



# APPENDIX A

## ANSI Standard Header Files

As you have learned in the past 24 hours, the C standard library comes with a set of include files called *header files*. These header files contain the declarations for the C library functions and macros, as well as relevant data types. Whenever a C function is invoked, the header file(s) with which the C function is associated has to be included in your programs.

The following are the ANSI standard header files:

<i>File</i>	<i>Description</i>
<code>assert.h</code>	Contains diagnostic functions.
<code>ctype.h</code>	Contains character testing and mapping functions.
<code>errno.h</code>	Contains constants for error processing.
<code>float.h</code>	Contains constants for floating-point values.
<code>limits.h</code>	Contains implementation-dependent values.

*continues*

<i>File</i>	<i>Description</i>
<code>locale.h</code>	Contains the <code>setlocale()</code> function, and is used to set locale parameters.
<code>math.h</code>	Contains mathematics functions.
<code>setjmp.h</code>	Contains the <code>setjmp()</code> and <code>longjmp()</code> functions, and is used to bypass the normal function call and return discipline.
<code>signal.h</code>	Contains signal-handling functions.
<code>stdarg.h</code>	Contains functions and macros for implementing functions that accept a variable number of arguments.
<code>stddef.h</code>	Contains definitions for the <code>ptrdiff_t</code> , <code>size_t</code> , <code>NULL</code> , and <code>errno</code> macros.
<code>stdio.h</code>	Contains input and output functions.
<code>stdlib.h</code>	Contains general utility functions.
<code>string.h</code>	Contains functions that are used to manipulate strings.
<code>time.h</code>	Contains functions for manipulating time.



If the C compiler on your machine is not 100 percent ANSI-conformable, some of the ANSI header files might not be available with the compiler.

# APPENDIX **B**



## Answers to Quiz Questions and Exercises

### Hour 1, “Taking the First Step”

#### Quiz

1. The lowest language mentioned in this hour that a computer can understand directly is the machine language—that is, the binary code. On the other hand, the highest language is the human language, such as Chinese, English, French, and so on. Most high-level programming languages, such as C, Java, and Perl, are close to the human language.
2. A computer cannot directly understand a program written in C. You have to compile the program and translate it into binary code so that the computer can read it.
3. Yes. That’s the beauty of the C language; you can write a program in C and save it into a library file. Later, you can invoke the program in another C program by including the library file.

4. We need the ANSI standard for C to guarantee the portability of the programs written in C. Most C compiler vendors support the ANSI standard. If you write your program by following the rules set up by the ANSI standard, you can port your program to any machine by simply recompiling your program with a compiler that supports those machines.

## Hour 2, "Writing Your First C Program"

### Quiz

1. No. Actually, the C preprocessor will filter out all comments you put into your program before the compiler can see them. Comments are written for you or other programmers who look at your program.
2. An `.obj` file is created after a program is compiled by the C compiler. You still need a linker to link all `.obj` files and other library files together to make the final executable file.
3. No, the `exit()` function doesn't return any values. However, the `return` statement does. In the `main()` function, if the `return` statement returns a value of `0`, it indicates to the operating system that the program has terminated normally; otherwise, an error occurs.
4. A file that is required by the `#include` directive and ends with the extension `.h` is called a *header file* in C. Later in this book, you'll learn that a header file contains the data or function declarations.

### Exercises

1. No. The angle brackets (`<` and `>`) in the `#include <stdio.h>` expression ask the C preprocessor to look for a header file in a directory other than the current one. On the other hand, the `#include "stdio.h"` expression tells the C preprocessor to check the current directory first for the header file `stdio.h`, and then look for the header file in another directory.

2. The following is one possible solution:

```
/* 02A02.c */
#include <stdio.h>

main()
{
    printf ("It's fun to write my own program in C.\n");
    return 0;
}
```

**OUTPUT**

The output of the program is:

It's fun to write my own program in C.

3. The following is one possible solution:

```
/* 02A03.c */
#include <stdio.h>

main()
{
    printf ("Howdy, neighbor!\nThis is my first C program.\n");
    return 0;
}
```

**OUTPUT**

The output of the program is:

Howdy, neighbor!  
This is my first C program.

4. The warning message I get when I try to compile the program is that the `main()` function should return a value of integer because, by default, the `main()` function returns an integer. Because the `exit()` function doesn't return any values, you can replace `exit()` with the `return` statement.
5. I got two error (warning) messages on my machine. The first one is 'printf' undefined; the second one is 'main' : 'void' function returning a value. To fix the first error, the header file, `stdio.h`, has to be included first before the `printf()` function can be called from the `main()` function; otherwise, you'll get an error message during the linking stage. To fix the second error, you can remove the `void` keyword from the code.

**B**

## Hour 3, "Learning the Structure of a C Program"

### Quiz

1. Yes. Both 74 and 571 are constants in C.
2. Yes. Both `x = 571 + 1` and `x = 12 + y` are expressions.
3. `2methods`, `*start_function`, and `.End_Exe` are not valid function names.
4. No. `2 + 5 * 2` is equivalent to `2 + 10`, which gives 12; `(2 + 5) * 2` is equivalent to `7 * 2`, which produces 14.
5. Yes. Both `7 % 2` and `4 % 3` produce 1.

## Exercises

1. The following is one possible solution:

```
{
    x = 3;
    y = 5 + x;
}
```

2. The function name, `3integer_add`, is illegal in C.
3. The second statement inside the function needs a semicolon at the end of the statement.
4. The following are two possible solutions:

```
/* Method 1: a C function */
int MyFunction( int x, int y)
{
    int result;
    result = x * y;
    return result;
}
```

or

```
/* Method 2: a C function */
int MyFunction( int x, int y)
{
    return (x * y);
}
```

5. The following is one possible solution:

```
/* 03A05.c */
#include <stdio.h>

int integer_multiply( int x, int y )
{
    int result;
    result = x * y;
    return result;
}

int main()
{
    int sum;

    sum = integer_multiply(3, 5);
    printf("The multiplication of 3 and 5 is %d\n", sum);
    return 0;
}
```

## Hour 4, “Understanding Data Types and Keywords”

### Quiz

1. Yes. Both  $134/100$  and  $17/10$  give the same result of 1.
2. Yes. The results of both  $3000 + 1.0$  and  $3000/1.0$  are floating-point values.
3. In scientific notation, we have the following expressions:
  - $3.5e3$
  - $3.5e-3$
  - $-3.5e-3$
4. Among the four names, `7th_calculation` and `Tom's_method` are not valid names in C.

