the price of hog bellies, the information (names, stock prices, expense amounts, or hog futures) is kept in your computer's RAM while the program is running. Now that you understand a little about the nuts and bolts of memory storage, you can get back to C programming and how C uses memory to store information.

Topic 3.2: Variables

*** Variables**

A *variable* is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

Topic 3.2.1: Variable Names

*** Rules for Naming Variables**

To use variables in your C programs, you must know how to create variable names. In C, variable names must adhere to the following rules:

The name can contain letters, digits, and the underscore character (_).

The first character of the name must be a letter. The underscore is also a legal first character, but its use is not advised.

Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables.

C keywords cannot be used as variable names. A keyword is a word that is part of the C language. Click the Examples link to see examples of variable names. Click the Tip button on the toolbar to see some tips about naming variables.

Examples

DO use variable names that are descriptive.
DO adopt and stick with a style for naming your variables.
DON'T start your variable names with an underscore unnecessarily.
DON'T name your variables with all capitals unnecessarily.

The following code contains some examples of legal and illegal C variable names:

```
percent           /* legal */
x2x5  fq7h        /* legal */
```

```
y2x5__tg7h        /* legal */
annual_profit     /* legal */
_1990_tax         /* legal but not advised */
savings#account   /* illegal: contains illegal character # */
double            /* illegal: is a C  keyword */
9winter           /* illegal: first character is a digit */
```

*** Using Lowercase in Names**

Because C is case-sensitive, the three names percent, PERCENT, and Percent are considered to refer to three distinct variables. C programmers commonly use only lowercase letters in variable names although it's not required. Uppercase letters are usually reserved for the names of constants (which are covered later in this unit).

*** Choosing Descriptive Names**

For many compilers, a C variable name can be up to 31 characters long. (It can actually be longer than that, but the compiler looks only at the first 31 characters of the name.) With this flexibility, you can create variable names that reflect the data being stored. Click the Example link for examples.

Example

A program that calculates loan payments could store the value of the prime interest rate in a variable named interest_rate. The variable name helps make its usage clear. You could as well have created a variable named x or even johnny_carson; it doesn't matter to the C compiler. The use of the variable, however, would not be nearly as clear to someone else looking at the source code. Although it may take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.

*** Naming Conventions**

Many naming conventions are used for variable names created from multiple words.

The Underscore

You've been shown one style: interest_rate. Using an underscore to separate words in a variable name makes it easy to interpret.

Camel Notation

The second style is called *camel notation*. Instead of using spaces, the first letter of each word is capitalized. Instead of interest_rate, the variable would be named InterestRate. Camel notation is gaining popularity because it is easier to type a capital letter than an underscore.

This Course's Convention

We use the underscore in this course because it is easier for most people to read. You should decide which style you wish to adopt.

Topic 3.2.2: Numeric Variable Types

*** The Need for Different Types of Variables**

C provides several different types of numeric variables. Why do you need different types of variables? Different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them. Click the Example link to see the C data types.

Example

Example

By using the appropriate *variable types* (also called *data types*), you ensure that your program runs as efficiently as possible. Click the Tip button on the toolbar for tips on using data types.

DO understand the number of bytes that variable types take for your computer.
DON'T use a float or double variable if you are only storing integers. Although they will work, using them is inefficient.
DON'T put negative numbers into variables with an unsigned type.

Small integer numbers (for example, 1, 199, –8) require less memory space for storage, and mathematical operations (addition, multiplication, and so on) with such numbers can be performed by your computer very quickly. In contrast, large integers and floating-point values (123,000,000 or 0.000000871256, for example) require more storage space and more time for mathematical operations.

*** Numeric Variable Categories**

C's numeric variables fall into the following two main categories:

Integer variables hold values that have no fractional part (that is, whole numbers only). Integer variables come in two flavors: signed integer variables can hold positive or negative values, whereas unsigned integer variables can hold only positive values (and 0, of course).

Floating-point variables hold values that have a fractional part (that is, real numbers).

Within each of these categories are two or more specific variable types. These are summarized here, which also shows the amount of memory, in bytes, required to hold a single variable of each type when you use a microcomputer with 32-bit architecture.

Table 3.2: C's Numeric Data Types (on a 32-bit machine)

Variable Type	Keyword	Bytes Required	Range
character	char	1	–128 to 127
integer	int	4	–2,147,483,648 to 2,147,483,647
short integer	short	2	–32768 to 32767
long integer	long	4	–2,147,483,648 to 2,147,483,647
unsigned character	unsigned char	1	0 to 255
unsigned integer	unsigned int	4	0 to 4,294,967,295
unsigned short	unsigned	2	0 to 65535

integer	short		
unsigned long integer	unsigned long	4	0 to 4,294,967,295
single-precision floating point	float	4	1.2E–38 to 3.4E38 Approximate range; precision = 7 digits.
double-precision floating-point	double	8	2.2E–308 to 1.8E308 Approximate range; precision = 19 digits.

*** Approximate Range**

Approximate range means the highest and lowest values a given variable can hold. (Space limitations prohibit listing exact ranges for the values of these variables.)

*** Precision**

Precision means the accuracy with which the variable is stored. (For example, if you evaluate 1/3, the answer is 0.33333 . . . with 3s going to infinity. A variable with a precision of 7 stores seven 3s.)

*** int Versus long**

Looking at Table 3.2, you may notice that the variable types int and long are identical. Why then have two different types? The int and long variable types are indeed identical on 32-bit IBM PC-compatible systems, but they may be different on other types of hardware. On a 16-bit MS-DOS system, a long and an int are not the same size. Instead, a long is 4 bytes, whereas an int is 2. Remember that C is a flexible, portable language, so it provides different keywords for the two types.

*** Integer Variables: Signed by Default**

No special keyword is needed to make an integer variable signed; integer variables are signed by default. You can, however, include the signed keyword if you wish. The keywords in Table 3.2 are used in variable declarations, discussed in the next section of this unit.

Listing 3.1 will help you determine the size of variables on your particular computer. Click the Listing button on the toolbar.

Listing 3.1: SIZEOF.C

```
Code  1: /* LIST0301.c: Day 3 Listing 3.1 */
      2: /* SIZEOF.C: Program to tell the size of the C */
      3: /* variable type in bytes */
      4:
      5: #include <stdio.h>
      6:
      7: int main(void) {
      8:   printf("\nA char      is %d bytes",
      9:          sizeof(char));
     10:   printf("\nAn int      is %d bytes",
     11:          sizeof(int));
     12:   printf("\nA short     is %d bytes",
     13:          sizeof(short));
     14:   printf("\nA long      is %d bytes",
```

```
15:           sizeof(long));
16:   printf("\nAn unsigned char      is %d bytes",
17:           sizeof(unsigned char));
18:   printf("\nAn unsigned int       is %d bytes",
19:           sizeof(unsigned int));
20:   printf("\nAn unsigned short     is %d bytes",
21:           sizeof(unsigned short));
22:   printf("\nAn unsigned long      is %d bytes",
23:           sizeof(unsigned long));
24:   printf("\nA float     is %d bytes",
25:           sizeof(float));
26:   printf("\nA double    is %d bytes",
27:           sizeof(double));
28:   return 0;
29: }
```

Output

```
A char      is 1 bytes
An int      is 4 bytes
A short     is 2 bytes
A long      is 4 bytes
An unsigned char  is 1 bytes
An unsigned int   is 4 bytes
An unsigned short is 2 bytes
An unsigned long  is 4 bytes
A float     is 4 bytes
A double    is 8 bytes
```

Description

The output of Listing 3.1 tells you exactly how many bytes each variable type on your computer takes. If you are using a 16-bit PC, your numbers should match those in Table 3.2.

Don't worry about trying to understand all the individual components of the program. Although some items are new, such as sizeof(), others should look familiar.

Lines 1 – 3 are comments about the name of the program and a brief description.

Line 5 includes the standard input/output header file to help print the information on the screen.

This is a simple program, in that it contains only a single function, main() (lines 7 – 29).

Lines 8 – 27 are the bulk of the program. Each of these lines prints a textual description with the size of each of the variable types, which is done using the sizeof operator. Day 19, "Exploring the Function Library," covers the sizeof operator in detail.

Line 28 of the program returns the value of 0 to the operating system before ending the program.

*** The Size of Variables**

*** The Size of Variables**

C does make some guarantees, thanks to the ANSI Standard. There are five things that can be counted on:

- The size of a char is 1 byte.
- The size of a short is less than or equal to the size of an int.
- The size of an int is less than or equal to the size of a long.
- The size of an unsigned int is equal to the size of an int.
- The size of a float is less than or equal to the size of a double.

Topic 3.2.3: Variable Declarations

*** Variable Declarations**

Before you can use a variable in a C program, it must be declared. A *variable declaration* informs the compiler of the name and type of a variable and optionally initializes the variable to a specific value. If your program attempts to use a variable that has not been declared, the compiler generates an error message.

*** Syntax**

A variable declaration has the following form:

typename varname;

typename specifies the variable type and must be one of the keywords given in Table 3.2. *varname* is the variable name, which must follow the rules mentioned earlier. You can declare multiple variables of the same type on one line by separating the variable names with commas.

```
int count, number, start;
/* three integer variables */
float percent, total;
/* two float variables */
```

*** Location**

On Day 12, "Variable Scope," you will learn that the location of variable declarations in the source code is important, because it affects the ways in which your program can use the variables. For now, you can place all the variable declarations together just before the start of the `main()` function.

Topic 3.2.4: The typedef Keyword

*** Usage**

The `typedef` keyword is used to create a new name for an existing data type. In effect, `typedef` creates a synonym.

Example

Note that `typedef` does not create a new data type, but only enables you to use a different name for a predefined data type. The most common use for `typedef` concerns *aggregate data types,* as explained on Day 11, "Structures." An aggregate data type consists of a combination of data types

presented in this unit.

DO use typedef to make your programs more readable.

The statement
`typedef int integer;`
creates `integer` as a synonym for `int`. You can then use `integer` to define variables of type int.
`integer count;`

Topic 3.2.5: Initializing Numeric Variables

*** Initialization Using Simple Assignment**
When you declare a variable, you instruct the compiler to set aside storage space for the variable. However, the value stored in that space—the value of the variable—is not defined. It may be zero, or some random "garbage" value. Before using a variable, you should always initialize it to a known value. This can be done independently of the variable declaration by using an assignment statement.

```
/* Set aside storage space for count */
int count;
/* Store 0 in count */
count = 0;
```

DO initialize variables when you declare them whenever possible.
DON'T use a variable that has not been initialized. Results can be unpredictable!
DON'T try to put numbers into variable types that are too small to hold them!

*** The Assignment Operator**
Note that this statement uses the equal sign (=), which is C's assignment operator and is discussed further on Day 4, "Statements, Expressions, and Operators." For now, you need to be aware that the equal sign in programming is not the same as the equal sign in algebra. If you write `x = 12` in an algebraic statement, you are stating a fact: "*x* equals 12." In C, however, it means something quite different: "Assign the value 12 to the variable named x."

*** Initialization within the Declaration**
You also can initialize a variable when it is declared. To do so, follow the variable name in the declaration statement with an equal sign and the desired initial value.

```
int count = 0;
double percent = 0.01, taxrate = 28.5;
```

*** Out-of-Range Initializations**
Be careful not to initialize a variable with a value outside the allowed range.
Examples
The C compiler does not catch such errors. Your program may compile and link, but you may get unexpected results when the program is run.

Here are some examples of out-of-range initializations:

```
int weight = 100000;
unsigned int value = -2500;
```

Topic 3.3: Constants

*** Constants**

Like a variable, a *constant* is a data storage location used by your program. Unlike a variable, the value stored in a constant cannot be changed during program execution. C has two types of constants, each with its own specific uses.

Topic 3.3.1: Literal Constants

*** Literal Constants**

A *literal constant* is a value that is typed directly into the source code wherever it is needed.
Examples

*** Floating-Point Constants: Decimal Notation**

A literal constant written with a decimal point is a *floating-point constant* and is represented by the C compiler as a double-precision number. Floating-point constants can be written in standard decimal notation, as shown in these examples:
Examples

Here are two examples of literal constants being assigned to variables:

```
int count = 20;
float tax_rate = 0.28;
```

The 20 and the 0.28 are literal constants. The preceding statements store these values in the variables `count` and `tax_rate`. Note that one of these constants contains a decimal point whereas the other does not. The presence or absence of the decimal point distinguishes floating-point constants from integer constants.

```
123.456
0.019
100.
```

Note that the third constant, 100., is written with a decimal point even though it is an integer (that is, it has no fractional part). The decimal point causes the C compiler to treat the constant as a double-precision value. Without the decimal point, it is treated as an integer constant.

*** Floating-Point Constants: Scientific Notation**

Floating-point constants also can be written in *scientific notation*. You may recall from high school math that scientific notation represents a number as a decimal part multiplied by 10 to a positive or

negative power. Scientific notation is particularly useful for representing extremely large and extremely small values. In C, scientific notation is written as a decimal number followed

extremely small values. In C, scientific notation is written as a decimal number followed immediately by an E or e and the exponent. Click the Examples link to see examples of scientific notation.

Examples

`1.23E2`	1.23 times 10 to the 2nd power, or 123
`4.08e6`	4.08 times 10 to the 6th power, or 4,080,000
`0.85e–4`	0.85 times 10 to the –4 power, or 0.000085

*** Integer Constants**

A constant written without a decimal point is represented by the compiler as an integer number. Integer constants can be written in three different notations:

A constant starting with any digit other than 0 is interpreted as a *decimal* integer (that is, the standard base-10 number system). Decimal constants can contain the digits 0 – 9 and a leading minus or plus sign. (Without a leading sign, a constant is assumed to be positive.)

A constant starting with the digit 0 is interpreted as an *octal* integer (the base 8 number system). Octal constants can contain the digits 0 – 7 and a leading minus or plus sign.

A constant starting with 0x or 0X is interpreted as a *hexadecimal* constant (the base-16 number system). Hexadecimal constants can contain the digits 0 – 9, the letters A – F, and a leading minus or plus sign.

Topic 3.3.2: Symbolic Constants

*** Symbolic Constants**

A *symbolic constant*, also called a named constant, is a constant that is represented by a name (symbol) in your program. Like a literal constant, a symbolic constant cannot change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it is first defined.

*** Advantages over Literal Constants**

Symbolic constants have two significant advantages over literal constants: they are easier to read and easier to change. Let's see why.

*** Using a Literal Constant**

Suppose that you are writing a program which performs a variety of geometrical calculations. The program frequently needs the value pi (3.14) for its calculations. (You may recall from geometry class that pi is the ratio of a circle's circumference to its diameter.) For example, to calculate the circumference and area of a circle with a known radius you could write

```
circumference = 3.14 * (2 * radius);
area = 3.14 * (radius)*(radius);
```

The asterisk (*) is C's multiplication operator. Thus, the first statement means "Multiply 2 times the value stored in the variable `radius`, and then multiply the result times 3.14. Finally, assign the result to the variable named `circumference`."

*** Using a Symbolic Constant Is Easier to Read**

If, however, you define a symbolic constant with the name `PI` and the value 3.14, you could write

If, however, you define a symbolic constant with the name PI and the value 3.14, you could write

```
circumference = PI * (2 * radius);
area = PI * (radius)*(radius);
```

The resulting code is clearer. Rather than puzzling over what the value 3.14 is for, you can see immediately that the constant `PI` is being used.

*** Easier to Change**

The second advantage of symbolic constants becomes apparent when you need to change a constant. Continuing the preceding example, you may decide that for greater accuracy your program needs to use a value of `PI` with more decimal places: 3.14159 rather than 3.14. If you had used literal constants for `PI`, you would have to go through your source code and change each occurrence of the value from 3.14 to 3.14159. With a symbolic constant, you would need to make a change only in the place where the constant is defined.

DO use constants to make your programs easier to read.
DON'T try to assign a value to a constant after it has already been initialized.

*** Defining a Symbolic Constant**

C has two methods for defining a symbolic constant, the `#define` directive and the `const` keyword.

*** The Syntax of `#define`**

The `#define` directive is one of C's preprocessor directives, discussed fully on Day 21, "Taking Advantage of Preprocessor Directives and More." The `#define` directive is used as follows:

```
#define CONSTNAME literal
```

This program line creates a constant named `CONSTNAME` with the value of `literal`. `literal` represents a numeric constant, as described earlier in this unit. `CONSTNAME` follows the same rules described for variable names earlier in this unit. By convention, the names of symbolic constants are uppercase. This makes them easy to distinguish from variable names, which by convention are lowercase. For the previous example, the required `#define` directive would be

```
#define PI 3.14159
```

Note that `#define` lines do not end with a semicolon (;).

*** The Placement of `#define`**

`#defines` can be placed anywhere in your source code, but are in effect only for the portions of the source code that follow the `#define` directive. Most commonly, programmers group all `#defines` together, near the beginning of the file and before the start of `main()`.

*** The Effect of `#define`**

The precise action of the `#define` directive is to instruct the compiler, "In the source code, replace `CONSTNAME` with `literal`." The effect is exactly the same as if you had used your editor to go through the source code and make the changes manually. Note that `#define` does not replace instances of its target that occur as parts of longer names, within double quotes, or as part of a program comment.

```
#define PI 3.14
/* You have defined a constant for PI. */ not changed
#define PIPETTE 100                       not changed
```

*** Description of `const`**

The second way to define a symbolic constant is with the `const` keyword. `const` is a modifier that can be applied to any variable declaration. A variable declared to be `const` can't be modified during

can be applied to any variable declaration. A variable declared to be const can't be modified during program execution—only initialized at the time of declaration.

*** Examples of `const`**

Here are some examples:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

`const` affects all variables on the declaration line. In the last example, `debt` and `tax_rate` are symbolic constants. If your program tries to modify a `const` variable, the compiler generates an error message. For example,

```
const int count = 100;
count = 200;   /* Does not compile! Cannot reassign or alter
                  the value of a constant. */
```

*** `#define` Versus `const`**

What are the practical differences between symbolic constants created with the `#define` directive and those created with the `const` keyword? The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming and are covered on Days 9 and 12, "Pointers" and "Variable Scope," respectively.

*** Example Program**

Look now at a program that demonstrates variable declarations and the use of literal and symbolic constants. The code in Listing 3.2 prompts the user to input his or her weight and year of birth. It then calculates and displays a user's weight in grams and his or her age in the year 2000. You can enter, compile, and run this program using the procedures explained on Day 1, "Getting Started." Click the Listing button on the toolbar to see the program.

Listing 3.2: The Use of Variables and Constants

```
Code  1: /* LIST0302.c: Day 3 Listing 3.2 */
      2: /* Demonstrates variables and constants. */
      3:
      4: #include <stdio.h>
      5: /* Define a constant to convert from pounds to */
      6: /* grams. */
      7: #define GRAMS_PER_POUND 454
      8: /* Define a constant for the start of the next */
      9: /* century. */
     10: const int NEXT_CENTURY = 2000;
     11: /* Declare the needed variables. */
     12: long weight_in_grams, weight_in_pounds;
     13: int year_of_birth, age_in_2000;
     14:
     15: int main(void) {
     16:   /* Input data from user. */
     17:
     18:   printf("Enter your weight in pounds: ");
     19:   scanf("%d", &weight_in_pounds);
     20:   printf("Enter your year of birth: ");
     21:   scanf("%d", &year_of_birth);
     22:
     23:   /* Perform conversions. */
     24:

     25:   weight_in_grams =
     26:     weight_in_pounds * GRAMS_PER_POUND;
```

```
27:   age_in_2000 = NEXT_CENTURY - year_of_birth;
28:
29:   /* Display results on the screen. */
30:
31:   printf("\nYour weight in grams = %ld",
32:           weight_in_grams);
33:   printf("\nIn 2000 you will be %d years old",
34:           age_in_2000);
35:   return 0;
36: }
```

Output

```
Enter your weight in pounds: 175
Enter your year of birth: 1960

Your weight in grams = 79450
In 2000 you will be 40 years old
```

Description

The program declares the two types of symbolic constants in lines 7 and 10. In line 7, a constant is being used to make the value 454 more understandable. Because it uses GRAMS_PER_POUND, lines 25 – 26 are easy to understand.

Lines 12 and 13 declare the variables used in the program. Notice the use of descriptive names such as weight_in_grams. By reading its name, you see what this variable is used for.

Lines 18 and 20 print prompts on the screen. The printf() function is covered in greater detail later.

To allow the user to respond to the prompts, lines 19 and 21 use another library function scanf() that is covered later. scanf() gets information from the screen. For now, accept that this works as shown in the listing. Later, you will learn exactly how it works.

Lines 25 – 27 calculate the user's weight in grams and his or her age in the year 2000. Those statements and others are covered in detail tomorrow.

To finish the program, lines 31 – 34 display the results for the user.

Topic 3.4: Day 3 Q&A

*** Questions & Answers**
Take a look at some questions that are frequently asked by programmers new to C.

*** Question 1**
`long int` variables hold bigger numbers, so why not always use them instead of `int` variables?
*** Answer**
A `long int` variable takes up more RAM than the smaller `int`. In smaller programs, this doesn't pose a problem. As programs get bigger, however, try to be efficient with the memory you use.

*** Question 2**
What happens if I assign a number with a decimal to an integer?
*** Answer**
You can assign a number with a decimal to an `int` variable. If you are using a constant variable, your compiler probably will give you a warning. The value assigned will have the decimal portion truncated. For example, if you assign 3.14 to an integer variable called `pi`, `pi` will only contain 3. The .14 will be chopped off and thrown away.