### Exercises

1. The following is one possible solution:

```
/* 04A01.c */
#include <stdio.h>

main()
{
    char c1;
    char c2;

    c1 = 'Z';
    c2 = 'z';
    printf("The numeric value of Z: %d.\n", c1);
    printf("The numeric value of z: %d.\n", c2);
    return 0;
}
```

#### OUTPUT

The output of the program is:

The numeric value of Z: 90.

The numeric value of z: 122.

2. The following is one possible solution:

```
/* 04A02.c */
#include <stdio.h>

main()
{
    char c1;
```

```

char c2;

c1 = 72;
c2 = 104;
printf("The character of 72 is: %c\n", c1);
printf("The character of 104 is: %c\n", c2);
return 0;
}

```

**OUTPUT**

The output of the program is:

```

The character of 72 is: H
The character of 104 is: h

```

3. No. 72368 is beyond the range of the int data type of 16 bits. If you assign a value that is too large for the data type, the resulting value will wrap around and the result will be incorrect.
4. The following is one possible solution:

```

/* 04A04.c */
#include <stdio.h>

main()
{
    double dbl_num;;

    dbl_num = 123.456;
    printf("The floating-point format of 123.456 is: %f\n",
           dbl_num);
    printf("The scientific notation format of 123.456 is: %e\n",
           dbl_num);

    return 0;
}

```

**OUTPUT**

The output of the program from my machine is:

```

The floating-point format of 123.456 is: 123.456000
The scientific notation format of 123.456 is: 1.234560e+002

```

5. The following is one possible solution:

```

/* 04A05.c */
#include <stdio.h>

main()
{
    char ch;

    ch = '\n';
    printf("The numeric value of newline is: %d\n", ch);

    return 0;
}

```



**OUTPUT**

The output of the program is:

The numeric value of newline is: 10

## Hour 5, “Handling Standard Input and Output”

### Quiz

1. Yes. By prefixing the minimum field specifier with the minus sign -.
2. The main difference between `putc()` and `putchar()` is that `putc()` requires the user to specify the file stream. For `putchar()`, the user doesn’t need to do so because the standard output (`stdout`) is used as the file stream.
3. The `getchar()` function returns a value of the `int` data type.
4. Within the `%10.3f` expression, `10` is the value of the minimum field width specifier; `.3` is called the precision specifier.

### Exercises

1. The following is one possible solution:

```
/* 05A01.c */
#include <stdio.h>

main()
{
    char c1, c2, c3;

    c1 = 'B';
    c2 = 'y';
    c3 = 'e';

    /* Method I */
    printf("%c%c%c\n", c1, c2, c3);

    /* Method II */
    putchar(c1);
    putchar(c2);
    putchar(c3);

    return 0;
}
```

2. The following is one possible solution:

```
/* 05A02.c */
#include <stdio.h>
```

```
main()
{
    int x;
    double y;

    x = 123;
    y = 123.456;
    printf("x: %-3d\n", x);
    printf("y: %-6.3f\n", y);

    return 0;
}
```

**OUTPUT**

The output of the program is:

```
x: 123
y: 123.456
```

3. The following is one possible solution:

```
/* 05A03.c */
#include <stdio.h>

main()
{
    int num1, num2, num3;

    num1 = 15;
    num2 = 150;
    num3 = 1500;
    printf("The hex format of 15 is: 0x%04X\n", num1);
    printf("The hex format of 150 is: 0x%04X\n", num2);
    printf("The hex format of 1500 is: 0x%04X\n", num3);

    return 0;
}
```

**OUTPUT**

The output of the program is:

```
The hex format of 15 is: 0x000F
The hex format of 150 is: 0x0096
The hex format of 1500 is: 0x05DC
```

4. The following is one possible solution:

```
/* 05A04.c */
#include <stdio.h>

main()
{
    int ch;

    printf("Enter a character:\n");
    ch = getchar();
}
```

```

    putchar(ch);

    return 0;
}

```

5. You will probably get two error (warning) messages; one stating that `getchar()` is undefined and another saying that `putchar()` is undefined. The reason is that the header file, `stdio.h`, is missing in the code.

## Hour 6, “Manipulating Data”

### Quiz

1. The `=` operator is an assignment operator that assigns the value of the operand on the right side of the operator to the one on the left side. On the other hand, `==` is one of the relational operators; it just tests the values of two operands on both sides and finds out whether they are equal to each other.
2. In the `x + - y - - z` expression, the first and third minus signs are unary minus operators; the second minus sign is a subtraction operator.
3. `15/4` evaluates to `3`. `(float)15/4` evaluates to `3.750000`.
4. No. The `y *= x + 5` expression is actually equal to the `y = y * (x + 5)` expression.