*** Question 3**
What happens if I put a number into a type that is not big enough to hold it?
*** Answer**
Answer Many compilers will allow this without signaling any errors. The number is wrapped to fit, however, and it isn't correct. For example, if you assign 2,147,483,648 to a four-byte signed integer, the integer really contains the value -2,147,483,648. If you assign the value 4,294,967,295 to this integer, it also really contains the value -1. Subtracting the maximum value the field will hold generally gives you the value that will be stored.

*** Question 4**
What happens if I put a negative number into an unsigned variable?
*** Answer**
As the previous answer indicated, your compiler may not signal any errors if you do this. The compiler does the same wrapping as if you assigned a number that was too big. For instance, if you assign –1 to an int variable that is two bytes long, the compiler will put the highest number possible in the variable (65535).

*** Question 5**
What are the practical differences between symbolic constants created with the `#define` directive and those created with the `const` keyword?
*** Answer**
The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming and are covered on Days 9 and 12, "Pointers," and "Variable Scope," respectively. For now, know that by using `#define` to create constants, you can make your programs much easier to read.

Topic 3.5: Day 3 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

Table 3.2: C's Numeric Data Types (on a 32-bit machine)			
Variable Type	**Keyword**	**Bytes Required**	**Range**
character	char	1	–128 to 127
integer	int	4	–2,147,483,648 to 2,147,483,647
short integer	short	2	–32768 to 32767
long integer	long	4	–2,147,483,648 to 2,147,483,648
unsigned character	unsigned char	1	0 to 255
unsigned integer	unsigned int	4	0 to 4,294,967,295
unsigned short integer	unsigned short	2	0 to 65535
unsigned long integer	unsigned long	4	0 to 4,294,967,295
single-precision floating point	float	4	1.2E–38 to 3.4E38 Approximate range; precision = 7 digits.
double-precision floating-point	double	8	2.2E–308 to 1.8E308 Approximate range; precision = 19 digits.

Topic 3.6: Day 3 Try Its

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
In what variable type would you best store the following values?

a. A person's weight in pounds.
b. Your annual salary.
c. The temperature.
d. A person's net worth.

Answer

a. unsigned int
b. If your expectations on yearly salary are not very high, a simple unsigned int variable would work. If you feel you have potential to go above $65,535, you probably should use a long. (Have faith in yourself; use a long.)
c. float (If you are only going to use whole numbers, use either int or long.)
d. Definitely a signed field. Either int, long, or float. See answer for 1b.

*** Exercise 2**
Determine appropriate variable names for the values in exercise one.
Answer

(Answers for 2 and 3 are combined)
Remember, a variable name should be representative of the value it holds. A variable declaration is the statement that initially creates the variable. The declaration may or may not initialize the variable to a value. You can use any name for a variable, except the C keywords.

a. `unsigned int age;`
b. `unsigned int weight;`
c. `float radius = 3;`
d. `long annual_salary;`
e. `float cost = 29.95;`
f. `const int max_grade = 100;`
or
`#define MAX_GRADE 100`
g. `float temperature;`
h. `long net_worth = -30000;`
i. `double star_distance;`

*** Exercise 3**
Write declarations for the variables in exercise two.
Answer

(Answers for 2 and 3 are combined)
Remember, a variable name should be representative of the value it holds. A variable declaration is the statement that initially creates the variable. The declaration may or may not initialize the variable to a value. You can use any name for a variable, except the C keywords.
a. `unsigned int age;`
b. `unsigned int weight;`
c. `float radius = 3;`
d. `long annual_salary;`
e. `float cost = 29.95;`
f. `const int max_grade = 100;`
or
```
#define MAX_GRADE 100
```
g. `float temperature;`
h. `long net_worth = -30000;`
i. `double star_distance;`

Topic 3.7: Day 3 Summary

Numeric Variables
This unit has explored numeric variables, which are used by a C program to store data during program execution. You've seen that there are two broad classes of numeric variables, integer and floating point. Within each class are specific variable types. Which variable type—`int`, `long`, `float`, or `double`—you use for a specific application depends on the nature of the data to be stored in the variable. You've also seen that in a C program, you must declare a variable before it can be used. A variable declaration informs the compiler of the name and type of a variable.
Constants
This unit has also covered C's two constant types, literal and symbolic. Unlike variables, the value of a constant cannot change during program execution. You type literal constants into your source code whenever the value is needed. Symbolic constants are assigned a name that is used wherever the constant value is needed. Symbolic constants can be created with the `#define` directive or with the `const` keyword.

Unit 4. Day 4 Statements, Expressions, and Operators

[**Skip Unit 4's navigation links**]
4. Day 4 Statements, Expressions, and Operators

4.1 Statements
4.1.1 Statements and Whitespace

C programs consist of statements, and most statements are composed of expressions and operators. You need an understanding of these three topics to be able to write C programs.

Today, you learn

- What an expression is.
- C's mathematical and relational operators and operator precedence.
- The if statement.
- The evaluation of relational expressions.
- C's logical operators.
- Other operators: compound assignment, conditional, and comma.

Topic 4.1: Statements

*** Statements**

A *statement* is a complete direction instructing the computer to carry out some task.
*

Guideline	Exception
In C, statements are usually written one per line.	Some statements span multiple lines.
C statements always end with a semicolon.	Except for preprocessor directives such as `#define` and `#include`, which are discussed on Day 21, "Taking Advantage of Preprocessor Directives and More".

For example, `x = 2 + 3;` is an *assignment statement*. It instructs the computer to add 2 to 3 and assign the result to the variable `x`.

Topic 4.1.1: Statements and Whitespace

*** Whitespace**
The term *whitespace* refers to spaces, tabs, and blank lines in your source code.
*** C Ignores Whitespace**
The C compiler is not sensitive to whitespace. When the compiler is reading a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores the whitespace. Click the Example link to see how whitespace works.
Example

DO stay consistent with how you use whitespace in statements.
DON'T spread a single statement across multiple lines if there is no need. Try to keep statements on one line.

The statements in the table are equivalent. This gives you a great deal of flexibility in formatting your source code. You shouldn't use formatting like Statement 3, however. Statements should be entered one per line with a standardized scheme for spacing around variables and operators.
If you follow the formatting conventions used in this course, you should be in good shape. As you become more experienced, you may discover that you prefer slight variations. The point is to keep your source code readable.

Statement 1	Statement 2	Statement 3

`x=2+3;`	`x = 2 + 3;`	`x =`
		`2`
		`+`
		`3;`

*** Exception**
The rule that C doesn't care about whitespace has, however, one exception. Within literal string constants, tabs and spaces are not ignored, but are considered part of the string. So, for example, `" 3"` is different than `"3"`. The first one is three spaces followed by the character 3. The second one is just the character 3.

*** Literal Strings**

A *string* is a series of characters, for example, text that you might want to print out on the screen, or text that your program takes as input. *Literal string* constants are strings that are enclosed within quotes and interpreted literally by the compiler, space for space.

*** Breaking a Literal String**

To make a literal string constant (often just called a "string") break across lines, you must use the backslash character (\) just before the break.

Examples

Legal:	Not Legal:

`printf("This is a \`
`really long character\`
`string.");`
	`printf("This is a`
	`really long character`
	`string.");`

*** Null Statement**

If you place a semicolon by itself on a line, you create a *null statement*, that is, a statement that doesn't perform any action. This is perfectly legal in C. Later in the course, you will learn how the null statement can be useful at times.

Topic 4.1.2: Compound Statements

*** Compound Statements**

A *compound statement*, also called a *block*, is a group of two or more C statements enclosed in braces. Click the Example link.

Example

*** Usage**

In C, a block can be used anywhere a single statement can be used. Many examples of this appear throughout the course.

Here's an example of a block:

```
{
  printf("Hello, ");
  printf("world!");
}
```

*** Placement of Enclosing Braces**

Note that the enclosing braces can be positioned in different ways. Click the Example link and the Tip button on the toolbar.

Example

It's a good idea to place braces on their own lines:

- The beginning and end of blocks are clearly visible.
- It is easier to see whether you've left one out.

Due to space constraints, the convention followed in this course is to put the opening brace on the same line as the statement to which it applies.

DO be consistent with placement of block braces. Consider putting block braces on their own lines. This makes the code easier to read.

The following statements are equivalent.

Statement 1	Statement 2

`{`
`  printf("Hello, ");`
`  printf("world!");`
`}`
	`{printf("Hello, ");`
	`printf("world!");}`

Topic 4.2: Expressions

*** Expressions**
In C, an *expression* is anything that evaluates to a numeric value. C expressions come in all levels of complexity.
Let's take a look at simple expressions and complex expressions.

Topic 4.2.1: Simple Expressions

*** Simple Expressions**
The simplest C expression consists of a single item: a simple variable, literal constant, or symbolic constant.
Examples

Here are four simple expressions:

```
PI    /* A symbolic constant (defined in the program). */
20    /* A literal constant. */
rate  /* A variable. */
-1.25  /* Another literal constant. */
```

*** Value Produced**
The following expressions evaluate to the values shown:

This expression	Evaluates to
A literal constant	Its own value.
A symbolic	The value it was given when you created it with the #define

A symbolic constant	The value it was given when you created it with the #define directive.
A variable	The value assigned to it by the program.

Topic 4.2.2: Complex Expressions

*** Complex Expressions**
More *complex expressions* consist of simpler expressions connected by operators.
Examples

`2 + 8` is an expression consisting of the subexpressions 2 and 8 and the addition operator +. The expression `2 + 8` evaluates, as you know, to 10.
You can write C expressions of great complexity:

```
1.25 / 8 + 5 * rate + rate * rate / cost
```

*** Evaluation Depends on Operator Precedence**
When an expression contains multiple operators, the evaluation of the expression depends on operator precedence. This concept, as well as details about all of C's operators, are covered later in this unit.
*** An Entire Statement Itself Is an Expression**
C expressions get even more interesting. Look at the following assignment statement: `x = a + 10;` This statement evaluates the expression `a + 10` and assigns the result to `x`. In addition. the entire statement `x = a + 10` is itself an expression that evaluates to the value of the variable on the left side of the equal sign. This is shown in Figure 4.1
Examples

The statement	Assigns
`y = x = a + 10;`	The value of the expression `a + 10` to both variables, `x` and `y`.
`x = 6 + (y = 4 + 5);`	The value 9 to `y` and the value 15 to `x`. **Note:** The parentheses are required for the statement to compile. The use of parentheses is covered later in this unit.

Topic 4.3: Operators

*** Operators**
An *operator* is a symbol that instructs C to perform some operation, or action, on one or more operands.
*** Operands**

An *operand* is something that an operator acts on. In C, all operands are expressions. C operators fall into several categories.

Topic 4.3.1: The Assignment Operator

*** The Equal Sign**
The assignment operator is the equal sign (=). Its use in programming is somewhat different from its use in regular math.

Expression	Meaning in a C Program	Meaning in Regular Math
`x = y;`	Assign the value of `y` to `x`	`x` is equal to `y`

*** Syntax**
In a C assignment statement, the right side can be any expression and the left side must be a variable name. Thus, the form is

```
variable = expression;
```

When executed, expression is evaluated, and the resulting value is assigned to variable.

Topic 4.3.2: Mathematical Operators

*** Two Unary and Five Binary Operators**
C's mathematical operators perform mathematical operations such as addition and subtraction. C has two unary mathematical operators and five binary mathematical operators.

Topic 4.3.3: The Unary Mathematical Operators

*** Unary Operator**
The *unary* mathematical operators are so named because they take a single operand. C has two unary mathematical operators, listed here.
*** Used with Variables**
The increment and decrement operators can be used only with variables, not with constants.
*** Operation Performed**
The operation performed is to add one to or subtract one from the operand.
Example

Table 4.1: C's Unary Mathematical Operators			
Operator	**Symbol**	**Action**	**Example**
Increment	`++`	Increments operand by one	`++x, x++`
Decrement	`--`	Decrements operand by one	`--x, x--`

Statement	Equivalent Statement

`++x;`	`x = x + 1;`
`--y;`	`y = y - 1;`

*** Prefix Versus Postfix Mode**

You should note from the table here that either unary operator can be placed before its operand *(prefix mode)* or after its operand *(postfix mode)*. These two modes are not equivalent. They differ in terms of *when* the increment or decrement is performed:

When used in this mode...	The increment and decrement operators modify their operand...
Prefix	*Before* it is used in the expression.
Postfix	*After* it is used in the expression.

Table 4.1: C's Unary Mathematical Operators

Operator	Symbol	Action	Example
Increment	`++`	Increments operand by one	`++x, x++`
Decrement	`--`	Decrements operand by one	`--x, x--`

*** Reminder: `=` Is Assignment, Not Equality**

Remember that = is the assignment operator and not a statement of equality. As an analogy, think of = as the "photocopy" operator. The statement `y = x` means to copy `x` into `y`. Subsequent changes to `x`, after the copy has been made, have no effect on `y`.

*** Example Program**

The program in Listing 4.1 illustrates the difference between prefix mode and postfix mode. Click the Listing button.

Listing 4.1: Unary Operator Prefix and Postfix Modes

```
Code  1: /* LIST0401.c: Day 4 Listing 4.1 */
      2: /* Demonstrates unary operator */
      3: /* prefix and postfix modes. */
      4:
      5: #include <stdio.h>
      6:
      7: int a, b;
      8:
      9: int main(void){
     10:   /* Set a and b both equal to 5. */
     11:
     12:   a = b = 5;
     13:
     14:   /* Print them, decrementing each time. */
```

```
15:   /* Use prefix mode for b, postfix mode for a. */
16:
17:   printf("\n%d   %d", a--, --b);
18:   printf("\n%d   %d", a--, --b);
19:   printf("\n%d   %d", a--, --b);
20:   printf("\n%d   %d", a--, --b);
21:   printf("\n%d   %d", a--, --b);
22:
23:   return 0;
24: }
```

Output

```
5   4
4   3
3   2
2   1
1   0
```

Description This program declares two variables, a and b, in line 7. In line 12, the variables are set to the value of 5. With the execution of each printf ( ) statement (lines 17 – 21), both a and b are decremented by one. The variable a is decremented after it is printed and b is decremented before it is printed.

Topic 4.3.4: The Binary Mathematical Operators

*** Binary Operator**

C's *binary operators* take two operands. The binary operators, which include the common mathematical operations found on a calculator, are listed in Table 4.2, which you can see by clicking here.

*** Arithmetic Operators**

These five operators are also known as the arithmetic operators. The first four operators in Table 4.2 should be familiar to you, and you should have little trouble using them. The fifth operator, modulus, may be new.

Table 4.2: C's Binary Mathematical Operators

Operator	Symbol	Action	Example
Addition	+	Adds its two operands	`x + y`
Subtraction	-	Subtracts the second operand from the first operand	`x - y`
Multiplication	*	Multiplies its two operands	`x * y`
Division	/	Divides the first operand by the second operand	`x / y`
Modulus	%	Gives the remainder when the first operand is divided by the second operand	`x % y`

*** Modulus**

Modulus returns the remainder when the first operand is divided by the second operand.

Example

The program in Listing 4.2 illustrates how you can use the modulus operator to convert a large number of seconds into hours, minutes, and seconds. Click the Listing button on the toolbar.

Listing 4.2: The Modulus Operator

Code

```
 1: /* LIST0402.c: Day 4 Listing 4.2 */
 2: /* Illustrates the modulus operator. */
 3: /* Inputs a number of seconds, and converts to */
 4: /* hours, minutes, and seconds. */
 5:
 6: #include <stdio.h>
 7:
 8: #define SECS_PER_MIN 60    /* Define constants. */
 9: #define SECS_PER_HOUR 3600
10:
11: unsigned seconds, minutes, hours;
12: unsigned secs_left, mins_left;
13:
14: int main(void) {
15:   /* Input the number of seconds. */
16:
17:   printf("Enter number of seconds (<65000): ");
18:   scanf("%d", &seconds);
19:
20:   hours = seconds / SECS_PER_HOUR;
21:   minutes = seconds / SECS_PER_MIN;
22:   mins_left = minutes % SECS_PER_MIN;
23:   secs_left = seconds % SECS_PER_MIN;
24:
25:   printf("%u seconds is equal to ", seconds);
26:   printf("%u h, %u m, and %u s",
27:           hours, mins_left, secs_left);
28:   return 0;
29: }
```

Output

```
E:\>list0402
Enter number of seconds (< 65000): 60
60 seconds is equal to 0 h, 1 m, and 0 s

E:\>list0402
Enter number of seconds (< 65000): 10000
10000 seconds is equal to 2 h, 46 m, and 40 s
```

Description

Listing 4.2 follows the same format that all the previous programs have followed. Lines 1 - 4 provide some comments to state what the program is going to do.

Line 5 is whitespace to make the program more readable. Just like the whitespace in statements and expressions, blank lines are ignored by the compiler.

Line 6 includes the necessary header file for this program.

Lines 8 and 9 define two constants, SECS_PER_MIN and SECS_PER_HOUR that are used to make the statements in the

	SECS_PER_HOUR, that are used to make the statements in the program easier to read. Lines 11 and 12 declare all the variables that will be used. Some people choose to declare each variable on an individual line. As with many elements of C, this is matter of style. Either method is correct. Line 14 is the main() function, which contains the bulk of the program. To convert seconds to hours and minutes, the program must first get the values it needs to work with. To do this, line 17 uses the printf() function to display a statement on the screen followed by line 18, which uses the scanf() function to get the number entered by the user. The scanf() statement then stores the number of seconds to be converted into the variable seconds. The printf() and scanf() functions are covered in more detail on Day 7, "Basic Input/Output." Line 20 contains an expression to determine the number of hours by dividing the number of seconds by the constant SECS_PER_HOUR. Because hours is an integer variable, the remainder value is ignored. Line 21 uses the same logic to determine the total number of minutes for the seconds entered. Because the total number of minutes figured in line 21 also contains minutes for the hours, line 22 uses the modulus operator to divide out the hours and keep the remaining minutes. Line 23 does a similar calculation for determining the number of seconds that are left. Lines 25, 26 and 27 are reflective of what you have seen before. They take the values that have been calculated in the expressions and display them. Line 28 finishes the program by returning 0 to the operating system before exiting.

11 modulus 4 equals 3 (that is, 4 goes into 11 two times with 3 left over).
Here are some more examples:

```
100 modulus 9 equals 1
10 modulus 5 equals 0
40 modulus 6 equals 4
```

Topic 4.3.5: Operator Precedence and Parentheses

*** Importance of Order of Evaluation**

In an expression that contains more than one operator, what is the order in which operations are performed?

performed?
Example

*** Operator Precedence**

Clearly, some rules are needed about the order in which operations are performed. This order, called *operator precedence*, is strictly spelled out in C. Each operator has a specific precedence.

*** Higher Precedence Performed First**

When an expression is evaluated, operators with higher precedence are performed first. For example, multiplication, division and modulus have a greater precedence than addition and subtraction. Click the Tip button on the toolbar.

DO use parentheses to make the order of expression evaluation clear.
DON'T overload an expression. It is often more clear to break an expression into two or more statements. This is especially true when using the unary operators (--) or (++).

The importance of this question is illustrated by the following assignment statement: `x = 4 + 5 * 3;`

If this operation is performed first ...	You have ...	And x is assigned the value ...
Addition `4+5`	`x = 9 * 3;`	`27`
Multiplication `5*3`	`x = 4 + 15;`	`19`

*** Mathematical Operator Precedence**

In any C expression, operations are performed in the following order:
1. Unary increment and decrement
2. Multiplication, division, and modulus
3. Addition and subtraction

*** Same Precedence: Left-To-Right**

If an expression contains more than one operator with the same precedence level, the operators are performed in left-to-right order as they appear in the expression.
Example

In the expression `12 % 5 * 2` the `%` and `*` have the same precedence level, but the `%` is the leftmost operator, so it is performed first. The expression evaluates to 4 (`12 % 5` evaluates to 2; 2 times 2 is 4).
Returning to the previous example, you see that the statement `x = 4 + 5 * 3;` assigns the value 19 to `x` because the multiplication is performed before the addition.

*** Parentheses Modify Evaluation Order**

What if the order of precedence does not evaluate your expression as needed? Using the previous example, what if you wanted to add 4 to 5 and then multiply the sum by 3? C uses parentheses to modify the evaluation order. A subexpression enclosed in parentheses is evaluated first, without regard to operator precedence.
Example

Example

Thus, you could write

```
x = (4 + 5) * 3;
```

The expression 4 + 5 inside the parentheses is evaluated first and so the value assigned to `x` is 27.

*** Multiple Parentheses**

You can use multiple and nested parentheses in an expression. When parentheses are nested, evaluation proceeds from the innermost expression outward.

Example

Look at the following complex expression:

```
x = 25 - (2 * (10 + (8 / 2)))
```

This evaluation proceeds in the following steps:

Step	Action
1	The innermost expression, `8 / 2`, is evaluated first, yielding the value 4. `25 - (2 * (10 + 4))`
2	Moving outward, the next expression, which becomes `10 + 4`, is evaluated, yielding the value 14. `25 - (2 * 14)`
3	The last, or outermost, expression becomes `2 * 14` and is evaluated, yielding the value 28. `25 - 28`
4	The final expression, `25 - 28`, is evaluated, assigning the value -3 to the variable `x`. `x = -3`

*** Parentheses for Clarity**

You may want to use parentheses in some expressions for the sake of clarity, even when they are not needed for modifying operator precedence. Parentheses must always be in pairs, or the compiler generates an error message.

Topic 4.3.6: Order of Subexpression Evaluation

*** Same Precedence: Left-To-Right**

As was mentioned in the previous section, if C expressions contain more than one operator with the same precedence level, they are evaluated left to right.

Example

*** Across Precedence Levels**

Across precedence levels, however, there is no guarantee of left-to-right order.

Example

In the expression `w * x / y * z`, `w` is first multiplied by `x`, the result of the multiplication is then divided by `y`, and the result of the division is then multiplied by `z`.

divided by `y`, and the result of the division is then multiplied by `z`.

Look at this expression: `w * x / y + z / y`. Because of precedence, the multiplication and division are performed before the addition. C does not specify, however, whether the subexpression `w * x / y` is to be evaluated before or after `z / y`. It may not be clear to you why this matters. Look at another example: `w * x / ++y + z / y`. If the first subexpression is evaluated first, `y` is incremented when the second expression is evaluated. If the second expression is evaluated first, `y` isn't incremented, and the result is different. You should, therefore, avoid this sort of indeterminate expression in your programming.

Topic 4.3.7: Relational Operators

*** Relational Operators**
C's *relational operators* are used to compare expressions, "asking" questions such as, "Is `x` greater than 100?" or "Is `y` equal to 0?" An expression containing a relational operator evaluates as either true (1) or false (0).
*** Six Relational Operators**
C's six relational operators are listed here.
*** Usage**
These examples use literal constants, but the same principles hold with variables.

Table 4.3: C's Relational Operators			
Operator	**Symbol**	**Question Asked**	**Example**
Equal	`==`	Is operand 1 equal to operand 2?	`x == y`
Greater than	`>`	Is operand 1 greater than operand 2?	`x > y`
Less than	`<`	Is operand 1 less than operand 2?	`x < y`
Greater than or equal	`>=`	Is operand 1 greater than or equal to operand 2?	`x >= y`
Less than or equal	`<=`	Is operand 1 less than or equal to operand 2?	`x <= y`
Not equal	`!=`	Is operand 1 not equal to operand 2?	`x != y`

Table 4.4: Relational Operators in Use	
Expression	**Evaluates As**
`5 == 1`	0 (false)
`5 > 1`	1 (true)
`5 != 1`	1 (true)

`(5 + 10) == (3 * 5)`	1 (true)

DO learn how C interprets true and false. When working with relational operators, true is equal to 1, and false is equal to 0.
DON'T confuse ==, the relational operator, with =, the assignment operator. This is one of the most common errors that C programmers make!

Topic 4.4: The if Statement

*** Use of Relational Operators**
Relational operators are used mainly to construct the relational expressions used in if and while statements, covered in detail on Day 6, "Basic Program Control." For now, it is useful to explain the basics of the if statement to show how relational operators are used to make program control statements.

*** Program Control Statements**
You may be wondering what a program control statement is. Statements in a C program normally execute from top to bottom, in the same order as they appear in your source code file. A *program control statement* modifies the order of statement execution. Program control statements can cause other program statements to execute multiple times or to not execute at all, depending on the circumstances. The if statement is one of C's program control statements. Others, such as do and while, are covered on Day 6, "Basic Program Control."

*** Basic Syntax**
In its basic form, the `if` statement evaluates an expression and directs program execution depending on the result of that evaluation. The form of an `if` statement is as follows:

```
if (expression)
statement;
```

DO remember that if you program too much in one day, you'll get C sick.
DON'T make the mistake of putting a semicolon at the end of an if statement. An if statement should end with the conditional statement that follows it. In the following, statement1 executes whether x equals 2 or not, because each line is evaluated as a separate statement, not together as intended!

```
if (x == 2);  /* Semicolon does not belong! */
statement1;
```

*** Process**
If the `if` expression evaluates as true, the statement is executed. If the `if` expression evaluates as

false, the statement is not executed. In either case, execution then passes to whatever code follows the `if` statement. You can say that execution of statement depends on the result of expression. Note that both the line `if (expression)` and the line `statement;` are considered to make up the

that both the line `if (expression)` and the line `statement;` are considered to make up the complete `if` statement. They are not separate statements.

*** Syntax with Multiple Statements**

An `if` statement can control the execution of multiple statements through the use of a compound statement, or block. As defined earlier in this unit, a block is a group of two or more statements enclosed in braces. A block can be used anywhere a single statement can be used. You could therefore write an `if` statement as follows:

```
if (expression) {
  statement-1;
  statement-2;
  /* additional code goes here */
  statement-n;
}
```

*** With Relational Expressions**

In your programming, you will find that `if` statements are used most often with relational expressions; in other words, "execute the following statement(s) only if such-and-such a condition is true."

Example

Here's an example:

```
if (x > y)
  y = x;
```

This code assigns the value of `x` to `y` only if `x` is greater than `y`. If `x` is not greater than `y`, no assignment takes place.

*** An Else Clause**

An if statement can optionally include an else clause. The else clause is included as follows:

```
if (expression)
  statement-1;
else
   statement-2;
```

If expression is true, statement-1 is executed. If expression is false, statement-2 is executed. Both statement-1 and statement-2 can be compound statements, or blocks.

*** Example Programs**

Listing 4.3 presents a short program that illustrates the use of if statements. Listing 4.4 shows the program in Listing 4.3 rewritten to use an if statement with an else clause. Click both Listing buttons on the toolbar to see the programs.

Listing 4.3: The if Statement

```
Code  1: /* LIST0403.c: Day 4 Listing 4.3 */
      2: /* Demonstrates the use of if statements. */
      3:
      4: #include <stdio.h>
      5:
      6: int x, y;
      7:
      8: int main(void) {
      9:   /* Input the two values to be tested. */
```

```
10:
11:   printf("\nInput an integer value for x: ");
12:   scanf("%d", &x);
13:   printf("\nInput an integer value for y: ");
14:   scanf("%d", &y);
15:
16:   /* Test values and print result. */
17:
18:   if (x == y)
19:     printf("x is equal to y");
20:
21:   if (x > y)
22:     printf("x is greater than y");
23:
24:   if (x < y)
25:     printf("x is smaller than y");
26:   return 0;
27: }
```

Output

```
E:\>list0403

Input an integer value for x: 100

Input an integer value for y: 10
x is greater than y

E:\>list0403

Input an integer value for x: 10

Input an integer value for y: 100
x is smaller than y

E:\>list0403

Input an integer value for x: 10

Input an integer value for y: 10
x is equal to y
```

Description

Listing 4.3 shows three if statements in action (lines 18 – 25). Many of the lines in this program should be familiar. Line 6 declares two variables, x and y, and lines 11 – 14 prompt the user for values to be placed into these variables.

Lines 18 – 25 use if statements to determine whether x is greater than, less than, or equal to y. Note that line 18 uses an if statement to see whether x is equal to y. Remember ==, the equal operator, is the same as "is equal to" and should not be confused with =, the assignment operator.

After the program checks to see whether the variables are equal, in line 21 it checks to see whether x is greater than y, followed by a check in line 24 to see whether x is less than y. You might think that this is inefficient, and you

are right. In the next program, you will see how to avoid this inefficiency. For now, run the program with different values for x and y to see the results.

Listing 4.4: if Statement with an else Clause

Code

```
 1: /* LIST0404.c: Day 4 Listing 4.4 */
 2: /* Demonstrates the use of an if statement */
 3: /* with an else clause. */
 4:
 5: #include <stdio.h>
 6:
 7: int x, y;
 8:
 9: int main(void) {
10:   /* Input the two values to be tested. */
11:
12:   printf("\nInput an integer value for x: ");
13:   scanf("%d", &x);
14:   printf("\nInput an integer value for y: ");
15:   scanf("%d", &y);
16:
17:   /* Test values and print result. */
18:
19:   if (x == y)
20:     printf("x is equal to y");
21:   else
22:     if (x > y)
23:       printf("x is greater than y");
24:     else
25:       printf("x is smaller than y");
26:   return 0;
27: }
```

Output

```
E:\>list0404

Input an integer value for x: 99

Input an integer value for y: 8
x is greater than y

E:\>list0404

Input an integer value for x: 8

Input an integer value for y: 99
x is smaller than y

E:\>list0404

Input an integer value for x: 99

Input an integer value for y: 99
x is equal to y
```

Description

Lines 19 – 25 are slightly different from the previous listing. Line 19 still checks to see whether x equals y. If x does equal y, x is equal to y is printed just as in Listing 4.3; however, the program then ends.

	Lines 21 – 25 are not executed. Line 22 is executed only if x is not equal to y, or to be more accurate, if the expression "x equals y" is false. If x does not equal y, line 22 checks to see whether x is greater than y. If so, line 23 prints x is greater than y, otherwise (else) line 25 is executed. Note that the program in Listing 4.4 uses a *nested* if statement. Nesting means to place (nest) one or more C statements inside another C statement. In the case of Listing 4.4, an if statement is part of the first if statement's else clause.

Topic 4.5: Evaluation of Relational Expressions

*** Evaluate to a Value**

Remember that expressions using relational operators are true C expressions that evaluate, by definition, to a value. Relational expressions evaluate to a value of either false (0) or true (1). Although the most common use for relational expressions is within if statements and other conditional constructions, they can be used as purely numeric values. This is illustrated by the program in Listing 4.5. Click the Listing button.

Listing 4.5: The Evaluation of Relational Expressions

Code

```
 1: /* LIST0405.c: Day 4 Listing 4.5 */
 2: /* Demonstrates the evaluation of relational */
 3: /* expressions. */
 4:
 5: #include <stdio.h>
 6:
 7: int a;
 8:
 9: int main(void) {
10:   a = (5 == 5); /* Evaluates to 1. */
11:   printf("\na = (5 == 5)\na = %d", a);
12:
13:   a = (5 != 5); /* Evaluates to 0. */
14:   printf("\na = (5 != 5)\na = %d", a);
15:
16:   a = (12 == 12) + (5 != 1); /* 1 + 1. */
17:   printf("\na = (12 == 12) + (5 != 1)\na = %d", a);
18:   return 0;
19: }
```

Output

```
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) + (5 != 1)

a = 2
```

Description

The output from this listing may seem a little confusing at first. Remember, the most common mistake people make when using the relational operators

is to use a single equal sign – the assignment operator – instead of a double equal sign.

The expression `x = 5` evaluates as 5 (and also assigns the value 5 to x). In contrast, the expression `x == 5` evaluates as either 0 or 1 (depending on whether x is equal to 5) and does not change the value of x. If by mistake you write

```
if (x = 5)
  printf("x is equal to 5");
```

the message always prints because the expression being tested by the if statement always evaluates as true, no matter what the original value of x happens to be.

Looking at Listing 4.5, you can begin to understand why a takes on the values that it does. In line 10, the value 5 does equal 5, therefore true (1) is assigned to a. In line 13, the statement "5 does not equal 5" is false, so 0 is assigned to a.

Topic 4.5.1: Precedence of Relational Operators

*** Lower Precedence**
Like the mathematical operators discussed earlier in the unit, the relational operators each have a precedence that determines the order in which they are performed in a multiple-operator expression. Similarly, you can use parentheses to modify precedence in expressions that use relational operators. First, all the relational operators have a lower precedence than the mathematical operators.
Example

Thus, if you write `if (x + 2 > y),` 2 is added to x, and the result is compared to y. This is the equivalent of `if ((x + 2) > y)` which is a good example of using parentheses for the sake of clarity. Although not required by the C compiler, the parentheses surrounding `(x + 2)` make it clear that it is the sum of x and 2 that is to be compared with y.

*** Two-Level Precedence**
There is also a two-level precedence within the relational operators. less than (<), less than or equal (<=), greater than (>) and greater than or equal (>=) have a greater precedence than not equal (`!=`) and equal (==).
Example

DON'T put assignment statements in if statements. This can be confusing to other people who look at your code. They may think it is a mistake and change your assignment to the logical equal

statement.
DON'T use the "not equal to" operator (!=) in an if statement containing an else. It is almost always clearer to use the "equal to" operator (==) with an else.

```
if (x != 5)
  statement-1;
else
  statement-2;
```

would be better as

```
if (x == 5)
  statement-2;
else
  statement-1;
```

Thus, if you write `x == y > z,` it is the same as `x == (y > z)` because C first evaluates the expression `y > z`, resulting in a value of 0 or 1. Next, C determines whether x is equal to the 1 or 0 obtained in the first step. You rarely, if ever, use this sort of construction, but you should know about it.

Topic 4.6: Logical Operators

*** Combining Relational Operators**
At times, you may need to ask more than one relational question at the same time. For example, "If it's 7:00 AM and a weekday and not your vacation, ring the alarm." C's logical operators enable you to combine two or more relational expressions into a single expression that evaluates as either true or false. C's three logical operators are listed here.

Table 4.5: C's Logical Operators		
Operator	**Symbol**	**Example**
and	`&&`	`exp1 && exp2`
or	`""`	`exp1 "" exp2`
not	`!`	`!exp1`

DO use `(expression == 0)` instead of `(!expression)`. When compiled, these two expressions evaluate the same; however, the first is more readable.
DO use the logical operators `&&` and `""` instead of nesting if statements.

Topic 4.6.1: Logical Operators in Use

*** Expressions Evaluate as True or False**
The way logical operators work is explained here.
You can see that expressions which use the logical operators evaluate as either true or false

depending on the true/false value of their operand(s).

Table 4.6: C's Logical Operators in Use	
Expression	**Evaluates As**
`(exp1 && exp2)`	True (1) only if both exp1 and exp2 are true; false (0) otherwise.
`(exp1 "" exp2)`	True (1) if either exp1 or exp2 is true; false (0) only if both are false.
`(!exp1)`	False (0) if exp1 is true; true (1) if exp1 is false.

Table 4.7: Code Examples of C's Logical Operators	
Expression	**Evaluates As**
`(5 == 5) && (6 != 2)`	True (1) because both operands are true.
`(5 > 1) "" (6 < 1)`	True (1) because one operand is true.
`(2 == 1) && (5 == 5)`	False (0) because one operand is false.
`!(5 == 4)`	True (1) because the operand is false.

*** Multiple Logical Operators**
You can create expressions that use multiple logical operators. For example, to ask the question "Is x equal to 2, 3, or 4?" you would write
`(x == 2) "" (x == 3) "" (x == 4)`
The logical operators often provide more than one way to ask a question. If x is an integer variable, the previous question also could be written in either of the following ways:
`(x > 1) && (x < 5)` or `(x >= 2) && (x <= 4)`

Topic 4.6.2: More on True/False Values

*** Numeric Values Interpreted as True or False**
You've seen that C's relational expressions evaluate to 0 to represent false and to 1 to represent true. It's important to be aware, however, that any numeric value is interpreted as either true or false when it is used in a C expression or statement that is expecting a logical (that is, a true or false) value.

*** Rules**
The rules for this are as follows:

- A value of zero represents false.
- Any nonzero value represents true.
- Any assignment statement represents the value of the variable after assignment

Example
You can further generalize this because, for any C expression, writing `(expression)` is equivalent to writing `(expression != 0).` Both evaluate as true if expression is nonzero, and as false if expression is 0. Using the not (!) operator, you can also write `(!expression)` which is equivalent to `(expression == 0).`

The following example illustrates the principle that a non-zero value represents true:

```
x = 125;
if (x)
  printf("%d", x);
```

In this case, the value of x is printed. Because x has a nonzero value, the expression (x) is interpreted as true by the if statement.

The following example illustrates the principle that an assignment statement represents the value of the variable after assignment:

```
x = 1;

if (x=0)

printf("%d", x);
```

In this case, the value of x is not printed because it has been assigned the value 0. Note that the addition of another equals sign would change the meaning of the expression inside the if statement. If the statement read `if (x==0)` the condition would be read as false, since x was assigned the value 1.

Topic 4.6.3: Precedence of Logical Operators

*** Precedence of ! Operator**
As you may have guessed, C's logical operators also have a precedence order, both among themselves and in relation to other operators. The ! operator has a precedence equal to the unary mathematical operators ++ and -- . Thus, ! has a higher precedence than all the relational operators and all the binary mathematical operators.

*** Precedence of `&&` and `""` Operators**
In contrast, the `&&` and `""` operators have much lower precedence, lower than all the mathematical and relational operators, although `&&` has a higher precedence than, `""`. As with all of C's operators, parentheses can be used to modify evaluation order when using the logical operators. Look at the following example.
Example

You want to write a logical expression that makes three individual comparisons:
1. Is a less than b?
2. Is a less than c?

3. Is c less than d?
You want the entire logical expression to evaluate as true if condition 3 is true and either condition 1 or condition 2 is true. You might write
`a < b "" a < c && c < d`
This does not, however, do what you intended. Because the `&&` operator has higher precedence than `""`, the expression is equivalent to
`a < b "" (a < c && c < d)`
and evaluates as true if `(a < b)` is true, whether or not the relationships `(a < c)` and `(c < d)` are true. You need to write
`(a < b "" a < c) && c < d`
which forces the `""` to be evaluated before the `&&`.

*** Example Program**
Listing 4.6 evaluates the expression `a < b "" a < c && c < d` written with and without parentheses. The variables are set so that, if written correctly, the expression should evaluate as false (0).

Listing 4.6: Logical Operator Precedence

Code

```
 1: /* LIST0406.c: Day 4 Listing 4.6 */
 2: /* Demonstrates logical operator precedence. */
 3:
 4: #include <stdio.h>
 5:
 6: /* Initialize variables. c is not less than d, */
 7: /* which is one of the conditions to test for. */
 8: /* Therefore, the entire expression should */
 9: /* evaluate as false. */
10:
11: int a = 5, b = 6, c = 5, d = 1;
12: int x;
13:
14: int main(void) {
15:   /* Evaluates expression without parentheses. */
16:
17:   x = a < b "" a < c && c < d;
18:   printf("\nWithout parentheses the expression "
19:          "evaluates as %d", x);
20:
21:   /* Evaluates expression with parentheses. */
22:
23:   x = (a < b "" a < c) && c < d;
24:   printf("\nWith parentheses the expression "
25:          "evaluates as %d", x);
26:   return 0;
27: }
```

Output

```
Without parentheses the expression evaluates as 1
With parentheses the expression evaluates as 0
```

Description

The two values printed for the expression are different. This program initializes four variables in line 11 with values to be used in the comparisons. Line 12 declares x to be used to store and print the results. Lines 17 and 23 use the logical operators. Line 17 does not use the parentheses, so the results are determined by operator precedence. In this case, the results are not those

	you desired. Line 23 uses parentheses to change the order in which the expressions are evaluated.

Topic 4.6.4: Compound Assignment Operators

*** Compound Assignment Operators**
C's *compound assignment operators* provide a shorthand method for combining a binary mathematical operation with an assignment operation.
Example
*** Syntax**
In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

```
exp1 op= exp2
```

which is equivalent to writing

```
exp1 = exp1 op exp2;
```

Say you want to increase the value of x by 5, or in other words, add 5 to x and assign the result to x. You could write

```
x = x + 5;
```

Using a compound assignment operator, which you can think of as a shorthand method of assignment, you would write

```
x += 5;
```

*** Usage**
You can create compound assignment operators with the five binary mathematical operators discussed earlier in this unit.
The compound operators provide a convenient shorthand, the advantages of which are particularly evident when the variable on the left side of the assignment operator is complex. As with all other assignment statements, a compound assignment statement is an expression and evaluates to the value assigned to the left side.
Example

Table 4.8: Examples of Compound Assignment Operators	
If you write...	**It is equivalent to...**
`x *= y`	`x = x * y`
`y -= z + 1`	`y = y - z + 1`
`a /= b`	`a = a / b`
`x += y / 8`	`x = x + y / 8`
`y %= 3`	`y = y % 3`

Thus, executing the statements
```
x = 12;
z = x += 2;
```
results in both x and z having the value 14.

Topic 4.6.5: The Conditional Operator

*** Ternary Operator**
The *conditional operator* is C's only *ternary operator*, meaning that it takes three operands.
*** Syntax**
The syntax of the conditional operator is
```
test-expression ? expression-1 : expression-2
```
If test-expression evaluates as true (that is, nonzero), the entire expression evaluates as the value of expression-1. If test-expression evaluates as false (that is, zero), the entire expression evaluates as the value of expression-2.
Example

For example, the statement
```
x = y ? 1 : 100;
```
assigns the value 1 to x if y is true and assigns 100 to x if y is false. Likewise, to make z equal to the larger of x and y, you could write
```
z = (x > y) ? x : y;
```

*** Compared to an if Statement**
Perhaps you've noticed that the conditional operator functions somewhat like an if statement. The previous statement could also be written
```
if (x > y)
  z = x;
else
  z = y;
```
The conditional operator can't be used in all situations in place of an if...else construction, but the conditional operator is more concise. The conditional operator can also be used in places you can't use an if statement, such as inside a single printf ( ) statement.

Topic 4.6.6: The Comma Operator

*** Comma as a Punctuation Mark**
The *comma* is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on.
*** Comma as an Operator**

In certain situations the comma acts as an operator rather than just as a separator. You can form an expression by separating two subexpressions with a comma. The result is as follows:

- Both expressions are evaluated, with the left expression being evaluated first.
- The entire expression evaluates as the value of the right expression.

While this is a legal statement, it leads to code that is difficult to read and maintain. Usually, the subexpressions should be written as separate lines of code.
Example
As the next unit teaches you, the most common use of the comma operator is in for statements.

The statement

```
x = (a++, b++);
```

assigns the value of b to x, then increments a, and then increments b. Because the ++ operator is used in postfix mode, the value of b – before it is incremented – is assigned to x. Using parentheses is necessary because the comma operator has a low precedence, even lower than the assignment operator.

Topic 4.7: Day 4 Q&A

*** Questions & Answers**
Take a look at some questions that are frequently asked by programmers new to C.

*** Question 1**
What effect do spaces and blank lines have on how a program runs?
*** Answer**
Whitespace (lines, spaces, tabs) makes the code listing more readable. When the program is compiled, whitespace is stripped and thus has no effect on the executable program. For this reason, whitespace should be used to make your program easier to read.