**B**

### Exercises

1. The following is one possible solution:

```

/* 06A01.c */
#include <stdio.h>

main()
{
    int x, y;

    x = 1;
    y = 3;
    x += y;
    printf("The result of x += y is: %d\n", x);

    x = 1;
    y = 3;
    x += -y;
    printf("The result of x += -y is: %d\n", x);

    x = 1;
    y = 3;

```

```

x -= y;
printf("The result of x -= y is: %d\n", x);

x = 1;
y = 3;
x -= -y;
printf("The result of x -= -y is: %d\n", x);

x = 1;
y = 3;
x *= y;
printf("The result of x *= y is: %d\n", x);

x = 1;
y = 3;
x *= -y;
printf("The result of x *= -y is: %d\n", x);

return 0;
}

```

**OUTPUT**

The output of the program is:

```

The result of x += y is: 4
The result of x += -y is: -2
The result of x -= y is: -2
The result of x -= -y is: 4
The result of x *= y is: 3
The result of x *= -y is: -3

```

2. The value of z is 1 (one), after the expression  $z=x*y=18$  expression is evaluated.
3. The following is one possible solution:

```

/* 06A03.c */
#include <stdio.h>

main()
{
    int x;

    x = 1;
    printf("x++ produces:  %d\n", x++);
    printf("Now x contains: %d\n", x);

    return 0;
}

```

**OUTPUT**

The output of the program is:

```

x++ produces:  1
Now x contains: 2

```

4. The following is one possible solution:

```
/* 06A04.c */
#include <stdio.h>

main()
{
    int x;

    x = 1;
    printf("x = x++ produces: %d\n", x = x++);
    printf("Now x contains:   %d\n", x);

    return 0;
}
```

I get 1 and 1 from the two `printf()` calls in this program. The reason is that, in the `x = x++` expression, the original value of `x` is copied into a temporary location first, and then `x` is incremented by 1. Last, the value saved in the temporary location is assigned back to `x`. That's why the final value saved in `x` is still 1.

5. The program incorrectly uses an assignment operator `=`, instead of an “equal to” relational operator `==`).

B

## Hour 7, “Working with Loops”

### Quiz

1. No.
2. Yes. The do-while loop prints out the character `d`, whose numeric value is 100.
3. Yes. Both for loops iterate 8 times.
4. Yes.

### Exercises

1. The first for loop contains a statement:

```
printf("%d + %d = %d\n", i, j, i+j);
```

But the second for loop has a semicolon right after the for statement. This is a null statement—a semicolon by itself, which is a statement that does nothing.

2. The following is one possible solution:

```
/* 07A02.c */
#include <stdio.h>

main()
{
```

```
int i, j;

for (i=0, j=1; i<8; i++, j++)
    printf("%d + %d = %d\n", i, j, i+j);

printf("\n");
for (i=0, j=1; i<8; i++, j++);
    printf("%d + %d = %d\n", i, j, i+j);

return 0;
}
```

3. The following is one possible solution:

```
/* 07A03.c */
#include <stdio.h>

main()
{
    int c;

    printf("Enter a character:\n(enter K to exit)\n");
    c = ' ';

    while( c != 'K' ) {
        c = getc(stdin);
        putchar(c);
    }
    printf("\nOut of the for loop. Bye!\n");

    return 0;
}
```

4. The following is one possible solution:

```
/* 07A04.c: Use a for loop */
#include <stdio.h>

main()
{
    int i;

    i = 65;
    for (i=65; i<72; i++){
        printf("The numeric value of %c is %d.\n", i, i);
    }

    return 0;
}
```

5. The following is one possible solution:

```
/* 07A05.c */
#include <stdio.h>

main()
{
    int i, j;

    i = 1;
    while (i<=3) { /* outer loop */
        printf("The start of iteration %d of the outer loop.\n", i);
        j = 1;
        do{ /* inner loop */
            printf("    Iteration %d of the inner loop.\n", j);
            j++;
        } while (j<=4);
        i++;
        printf("The end of iteration %d of the outer loop.\n", i);
    }

    return 0;
}
```

B

## Hour 8, “Using Conditional Operators”

### Quiz

1. The  $(x=1)\&\&(y=10)$  expression returns 1;  $(x=1)\&(y=10)$  returns 0.
2. In the  $!y \ ? \ x == z \ : \ y$  expression,  $!y$  produces 0, thus the value of the third operand  $y$  is taken as the value of the expression. That is, the expression evaluates to 1.
3. 1100111111000110 and 0011000000111001.
4. The  $(x\%2==0) \ || \ (x\%3==0)$  expression yields 1, and the  $(x\%2==0)\&\&(x\%3==0)$  expression evaluates to 0.
5. Yes.  $8 \gg 3$  is equivalent to  $8/2^3$ .  $1 \ll 3$  is equivalent to  $2^3$ .

### Exercises

1.  $\sim x$  yields 0x1000 because  $\sim 0xFFFF$  is equivalent to  $\sim 0111111111111111$  (in binary), which produces 1000000000000000 (in binary), (that is, 0x1000 in hex format). Likewise,  $\sim y$  evaluates to 0xFFFF because  $\sim 0x1000$  is equivalent to  $\sim 1000000000000000$  (in binary), which yields 0111111111111111 (in binary) (that is, 0xFFFF in hex format).

2. The following is one possible solution:

```
/* 08A02.c */
#include <stdio.h>

int main()
{
    int x, y;

    x = 0xEFFF;
    y = 0x1000;

    printf("!x yields: %d (i.e., %u)\n", !x, !x);
    printf("!y yields: %d (i.e., %u)\n", !y, !y);

    return 0;
}
```

The output of the program is:

```
!x yields: 0 (i.e., 0)
!y yields: 0 (i.e., 0)
```

3. The following is one possible solution:

```
/* 08A03.c */
#include <stdio.h>

int main()
{
    int x, y;

    x = 123;
    y = 4;
    printf("x << y yields: %d\n", x << y);
    printf("x >> y yields: %d\n", x >> y);

    return 0;
}
```

[ic:output]The output of the program is:

```
x << y yields: 1968
x >> y yields: 7
```

4. The following is one possible solution:

```
/* 08A04.c */
#include <stdio.h>

int main()
{
    printf("0xFFFF ^ 0x8888 yields: 0x%X\n",
```



```

        0xFFFF ^ 0x8888);
printf("0xABCD & 0x4567 yields: 0x%X\n",
       0xABCD & 0x4567);
printf("0xDCBA | 0x1234 yields: 0x%X\n",
       0xDCBA | 0x1234);

return 0;
}

```

**OUTPUT**

The output of the program is:

```

0xFFFF ^ 0x8888 yields: 0x7777
0xABCD & 0x4567 yields: 0x145
0xDCBA | 0x1234 yields: 0xDEBE

```

5. The following is one possible solution:

```

/* 08A05.c */
#include <stdio.h>

main()
{
    int x;

    printf("Enter a character:\n(enter q to exit)\n");
    for ( x=' '; x != 'q' ? 1 : 0; ) {
        x = getc(stdin);
        putchar(x);
    }
    printf("\nOut of the for loop. Bye!\n");

    return 0;
}

```

**B**

## Hour 9, “Working with Data Modifiers and Math Functions”

### Quiz

1. No. `x` contains a negative number that is not the same as the number contained by the unsigned `int` variable `y`.
2. You can use the `long` modifier, which increases the range of values that the `int` can hold. Or, if you are storing a positive number, you can also use the unsigned modifier to store a larger value.
3. `%lu`.
4. The header file is `math.h`.