*** Question 2**
Is it better to code a compound if statement or to nest multiple if statements?
*** Answer**
You should make your code easy to understand. If you nest if statements, they are evaluated as shown in the unit. If you use a single compound statement, the expressions are evaluated from left to right until the truth or falsehood of the entire statement is known.

*** Question 3**
What is the difference between unary and binary operators?
*** Answer**
As the names imply, unary operators work with one variable and binary work with two.

*** Question 4**
Is the subtraction operator (-) binary or unary?
*** Answer**

It's both! The compiler is smart enough to know which you are using. It knows which form to use based on the number of variables in the expression that is used. In the following statement, it is a unary:

```
x = -y;
```
versus the following binary use:
```
x = a - b;
```

*** Question 5**
Are negative numbers considered to be true or false?
*** Answer**
Remember 0 is false, and any other value is true. This includes negative numbers.

Topic 4.8: Day 4 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
#include <stdio.h>
int a, b;
int main(void){

  a = b = 5;
  printf("\n%d   %d", a--, --b);
  printf("\n%d   %d", a--, --b);
  printf("\n%d   %d", a--, --b);
  printf("\n%d   %d", a--, --b);
  printf("\n%d   %d", a--, --b);


  return 0;
}
```

Sample 1

```
if ((x >= 1) && (x <= 20))
  y = x;
```

Sample 2

```
y = ((x >= 1) && (x <= 20)) ? x : y;
```

Sample 1

```
if (x < 1)
  if ( x > 10 )
    statement1;
```

Sample 2

```
if (x < 1 && x > 10)
  statement1;
```

```
01: /* a program with problems... */
02: #include <stdio.h>
03: int x= 1:
04: main() {
05:     if(x = 1);
06:         printf("x equals 1");
07:     otherwise
08:         printf("x does not equal 1");
09:     return 0;
10: }
```

Topic 4.9: Day 4 Try Its

*** Try Its**

Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**

The following code is not well formatted. Enter and compile it to see whether it works.

```
#include <stdio.h>
int x,y;int main(void){ printf(
"\nEnter two numbers");scanf(
"%d %d" , &x, &y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

Answer

The listing should have worked even though it was poorly structured. The purpose of the listing was to demonstrate that whitespace is irrelevant to how the program runs. You should use whitespace to make your programs readable.

*** Exercise 2**

The code from exercise one follows. Rewrite the code to be more readable.

```
#include <stdio.h>
int x,y;int main(void){ printf(
"\nEnter two numbers");scanf(
"%d %d" , &x, &y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

Answer

The following is a better way to structure the exercise one listing:

```
#include <stdio.h>
int x, y;
int main(void) {
  printf("\nEnter two numbers");
  scanf("%d %d" , &x, &y);
  printf("\n\n%d is bigger", (x > y) ? x : y);

  return 0;
}
```

The listing asks for two numbers, x and y, and then prints which is bigger.

*** Exercise 3**

Change the Exercise 4.3 program to count upward instead of downward.

Answer

Listing 4.1: Unary Operator Prefix and Postfix Modes

```
Code  1: /* LIST0401.c: Day 4 Listing 4.1 */
      2: /* Demonstrates unary operator */
      3: /* prefix and postfix modes. */
      4:
      5: #include <stdio.h>
      6:
      7: int a, b;
      8:
      9: int main(void){
     10:   /* Set a and b both equal to 5. */
     11:
     12:   a = b = 5;
     13:
     14:   /* Print them, decrementing each time. */
     15:   /* Use prefix mode for b, postfix mode for a. */
     16:
     17:   printf("\n%d   %d", a--, --b);
     18:   printf("\n%d   %d", a--, --b);
```

```
19:   printf("\n%d   %d", a--, --b);
20:   printf("\n%d   %d", a--, --b);
21:   printf("\n%d   %d", a--, --b);
22:
23:   return 0;
24: }
```

Output

```
5   4
4   3
3   2
2   1
1   0
```

Description

This program declares two variables, a and b, in line 7. In line 12, the variables are set to the value of 5. With the execution of each printf ( ) statement (lines 17 – 21), both a and b are decremented by one. After it is printed, a is decremented. b is decremented before it is printed.

The only changes needed in Listing 4.1 are the following:

```
11:  printf("\n%d   %d", a++, ++b);
12:  printf("\n%d   %d", a++, ++b);
13:  printf("\n%d   %d", a++, ++b);
14:  printf("\n%d   %d", a++, ++b);
15:  printf("\n%d   %d", a++, ++b);
```

*** Exercise 4**

Write an if statement that assigns the value of x to the variable y only if x is between 1 and 20. Leave y unchanged if x is not in that range.

Answer

The following code fragment is just one of many possible examples. It checks to see if x is greater than or equal to 1 and if x is less than or equal to 20. If these two conditions are met, x is assigned to y. If these conditions are not met, s is not assigned to y; therefore, y remains the same.

```
if ((x > 1) && (x < 20))
  y = x;
```

*** Exercise 5**

Use the conditional operator to perform the same task as in exercise four.

Answer

```
y = ((x >= 1) && (x <= 20)) ? x : y;
```

*** Exercise 6**

*** Exercise 6**

Rewrite the following nested if statements using a single if statement and compound operators.

```
if (x > 1)
  if ( x < 10 )
    statement;
```

Answer

```
if (x > 1 && x < 10)
  statement;
```

*** Exercise 7**

To what value do each of the following expressions evaluate?

a. `(1 + 2 * 3)`
b. `10 % 3 * 3 - (1 + 2)`
c. `((1 + 2) * 3)`
d. `(5 == 5)`
e. `(x = 5)`

Answer

a. `(1 + 2 * 3) = 7`
b. `10 % 3 * 3 - (1 + 2) = 0`
c. `((1 + 2) * 3) = 9`
d. `(5 == 5) = 1 (true)`
e. `(x = 5) = 5`

*** Exercise 8**

Write an if statement that determines whether someone is legally an adult (age 21), but not a senior citizen (age 65).

Answer

We did not do what this exercise asked; however, you can get the answer to the question from this nested if. (You also could have indented this differently.)

```
if(age < 21)
  printf("You are not an adult");
else if(age >= 65)
    printf("You are a senior citizen!");
  else
    printf("You are an adult");
```

*** Exercise 9**

BUG BUSTERS: Fix the following program so that it runs correctly.

```
/* a program with problems... */
#include <stdio.h>
```

```
#include <stdio.h>
int x= 5:
main() {
  if(x = 5);
    printf("x equals 5");
  otherwise
    printf("x does not equal 5");
  return 0;
}
```

Answer

This program had four problems. The first was on line 3. The assignment statement should end with a semicolon, not a colon.

The second problem was the semicolon at the end of the if statement on line 5. Only the statements inside the if statment get semicolons.

The third problem is common; the assignment operator (=) was used rather than the relational operator (==) in the if statement.

The final problem was the word `otherwise` on line 7. This should be `else`.

```
01: /* a program with problems... */
02: #include <stdio.h>
03: int x= 5;
04: main() {
05:   if(x == 5)
06:     printf("x equals 5");
07:   else
08:     printf("x does not equal 5");
09:   return 0;
10: }
```

Topic 4.10: Day 4 Summary

Statements

This unit has covered a lot of material. You have learned what a C statement is, that whitespace does not matter to a C compiler, and that statements always terminate with a semicolon. You've also learned that a compound statement (or block), which consists of two or more statements enclosed in braces, can be used anywhere a single statement can be used.

Expressions

Many statements are made up of some combination of expressions and operators. Remember that an expression is anything that evaluates to a numeric value. Complex expressions can contain many simpler expressions, which are called subexpressions.

Operators

Operators are C symbols that instruct the computer to perform an operation on one or more expressions. Some operators are unary, which means that they operate on a single operand. Most of C's operators are binary, however, operating on two operands. One operator, the conditional operator, is ternary. C's operators have a defined hierarchy of precedence that determines the order in

operator, is ternary. C's operators have a defined hierarchy of precedence that determines the order in which operations are performed in an expression that contains multiple operators.

Three Categories of Operators
The C operators covered by this unit fall into three categories, indicating that

- Mathematical operators perform arithmetic operations on their operands (for example, addition).
- Relational operators perform comparisons between their operands (for example, greater than).
- Logical operators operate on true/false expressions. Remember that C uses 0 and 1 to represent false and true, respectively, and that any nonzero value is interpreted as being true.

The if Statement
You've also been introduced to C's `if` statement, which enables you to control program execution based on the evaluation of relational expressions.

Unit 5. Day 5 Functions: The Basics

Functions are central to C programming and to the philosophy of C program design. You've already been introduced to some of C's library functions, which are complete functions supplied as part of your compiler. This unit covers user-defined functions which, as the name implies, are functions that you, the programmer, create.

Today you learn

- What a function is and what its parts are.
- About the advantages of structured programming with functions.
- How to create a function.
- About the declaration of local variables in a function.
- How to return a value from a function to the program.
- How to pass arguments to a function.

Topic 5.1: What Is a Function?

*** What Is a Function?**
This unit approaches the question "What is a function?" in two ways. First, it tells you what functions are and then shows you how they're used.

Topic 5.1.1: A Function Defined

*** Functions**
First the definition: a *function* is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program.

*** Parts of the Definition**
Now, look at the parts of this definition.

Component	Details
A function is named.	Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function.
A function is independent.	A function can perform its task without interference from or interfering with other parts of the program.
A function performs a specific task.	This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.
A function can return a value to the calling program.	When your program calls a function, the statements it contains are executed. These statements, if desired, can pass information back to the calling program.

That's all there is to the "telling" part. Keep the previous definition in mind as you look at the next

section.

Topic 5.1.2: A Function Illustrated

*** Example Program**

The program in Listing 5.1 contains a user-defined function. Click the Listing button.

Listing 5.1: Calculating Cubes

Code

```
/* LIST0501.c: Day 5 Listing 5.1 */
/* Demonstrates a simple function */

#include <stdio.h>

long cube(long x);
long input, answer;

int main(void) {
  printf("Enter an integer value: ");
  scanf("%d", &input);
  answer = cube(input);
  /* Note: %ld is the conversion specifier */
  /* for a long integer. */
  printf("\n\nThe cube of %ld is %ld.",
          input, answer);
  return 0;
}

long cube(long x) {
  long x_cubed;

  x_cubed = x * x * x;
  return x_cubed;
}
```

Output

```
E:\>list0501
Enter an integer value: 100
The cube of 100 is 1000000.
E:\>list0501
Enter an integer value: 9
The cube of 9 is 729.
E:\>list0501
Enter an integer value: 3
The cube of 3 is 27.
```

Description

We're going to concentrate on the components of the program that relate directly to the function rather than explain the entire program.

The Function Prototype

Line 6 contains the *function prototype*, a model for a function that will appear later in the program. A function's prototype contains the name of the function, a list of variables that must be passed to it, and the type of variable it returns, if any.

Looking at line 6 you can tell that the function is named cube(), that it

Looking at line 6 you can tell that the function is named cube(), that it requires a variable of the type long, and that it will return a value of type long. The list of variables to be passed to the function are called arguments and appear between the parentheses following the function's name. In this example, the function's argument is long x. The keyword before the name of the function indicates the type of variable the function returns. In this case, a type long variable is returned.

The Function Call
Line 12 calls the function cube() and passes the variable input to it as the function's argument. The function's return value is assigned to the variable answer. Notice that both input and answer are declared on line 7 as long variables, keeping with the function prototype on line 6.

The Function Definition
The function itself is called the *function definition*. In this case, it's called cube() and is contained on program lines 20 – 24. Like the prototype, the function definition has several parts. The function starts out with a function header on line 20. The function header is at the start of a function and gives the function's name (in this case, the name is cube). The header also gives the function's return type and describes its arguments. Note that the function header is identical to the function prototype (minus the semicolon).

The Function Body
The body of the function, lines 21 – 24, is enclosed in braces. The body contains statements, such as shown on line 23, that are executed whenever the function is called. Line 21 is a variable declaration that looks like the declarations you have seen before, with one difference: it is local. *Local* variables are those that are declared within a function body. (Local declarations are discussed further on Day 12, "Variable Scope.")

Finally, the function concludes with a return statement on line 24, which signals the end of the function. A return statement also passes a value back to the calling program. In this case, the value of the variable x_cubed is returned.

cube() Versus main()
If you compare the structure of the cube() function with that of the main() function, you see that they are the same. main() is also a function. Other functions that you already have used are printf() and scanf(). Although printf() and scanf() are library functions (as opposed to user-defined functions) they are functions that can take arguments and return values just like the functions you create.

Topic 5.2: How a Function Works

*** Operation of Functions**

A C program does not execute the statements in a function until the function is called by another part of the program.

Stage	Description
1	When a function is called, the program can send the function information in the form of one or more arguments. An argument is program data needed by the function to perform its task.
2	The statements in the function then execute, performing whatever task each was designed to do.
3	When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

*** Example**

Figure 5.1 shows a program with three functions, each of which is called once. Each time a function is called, execution passes to that function. When the function is finished, execution passes back to the place from which the function was called. A function can be called as many times as needed, and functions can be called in any order.

You now know what a function is and the importance of functions. Lessons on how to create and use your own functions follow.

Note: Some compilers go against the ANSI standard and fail to initialize global variables to 0.

Topic 5.3: Functions and Structured Programming

*** Structured Programming**

By using functions in your C programs, you can practice *structured programming* in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

Topic 5.3.1: The Advantages of Structured Programming

*** Easier to Write**

Why is structured programming so great? There are two important reasons:

It's easier to write a structured program because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can make progress faster dealing one at a time with these relatively simple tasks.

*** Easier to Debug**

It's easier to debug a structured program. If your program has a *bug* (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

*** Saves You Time**

A related advantage of structured programming is the time you can save. If you write a function to

A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you quickly and easily can use it in another program that needs to execute the same task. Even if the new program needs to accomplish a slightly different task, you often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions printf() and scanf() even though you probably haven't seen the code they contain. If your functions have been created to do a single task, using them in other programs is much easier.

Topic 5.3.2: Planning a Structured Program

*** List the Specific Tasks**
If you're going to write a structured program, you need to do some planning first. This planning should take place before you write a single line of code, and usually can be done with nothing more than pencil and paper. Your plan should be a list of the specific tasks that your program performs. Begin with a global idea of the program's function. Click the Example link.
Example
*** Divide Tasks into Subtasks**
Now you can go a step further, dividing these tasks into subtasks.
Example

If you were planning a program to manage your name and address list, what would you want the program to do? Here are some obvious things:
Enter new names and addresses.
Modify existing entries.
Sort entries by last name.
Print mailing labels.
With this list, you've divided the program into four main tasks, each of which can be assigned to a function.

For example, the "Enter new names and addresses" task can be subdivided into these subtasks:
Read the existing address list from disk.
Prompt the user for one or more new entries.
Add the new data to the list.
Save the updated list to disk.
Likewise, the "Modify existing entries" task can be subdivided as follows:
Read the existing address list from disk.
Modify one or more entries.
Save the updated list to disk.

*** Look for Common Tasks**
You might have noticed that these two lists have two subtasks in common—the ones dealing with reading from and saving to disk. You can write one function to "Read the existing address list from disk," and that function can be called by both the "Enter new names and addresses" function and the "Modify existing entries" function. The same is true for "Save the updated list to disk."

*** Write "Double-Duty" Functions**

*** Write "Double-Duty" Functions**

Already, you should see at least one advantage of structured programming. By carefully dividing the program into tasks, you can identify parts of the program that share common tasks. You can write "double-duty" disk access functions, saving yourself time and making your program smaller and more efficient.

*** Summary**

This method of programming results in a hierarchical, or layered, program structure. Figure 5.2 illustrates hierarchical programming for the address list program. A diagram like this of the way a program needs to work is called an *algorithm*.

When you follow this planned approach, you quickly make a list of discrete tasks that your program needs to perform. Then you can tackle the tasks one at a time, giving all your attention to one relatively simple task. When that function is written and working properly, you can move on to the next task. Before you know it, your program starts to take shape.

Topic 5.3.3: The Top-Down Approach

*** Real Work Performed by Functions**

By using structured programming, C programmers take the *top-down approach*. You saw this illustrated in Figure 5.2 where the program's structure resembles an inverted tree. Many times, most of the real work of the program is performed by the functions at the "tips of the branches." The functions closer to the "trunk" primarily direct program execution among these functions.

*** Result: main() is Small**

As a result, many C programs have a small amount of code in the main body of the program, that is, in main(). The bulk of the program's code is found in functions. In main(), all you may find is a few dozen lines of code that direct program execution among the functions. Often, a menu is presented to the person using the program, with program execution branched according to the user's choices.

This is a good approach to program design. Day 13, "More Program Control," shows how you can use the switch statement to create a versatile menu-driven system.

Now that you know what functions are and why they're so important, the time has come for you to learn how to write your own functions. Click the Tip button on the toolbar.

DO plan before starting to code. By determining your program's structure ahead of time, you can save time writing the code and debugging it.

DON'T try to do everything in one function. A single function should do a single task, such as reading information from a file.

Topic 5.4: Writing a Function

*** Writing a Function**

The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function are not particularly difficult.

Topic 5.4.1: The Function Header

*** Function Header**
The first line of every function is the function header.
*** No Semicolon**
You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler generates an error message.
*** Three Components**
A function header has three components, each serving a specific function. They are diagrammed in Figure 5.3 and explained on the following pages.

*** The Function Return Type**
The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value. This type of function has the return type `void`. Here are some examples:

```
int func1(...)          /* Returns a type int.   */
float func2(...)        /* Returns a type float. */
void func3(...)         /* Returns nothing.      */
```

*** The Function Name**
You can name a function anything you like, as long as you follow the rules for C variable names (given in Day 3, "Numeric Variables and Constants"). A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.
*** The Parameter List**
Many functions use *arguments,* which are values passed to the function when it is called. A function needs to know what kinds of arguments to expect—the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list.
For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter.

*** Example: One Parameter**
For example, here's the header from the function in Listing 5.1:
`long cube(long x)`
The parameter list reads long x, specifying that this function takes one type long argument, represented by the parameter x.

*** Example: More Than One Parameter**
If there is more than one parameter, each must be separated by a comma. The function header
`void func1(int x, float y, char z)`
specifies a function with three arguments: a type int named x, a type float named y, and a type char named z.

*** Example: No Parameters**
Some functions take no arguments, in which case, the parameter list should read void:
`void func2(void)`

*** Arguments Versus Parameters**

Sometimes confusion arises about the distinction between a parameter and an argument. A *parameter* is an entry in a function header; it serves as a "place holder" for an argument. A function's parameters are fixed; they do not change during program execution.

An argument is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. A function must be passed the same number and type of arguments each time it is called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name. Click the Tip button on the toolbar.

DO use a function name that describes the purpose of the function.
DON'T pass values to a function that it doesn't need.
DON'T try to pass fewer (or more) arguments to a function than there are parameters!

*** Example Program**

An example makes this clearer. Listing 5.2 presents a very simple program with one function that is called twice. Click the Listing button.

*** Schematic Representation**

Figure 5.4 shows the relationship between arguments and parameters schematically.

Listing 5.2: Difference Between Arguments and Parameters

Code

```
 1: /* LIST0502.c: Day 5 Listing 5.2 */
 2: /* Illustrates the difference between arguments */
 3: /* and parameters. */
 4:
 5: #include <stdio.h>
 6:
 7: float x = 3.5F, y = 65.11F, z;
 8: float half_of(float k);
 9:
10: int main(void) {
11:
12:   /* In this call, x is argument to half_of(). */
13:   z = half_of(x);
14:   printf("The value of z = %f\n", z);
15:
16:   /* In this call, y is argument to half_of(). */
17:   z = half_of(y);
18:   printf("The value of z = %f\n", z);
19:   return 0;
20: }
21:
22: float half_of(float k) {
23:   /* k is the parameter.  Each time half_of() */
24:   /* is called, k has the value that was      */
25:   /* passed as an argument.                   */
26:   return (k/2);
27: }
```

Output

```
The value of z = 1.750000
The value of z = 32.555000
```

Description Looking at Listing 5.2, you can see that the function half_of() prototype is

Description	Looking at Listing 5.2, you can see that the function half_of() prototype is declared on line 8. Lines 13 and 17 call half_of() and lines 22 – 27 contain the actual function. Lines 13 and 17 each send a different argument to half_of(). Line 13 sends x, which contains a value of 3.5, and line 17 sends y, which contains a value of 65.11. When the program runs, it prints the correct number for each. The values in x and y are passed into the argument k of half_of(). This is like copying the values from x to k, and then y to k. half_of() then returns this value after dividing it by 2 (line 26).

Topic 5.4.2: The Function Body

*** Components**
The *function body* is enclosed in braces and follows immediately after the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

*** Local Variables**
You can declare variables within the body of a function. Variables declared in a function are called *local variables*. The term *local* means the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This is explained shortly; for now, you should learn how to declare local variables.
*** Declaring Local Variables**
A local variable is declared like any other variable, using the same variable types and rules for names that you learned on Day 3, "Numeric Variables and Constants." Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function.
Example

Here are some examples:

```
int func1(int y) {
    int a, b = 10;
    float rate;
    double cost = 12.55;
...
}
```

The preceding declarations create local variables a, b, rate, and cost that can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list are also available.

*** Local Variables Are Independent**
When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name. The program in Listing 5.3 demonstrates this independence. Click the Listing button.

Listing 5.3: Demonstration of Local Variables