## Exercises

1. The following is one possible solution:

```
/* 09A01 */
#include <stdio.h>

main()
{
    int x;
    unsigned int y;

    x = 0xAB78;
    y = 0xAB78;

    printf("The decimal value of x is %d.\n", x);
    printf("The decimal value of y is %u.\n", y);

    return 0;
}
```

### OUTPUT

The output of the program from my machine is:

The decimal value of x is -21640.

The decimal value of y is 43896.

2. The following is one possible solution:

```
/* 09A02 */
#include <stdio.h>

main()
{
    printf("The size of short int is %d.\n",
           sizeof(short int));
    printf("The size of long int is %d.\n",
           sizeof(long int));
    printf("The size of long double is %d.\n",
           sizeof(long double));

    return 0;
}
```

3. The following is one possible solution:

```
/* 09A03 */
#include <stdio.h>

main()
{
    int x, y;
    long int result;

    x = 7000;
```

```
y = 12000;
result = x * y;
printf("x * y == %lu.\n", result);

return 0;
}
```

**OUTPUT**

The output of the program from my machine is:

x \* y == 84000000.

4. The following is one possible solution:

```
/* 09A04 */
#include <stdio.h>

main()
{
    int x;

    x = -23456;
    printf("The hex value of x is 0x%X.\n", x);

    return 0;
}
```

**OUTPUT**

The output of the program from my machine is:

The hex value of x is 0xA460.

5. The following is one possible solution:

```
/* 09A05.c */
#include <stdio.h>
#include <math.h>

main()
{
    double x;

    x = 30.0; /* 45 degrees */
    x *= 3.141593 / 180.0; /* convert to radians */
    printf("The sine of 30 is: %f.\n", sin(x));
    printf("The tangent of 30 is: %f.\n", tan(x));

    return 0;
}
```

6. The following is one possible solution (note that I've used the type casting (double) in the assignment statement `x=(double)0x19A1;`):

```
/* 09A06.c */
#include <stdio.h>
#include <math.h>
```

```

main()
{
    double x;

    x = (double)0x19A1;
    printf("The square root of x is: %2.0f\n", sqrt(x));

    return 0;
}

```

## Hour 10, “Controlling Program Flow”

### Quiz

1. No.
2. The final result saved in x is 2, after the execution of three cases, ' - ', ' \* ', and ' / '.
3. The final result saved in x is 2. This time, only the case of operator = ' - ' is executed due to the break statement.
4. The result saved by x is 27.

### Exercises

1. The following is one possible solution:

```

/* 10A01.c Use the if statement */
#include <stdio.h>

main()
{
    int i;

    printf("Integers that can be divided by both 2 and 3\n");
    printf("(within the range of 0 to 100):\n");
    for (i=0; i<=100; i++)
        if (i%6 == 0)
            printf("    %d\n", i);

    return 0;
}

```

2. The following is one possible solution:

```

/* 10A02.c Use the if statement */
#include <stdio.h>

```

```
main()
{
    int i;

    printf("Integers that can be divided by both 2 and 3\n");
    printf("(within the range of 0 to 100):\n");
    for (i=0; i<=100; i++)
        if (i%2 == 0)
            if (i%3 == 0)
                printf("    %d\n", i);

    return 0;
}
```

3. The following is one possible solution:

```
/* 10A03.c */
#include <stdio.h>

main()
{
    int letter;

    printf("Please enter a letter:\n");
    letter = getchar();
    switch (letter){
        case 'A':
            printf("The numeric value of A is: %d\n", 'A');
            break;
        case 'B':
            printf("The numeric value of B is: %d\n", 'B');
            break;
        case 'C':
            printf("The numeric value of C is: %d\n", 'C');
            break;
        default:
            break;
    }

    return 0;
}
```

4. The following is one possible solution:

```
/* 10A04.c */
#include <stdio.h>

main()
{
    int c;
```

```

printf("Enter a character:\n(enter q to exit)\n");
while ((c = getc(stdin)) != 'q') {

    /* no statements inside the while loop */
}
printf("\nBye!\n");

return 0;
}

```

5. The following is one possible solution:

```

/* 10A05.c */
#include <stdio.h>

main()
{
    int i, sum;

    sum = 0;
    for (i=1; i<8; i++){
        if ((i%2 == 0) && (i%3 == 0))
            continue;
        sum += i;
    }
    printf("The sum is: %d\n", sum);
    return 0;
}

```

## Hour 11, "Understanding Pointers"

### Quiz

1. By using the address-of operator, &. That is, the &ch expression gives the left value (the address) of the character variable ch.
2. The answers are as follows:
  - Dereference operator
  - Multiplication operator
  - Multiplication operator
  - The first and third asterisks are dereference operators; the second asterisk is a multiplication operator.
3. ptr\_int yields the value of the address 0x1A38; \*ptr\_int yields the value of 10.
4. x now contains the value of 456.

## Exercises

1. The following is one possible solution:

```
/* 11A01.c */
#include <stdio.h>

main()
{
    int x, y, z;

    x = 512;
    y = 1024;
    z = 2048;

    printf("The left values of x, y, and z are:\n");
    printf("0x%p, 0x%p, 0x%p\n", &x, &y, &z);
    printf("The right values of x, y, and z are:\n");
    printf("%d, %d, %d\n", x, y, z);

    return 0;
}
```

2. The following is one possible solution:

```
/* 11A02.c */
#include <stdio.h>

main()
{
    int *ptr_int;
    char *ptr_ch;

    ptr_int = 0; /* null pointer */
    ptr_ch = 0; /* null pointer */

    printf("The left value of ptr_int is: 0x%p\n",
           ptr_int);
    printf("The right value of ptr_int is: %d\n",
           *ptr_int);
    printf("The left value of ptr_ch is: 0x%p\n",
           ptr_ch);
    printf("The right value of ptr_ch is: %d\n",
           *ptr_ch);

    return 0;
}
```

3. The following is one possible solution:

```
/* 11A03.c */
#include <stdio.h>
```

```

main()
{
    char ch;
    char *ptr_ch;

    ch = 'A';
    printf("The right value of ch is: %c\n",
           ch);
    ptr_ch = &ch;
    *ptr_ch = 'B'; /* decimal 66 */
    /* prove ch has been updated */
    printf("The left value of ch is: 0x%p\n",
           &ch);
    printf("The right value of ptr_ch is: 0x%p\n",
           ptr_ch);
    printf("The right value of ch is: %c\n",
           ch);

    return 0;
}

```

4. The following is one possible solution:

```

/* 11A04.c */
#include <stdio.h>

main()
{
    int x, y;
    int *ptr_x, *ptr_y;

    x = 5;
    y = 6;
    ptr_x = &x;
    ptr_y = &y;

    *ptr_x *= *ptr_y;

    printf("The result is: %d\n",
           *ptr_x);

    return 0;
}

```

## Hour 12, “Understanding Arrays”

### Quiz

1. It declares an `int` array called `array_int` with four elements. The statement also initializes the array with four integers, 12, 23, 9, and 56.



2. Because there are only three elements in the `int` array `data`, and the last element is `data[2]`, the third statement is illegal. It may overwrite some valid data in the memory location of `data[3]`.
3. The first array, `array1`, is a two-dimensional array, the second one, `array2`, is one-dimensional, the third one, `array3`, is three-dimensional, and the last one, `array4`, is a two-dimensional array.
4. In a multidimensional array declaration, only the size of the leftmost dimension can be omitted. Therefore, this declaration is wrong. The correct declaration looks like this:

```
char list_ch[][2] = {
    'A', 'a',
    'B', 'b',
    'C', 'c',
    'D', 'd',
    'E', 'e'};
```

## Exercises

1. The following is one possible solution:

```
/* 12A01.c */
#include <stdio.h>

main()
{
    int i;
    char array_ch[5] = {'A', 'B', 'C', 'D', 'E'};

    for (i=0; i<5; i++)
        printf("%c ", array_ch[i]);

    return 0;
}
```

2. The following is one possible solution:

```
/* 12A02.c */
#include <stdio.h>

main()
{
    int i;
    char array_ch[5];

    for (i=0; i<5; i++)
        array_ch[i] = 'a' + i;
    for (i=0; i<5; i++)
        printf("%c ", array_ch[i]);

    return 0;
}
```

3. The following is one possible solution:

```
/* 12A03.c */
#include <stdio.h>

main()
{
    int i, size;
    char list_ch[][2] = {
        '1', 'a',
        '2', 'b',
        '3', 'c',
        '4', 'd',
        '5', 'e',
        '6', 'f'};

    /* method I */
    size = &list_ch[5][1] - &list_ch[0][0] + 1;
    size *= sizeof(char);
    printf("Method I: The total bytes are %d.\n", size);

    /* method II */
    size = sizeof(list_ch);
    printf("Method II: The total bytes are %d.\n", size);

    for (i=0; i<6; i++)
        printf("%c  %c\n",
            list_ch[i][0], list_ch[i][1]);

    return 0;
}
```

4. The following is one possible solution:

```
/* 12A04.c */
#include <stdio.h>

main()
{
    char array_ch[11] = {'I', ' ',
                        'l', 'i', 'k', 'e', ' ',
                        'C', '!', '\0'};

    int i;
    /* array_ch[i] in logical test */
    for (i=0; array_ch[i]; i++)
        printf("%c", array_ch[i]);

    return 0;
}
```

5. The following is one possible solution:

```
/* 12A05.c */
#include <stdio.h>

main()
{
    double list_data[6] = {
        1.12345,
        2.12345,
        3.12345,
        4.12345,
        5.12345};
    int size;

    /* Method I */
    size = sizeof(double) * 6;
    printf("Method I: The size is %d.\n", size);

    /* Method II */
    size = sizeof(list_data);
    printf("Method II: The size is %d.\n", size);

    return 0;
}
```

**B**

## Hour 13, "Manipulating Strings"

### Quiz

1. The following two statements are legal:
  - `char str2[] = "A character string";`
  - `char str3 = "A";`
2. The following two statements are illegal:
  - `ptr_ch = 'x';`
  - `*ptr_ch = "This is Quiz 2.";`
3. No. The `puts()` function appends a newline character to replace the null character at the end of a character array.
4. The `%s` format specifier is used for reading in a string; the `%f` is for a float number.