```
Code  1: /* LIST0503.c: Day 5 Listing 5.3 */
      2: /* Demonstrates local variables. */
      3:
      4: #include <stdio.h>
      5: /* Declare and initialize two global variables. */
      6: int x = 1, y = 2;
      7:
      8: void demo(void);
      9:
     10: int main(void) {
     11:   printf("\nBefore calling demo(),"
     12:          "x = %d and y = %d.", x, y);
     13:   demo();
     14:   printf("\nAfter calling demo(),"
     15:          "x = %d and y = %d.", x, y);
     16:   return 0;
     17: }
     18:
     19: void demo(void) {
     20:   /* Declare and initialize two local variables.
     21:    * with the same names as the global variables. */
     22:   int x = 88, y = 99;
     23:   /* Display their values. The local variables are displayed.*/
     24:   printf("\nWithin demo(), x = %d and y = %d.",
     25:          x, y);
           }
```

Output

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

Description

Listing 5.3 is similar to the previous programs in this unit. Line 6 declares variables x and y. These are declared outside of any functions and therefore are considered global.

Line 8 contains the prototype for our demonstration function, named demo(). It is a function that does not take any parameters, and therefore has void in the prototype. It also does not return any values, giving it a type of void.

Line 10 starts our main() function, which is very simple. First, printf() is called on line 11 to display the values of x and y, and then the demo() function is called.

Notice that demo() declares its own local versions of x and y on line 21. Line 23 shows that the local variables take precedence over any others.

After the demo function is called, line 14 again prints the values of x and y. Because you are no longer in demo(), the original global values are printed.

* Rules for Use of Variables in Functions

As you can see in Listing 5.3, local variables x and y in the function are totally independent from the global variables x and y declared outside the function. Three rules govern the use of variables in functions.

- To use a variable in a function, you must declare it in the function header or the function body (except for global variables, which are covered on Day 12, "Variable Scope").
- For a function to obtain a value from the calling program, the value must be passed as an argument.
- For a calling program to obtain a value from a function, the value must be explicitly returned from the function.

To be honest, these "rules" are not strictly applied, because you learn how to get around them later in the course. However, follow these rules for now, and you should stay out of trouble!

*** Summary**
Keeping the function's variables separate from other program variables is one way in which functions are independent. A function can perform any sort of data manipulation you want, using its own set of local variables. There's no worry that these manipulations have an unintended effect on another part of the program.

*** Permitted Statements**
There is essentially no limitation on the statements that can be included within a function. The only thing you can't do inside a function is define another function. You can, however, use all other C statements, including loops (these are covered on Day 6, "Basic Program Control"), if statements, and assignment statements. You can call library functions and other user-defined functions.
*** Permitted Length**
C places no length restriction on functions, however, for practical purposes, keep your functions short. In structured programming, each function is supposed to perform a relatively simple task. If you are trying to perform a task too complex for one function, break it into two or more smaller functions.

*** How Long Is Too Long?**
There's no definite answer to that question, but in practical experience it's rare that you find a function longer than 25 to 30 lines of code. Use your own judgment. Some programming tasks require longer functions, whereas many functions are only a few lines. As you gain programming experience, you will become more adept at determining what should and should not be broken into smaller functions.

*** return Keyword**
To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression.
Example

*** Multiple return Statements**

A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements are an efficient way to return different values from a function. Look at the example in Listing 5.4. Also click the Tip button.

DON'T try to return a value that has a different type than the function's type.
DO use local variables whenever possible.
DON'T let functions get too long. If a function starts getting long, try to break it into separate, smaller tasks.
DO limit each function to a single task.
DON'T have multiple return statements if they are not needed. You should try to have one return when possible; however, sometimes having multiple return statements is easier and clearer.

Listing 5.4: Using Multiple Return Statements in a Function

Code

```
 1: /* LIST0504.c: Day 5 Listing 5.4 */
 2: /* Demonstrates using multiple return */
 3: /* statements in a function. */
 4:
 5: #include <stdio.h>
 6:
 7: int x, y, z;
 8: int larger_of(int a, int b);
 9:
10: int main(void) {
11:
12:   puts("Enter two different integer values: ");
13:   scanf("%d%d", &x, &y);
14:
15:   z = larger_of(x, y);
16:
17:     printf("\nThe larger value is %d.", z);
18:     return 0;
19: }
20:
21:     int larger_of(int a, int b) {
22:         if (a > b)
23:             return a;
24:         else
25:             return b;
26: }
```

Output

```
E:\>list0504
Enter two different integer values:
200 300

The larger value is 300.

E:\>list0504

Enter two different integer values:
300
200

The larger value is 300.
```

Description	As in other examples, Listing 5.4 starts with a comment to describe what the program does (lines 1 – 3). The STDIO.H header file is included for the standard input/output functions that allow the program to display information to the screen and get user input. Line 8 is the function prototype for larger_of(). Notice that it takes two int variables for parameters and returns an int. Line 15 calls larger_of() with x and y. The function larger_of() has the multiple return statements. Using an if statement, the function checks to see whether a is bigger than b on line 22. If it is, line 23 executes a return statement and the function immediately ends. Lines 24 and 25 are ignored in this case. If a is not bigger than b, line 23 is skipped, the else clause is instigated, and the return on line 25 executes. You should be able to see that, depending on the arguments passed to the function larger_of(), either the first or the second return statement is executed, and the appropriate value is passed back to the calling function. One final note on this program. Line 12 is a new function that you have not seen before. puts()—read *put string*—is a simple function that displays a string to the standard output, usually the computer screen. (Strings are covered on Day 10, "Characters and Strings"; for now, know that they are just quoted text). Remember that a function's return value has a type that is specified in the function header and function prototype. The value returned by the function must be of the same type or the compiler generates an error message.

Look at this function:

```
int func1(int var)
{
    int x;
    ...
    ...
    return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

Topic 5.4.3: The Function Prototype

*** Prototypes are Required**

A program must include a prototype for each function that it uses. You saw an example of a

A program must include a prototype for each function that it uses. You saw an example of a function prototype on line 6 of Listing 5.1, and there have been function prototypes in the other listings as well. What is a function prototype, and why is it needed?

*** The Prototype's Format**

You can see from the earlier examples that the prototype for a function is identical to the function header, with a semicolon added at the end. Like the function header, the function prototype includes information about the function's return type, name, and parameters.

*** The Prototype's Job**

The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code calls the function and verify that you are passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message.

*** Should the Prototype Match the Function Header?**

Strictly speaking, a function prototype need not exactly match the function header. The parameter names can be different, as long as they are the same type, number, and in the same order. There's no reason for the header and prototype not to match; having them identical makes source code easier to understand. Matching the two also makes writing a program easier. When you complete a function definition, use your editor's cut-and-paste feature to copy the function header and create the prototype. Be sure to add a semicolon at the end.

*** The Prototype's Location**

Where should function prototypes be placed in your source code? They must be placed before the start of main() or before the first function is defined. For readability, it's best to group all prototypes together in one location.

Topic 5.5: Passing Arguments to a Function

*** Number and Type Must Match**

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype. For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments—no more, no less—and no other type. If you try to pass a function an incorrect number and/or type of arguments, the compiler detects it, based on the information in the function prototype.

*** Arguments Assigned to Parameters in Order**

If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on, as illustrated in Figure 5.5

*** An Argument Can Be Any Valid C Expression**

Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value). Click the Example link to see a

complex example.
Example

If half(), square(), and third() are all functions with return values, you could write

```
x = half(third(square(half(y))));
```

Step	Action
1	The program first calls half(), passing it y as an argument.
2	When execution returns from half(), the program calls square(), passing half()'s return value as an argument.
3	Next, third() is called with square()'s return value as the argument.
4	Then, half() is called again, this time with third()'s return value as an argument.
5	Finally, half()'s return value is assigned to the variable x.

The following is an equivalent piece of code:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

Topic 5.6: Calling Functions

*** Discarding the Return Value**

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement. If the function has a return value, it is discarded. For example,

```
wait(12);
```

*** Using the Return Value**

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. If you try to use a function with a void return type as an expression, the compiler generates an error message.

You've already seen an expression with a return value used as the right side of an assignment statement. Here are some other examples.

Example1
Example2
Example3
Example4

Now click the Tip button on the toolbar.

DO pass parameters to functions in order to make the function generic and thus reusable!
DO take advantage of the ability to put functions into expressions.
DON'T make an individual statement confusing by putting a bunch of functions in it. Only put functions into your statements if they don't make the code more confusing.

```
printf("Half of %d is %d.", x, half_of(x));
```

In this example, half_of() is a parameter of a function. First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

```
y = half_of(x) + half_of(z);
```

For this second example, multiple functions are being used in an expression. Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line.

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions:

```
if ( half_of(x) > 10 ) {
  /* Any statement can go here. */
     statement(s)
}
```

Here a function is being used with the if statement. If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned value does not meet the criteria, the if's statements are not executed. The next example is even better.

```
if ( do_a_process() != OKAY ) {
     statements                /* do error routine */
}
```

Again, you don't see the actual statements, nor is do_a_process() a real function; however, this is an important example that checks the return value of a process to see whether it did not run all right. If it didn't, the statements take care of any error handling or cleanup. This is used commonly with accessing information in files, comparing values, and allocating memory.

Topic 5.6.1: Recursion

*** Recursion**

The term *recursion* refers to a situation where a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations. Click the Example link and then the Tip button.

Example

DO understand and work with recursion before you use it.

DON'T use recursion if there will be several iterations. (An iteration is the repetition of a program statement.) Recursion uses many resources because the function has to remember where it is.

Recursion can be used to calculate the factorial of a number. The factorial of a number x is written x!, and is calculated as follows:

```
x! = x * (x-1) * (x-2) * (x-3) .... * (2) * 1
```

However, you also can calculate x! as follows:

```
x! = x * (x-1)!
```

Going one step further, you can calculate (x-1)! by the same procedure:

```
(x-1)! = (x-1) * (x-2)!
```

You can continue calculating recursively until you're down to a value of 1, in which case, you're finished.

*** Example Program**

The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers. Click the Listing button.

Listing 5.5: Use of a Recursive Function to Calculate Factorials

Code

```
/* LIST0505.c: Day 5 Listing 5.5 */
/* Demonstrates function recursion. */
/* Calculates the factorial of a number. */

#include <stdio.h>

unsigned int f,x;
unsigned int factorial(unsigned int a);

int main(void) {
  puts("Enter an integer value between 1 and 8: ");
  scanf("%d", &x);

  if (x > 8 "" x < 1) {
    printf("Only values from 1 to 8 are"
           " acceptable!");
  }
  else {
    f = factorial(x);
    printf("%u factorial equals %u", x, f);
  }
  return 0;
}

unsigned int factorial(unsigned int a) {
  if (a == 1)
    return 1;
  else {
    a *= factorial(a-1);
    return a;
  }
}
```

Output

```
Enter an integer value between 1 and 8: 6
6 factorial equals 720
```

Description The first half of this program is like many of the other programs you have

Description	The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 – 3. On line 4, the appropriate header file is included for the input/output routines. Line 7 declares a couple of unsigned integer values. Line 8 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 10 – 23 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8, and then accept an entered value. Lines 14 – 21 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it is greater than 8, an error message is printed; otherwise, the program figures the factorial on line 19 and prints the result on line 20. When you know there could be a problem, such as a limit on the size of a number, add code like this to detect the problem and prevent it. Our recursive function, factorial(), is located on lines 25 – 31. The value passed is assigned a. On line 26, the value of a is checked. If it is 1, the program returns the value of 1. If the value is not 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) is not equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 26 is true. If the value of the factorial is 3, the factorial is evaluated to the following: `3 * (3-1) * ((3-1)-1)`

Topic 5.7: Where to Put Functions

*** Location of Function Definitions**

You may be wondering where in your source code you should place your function definitions. For now, they should go in the same source code file as main() and after the end of main(). The basic structure of a program that uses functions is shown in Figure 5.6

But as you learn more, you will probably want to keep your user-defined functions in a separate source-code file, apart from main(). This technique is useful with large programs, and when you want to use the same set of functions in more than one program. This technique is discussed on Day 21, "Taking Advantage of Preprocessor Directives and More."

Topic 5.8: Day 5 Q&A

*** Questions & Answers**

Here are some questions to help you review what you have learned in this unit.

Here are some questions to help you review what you have learned in this unit.

*** Question 1**
What if I need to return more than one value from a function?
*** Answer**
Many times you will need to return more than one value from a function, or more commonly, you will want to change a value you send to the function and keep the change after the function ends. This is covered on Day 18, "Getting More from Functions."

*** Question 2**
How do I know what a good function name is?
*** Answer**
A good function name describes as specifically as possible what the function does.

*** Question 3**
When variables are declared at the top of the listing, before main(), they can be used anywhere, but local variables can only be used in the specific function. Why not just declare everything before main()?
*** Answer**
On Day 12, "Variable Scope," variable scope is discussed in more detail.

*** Question 4**
What other ways are there to use recursion?
*** Answer**
The factorial function is a prime example of using recursion. In many statistical calculations, the factorial number is needed. Recursion is just a loop; however, it has one difference from other loops. With recursion, each time a recursed function is called, a new set of variables is created. This is not true of the other loops that you will learn about in the next unit.

*** Question 5**
Does main() have to be the first function in a program?
*** Answer**
No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it is easy to locate.

Topic 5.9: Day 5 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
01: /* EXER0504.c: Day 5 Exercise 5.4 */
02: #include <stdio.h>
03:
04: void print_msg(void);
05:
06: int main(void) {
07:   print_msg("This is a message to print");
08:   return 0;
09: }
10:
11: void print_msg(void) {
12: puts("This is a message to print");
13:   return 0;
14: }
```

```
int product(int x, int y) {
return (x * y);
}
```

Topic 5.10: Day 5 Try Its

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write a header for a function named do_it() that takes three type char arguments and returns a type float to the calling program.
Answer

```
float do_it(char a, char b, char c)
```
Add a semicolon to the end, and you have the function prototype. As a function header, it should be followed by the function's statements enclosed by braces.

be followed by the function's statements enclosed by braces.

*** Exercise 2**

BUG BUSTER: What is wrong with the following function definition?

```
int twice(int y);
{
    return (2 * y);
}
```

Answer

There should not be a semicolon at the end of the function header.

*** Exercise 3**

Rewrite Listing 5.4 so that it needs only one return statement.

Answer

Listing 5.4: Using Multiple Return Statements in a Function

```
Code  1:  /* LIST0504.c: Day 5 Listing 5.4 */
      2:  /* Demonstrates using multiple return */
      3:  /* statements in a function. */
      4:
      5:  #include <stdio.h>
      6:
      7:  int x, y, z;
      8:  int larger_of(int a, int b);
      9:
     10:  int main(void) {
     11:
     12:    puts("Enter two different integer values: ");
     13:    scanf("%d%d", &x, &y);
     14:
     15:    z = larger_of(x, y);
     16:
     17:      printf("\nThe larger value is %d.", z);
     18:      return 0;
     19:  }
     20:
     21:      int larger_of(int a, int b) {
     22:          if (a > b)
     23:              return a;
     24:          else
     25:              return b;
     26:  }
```

Only the larger_of() function needs to be changed:

```
21: int larger_of(int a, int b) {
```

```
21: int larger_of(int a, int b) {
22:   int save;
23:     if (a > b)
24:         save = a;
25:     else
26:         save = b;
27:     return save;
28: }
```

*** Exercise 4**
Write a function that receives two numbers as arguments and returns the value of their product.
Answer

The following assumes that two values are integers, and an integer is returned:

```
int product(int x, int y) {
return (x * y);
}
```

*** Exercise 5**
Write a function that receives two numbers as arguments. The function should divide the first number by the second. Don't divide by the second number if it is zero. (Hint: use an if statement.)
Answer

This listing asks you to check the values passed. You should never assume that the values passed are correct.

```
int divide_em(int a, int b) {
  int answer = 0;
  if (b == 0)
    answer = 0;
  else
    answer = a/b;
  return answer;
}
```

*** Exercise 6**
Write a function that calls the functions in exercises four and five.
Answer

Although the following uses main(), it could be any function. Lines 12, 13, and 14 show the calls to the two functions. Lines 16 – 20 print the values.

```
01: /* EXER0506.c: Day 5 Exercise 5.6 */
02:
03: #include <stdio.h>
04:
05: int product(int x, int y);
06: int divide_em(int a, int b);
```

```
06: int divide_em(int a, int b);
07:
08: int main(void) {
09:   int number1 = 10, number2 = 5;
10:   int x, y, z;
11:
12:   x = product(number1, number2);
13:   y = divide_em(number1, number2);
14:   z = divide_em(number1, 0);
15:
16:   printf("\nnumber1 is %d and number2 is %d", number1,
17:           number2);
18:   printf("\nnumber1 * number2 is %d", x);
19:   printf("\nnumber1 / number2 is %d", y);
20:   printf("\nnumber1 / 0 is %d", z);
21:   return 0;
22: }
23:
24:   int product(int x, int y) {
25:     return (x * y);
26:     }
27:
28:   int divide_em(int a, int b) {
29:     int answer = 0;
30:     if (b == 0)
31:       answer = 0;
32:     else
33:       answer = a/b;
34:     return answer;
35:     }
```

*** Exercise 7**

Write a program that uses a function to find the average of five type float values entered by the user.

Answer

```
/* EXER057.c: Day 5 Exercise 5.7 */
/* Averages five float values entered by the user. */
#include <stdio.h>
float v, w, x, y, z, answer;
float average(float a, float b, float c,
              float d, float e);
int main(void) {
  puts("Enter five numbers: ");
  scanf("%f%f%f%f%f", &v, &w, &x, &y, &z);

  answer = average(v, w, x, y, z);
  printf("\nThe average is %f.", answer);
  return 0;
}
float average(float a, float b, float c, float d, float e) {
  return ((a+b+c+d+e)/5);
}
```

*** Exercise 8**

Exercise 8

Write a recursive function to take the value 3 to the power of another number. For example, if 4 is passed, the function will return 81.

Answer

The following is the answer using type int variables. It can only run with values less than or equal to 9. To use values larger than 9, you need to change the values to type long.

```
/* EXER058.c: Day 5 Exercise 5.8 */
/* This is a program with a recursive function. */
#include <stdio.h>
int three_powered(int power);
int main(void) {
  int a = 4;
  int b = 9;

  printf("\n3 to the power of %d is %d", a,
         three_powered(a));
  printf("\n3 to the power of %d is %d", b,
         three_powered(b));
  return 0;
}
int three_powered(int power) {
  if (power < 1)
    return(1);
  else
    return(3 * three_powered(power - 1));
}
```

Topic 5.11: Day 5 Summary

What a Function Is

This unit introduced you to functions, an important part of C programming. Functions are independent sections of code that perform specific tasks. When your program needs a task performed, it calls the function that performs that task.

Advantages of Structured Programming with Functions

The use of functions is essential for structured programming—a method of program design that emphasizes a modular, top-down approach. Structured programming creates more efficient programs and also is much easier for you, the programmer, to use.

Parts of a Function

You learned, too, that a function consists of a header and body. The header includes information about the function's return type, name, and parameters. The body contains local variable declarations and the C statements that are executed when the function is called.

Independence of Local Variables

Finally, you saw that local variables—those declared within a function—are totally independent from any other program variables declared elsewhere.

Unit 6. Day 6 Basic Program Control

Day 4, "Statements, Expressions, and Operators," covered the if statement, which gives you some control over the flow of your programs. Many times, though, you need more than just the ability to make true and false decisions. This unit introduces three new ways to control the flow of the program.

Today you learn

- How to use simple arrays.
- How to use for, while, and do...while loops to execute statements multiple times.
- How you can nest program control statements: nesting for statements, nesting while statements and nested loops.

*** More Details**
This unit is not intended to be a complete treatment of these topics, but we hope to provide enough information for you to be able to start writing real programs. These topics are covered in greater detail on Day 13, "More Program Control."

Topic 6.1: Arrays: The Basics

*** The for Statement and Arrays**
Before we cover the for statement, let's take a short detour and learn the basics of arrays. (See Day 8, "Numeric Arrays," for a complete treatment of arrays.) The for statement and arrays are closely linked in C, so it is difficult to define one without explaining the other. To help you understand the arrays used in the for statement examples to come, a quick treatment of arrays follows.

*** Arrays**

So far the data types and variables we've studied have held only a single value. An *array* is simply a collection of values. The technical definition of an array would be something like "an indexed group of data storage locations that have the same name and are distinguished from each other by a *subscript*, or *index*—a number following the variable name, enclosed in brackets." This definition will become clearer as you continue

will become clearer as you continue.

*** Array Declarations**
Like other C variables, arrays must be declared. An array declaration includes both the data type and the size of the array (the number of elements in the array). Click the Example link and then the Tip button.
Example

DON'T declare arrays with subscripts larger than you will need; it wastes memory.
DON'T forget that in C, arrays are referenced starting with subscript 0, not 1.

The statement
```
int data[1000];
```
declares an array named data that is type int and has 1,000 elements.

*** Array Elements**
The individual elements are referred to by subscript as `data[0]` through `data[999]`. The first element is `data[0]`—not `data[1]`. (In a few other languages, such as BASIC, the first element of an array is 1.)
Each element of this array is equivalent to a normal integer variable and can be used the same way. The subscript of an array can be another C variable, such as in this example:
Example