## Exercises

1. The following is one possible solution:

```
/* 13A01.c: Copy a string to another */
#include <stdio.h>
#include <string.h>

main()
{
    int i;
    char str1[] = "This is Exercise 1.";
    char str2[20];

    /* Method I */
    strcpy(str2, str1);

    /* confirm the copying */
    printf("from Method I: %s\n", str2);

    /* Method II */
    for (i=0; str1[i]; i++)
        str2[i] = str1[i];
    str2[i] = '\0';
    /* confirm the copying */
    printf("from Method II: %s\n", str2);

    return 0;
}
```

2. The following is one possible solution:

```
/* 13A02.c: Measure a string */
#include <stdio.h>
#include <string.h>

main()
{
    int i, str_length;
    char str[] = "This is Exercise 2.";

    /* Method I */
    str_length = 0;
    for (i=0; str[i]; i++)
        str_length++;
    printf("The string length is %d.\n", str_length);

    /* Method II */
    printf("The string length is %d.\n",
        strlen(str));

    return 0;
}
```

3. The following is one possible solution:

```
/* 13A03.c: Use gets() and puts() */
#include <stdio.h>

main()
{
    char str[80];
    int i, del;

    printf("Enter a string less than 80 characters:\n");
    gets( str );
    del = 'a' - 'A';
    i = 0;
    while (str[i]){
        if ((str[i] >= 'A') && (str[i] <= 'Z'))
            str[i] += del; /* convert to lowercase */
        ++i;
    }
    printf("The entered string is (in lowercase):\n");
    puts( str );

    return 0;
}
```

4. The following is one possible solution:

```
/* 13A04.c: Use scanf() */
#include <stdio.h>

main()
{
    int x, y, sum;

    printf("Enter two integers:\n");
    scanf("%d%d", &x, &y);
    sum = x + y;
    printf("The sum is %d\n", sum);

    return 0;
}
```

**B**

## Hour 14, "Understanding Scope and Storage Classes"

### Quiz

1. The `int` variable `x` and `float` variable `y`, declared outside the `myFunction()` function, are global variables. The `int` variables, `i` and `j`, and the `float` variable `y`,

declared inside the function, are local variables. Also, the two `int` variables, `x` and `y`, declared within a block inside `myFunction()`, are local variables with scope limited to the block.

2. For two variables sharing the same name, the compiler can figure out which one to use by checking their scopes. The latest declared variable becomes visible by replacing the variable that has the same name but is declared in the outer block. If, however, two variables sharing the same name are declared in the same block, the compiler will issue an error message.
3. The `int` variable `i` declared outside the `myFunction()` function has the same static storage class as the `int` variable `x`. The `float` variable `y` has an extern storage class.

Inside the `myFunction()` function, the two integer variables, `i` and `j`, have the auto storage class. The `float` variable `z` has an extern storage class, and the long variable `s` has a register storage class. `index` is an integer variable with a static storage class. The content of the character array `str` cannot be changed due to the `const` specifier.

4. No, it's not legal. You cannot change the content of an array specified by the `const` specifier.

## Exercises

1. The answers are as follows:

- `{ int x; }`
- `{ const char ch; }`
- `{ static float y; }`
- `register int z;`
- `char *ptr_str = 0;`

2. The following is one possible solution:

```
/* 14A02.c */
#include <stdio.h>

int x = 1234;          /* program scope */
float y = 1.234567f;   /* program scope */

void function_1(int x, double y)
{
    printf("From function_1:\n x=%d, y=%f\n", x, y);
}

main()
```

```

{
    int x = 4321;    /* block scope 1*/

    function_1(x, y);
    printf("Within the main block:\n  x=%d, y=%f\n", x, y);
    /* a nested block */
    {
        float y = 7.654321f; /* block scope 2 */
        function_1(x, y);
        printf("Within the nested block:\n  x=%d, y=%f\n", x, y);
    }
    return 0;
}

```

3. The following is what I obtained from running the C program given in this exercise:

```

x=0, y=0
x=0, y=1
x=0, y=2
x=0, y=3
x=0, y=4

```

Because *x* has a temporary storage with the block scope, and *y* has a permanent storage, *x* is set to 0 every time the program execution enters the for loop, but the value saved in *y* is kept.

4. The following is one possible solution:

```

/* 14A04.c: Use the static specifier */
#include <stdio.h>
/* the add_two function */
int add_two(int x, int y)
{
    static int counter = 1;
    static int sum = 0;

    printf("This is the function call of %d,\n", counter++);
    printf("the previous value of sum is %d,\n", sum);
    sum = x + y;
    return sum;
}
/* the main function */
main()
{
    int i, j;

    for (i=0, j=5; i<5; i++, j--)
        printf("the addition of %d and %d is %d.\n\n",
            i, j, add_two(i, j));
    return 0;
}

```

## Hour 15, "Working with Functions"

### Quiz

1. The answers are as follows:
  - `int function_1(int x, int y);` is a function declaration with a fixed number of arguments.
  - `void function_2(char *str);` is a function declaration with a fixed number of arguments.
  - `char *asctime(const struct tm *timeptr);` is a function declaration with a fixed number of arguments.
  - `int function_3(void);` is a function declaration with no arguments.
  - `void function_5(void);` is a function declaration with no arguments.
  - `char function_4(char c, ...);` is a function declaration with a variable number of arguments.
2. The second expression is a function definition; that is,  
`int function_2(int x, int y) {return x+y;}`
3. The `int` data type is the default data type returned by a function if a type specifier is omitted.
4. The third one, `char function_3(...);`, is illegal.

### Exercises

1. The following is one possible solution:

```
/* 15A01.c: */
#include <stdio.h>
#include <time.h>

void GetDateTime(void);

main()
{
    printf("Before the GetDateTime() function is called.\n");
    GetDateTime();
    printf("After the GetDateTime() function is called.\n");
    return 0;
}
/* GetDateTime() definition */
void GetDateTime(void)
{
    time_t now;
    int i;
```



```
char *str;

printf("Within GetDateTime().\n");
time(&now);
str = asctime(localtime(&now));
printf("Current date and time is: ");
for (i=0; str[i]; i++)
    printf("%c", str[i]);
}
```

2. The following is one possible solution:

```
/* 15A02.c */
#include <stdio.h>

int MultiTwo(int x, int y);

main ()
{
    printf("The result returned by MultiTwo() is: %d\n",
        MultiTwo(32, 10));
    return 0;
}

/* function definition */
int MultiTwo(int x, int y)
{
    return x * y;
}
```

3. The following is one possible solution:

```
/* 15A03.c */
#include <stdio.h>
#include <stdarg.h>

int MultiInt(int x, ...);

main ()
{
    int d1 = 1;
    int d2 = 2;
    int d3 = 3;
    int d4 = 4;

    printf("Given an argument: %d\n", d1);
    printf("The result returned by MultiInt() is: %d\n\n",
        MultiInt(1, d1));
    printf("Given an argument: %d, %d, %d, and %d\n", d1, d2, d3, d4);
    printf("The result returned by MultiInt() is: %d\n\n",
        MultiInt(4, d1, d2, d3, d4));

    return 0;
}
```

```

}

/* definition of MultiInt() */
int MultiInt(int x, ...)
{
    va_list arglist;
    int i;
    int result = 1;

    printf("The number of arguments is: %d\n", x);
    va_start (arglist, x);
    for (i=0; i<x; i++)
        result *= va_arg(arglist, int);
    va_end (arglist);
    return result;
}

```

4. The `va_arg()` fetches arguments from left to right on my machine. The following is one possible solution:

```

/* 15A04.c */
#include <stdio.h>
#include <stdarg.h>

double AddDouble(int x, ...);

main ()
{
    double d1 = 1.5;
    double d2 = 2.5;
    double d3 = 3.5;
    double d4 = 4.5;

    printf("Given an argument: %2.1f\n", d1);
    printf("The result returned by AddDouble() is: %2.1f\n\n",
        AddDouble(1, d1));
    printf("Given arguments: %2.1f and %2.1f\n", d1, d2);
    printf("The result returned by AddDouble() is: %2.1f\n\n",
        AddDouble(2, d1, d2));
    printf("Given arguments: %2.1f, %2.1f and %2.1f\n", d1, d2, d3);
    printf("The result returned by AddDouble() is: %2.1f\n\n",
        AddDouble(3, d1, d2, d3));
    printf("Given arguments: %2.1f, %2.1f, %2.1f, and %2.1f\n", d1, d2, d3,
d4);
    printf("The result returned by AddDouble() is: %2.1f\n",
        AddDouble(4, d1, d2, d3, d4));
    return 0;
}

/* definition of AddDouble() */

```

```
double AddDouble(int x, ...)
{
    va_list arglist;
    int i;
    double argument, result = 0.0;

    printf("The number of arguments is: %d\n", x);
    va_start (arglist, x);
    for (i=0; i<x; i++){
        argument = va_arg(arglist, double);
        printf("Argument passed to this function: %f\n", argument);
        result += argument;
    }

    va_end (arglist);

    return result;
}
```

## Hour 16, "Applying Pointers"

### Quiz

1. I obtain the following answers from my machine:
  - 4 bytes
  - 4 bytes
  - 4 bytes
  - 12 bytes
  - 12 bytes
  - 12 bytes
2. Because  $0x100A - 0x1006$  gives 4, and one int takes 2 bytes, ptr1 and ptr2 are two integers apart. Therefore, the answer is 2.
3.  $0x0230$  and  $0x0260$ .
4. The answers are as follows:
  - $*(ptr + 3)$  fetches 'A'.
  - $ptr - ch$  gives 1.
  - $*(ptr - 1)$  fetches 'a'.
  - $*ptr = 'F'$  replaces 'b' with 'F'.

## Exercises

1. The following is one possible solution:

```
/* 16A01.c */
#include <stdio.h>

void StrPrint(char *str);
main()
{
    char string[] = "I like C!";

    StrPrint(string);

    return 0;
}

void StrPrint(char *str)
{
    printf("%s\n", str);
}
```

2. The following is one possible solution:

```
/* 16A02.c */
#include <stdio.h>

void StrPrint(char *str);
main()
{
    char string[] = "I like C!";
    char *ptr;
    int i;

    ptr = string;
    for (i=0; ptr[i]; i++){
        if (ptr[i] == 'i')
            ptr[i] = 'o';
        if (ptr[i] == 'k')
            ptr[i] = 'v';
    }
    StrPrint(ptr);

    return 0;
}

void StrPrint(char *str)
{
    printf("%s\n", str);
}
```

3. The following is one possible solution:

```
/* 16A03.c */
#include <stdio.h>

void StrPrint(char str[][15], int max);
main()
{
    char str[2][15] = {
        "You know what,",
        "C is powerful." };

    StrPrint(str, 2);

    return 0;
}

void StrPrint(char str[][15], int max)
{
    int i;

    for (i=0; i<max; i++)
        printf("%s\n", str[i]);
}
```

4. The following is one possible solution:

```
* 16A04.c */
#include <stdio.h>

/* function declarations */
void StrPrint1(char **str1, int size);
void StrPrint2(char *str2);

/* main() function */
main()
{
    char *str[7] = {
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday"};
    int i, size;

    size = 7;
    StrPrint1(str, size);
    for (i=0; i<size; i++)
        StrPrint2(str[i]);
}
```

```
        return 0;
    }

    /* function definition */
    void StrPrint1(char **str1, int size)
    {
        int i;

        for (i=0; i<size; i++)
            printf("%s\n", str1[i]);
    }

    /* function definition */
    void StrPrint2(char *str2)
    {
        printf("%s\n", str2);
    }
```

## Hour 17, “Allocating Memory”

### Quiz

1. The answers are as follows:
  - 200 bytes
  - 200 bytes
  - 200 bytes
  - 0 bytes
2. The statement is  
`ptr = realloc(ptr, 150 * sizeof(int));`
3. The final size is 120 bytes, provided the `int` data type is one byte long.
4. The final size is 0. In other words, all allocated memory blocks have been released by the last statement.