```
int data[1000];
int count;
count = 100;
data[count] = 12;      /* The same as data[100] = 12 */
```

*** More Details**
This has been a quick introduction to arrays. However, you now should be able to understand how arrays are used in the program examples later in this unit. If every detail of arrays is not clear to you, don't worry. You can find more about arrays on Day 8, "Numeric Arrays."

Topic 6.2: Controlling Program Execution

*** Program Control Statements**
The default order of execution in a C program is top-down. Execution starts at the beginning of the main() function and progresses, statement by statement, until the end of main() is reached. However, this order is rarely encountered in real C programs.
The C language includes a variety of program control statements that enable you to control the order of program execution. You have already learned how to use C's fundamental decision operator, the if statement, so let's explore three additional control statements you will find useful.

Topic 6.2.1: The for Statement

*** The for Loop**

The for statement is a C programming construct that executes a block of one or more statements a certain number of times. It is sometimes called the *for loop* because program execution typically loops through the statement more than one time. You've seen a few for statements used in programming examples earlier in this course. Now you're ready to see how the for statement works.

*** Syntax**

A for statement has the following structure:

```
for (start; test; increment)
    statements
```

*** Parameters**

`start`, `test`, and `increment` are all C expressions, and `statements` is a single or compound C statement.

*** Process**

```
for (start; test; increment)
    statements
```

When a for statement is encountered during program execution, the following events occur:

Stage	Description
1	The expression `start` is evaluated. `start` is usually an assignment statement that sets a variable to a particular value, for example `i=1`.
2	The expression `test` is evaluated. `test` is typically a relational (conditional) expression, for example, `i<=100`.
3	If `test` evaluates as false (that is, as zero), the for statement terminates, and execution passes to the first statement following statements.
4	If `test` evaluates as true (that is, as nonzero), the C statement(s) in `statements` are executed.
5	Finally, the expression `increment` is evaluated. As you might guess, `increment` usually changes the value of the variable used in the `start` and `test` expressions, for example `i++`. Then, execution returns to Step 2 with the new variable value.

*** Schematic Representation**

The operation of a for statement is shown schematically in Figure 6.1. Please note that statements never execute if test is false the first time it is evaluated.

Here is a simple example. The program in Listing 6.1 uses a for statement to print the numbers 1 through 20. You can see that the resulting code is much more compact than it would be if a separate printf() statement were used for each of the 20 values. Click the Listing button.

Listing 6.1: Demonstration of a Simple for Statement

```
Code  1:  /* LIST0601.c: Day 6 Listing 6.1 */
      2:  /* Demonstrates a simple for statement */
      3:
      4:  #include <stdio.h>
      5:
      6:  int count;
```

```
 7:
 8: int main(void) {
 9:   /* Print the numbers 1 through 20. */
10:
11:   for (count = 1; count <= 20; count++)
12:     printf("\n%d", count);
13:   return 0;
14: }
```

Output

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Description

The diagram in Figure 6.2 illustrates the operation of the for loop in Listing 6.1.

Line 4 includes the standard input/output header file. Line 6 declares a type int variable, named count, that will be used in the for loop.

Lines 11 and 12 are the for loop. When the for statement is reached, the initial statement is executed first. In this listing, the initial statement is `count = 1`. This initializes count so that it can be used by the rest of the loop.

The second step in executing this for statement is the evaluation of the condition `count <= 20`. Because count was just initialized to 1, you know that it is less than 20, so the statement in the for command, the printf(), is executed. After executing the printing function, the increment expression, `count++`, is evaluated. This adds 1 to count, making it 2.

Next, the program loops back and checks the condition again. If it is true, the printf() re-executes, the increment adds to count (making it 3), and the condition is checked. This loop continues until the condition evaluates to false, at which point the program exits the loop and continues to the next line (line 13), which in this listing is the return statement.

* The Counter Variable

The for statement is frequently used, as in the previous example, to "count up," incrementing a

The for statement is frequently used, as in the previous example, to "count up," incrementing a counter from one value to another. You also can use it to "count down," decrementing, rather than incrementing, the counter variable.

```
for (count = 100; count > 0; count--)
```

You also can "count by" a value other than 1.

```
for (count = 0; count < 1000; count += 5)
```

*** Omitting the Initialization (start) Expression**

The for statement is quite flexible. For example, you can omit the initialization expression if the test variable has been initialized previously in your program. (You must still use the semicolon separator, however, as shown.)

```
count = 1;
for (; count < 1000; count++)
```

The initialization expression need not be an actual initialization; it can be any valid C expression. Whatever it is, it is executed once when the for statement is first reached. Click the Example link to see an example of an initialization expression that does not initialize at all and is a print statement.

Example

The following prints a statement, `"Now sorting the array..."`

```
count = 1;
for (printf("Now sorting the array...") ; count < 1000;
     count++)
    /* Sorting statements here */
```

*** Omitting the increment Expression**

You can also omit the increment expression, performing the updating in the body of the for statement. The semicolon, again, must be included.

Example

To print the numbers from 0 to 99, you could write

```
for (count = 0; count < 100;)
printf("%d", count++);
```

*** The test Expression**

The test expression that terminates the loop can be any C expression. As long as it evaluates as true (nonzero), the for statement continues to execute. You can use C's logical operators to construct complex test expressions.

Example

*** Using a Null Statement**

You can follow the for statement with a null statement, enabling all the work to be done in the for statement itself. Remember, the null statement is a semicolon alone on a line.

Example

The following for statement prints the elements of an array named `array[]`, stopping when all elements have been printed or an element with a value of 0 is encountered.

```
for (count = 0; count < 1000 && array[count] != 0; count++)
    printf("%d", array[count]);
```

You could simplify the previous for loop even further, writing it as follows. (If you don't understand the change made to the test expression, you need to review Day 4, "Statements, Expressions, and Operators.")

```
for (count = 0; count < 1000 && array[count]; count++)
printf("%d", array[count]);
```

To initialize all elements of a 1,000-element array to the value 50, you could write

```
for (count = 0; count < 1000; array[count++] = 50)
;
```

In this for statement, 50 is assigned to each member of the array by the increment part of the statement.

*** Using the Comma Operator**

Day 4, "Statements, Expressions, and Operators," mentioned that C's comma operator is most often used in for statements. You can create an expression by separating two subexpressions with the comma operator. The two subexpressions are evaluated (in left-to-right order), and the entire expression evaluates to the value of the right subexpression. By using the comma operator, you can make each part of a for statement perform multiple duty.

Example

Imagine that you have two 1,000-element arrays, `a[]` and `b[]`. You want to copy the contents of `a[]` to `b[]` in reverse order so that after the copy operation, `b[0] = a[999]`, `b[1] = a[998]`, and so on. The following for statement does the trick:

```
for (i = 0, j = 999; i < 1000; i++, j--)
    b[j] = a[i];
```

The comma operator is used to initialize two variables, i and j. It also is used to increment part of these two variables with each loop.

Topic 6.2.2: Nesting for Statements

*** Nesting**

A for statement can be executed within another for statement. This is called *nesting*. (You saw this on Day 4, "Statements, Expressions, and Operators," with the if statement.) By nesting for statements, some complex programming can be done.

Listing 6.2 is not a complex program, but it illustrates the nesting of two for statements. Click the Listing button and then the Tip button.

DON'T put too much processing in the for statement. Although you can use the comma separator, it is often clearer to put some of the functionality into the body of the loop.

DO remember the semicolon if you use a for with a null statement. Put the semicolon placeholder on a separate line or place a space between it and the end of the for statement.

```
for(count = 0; count < 1000; array[count] = 50) ;
/* note space! */
```

Listing 6.2: Demonstration of Nested for Statements

Code

```
 1: /* LIST0602.c: Day 6 Listing 6.2 */
 2: /* Demonstrates nesting two for statements */
 3:
 4: #include <stdio.h>
 5:
 6: void draw_box(int row, int column);
 7:
 8: int main(void) {
 9:   draw_box(8, 35);
10:   return 0;
11: }
12:
13: void draw_box(int row, int column) {
14:   int col;
15:   for(; row > 0; row--) {
16:     for(col = column; col > 0; col--)
17:       printf("X");
18:
19:     printf("\n");
20:   }
21: }
```

Output

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Description

The main work of this program is accomplished on line 17. When you run this program, 280 Xs are printed on the screen, forming an 8 by 35 square. The program has only one command to print an X, but it is nested in two loops.

In this listing, a function prototype for draw_box() is declared on line 6. This function takes two type int variables, row and column, which contain the dimensions of the box of Xs to be drawn. In line 9, main() calls draw_box() and passes the value 8 as the row and the value 35 as the column.

Looking closely at the draw_box() function, you might see a couple things you don't readily understand. The first is why the local variable col was declared. The second is why the second printf() in line 19 was used. Both of these will become clearer after looking at the two for loops.

Line 15 starts the first for loop. The initialization is skipped because the initial value of row was passed to the function. Looking at the condition, you see that this for loop is executed until the row is 0. On first executing line 15, row is 8; therefore, the program continues to line 16.

Line 16 contains the second for statement. Here the passed parameter, column, is copied to a local variable, col, of type int. The value of col is 35 initially (the value passed via column), and column retains its original value. Because col is greater than 0, line 17 is executed, printing an X. col is then

	decremented and the loop continues. When col is 0, the for loop ends and control goes to line 19. Line 19 causes the printing on the screen to start on a new line. (On Day 7, "Basic Input/Output," printing is covered in detail). After moving to a new line on the screen, control reaches the end of the first for loop's statements, thus executing the increment expression, which subtracts 1 from row, making it 7. This puts control back at line 16. Notice that the value of col was 0 when last used. If column had been used instead of col, it would fail the condition test because it will never be greater than 0. Only the first line would be printed. Take the initializer out of line 16 and change the two col variables to column to see what actually happens.

Topic 6.2.3: The while Statement

*** The while Loop**
The while statement, also called the *while loop*, executes a block of statements as long as a specified condition is true.
*** Syntax**
The while statement has the following form:

```
while (expression)
    statements
```

*** Parameters**
The expression is any C expression, and `statements` is a single or compound C statement.

*** Process**

```
while (expression)
statements
```

When program execution reaches a while statement, the following events occur:

Stage	Description
1	`expression` is evaluated.
2	If expression evaluates as false (that is, as zero), the while statement terminates and execution passes to the first statement following statements.
3	If `expression` evaluates as true (that is, as nonzero), the C statement(s) in statements are executed.
4	Execution returns to Step 1.

*** Schematic Representation**
The operation of a while statement is diagrammed in Figure 6.3.
Listing 6.3 is a simple program that uses a while statement to print the numbers 1–20. Incidently, this is the same task that is performed by a for statement in Listing 6.1. Click the Listing button.

Listing 6.3: Demonstration of a Simple while Statement

Code

```
 1: /* LIST0603.c: Day 6 Listing 6.3 */
 2: /* Demonstrates a simple while statement */
 3:
 4: #include <stdio.h>
 5:
 6: int count;
 7:
 8: int main(void) {
 9:   /* Print the numbers 1 through 20. */
10:
11:   count = 1;
12:
13:   while (count <= 20) {
14:     printf("\n%d", count);
15:     count++;
16:   }
17:   return 0;
18: }
```

Output

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Description

Examine Listing 6.3 and compare it with Listing 6.1, which uses a for statement to perform the same task.

In line 11, count is initialized to 1. Because the while statement does not contain an initialize section, you must take care of initializing any variables before starting the while.

Line 13 is the actual while statement, and it contains the same condition statement from Listing 6.1, `count <= 20`. In the while loop, line 15 takes care of incrementing count.

What do you think would happen if you forgot to put line 15 in this program? Your program would not "know" when to stop because count would always be 1, which is always less than 20.

*** while Versus for**
You might have noticed that a while statement is essentially a for statement without the initialization (start) and increment components. Thus,
`for (; test ;)`
is equivalent to
`while (expression)`
Because of this equivalence, anything that can be done with a for statement can also be done with a while statement. When you use a while statement, any needed initialization must first be performed in a separate statement, and the updating must be performed by a statement that is part of the while loop.

*** Preference: Use a for Statement**
When initialization and updating are required, most experienced C programmers prefer to use a for statement rather than a while statement. This preference is based primarily on source code readability. When you use a for statement, the `initialization` (start), `test`, and `increment` expressions are located together and are easy to find and modify. With a while statement, the initialization and update expressions are located separately and may be less obvious.

Topic 6.2.4: Nesting while Statements

*** Nesting**
Just like the for and if statements, while statements can also be nested.
Listing 6.4 shows an example of nested while statements. Although this is not the best use of a while statement, the example does present some new ideas. Click the Listing button and then the Tip button.

DON'T use the following convention if it is not necessary.
`while(x)`
Instead use
`while(x != 0)`
Although both work, the second is clearer when debugging the code. When compiled, these produce virtually the same code.
DO use the for statement instead of the while statement if you need to initialize and increment within your loop. The for statement keeps the initialization, condition, and increment statements all together. The while statement does not.

Listing 6.4: Demonstration of Nested while Statements

```
Code  1: /* LIST0604.c: Day 6 Listing 6.4 */
      2: /* Demonstrates nested while statements */
      3:
      4: #include <stdio.h>
      5:
      6: int array[5];
```

```
 7:
 8: int main(void) {
 9:   int ctr = 0,
10:       nbr = 0;
11:
12:   printf("This program prompts you to enter 5 "
13:          "numbers\n");
14:   printf("Each number should be from 1 to 10\n");
15:
16:   while (ctr < 5) {
17:     nbr = 0;
18:     while (nbr < 1 "" nbr > 10) {
19:       printf("\nEnter number %d of 5: ", ctr + 1);
20:       scanf("%d", &nbr);
21:     }
22:
23:     array[ctr] = nbr;
24:     ctr++;
25:   }
26:
27:   for(ctr = 0; ctr < 5; ctr++)
28:     printf("\nValue %d is %d", ctr + 1,
29:            array[ctr]);
30:   return 0;
31: }
```

Output

```
This program prompts you to enter 5 numbers
Each number should be from 1 to 10
Enter number 1 of 5: 3
Enter number 2 of 5: 6
Enter number 3 of 5: 3
Enter number 4 of 5: 9
Enter number 5 of 5: 2
Value 1 is 3
Value 2 is 6
Value 3 is 3
Value 4 is 9
Value 5 is 2
```

Description

Like previous listings, lines 1 – 2 contain a comment with a description of the program, and line 4 contains a #include statement for the standard input/output header file. Line 6 contains a declaration for an array (named array) that can hold 5 integer values. The function main() contains two additional local variables, ctr and nbr (lines 9 and 10). Notice that these variables are initialized to zero at the same time they are declared. Also notice that the comma operator is used as a separator at the end of line 9, allowing nbr to be declared as an int without restating the int type command. Stating declarations in this manner is a common practice for many C programmers. Lines 12 – 14 print messages stating what the program does and what is expected of the user. Lines 16 – 25 contain the first while command and its statements. Lines 18 – 21 also contain a nested while loop with its own statements that are all part of the outer while.

This outer loop continues to execute while ctr is less than 5 (line 16). As long as ctr is less than 5, line 17 sets nbr to 0, lines 18 – 21 (the nested while statement) gather a number in variable nbr, line 23 places the number in array, and line 24 increments ctr. Then the loop starts again. Therefore, the

	outer loop gathers 5 numbers and places each into array, indexed by ctr. The inner loop is a good use of a while statement. Only the numbers from 1 to 10 are valid, so until the user enters a valid number, there is no point continuing the program. Lines 18 – 21 prevent continuation. This while statement states that while the number is less than 1 or while it is greater than 10, the program should print a message to enter a number, and then get the number. Lines 27 – 29 print the values that are stored in array. Notice that because the while statements are done with the variable ctr, the for command can reuse it. Starting at zero and incrementing by one, the for loops five times, printing the value of ctr plus one (because the count started at zero), and printing the corresponding value in array. For additional practice, there are two things you can change in this program. The first is the values that are accepted by the program. Instead of 1 to 10, try making it accept from 1 to 100. You can also change the number of values that it accepts. Currently it allows for 5 numbers. Try making it accept 10.

Topic 6.2.5: The do...while Loop

*** The do...while Loop**

C's third loop construct is the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning as is done by the for loop and the while loop.

*** Syntax**

The structure of the do...while loop is as follows:

```
do
    statements
while (expression);
```

*** Parameters**

`expression` is any C expression, and `statements` is a single or compound C statement.

*** Process**

```
do
    statements
while (expression);
```

When program execution reaches a do...while statement, the following events occur:

Stage	Description
1	The statements in `statements` are executed (at least once).
2	`expression` is evaluated. If it is true, execution returns to step one. If it is false, the loop terminates.

The operation of a do...while loop is summarized graphically in Figure 6.4

The operation of a do...while loop is shown schematically in Figure 6.4.

*** Exit-Condition Loop**

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In contrast, for loops and while loops evaluate the test condition at the start of the loop, and so the associated statements are not executed at all if the test condition is initially false.

*** Usage**

The do...while loop is used less frequently than while and for loops. It is most appropriate when the statement(s) associated with the loop must be executed at least once. You could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop. A do...while loop probably would be more straightforward, however.

*** Example Program**

Listing 6.5 shows an example of a do...while loop.

Listing 6.5: Demonstration of a Simple do...while Loop

Code

```
/* LIST0605.c: Day 6 Listing 6.5 */
/* Demonstrates a simple do...while statement */

#include <stdio.h>

int get_menu_choice(void);

int main(void) {
  int choice;

  choice = get_menu_choice();

  printf("You chose Menu Option %d", choice);
  return 0;
}

int get_menu_choice(void) {
    int selection = 0;

    do {
        printf("\n");
        printf("\n1 - Add a record");
        printf("\n2 - Change a record");
        printf("\n3 - Delete a record");
        printf("\n4 - Quit");
        printf("\n");
        printf("\nEnter a selection:");

        scanf("%d", &selection);

    }while (selection < 1 "" selection > 4);

    return selection;
}
```

Output

```
1 - Add a record
2 - Change a record
3 - Delete a record
```

```
3 - Delete a record
4 - Quit

Enter a selection:8

1 - Add a record
2 - Change a record
3 - Delete a record
4 - Quit

Enter a selection:4
You chose Menu Option 4
```

Description

This program provides a menu with four choices. The user selects one of the four choices, and then the program prints the number selected. Programs later in this course use and expand on this concept. For now, you should be able to follow most of the listing.

The main() function (lines 8 – 15) adds nothing to what you already know. All of main() could have been written into one line.

```
printf("You chose Menu Option %d",
       get_menu_choice());
```

If you were to expand this program, and act on the selection, you would need the value returned by get_menu_choice(), so it is wise to assign the value to a variable (such as choice).

Lines 17 – 34 contain get_menu_choice(). This function displays a menu on the screen (lines 21 – 27), and then gets a selection. Because you have to display a menu at least once to get an answer, it is appropriate to use a do...while loop. In the case of this program, the menu is displayed until a valid choice is entered.

Line 31 contains the while part of the do...while statement and validates the value of the selection, appropriately named selection. If the value entered is not between 1 and 4, the menu is redisplayed and the user is prompted for a new value. When a valid selection is entered, the program continues to line 33, which returns the value in the variable selection.

Topic 6.3: Nested Loops

*** Nested Loops**

The term *nested loop* refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops except that each inner loop must be enclosed completely in the outer loop; you cannot have overlapping loops. Click the Example links, then click the Tip button on the toolbar.

Illegal Example

Legal Example

DON'T try to overlap loops. You can nest them, but they must be entirely within each other.
DO use the do...while loop when you know that a loop should be executed at least once.

Thus, the following is not allowed:

```
for (count = 1; count < 100; count++)
{
    do
    {
    /* the do...while loop */
} /* end of for loop */
    } while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem.

```
for (count = 1; count < 100; count++)
{
    do
    {
    /* the do...while loop */
    } while (x != 0);
} /* end of for loop */
```

*** Inner/Outer Loop Dependence**
When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. In the previous example, if the inner do...while loop modifies the value of count, the number of times the outer for loop executes is affected. Note, however, that the inner loop may be independent from any variables in the outer loop; in this example, they are not.
*** Readability**
Good indenting style makes code with nested loops easier to read. Each level of loop should be indented one step farther than the last level. This clearly labels the code associated with each loop.

Topic 6.4: Day 6 Q&A

*** Questions & Answers**
Here are some questions to help you review what you have learned in this unit.

*** Question 1**
How do I know which programming control statement to use, for, the while, or the do...while?
*** Answer**

If you look at the syntax boxes provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you are not dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best.

Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.

*** Question 2**
How deep can I nest my loops?
*** Answer**
You can nest as many loops as you want. If your program requires more than two loops, however, consider using a function instead. You might find sorting through all those braces difficult, and a function is easier to follow in code.

*** Question 3**
Can I nest different loop commands?
*** Answer**
You can nest if, for, while, do...while, or any other command. You will find that many of the programs you try to write will require that you nest at least a few of these.

Topic 6.5: Day 6 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
for (count = 1; count < 100; count++)
{
    do
    {
    /* the do...while loop */
} /* end of for loop */
    } while (x != 0);
```

```
for(x = 0; x < 10; x++)
  for(y = 5; y > 0; y--)
/* puts() is a C library function that prints to screen. */
```

```
    puts("X");



record = 0;
while(record < 100) {
    printf("\nRecord %d ", record);
    printf("\nGetting next number...");
}



/* EXER0204.c: Day 2 Exercise 2.4 */
#include <stdio.h>
int main(void) {
  int ctr;

  for (ctr = 65; ctr < 91; ctr++)
    printf("%c", ctr);
  return 0;
}



for (counter = 1; counter < MAXVALUE; counter++);
    printf("\nCounter = %d", counter);
```

Topic 6.6: Day 6 Try Its

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write a declaration for an array that will hold 50 type long values.
Answer

```
long array[50];
```

*** Exercise 2**
Show a statement that assigns the value of 123.456 to the 50th element in the array from exercise one.
Answer

Notice that in the following answer, the 50th element is indexed to 49. Remember that arrays start at

0.

```
array[49] = 123.456;
```

*** Exercise 3**

Write a for statement to count from 1 to 100 by 3s.

Answer

```
int x;
for(x = 1; x <= 100; x += 3) ;
```

*** Exercise 4**

Write a while statement to count from 1 to 100 by 3's.

Answer

```
int x = 1;
while(x <= 100)
    x += 3;
```

*** Exercise 5**

Write a do...while statement to count from 1 to 100 by 3's.

Answer

```
int ctr = 1;
do {
    ctr += 3;
} while(ctr < 100);
```

Controlling Program Execution

C has three loop statements that control program execution: for, while, and do...while. Each of these constructs enables your program to execute a block of statements zero, one, or more than one time, based on the condition of certain program variables. Many programming tasks are well served by the repetitive execution allowed by these loop statements.

The for, while and do...while Statements

Although all three can be used to accomplish the same task, each is different. The for statement enables you to initialize, evaluate, and increment all in one command. The while statement operates as long as a condition is true. The do...while statement always executes its statements at least once

and continues to execute them until a condition is false.

Nested Loops

Nesting is the placing of one command within another. C allows for the nesting of any of its commands. Nesting the if statement was demonstrated on Day 4, "Statements, Expressions, and Operators." In this unit, the for, while, and do...while statements were nested.

Unit 7. Day 7 Basic Input/Output

In most programs you create, you will need to display information on the screen or read information from the keyboard. Many of the programs presented in earlier units performed these tasks, but you might not have understood exactly how.

Today, you learn

- Something about C's input and output statements.
- How to display information on the screen with the printf() and puts() library functions.
- How to format the information that is displayed on the screen.
- How to read data from the keyboard with the scanf() library function.

This unit is not intended to be a complete treatment of these topics, but provides enough information so that you can start writing real programs. These topics are covered in greater detail in Week 2.

Topic 7.1: Displaying Information on the Screen

*** Most Frequent Ways**
You will need most programs to display information on the screen. The two most frequent ways to do this are C's library functions printf() and puts().

Topic 7.1.1: The printf() Function

*** Description**

The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on the screen. You've already seen printf() used in many of the examples in this course. Now you need to see how printf() works.

*** Displaying a Text Message**
Printing a text message on the screen is simple. Call the printf() function, passing the desired

Printing a text message on the screen is simple. Call the printf() function, passing the desired message enclosed in double quotations.
Example

*** Displaying the Value of Variables**

In addition to text messages, however, you frequently need to display the value of program variables. This is a little more complicated than displaying only a message. Look at the examples, then click the Tip button on the toolbar.
Example

DON'T try to put multiple lines of text into one printf() statement. In most instances, it is clearer to print multiple lines with multiple print statements than it is to use just one with several newline (\n) escape characters.
DON'T forget to use the newline escape character when printing multiple lines of information in separate printf() statements.

To display *An error has occurred!* on the screen, you write

```
printf("An error has occurred!");
```

Suppose you want to display the value of the numeric variable x on the screen, along with some identifying text. Furthermore, you want the information to start at the beginning of a new line. You could use the printf() function as follows:

```
printf("\nThe value of x is %d", x);
```

and the resulting screen display, assuming the value of x is 12, would be

```
The value of x is 12
```

In this example, two arguments are passed to printf(). The first argument is enclosed in double quotation marks and is called the *format string*. The second argument is the name of the variable (x) with the value to be printed.

*** Format String**

A printf() format string specifies how the output is formatted. The three possible components of a format string are as follows:
Literal text
Escape sequences
Conversion specifiers (also known as format specifiers)
Let's look at each of these in more detail.

*** Literal Text**

Literal text is displayed exactly as entered in the format string. In the example, characters starting with the "T" (in The) and up to, but not including, the % sign comprise a literal string. The literal text is bolded here:

```
printf("\nThe value of x is %d", x);
```

*** Details**

The literal text (also known as plain text) of a format specifier is anything that doesn't qualify as either an escape sequence or a conversion specifier. Literal text is simply printed as is, including all spaces.

*** Escape Sequences**

Escape sequences provide special formatting control. An escape sequence consists of the backslash (\) followed by a single character. In the example, \n is an escape sequence:

```
printf("\nThe value of x is %d", x);
```

The \n is called the *newline* character and means "move to the start of the next line." Escape sequences are also used to print certain characters. More escape sequences are listed in Table 7.1. Listing 7.1 is a program that demonstrates some of the frequently used escape sequences. Click the Table button and then the Listing button on the toolbar.

Listing 7.1: Using printf() Escape Sequences

```
Code  1: /* LIST0701.c: Day 7 Listing 7.1 */
      2: /* Demonstration of frequently used escape */
      3: /* sequences */
      4:
      5: #include <stdio.h>
      6:
      7: #define QUIT 3
      8:
      9: int get_menu_choice(void);
     10: void print_report(void);
     11:
     12: int main(void) {
     13:   int choice = 0;
     14:
     15:   while(choice != QUIT) {
     16:     choice = get_menu_choice();
     17:
     18:     if (choice == 1)
     19:       printf("\nBeeping the computer\a\a\a");
     20:     else {
     21:       if (choice ==2)
     22:         print_report();
     23:     }
     24:   }
     25:   printf("You chose to quit!");
     26:   return 0;
     27: }
     28: int get_menu_choice(void) {
     29:   int selection = 0;
     30:
     31:   do {
     32:         printf("\n");
     33:         printf("\n1 - Beep computer");
     34:         printf("\n2 - Display Report");
     35:         printf("\n3 - Quit");
     36:         printf("\n");
     37:         printf("\nEnter a selection:");
     38:
     39:         scanf("%d", &selection);

     40:
     41:     }while (selection < 1 "" selection > 3);
     42:
     43:     return selection;
     44: }
     45:
     46: void print report(void) {
```

```
47:         printf("\nSAMPLE REPORT");
48:         printf("\n\nSequence\tMeaning");
49:         printf("\n========\t=======");
50:         printf("\n\\a\t\tbell (alert)");
51:         printf("\n\\b\t\tbackspace");
52:         printf("\n...\t\t...");
53: }
```

Output

```
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
========        =======
\a              bell (alert)
\b              backspace
...             ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

Description

Listing 7.1 seems long compared with previous examples, but it offers some additions that are worth noting. The STDIO.H header was included in line 5 because printf() is used in this listing. In line 7, a constant named QUIT is defined. From Day 3, "Numeric Variables and Constants," you know that #define makes using the constant QUIT equivalent to using the value 3.

Lines 9 and 10 are function prototypes. This program has two functions, get_menu_choice() and print_report(). get_menu_choice() is defined in lines 28 – 44. This is similar to the menu function in Listing 6.5.

Lines 32 and 36 contain calls to printf() that print the newline escape sequence. Lines 33, 34, 35, and 37 also use the newline escape character, and they print text. Line 32 could have been eliminated by changing line 33 to the following:

```
printf( "\n\n1 - Beep Computer" );
```

However, leaving line 32 makes the program easier to read.

Looking at the main() function, you see the start of a while loop on line 15. The while's statements are going to keep looping as long as choice is not equal to QUIT. Because QUIT is a constant, you could have replaced it with 3; however, the program would not be as clear.

Line 16 gets the variable choice, which is then analyzed in lines 18 – 24 in an if statement. If the user chose 1, line 19 prints the newline character, a message, and then three beeps. If the user selected 2 on the menu, line 22

	message, and then three beeps. If the user selected 2 on the menu, line 22 calls the function print_report(). print_report() is defined on lines 46–53. This simple function shows the ease of using printf() and the escape sequences to print formatted information to the screen. You've already seen the newline character. Lines 48 to 52 also use the tab escape character, \t. It lines the columns of the report vertically. Lines 50 and 51 might seem confusing at first, but if you start at the left and work right, they make sense. Line 50 prints a newline (\n), then a backslash (\), then the letter a, followed by two tabs (\t\t). The line ends with some descriptive text, (bell (alert)). Line 51 follows the same format. This program prints the first two lines of Table 7.1, along with a report title and column headings. In exercise nine, you will complete this program by making it print the rest of the table.

Table 7.1: The Most Frequently Used Escape Sequences

Sequence	Meaning
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash
\?	Question mark
\'	Single quotation
\"	Double quotations

*** Details**

Now look at the format string components in more detail. Escape sequences are used to control the location of output by moving the screen cursor. They are also used to print characters that would otherwise have a special meaning to printf().

For example, to print a single backslash character, include a double backslash (\\) in the format string. The first backslash tells printf() that the second backslash is to be interpreted as a literal character, not the start of an escape sequence. In general, the backslash tells printf() to interpret the next character in a special manner. Here are some examples:

Character	Description
n	The character *n*
\n	Newline

\"	The double quotation character
"	The start or end of a string

*** Description**
Conversion specifiers (also known as format specifiers) consist of the percent sign followed by a single character. In the example, the conversion specifier is %d. A conversion specifier tells printf() how to interpret the variable(s) being printed. The %d tells printf() to interpret the variable x as a signed decimal (base 10) integer.

*** Details**
The format string must contain one conversion specifier (also known as format specifiers) for each printed variable. printf() then displays each variable as directed by its corresponding conversion specifier. You learn more about this process on Day 15, "More on Pointers." For now, be sure to use the conversion specifier that corresponds to the type of variable being printed.
What exactly does this mean? If you're printing a variable that is a *signed decimal integer* (types int and long), use the %d conversion specifier. For an *unsigned decimal integer* (types unsigned int and unsigned long), use %u. For a *floating-point variable* (types float and double), use the %f specifier.
The conversion specifiers you need most often are listed here.

Table 7.2: The Most Commonly Needed Conversion Specifiers	
Specifier	**Meaning**
%c	Single character
%d	Signed decimal integer (types int and long)
%f	Decimal floating-point number (types float and double)
%s	Character string
%u	Unsigned decimal integer (types unsigned int and unsigned long)

*** Displaying the Values of More Than One Variable**
What about printing the values of more than one variable? A single printf() statement can print an unlimited number of variables, but the format string must contain one conversion specifier for each variable. The conversion specifiers are paired with variables in left-to-right order, as shown in the Example.
Example
The positions of the conversion specifiers in the format string determine the position of the output. If there are more variables passed to printf() than there are conversion specifiers, the unmatched variables are not printed. If there are more specifiers than variables, the unmatched specifiers print "garbage."

If you write

```
printf("Rate = %f, amount = %d", rate, amount);
```

the variable rate is paired with the %f specifier, and the variable amount is paired with the %d specifier.

*** Displaying Other Values**
You are not limited to printing the value of variables with printf(). The arguments can be any valid C expression.
Example
*** STDIO.H**
Any program that uses printf() should include the header file STDIO.H.

To print the sum of x and y you could write

```
z = x + y;
printf("%d", z);
```

You could also write

```
printf("%d", x + y);
```

*** Example Program**
Listing 7.2 demonstrates the use of printf().

Day 15, "More on Pointers," gives more details on printf().

Click the Listing button.

Listing 7.2: Using printf() to Display Numerical Values

```
Code  1: /* LIST0702.c: Day 7 Listing 7.2 */
      2: /* Demonstration using printf() to display */
      3: /* numerical values. */
      4:
      5: #include <stdio.h>
      6:
      7: int a = 2, b = 10, c = 50;
      8: float f = 1.05F, g = 25.5F, h = -0.1F;
      9:
     10: int main(void) {
     11:   printf("\nDecimal values without tabs:"
     12:          " %d %d %d", a, b, c);
     13:   printf("\nDecimal values with tabs:"
     14:          " \t%d \t%d \t%d", a, b, c);
     15:
     16:   printf("\nThree floats on 1 line:"

     17:          " \t%f\t%f\t%f", f, g, h);
     18:   printf("\nThree floats on 3 lines: "
     19:          "\n\t%f\n\t%f\n\t%f", f, g, h);
     20:
     21:   printf("\nThe rate is %f%%", f);
     22:   printf("\nThe result of %f/%f = %f", g, f, g/f);
     23:   return 0;
```

24:	}

Output	`Decimal values without tabs: 2 10 50`
	`Decimal values with tabs: 2 10 50`
	`Three floats on 1 line:1.050000 25.500000 -0.100000`
	`Three floats on 3 lines:`
	`1.050000`
	`25.500000`
	`-0.100000`
	`The rate is 1.050000%`
	`The result of 25.500000/1.050000 = 24.285715`

Description	Listing 7.2 prints six lines of information. Lines 11 – 14 print three decimals, a, b, and c. Lines 11 – 12 print them without tabs, and lines 13 – 14 print them with tabs. Lines 16 – 19 print three float variables, f, g, and h. Lines 16 – 17 print them on one line, and lines 18 – 19 print them on three lines. Line 21 prints out a float variable, f, followed by a percentage sign. Because a percentage sign is normally a message to print a variable, you must place two in a row to print a single percent sign. This is exactly like the backslash escape character. Line 22 shows one final concept. When printing values in conversion specifiers, you do not have to use variables. You can also use expressions such as g / f, or even constants.

Topic 7.1.2: Displaying Messages with puts()

*** The puts() Function**
The puts() function also can be used to display text messages on the screen, but it cannot display numeric variables. puts() takes a single string as its argument and displays it, automatically adding a newline at the end.
Example
*** Escape Sequences**
You can include escape sequences (including \n) in a string passed to puts(). They have the same effects as when they are used with printf() (see Table 7.1).
*** STDIO.H**
Any program that uses puts() should include the header file STDIO.H. Note that STDIO.H should be included only once in any program. Click the Tip button on the toolbar.

DO use the puts() function instead of the printf() function whenever you want to print text but don't need to print any variables.

DON'T try to use conversion specifiers with the puts() statement.

The statement

```
puts("Hello, world.");
```

performs the same action as

```
printf("Hello, world.\n");
```

Topic 7.2: Inputting Numeric Data with scanf()

*** The scanf() Function**

Just as most programs need to output data to the screen, so also do they need to input data from the keyboard. The most flexible way your program can read numeric data from the keyboard is by using the scanf() library function.

*** Reading Data from the Keyboard**

The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() uses a format string to describe the format of the input. The format string utilizes the same conversion specifiers as the printf() function. Click the Example link.

Example

DON'T forget to include the *address of* operator (&) when using scanf() variables.
DO use printf() or puts() in conjunction with scanf(). Use the printing functions to display a prompting message for the data you want scanf() to get.

The statement

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

reads a decimal integer from the keyboard and assigns it to the integer variable x. Likewise, the statement

```
scanf("%f", &rate);
```

reads a floating-point value from the keyboard and assigns it to the variable rate.

*** The address of Operator**

What is the ampersand (&) doing before the variable's name? The & symbol is C's *address of* operator, which is fully explained on Day 9, "Pointers." For now, all you need to remember is that scanf() requires the & symbol before each numeric variable name in its argument list (unless the variable is a *pointer,* which is also explained on Day 9).

*** Reading More Than One Value**

A single scanf() can input more than one value if you include multiple conversion specifiers in the format string and variable names (again, each preceded by & in the argument list).

Example

The statement

```
scanf("%d %f", &x, &rate);
```

inputs an integer value and a floating point value and assigns them to the variables x and rate, respectively.

*** Function of Whitespace**

When multiple variables are entered, scanf() uses whitespace to separate input into fields. Whitespace can be spaces, tabs, or new lines. Each conversion specifier in the scanf() format string is matched with an input field; the end of each input field is identified by whitespace. This gives you considerable flexibility.
Example

In response to the scanf() statement
```
scanf("%d %f", &x, &rate);
```
you could enter
```
10 12.45
```
You could also enter
```
10                    12.45
```
or
```
10
12.45
```
As long as there's some whitespace between values, scanf() can assign each value to its variable.

*** Example Program**
Although Listing 7.3 gives an example of using scanf(), a more complete description will be presented on Day 15, "More on Pointers." Click the Listing button and then the Note button on the toolbar.

Listing 7.3: Using scanf() to Obtain Numerical Values

```
Code  1: /* LIST0703.c: Day 7 Listing 7.3 */
      2: /* Demonstration on using scanf() */
      3:
      4: #include <stdio.h>
      5:
      6: #define QUIT 4
      7:
      8: int get_menu_choice(void);
      9:
     10: int main(void) {
     11:   int choice      = 0;
     12:   int int_var     = 0;
     13:   float float_var = 0.0;
     14:   unsigned unsigned_var = 0;
     15:
     16:   while (choice != QUIT) {
     17:     choice = get_menu_choice();
     18:
     19:     if (choice == 1) {
     20:       puts("\nEnter a signed decimal integer"
     21:            " (i.e. -123)");
     22:       scanf("%d", &int_var);
     23:     }
     24:     if (choice == 2) {
     25:       puts("\nEnter a decimal floating point"
     26:            " number (i.e. 1.23)");
     27:       scanf("%f", &float_var);
     28:     }
     29:     if (choice == 3) {
     30:       puts("\nEnter an unsigned decimal integer"
```

```
31:             " (i.e. 123)");
32:       scanf("%u", &unsigned_var);
33:     }
34:   }
35:   printf("\nYour values are: int: %d "
36:          " float: %f   unsigned: %u",
37:          int_var, float_var, unsigned_var);
38:   return 0;
39: }
40:
41: int get_menu_choice(void) {
42:   int selection = 0;
43:
44:   do {
45:     puts("\n1 - Get a signed decimal integer");
46:     puts("2 - Get a decimal floating-point"
47:          " number");
48:     puts("3 - Get an unsigned decimal integer");
49:     puts("4 - Quit");
50:     puts("\nEnter a selection:");
51:
52:     scanf("%d", &selection);
53:
54:   }while (selection < 1 "" selection > 4);
55:
56:   return selection;
57: }
```

Output

```
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned:
321
```

Description Listing 7.3 uses the same menu concepts that were used in Listing 7.1. The

Description	Listing 7.3 uses the same menu concepts that were used in Listing 7.1. The differences in get_menu_choice() (lines 41 – 57) are minor, but should be noted. First, puts() is used instead of printf(). Because no variables are printed, there is no need to use printf(). Because puts() is being used, the newline escape characters have been removed from lines 46 –49. Line 54 was also changed to allow values from 1 to 4 because there are now four menu options. Notice that line 52 has not changed; however, now it should make a little more sense. scanf() gets a decimal value and places it in the variable selection. The function returns selection to the calling program in line 56. Listings 7.1 and 7.3 use the same main() structure. An if statement evaluates choice, which is the return value of get_menu_choice(). Based on choice's value, the program prints a message, asks for a number to be entered, and reads the value with scanf(). Notice the differences between lines 22, 27, and 32. Each is set up to get a different type of variable. Lines 12–14 declare variables of the appropriate types. When the user selects quit, the program prints the last-entered number for all three types. If the user did not enter a value, 0 is printed because lines 12 and 13 initialized all three types. One final note on lines 19 through 34: The if statements used here are not structured well. If you are thinking that an if...else structure would be better, you are correct. On Day 14, "Working with the Screen, Printer, and Keyboard," a new control statement, switch, is introduced. This statement would be the best option.

*** STDIO.H**
As with the other functions discussed in this unit, programs that use scanf() must include the STDIO.H header file.

Topic 7.3: Day 7 Q&A

*** Questions & Answers**

Here are some questions to help you review what you have learned in this unit.

*** Question 1**
Why should I use puts() if printf() does everything puts() does and more?
*** Answer**
Because printf() does more, it has additional overhead. When you are trying to write a small

efficient program or when your programs get big and resources are valuable, you will want to take advantage of the smaller overhead of puts(). In general, use the simplest available resource.

*** Question 2**
Why do I need to include STDIO.H when I use printf(), puts(), or scanf().
*** Answer**
STDIO.H contains the prototypes for the standard input/output functions. printf(), puts(), and scanf() are three of these standard functions. Try running a program without the STDIO.H header and see the errors and warnings you get.

*** Question 3**
What happens if I leave the address of operator (&) off a scanf() variable?
*** Answer**
This is an easy mistake to make. Unpredictable results can occur if you forget the address of operator. When you cover pointers on Days 9 and 13, "Pointers" and "More Program Control," you will understand this more. For now know that if you omit the address of operator, scanf() doesn't place the entered information in your variable, but in some other place in memory. This could do anything from apparently having no effect at all to locking up your computer so that you have to reboot.

Topic 7.4: Day 7 Think About Its

*** Think About Its**
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

b prints the literal character b

\b prints a backspace character
\ looks at the next character to determine an escape character
\\ prints a single backslash

```
01: #include <stdio.h>
02:
03: int x;
04:
05: int main(void) {
06:   puts("Enter an integer value:");
07:   ??????("%d", &x);
08:
09:   ??????("\nThe value entered is %d", x);
10:   return 0;
11: }
```

```
int get_1_or_2( void )
{
    int answer = 0;
    while( answer < 1 "" answer > 2 )
    {
        printf(Enter 1 for Yes, 2 for No);
        scanf( "%f", answer );
    }
    return answer;
}
```

Topic 7.5: Day 7 Try Its

*** Try Its**
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write both a printf() and a puts() statement to start a new line.
Answer

puts() automatically adds the newline, printf() does not.

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

*** Exercise 2**
Write a scanf() statement that could be used to get a character, an unsigned decimal integer, and another single character.
Answer

```
char c1, c2;
int d1;
scanf("%c %u %c", &c1, &d1, &c2);
```

*** Exercise 3**
Write the statements to get an integer value and print it.
Answer

Your answer may vary.

```
/* EXER0703.c: Day 7 Exercise 7.3 */
#include <stdio.h>
int x;
int main(void) {
  puts("Enter an integer value:");
  scanf("%d", &x);

  printf("\nThe value entered is %d", x);
  return 0;
}
```

*** Exercise 4**
Modify exercise three so that it accepts only even values (2, 4, 6, etc.).
Answer

It's typical to add edits that allow only specific values to be accepted. The following is one way to accomplish this exercise.

```
/* EXER0704.c: Day 7 Exercise 7.4 */
#include <stdio.h>
int x;
int main(void) {
  puts("Enter an even integer value:");
  scanf("%d", &x);
  while(x % 2 != 0) {
    printf("\n%d is not even. Enter an even number: ", x);
    scanf("%d", &x);
  }
  printf("\nThe value entered is %d", x);
  return 0;
}
```

*** Exercise 5**
Modify exercise four so that it returns values until the number 99 is entered, or until six even values have been entered. Store the numbers in an array. (Hint: You need a loop!)
Answer

```
/* EXER0705.c: Day 7 Exercise 7.5 */
#include <stdio.h>
int array[6], x, number;
int main(void) {
  /* Loop 6 times or until last entered element is 99. */
  for (x=0; x < 6 && number != 99; x++) {
    puts("Enter an even integer value, or 99 to quit");
    scanf("%d", &number);
    while(number % 2 == 1 && number != 99) {
      printf("\n%d is not even. Enter an even number: ",
             number);
      scanf("%d", &number);
    }
    array[x] = number;
  }
  /* Now print them out. */
  for (x=0; x < 6 && array[x] != 99; x++) {
    printf("\nThe value entered is %d", array[x]);
  }
  return 0;
}
```

*** Exercise 6**

Turn exercise five into an executable program. Add a function that prints the values in the array on a single line, separated by tabs. (Print only the values that were entered into the array.)

Answer

The previous answers already are executable programs. The only change that needs to be made is in the final printf(). To print each value separated by a tab, change the printf() statement to the following:

```
printf("%d\t", array[x]);
```

*** Exercise 7**

Using Listing 7.1, complete the print_report() function so that it prints the rest of Table 7.1. Click the Table button and then the Listing button on the toolbar.

Answer

Listing 7.1: Using printf() Escape Sequences

```
Code  1:  /* LIST0701.c: Day 7 Listing 7.1 */
      2:  /* Demonstration of frequently used escape */
      3:  /* sequences */
      4:
      5:  #include <stdio.h>
      6:
      7:  #define QUIT 3
      8:
      9:  int get_menu_choice(void);
     10:  void print_report(void);
     11:
```

```
11:
12: int main(void) {
13:   int choice = 0;
14:
15:   while(choice != QUIT) {
16:     choice = get_menu_choice();
17:
18:     if (choice == 1)
19:       printf("\nBeeping the computer\a\a\a");
20:     else {
21:       if (choice ==2)
22:         print_report();
23:     }
24:   }
25:   printf("You chose to quit!");
26:   return 0;
27: }
28: int get_menu_choice(void) {
29:   int selection = 0;
30:
31:   do {
32:         printf("\n");
33:         printf("\n1 - Beep computer");
34:         printf("\n2 - Display Report");
35:         printf("\n3 - Quit");
36:         printf("\n");
37:         printf("\nEnter a selection:");
38:
39:         scanf("%d", &selection);
40:
41:     }while (selection < 1 "" selection > 3);
42:
43:     return selection;
44: }
45:
46: void print_report(void) {
47:         printf("\nSAMPLE REPORT");
48:         printf("\n\nSequence\tMeaning");
49:         printf("\n========\t=======");
50:         printf("\n\\a\t\tbell (alert)");
51:         printf("\n\\b\t\tbackspace");
52:         printf("\n...\t\t...");
53: }
```

Table 7.1: The Most Frequently Used Escape Sequences

Sequence	Meaning
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash
\?	Question mark

\?	Question mark
\'	Single quotation
\"	Double quotations

The following is the completed print_report function for Listing 7.1:

```
void print_report(void) {
        printf("\nSAMPLE REPORT");
        printf("\n\nSequence\tMeaning");
        printf("\n========\t=======");
        printf("\n\\a\t\tbell (alert)");
        printf("\n\\b\t\tbackspace");
        printf("\n\\n\t\tnew line");
        printf("\n\\t\t\thorizontal tab");
        printf("\n\\\\\t\tbackslash");
        printf("\n\\\?\t\tquestion mark");
        printf("\n\\\'\t\tsingle quote");
        printf("\n\\\"\t\tdouble quote");
        printf("\n...\t\t...");
}
```

*** Exercise 8**

Write a program that inputs two floating-point values from the keyboard, and then displays their product.

Answer

```
/* EXER078.c: Day 7 Exercise 7.8 */
#include <stdio.h>
float x, y;
int main(void) {
  puts("Enter two values: ");
  scanf("%f %f", &x, &y);
  printf("\nThe product is %f", x * y);
  return 0;
}
```

*** Exercise**

Write a program that inputs 10 integer values from the keyboard, and then displays their sum.

Answer

The following program prompts for 10 integers and displays their sum.

```
/* EXER079.c: Day 7 Exercise 7.9 */
/* Input 10 integers and display their sum. */
#include <stdio.h>
int count, temp;
long total = 0;  /* Use type long to ensure we don't */
```

```
                  /* exceed the maximum for type int. */
int main(void) {
  for (count = 1; count <= 10; count++) {
      printf("Enter integer # %d: ", count);
      scanf("%d", &temp);
      total += temp;
    }
  printf("\n\nThe total is %d", total);
  return 0;
}
```

*** Exercise 1**

Write a program that inputs integers from the keyboard, storing them in an array. Input should stop when a zero is entered or when the end of the array is reached. Then, find and display the array's largest and smallest values. (Note: This is a tough problem because arrays haven't been completely covered yet. If you have difficulty, try to solve it again after reading Day 8, "Numeric Arrays."

Answer

```
/* EXER0710.c: Day 7 Exercise 7.10 */
/* Inputs integers and stores them in an array, */
/* stopping when a zero is entered. Finds and displays */
/* the array's largest and smallest values */
#include <stdio.h>
#define MAX 100
int array[MAX];
int count = -1, maximum, minimum, num_entered, temp;
long total = 0;  /* Use type long to ensure we don't */
                 /* exceed the maximum for type int. */
int main(void) {
  puts("Enter integer values one per line.");
  puts("Enter 0 when finished.");

  /* Input the values. */

  do {
    scanf("%d", &temp);
    array[++count] = temp;
  }while (count < (MAX-1) && temp != 0);

  num_entered = count;

  /* Find the largest and smallest. */
  /* First set maximum to a very small value, */
  /* and minimum to a very large value. */

  maximum = -32000;
  minimum = 32000;

  for (count = 0; count <= num_entered
      && array[count] != 0; count++) {
      if (array[count] > maximum)
        maximum = array[count];

      if (array[count] < minimum)
        minimum = array[count];
    }
```

```
    }

    printf("\nThe maximum value is %d", maximum);
    printf("\nThe minimum value is %d", minimum);
    return 0;
}
```

Topic 7.6: Day 7 Summary

Tools to Write Simple Programs
With the completion of this unit, you are ready to write your own C programs. By combining the printf(), puts(), and scanf() functions and the programming control you learned in earlier units, you have the tools needed to write simple programs.
Displaying Information on the Screen
Screen display is performed with the printf() and puts() functions. The puts() function can display text messages only, whereas printf() can display text messages and variables. Both functions use escape sequences for special characters and printing controls.
Inputting Numeric Data
The scanf() function reads one or more numeric values from the keyboard and interprets each one according to a conversion specifier. Each value is assigned to a program variable.

Unit 8. Week 1 in Review

[Skip Unit 8's navigation links]
8. Week 1 in Review

*** Week 1 in Review**
After finishing your first week of learning how to program in C, you should feel comfortable entering programs and using your editor and compiler. Listing week1.c pulls together many of the topics from the previous week.
*** More Information**
The numbers to the left of the line numbers indicate the day (e.g. D02 = Day 2) that covers the concept presented on that line. If you are confused by the line, refer to the referenced day for more information. Click the Listing button.

Listing R1.1: Week One Review Listing

```
Code   1: /* Program: week1.c                                  */
D02    2: /* A program to enter the ages and incomes of up */

       3: /* to 100 people.  The program prints a report   */
       4: /* based on the numbers entered.                  */
       5: /* MS-DOS Application */
       6:
       7: /* Included files. */
       8:
D02    9: #include <stdio.h>
      10:
```

```
D02  11: /* Defined constants. */
     12:
D03  13: #define MAX 100
     14: #define YES 1
     15: #define NO  0
     16:
D02  17: /* Variables. */
     18:
D03  19: long income[MAX];        /* To hold incomes.     */
     20: int  month[MAX], day[MAX], year[MAX];
     21:                         /* To hold birthdays.   */
     22: int   x, y, ctr;         /* For counters.        */
     23: int   cont;              /* For program control. */
     24: long month_total, grand_total; /* For totals.    */
     25:
D02  26: /* Function prototypes. */
     27:
D05  28: int display_instructions(void);
     29: void get_data(void);
     30: void display_report(void);
     31: int continue_function(void);
     32:
     33: /* Start of program. */
     34:
D02  35: int main(void) {
D05  36:   cont = display_instructions();
     37:
D04  38:   if (cont == YES) {
D05  39:     get_data();
D05  40:     display_report();
     41:   }
D04  42:   else
D07  43:     printf("\nProgram Aborted by User!\n\n");
     44:   return 0;
     45: }
D02  46: /* Function: display_instructions()             */
     47: /* Purpose: This function displays information  */
     48: /* on how to use this program and asks the user */
     49: /* to enter 0 to quit, or 1 to continue.        */
     50: /* Returns: NO - if the user enters 0           */
     51: /* YES - if user enters any number other than 0 */
     52:
D05  53: int display_instructions(void) {
D07  54:   printf("\n\n");
     55:   printf("\nThis program enables you to enter up"
     56:          " to 99 people\'s");
     57:   printf("\nincomes and birthdays. It then prints"
     58:          " the incomes by");
     59:   printf("\nmonth along with the overall income"
     60:          " and overall average.");
     61:   printf("\n");
     62:


D05  63:   cont = continue_function();
     64:
D05  65:   return(cont);
     66: }
     67:
D02  68: /* Function: get_data()                         */
     69: /* Purpose: This function gets the data from the */
     70: /* user. It continues to get data until either   */
```

```
     70: /* user is continued to get data until sentinel    */
     71: /* 100 people are entered, or until the user     */
     72: /* enters 0 for the month.                       */
     73: /* Returns: Nothing                              */
     74: /* Notes:   This allows 0/0/0/ to be entered for */
     75: /* birthdays in case the user is unsure. It also */
     76: /* allows for 31 days in each month.             */
     77:
D05  78: void get_data(void) {
D06  79:   for (ctr = 0; ctr < MAX && cont == YES; ctr++) {
D07  80:     printf("\nEnter information for Person %d.",
     81:            ctr + 1);
     82:     printf("\n\tEnter Birthday:");
     83:
D06  84:     do {
D07  85:       printf("\n\tMonth (0-12): ");
D07  86:       scanf("%d", &month[ctr]);
D06  87:     } while (month[ctr] < 0 "" month[ctr] > 12);
     88:
D06  89:     do {
D07  90:       printf("\n\tDay (0-31): ");
D07  91:       scanf("%d", &day[ctr]);
D06  92:     } while (day[ctr] < 0 "" day[ctr] > 31);
     93:
D06  94:     do {
D07  95:       printf("\n\tYear (0-1999): ");
D07  96:       scanf("%d", &year[ctr]);
D06  97:     } while (year[ctr] < 0 "" year[ctr] > 1999);
     98:
D07  99:     printf("\nEnter Yearly Income (whole $s): ");
D07 100:     scanf("%ld", &income[ctr]);
    101:
D05 102:     cont = continue_function();
    103:   }
    104: }
D07 105: /* ctr equals number of people entered. */
    106:
D02 107: /* Function: display_report()                    */
    108: /* Purpose: This function displays a report to  */
    109: /* the screen. Returns: Nothing                 */
    110: /* Notes: More information could be displayed.  */
    111:
D05 112: void display_report(void) {
D04 113:   grand_total = 0;
D07 114:   printf("\n\n\n");
    115:   printf("\n         SALARY SUMMARY");
    116:   printf("\n         ==============");
    117:
D06 118:   for (x = 0; x <= 12; x++) {
    119:     /* For each month, including 0 */
    120:     month_total = 0;
D04 121:     for (y = 0; y < ctr; y++) {
D06 122:       if (month[y] == x)

D04 123:         month_total += income[y];
D04 124:     }
    125:     printf("\nTotal for month %d is %ld", x,
    126:            month_total);
D07 127:     grand_total += month_total;
D04 128:   }
D07 129:   printf("\n\nReport totals:");
    130:   printf("\nTotal Income is %ld", grand_total);
```

```
    131:   printf("\nAverage Income is %ld",
    132:          grand_total/ctr);
    133:
    134:   printf("\n\n* * * End of Report * * *");
    135: }
    136:
D02 137: /* Function: continue_function()                 */
    138: /* Purpose: This function asks if user wants to  */
    139: /* continue. Returns: YES - if user wants to     */
    140: /* continue, NO - if user wants to quit          */
    141:
D05 142: int continue_function(void) {
D07 143:   printf("\n\nDo you wish to continue? "
    144:          "(0=NO/1=YES): ");
    145:   scanf("%d", &x);
    146:
D06 147:   while (x < 0 "" x > 1) {
D07 148:     printf("\n%d is invalid!", x);
    149:     printf("\nPlease enter 0 to Quit or "
    150:            "1 to Continue: ");
    151:     scanf("%d", &x);
    152:   }
D04 153:   if (x == 0)
D05 154:     return(NO);
D04 155:   else
D05 156:     return(YES);
    157: }
```

Description

After completing Day 1 and Day 2, you should be able to enter and compile this program. This program contains more comments than other listings throughout this course. These comments are typical of a "real world" C program. In particular, you should notice the comments at the beginning of the program and before each major function. The comments on lines 1 – 5 contain an overview of the entire program, including the program name.

Some programmers also include information such as the author of the program, the compiler used, its version number, the libraries linked into the program, and the date the program was created. The comments before each function describe the purpose of the function, possible return values, the function's calling conventions, and anything relating specifically to that function.

The comments on lines 1 – 5 specify that you can enter information in this program for up to 100 people. Before you can enter the data, the program calls display_instructions() (line 36). This function displays instructions for using the program, asking you whether you want to continue or quit. On lines 54 – 61, you can see that this function uses the printf() function from Day 7, "Basic Input/Output," to display the instructions.

The continue_function() on lines 142 – 157 uses some of the features covered at the end of the week. The function asks whether you want to continue (line 143). Using the while control statement from Day 6, "Basic Program Control," the function verifies that the answer entered was a 0 or a 1. As long as the answer is not one of these two values, the function keeps prompting for a response. Once the program receives an appropriate answer

	prompting for a response. Once the program receives an appropriate answer, an if...else statement (Day 4, "Statements, Expressions, and Operators") returns a constant variable of either YES or NO. The heart of this program lies in two functions: get_data() and display_report(). The get_data() function prompts you to enter data, placing the information into the arrays declared near the beginning of the program. Using a for statement on line 79, you are prompted to enter data until cont is not equal to the defined constant YES (returned from continue_function()) or the counter, ctr, is greater than or equal to the maximum number of array elements, MAX. This program checks each piece of information entered to ensure that it is appropriate. For example, lines 84 – 87 prompt you to enter a month. The only values that the program accepts are 0 – 12. If you enter a number greater than 12, the program prompts for the month again. Line 102 calls continue_function() to check whether you want to continue adding data. When you respond to the continue function with a 0, or the maximum number of sets of information is entered (MAX sets), the program returns to line 40 in main() where it calls display_report(). The display_report() function on lines 107 – 135 prints a report to the screen. This report uses a nested for loop to total incomes for each month and a grand total for all the months. This report might seem complicated; if so, review Day 6, "Basic Program Control," for coverage of nested statements. Many of the reports that you create as a programmer are more complicated than this one. This program uses what you learned in your first week of teaching yourself C. This was a large amount of material to cover in just one week, but you did it! If you use everything you learned this week, you can write your own programs in C. However, there are still limits to what you can do.

Unit 9. Reference

[**Skip Unit 9's navigation links**]
9. Reference
9.1 C Reserved Words
9.2 Binary and Hexadecimal Notation

Topic 9.1: C Reserved Words

*** C Reserved Words**
The following identifiers are reserved C keywords. They should not be used for any other purpose in a C program. They are allowed, of course, within double quotation marks.

Keyword	Description
asm	A C keyword that denotes inline assembly language code.
auto	The default storage class.
break	A C command that exits for, while, switch, and do...while statements unconditionally.
case	A C command used within the switch statement.
char	The simplest C data type.
const	A C data modifier that prevents a variable from being changed. See volatile.
continue	A C command that resets a for, while, or do...while statement to the next iteration.
default	A C command used within the switch statement to catch any instances not specified with a case statement.
do	A C looping command used in conjunction with the while statement. The loop will always execute at least once.
double	A C data type that can hold double-precision floating-point values.
else	A statement signaling alternative statements to be executed when an if statement evaluates to FALSE.
enum	A C data type that allows variables to be declared that accept only certain values.
extern	A C data modifier indicating that a variable will be declared in another area of the program.
float	A C data type used for floating-point numbers.
for	A C looping command that contains initialization, incrementation, and conditional sections.
goto	A C command that causes a jump to a predefined label.
if	A C command used to change program flow based on a TRUE/FALSE decision.
int	A C data type used to hold integer values.
long	A C data type used to hold larger integer values than int.
register	A storage modifier that specifies that a variable should be stored in a register, if possible.
return	A C command that causes program flow to exit from the current function and return to the calling function. It also can be used to return a single value.
short	A C data type that is used to hold integers. It is not commonly used, and is the same size as an int on most computers.
signed	A C modifier that is used to signify that a variable can have both positive and negative values.
sizeof	A C operator that returns the size (number of bytes) of the item.
static	A C modifier that is used to signify that the compiler should retain the

	A C modifier that is used to signify that the compiler should retain the variable's value.
`struct`	A C keyword used to combine C variables of any data types into a group.
`switch`	A C command used to change program flow into a multitude of directions. Used in conjunction with the case statement.
`typedef`	A C modifier used to create new names for existing variable and function types.
`union`	A C keyword used to allow multiple variables to share the same memory space.
`unsigned`	A C modifier that is used to signify that a variable will only contain positive values. See signed.
`void`	A C keyword used to signify either that a function does not return anything or that a pointer being used is considered generic or able to point to any data type.
`volatile`	A C modifier that signifies that a variable can be changed. See const.
`while`	A C looping statement that executes a section of code as long as a condition remains TRUE.

Topic 9.2: Binary and Hexadecimal Notation

*** Binary and Hexadecimal Notation**
Binary and hexadecimal numeric notations are frequently used in the world of computers, so it's a good idea to be familiar with them. All number notation systems use a certain base. For the binary system, the base is 2, and for the hexadecimal system, it is 16.

*** Base 10**
To understand what *base* means, consider the familiar decimal notation system, which uses a base of 10. Base 10 requires 10 different digits, 0–9. In a decimal number, each successive digit (starting right and moving to the left) indicates a successively increasing power of 10. The rightmost digit specifies 10 to the 0 power, the second digit specifies 10 to the 1 power, the third digit specifies 10 to the 2 power, and so on. Because any number to the 0 power equals 1 and any number to the 1 power equals itself, you have

```
first digit:    10^0 = ones
second digit:   10^1= tens
third digit:    10^2 = hundreds
...
nth digit:      10^(n-1)
```

Example

You can break down the decimal number 382 as follows:

```
382 (decimal)
  "---------- 2 x 10^0 = 2 x 1      =   2
 "----------- 8 x 10^1 = 8 x 10     =  80
"------------ 3 x 10^2 = 3 x 100    = 300
```

```
                                  ------
                              sum = 382
```

*** Base 16**

The hexadecimal system works in the same way except that it uses powers of 16. Because the base is 16, the hexadecimal system requires 16 digits. It uses the regular digits 0–9, and then represents the decimal values 10–15 by the letters A–F. Here are some hex/decimal equivalents:

Hex/Decimal Equivalents

Because some hex numbers (those without letters) look like decimal numbers, they are written with the prefix 0X in order to distinguish them.

Example

Hexadecimal Values	Decimal Values
9	9
A	10
F	15
10	16
1F	31

The following is a three-digit hex number broken down into its decimal components:

```
2DA (hex)
  "--------------  10 x 16^0 = 10 x 1   =  10 (all in decimal)
 "---------------- 13 x 16^1 = 13 x 16  = 208
"-----------------  2 x 16^2 =  2 x 256 = 512
                                        ------
                                           730
```

*** Base 2**

Binary is a base 2 system, and as such requires only two digits, 0 and 1. Each place in a binary number represents a power of 2.

Example

As you can see, binary notation requires many more digits than either decimal or hexadecimal to represent a given value. It is useful, however, because the way a binary number represents values is very similar to the way a computer stores integer values in memory.

Breaking down a binary number gives you the following:

```
10010111 (binary)
       "----------- 1 x 2^0 = 1 x 1      =   1
      "------------ 1 x 2^1 = 1 x 2      =   2
     "------------- 1 x 2^2 = 1 x 4      =   4
    "--------------   x 2^3 = 0 x 8      =   0
   "--------------- 1 x 2^4 = 1 x 16     =  16
```

```
     "----------------   x 2^5 = 0 x 32      =   0
    "-----------------   x 2^6 = 0 x 64      =   0
   "------------------ 1 x 2^7 = 1 x 128     = 128
                                            ------
                                               151
```