### Exercises

1. The following is one possible solution:

```
/* 17A01.c */
#include <stdio.h>
#include <stdlib.h>

/* main() function */
main()
{
```

```
int *ptr_int;
int i, sum;
int max = 0;
int termination = 0;

printf("Enter the total number of integers:\n");
scanf("%d", &max);
/* call malloc() */
ptr_int = malloc(max * sizeof(int));
if (ptr_int == NULL){
    printf("malloc() function failed.\n");
    termination = 1;
}
else{
    for (i=0; i<max; i++)
        ptr_int[i] = i + 1;
}
sum = 0;
for (i=0; i<max; i++)
    sum += ptr_int[i];
printf("The sum is %d.\n", sum);
free(ptr_int);

return termination;
}
```

2. The following is one possible solution:

```
/* 17A02.c */
#include <stdio.h>
#include <stdlib.h>

/* main() function */
main()
{
    float *ptr_flt;
    int termination = 0;

    /* call calloc() */
    ptr_flt = calloc(100, sizeof(float));
    if (ptr_flt == NULL){
        printf("calloc() function failed.\n");
        termination = 1;
    }
    else{
        ptr_flt = realloc(ptr_flt, 150 * sizeof(float));
        if (ptr_flt == NULL){
            printf("realloc() function failed.\n");
            termination = 1;
        }
    }
}
```

```

        else
            free(ptr_flt);
    }
    printf("Done!\n");

    return termination;
}

```

3. The following is one possible solution:

```

/* 17A03.c */
#include <stdio.h>
#include <stdlib.h>

/* main() function */
main()
{
    float *ptr1, *ptr2;
    int i;
    int termination = 1;
    int max = 0;

    printf("Enter the total number:\n");
    scanf("%d", &max);

    ptr1 = malloc(max * sizeof(float));
    ptr2 = calloc(max, sizeof(float));

    if (ptr1 == NULL)
        printf("malloc() failed.\n");
    else if (ptr2 == NULL)
        printf("calloc() failed.\n");
    else{
        for (i=0; i<max; i++)
            printf("ptr1[%d]=%5.2f, ptr2[%d]=%5.2f\n",
                i, *(ptr1 + i), i, *(ptr2 + i));
        free(ptr1);
        free(ptr2);
        termination = 0;
    }
    printf (" \nBye!\n");

    return termination;
}

```

4. The following is one possible solution:

```

/* 17A04.c: Use the realloc() function */
#include <stdio.h>
#include <stdlib.h>

```



```
#include <string.h>
/* function declaration */
void StrCopy(char *str1, char *str2);
/* main() function */
main()
{
    char *str[4] = {"There's music in the sighing of a reed;",
                    "There's music in the gushing of a rill;",
                    "There's music in all things if men had ears;",
                    "There earth is but an echo of the spheres.\n"};

    char *ptr;
    int i;

    int termination = 0;

    ptr = realloc(NULL, strlen(str[0]) + 1) * sizeof(char));
    if (ptr == NULL){
        printf("realloc() failed.\n");
        termination = 1;
    }
    else{
        StrCopy(str[0], ptr);
        printf("%s\n", ptr);
        for (i=1; i<4; i++){
            ptr = realloc(ptr, (strlen(str[i]) + 1) * sizeof(char));
            if (ptr == NULL){
                printf("realloc() failed.\n");
                termination = 1;
                i = 4; /* break the for loop */
            }
            else{
                StrCopy(str[i], ptr);
                printf("%s\n", ptr);
            }
        }
    }
    realloc(ptr, 0);
    return termination;
}

/* function definition */
void StrCopy(char *str1, char *str2)
{
    int i;

    for (i=0; str1[i]; i++)
        str2[i] = str1[i];
    str2[i] = '\0';
}
```

## Hour 18, “Using Special Data Types and Functions”

### Quiz

1. The enumerated names Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec represent the values of 0 to 11, respectively.
2. The values of 0, 10, 11, and 12 are represented by the enumerated names name1, name2, name3, and name4, respectively.
3. The typedef long int BYTE32; and BYTE32 x, y, z; statements are equivalent to long int x, y, z;.  
typedef char \*STRING[16]; and STRING str1, str2, str3; are equivalent to char \*str1[16], \*str2[16], \*str3[16];
4. No. The void keyword in the main() function indicates that there is no argument passed to the function.

### Exercises

1. The following is one possible solution:

```
/* 18A01.c */
#include <stdio.h>

main(void)
{
    enum tag {name1,
              name2 = 10,
              name3,
              name4 };

    printf("The value represented by name1 is: %d\n",
           name1);
    printf("The value represented by name2 is: %d\n",
           name2);
    printf("The value represented by name3 is: %d\n",
           name3);
    printf("The value represented by name4 is: %d\n",
           name4);

    return 0;
}
```

2. The following is one possible solution:

```
/* 18A02.c */
#include <stdio.h>
```

```
main(void)
{
    typedef char WORD;
    typedef int SHORT;
    typedef long LONG;
    typedef float FLOAT;
    typedef double DFLOAT;

    printf("The size of WORD is: %d-byte\n", sizeof(WORD));
    printf("The size of SHORT is: %d-byte\n", sizeof(SHORT));
    printf("The size of LONG is: %d-byte\n", sizeof(LONG));
    printf("The size of FLOAT is: %d-byte\n", sizeof(FLOAT));
    printf("The size of DFLOAT is: %d-byte\n", sizeof(DFLOAT));

    return 0;
}
```

3. The following is one possible solution:

```
/* 18A03.c */
#include <stdio.h>

enum con{MIN_NUM = 0,
          MAX_NUM = 100};

int fRecur(int n);

main()
{
    int i, sum1, sum2;

    sum1 = sum2 = 0;
    for (i=1; i<=MAX_NUM; i++)
        sum1 += i;
    printf("The value of sum1 is %d.\n", sum1);
    sum2 = fRecur(MIN_NUM);
    printf("The value returned by fRecur() is %d.\n", sum2);

    return 0;
}

int fRecur(int n)
{
    if (n > MAX_NUM)
        return 0;
    return fRecur(n + 1) + n;
}
```

4. The following is one possible solution:

```
/* 18A04.c: Command-line arguments */
#include <stdio.h>

main (int argc, char *argv[])
```

```

{
    int i;

    if (argc < 2){
        printf("The usage of this program is:\n");
        printf("18A04.EXE argument1 argument2 [...argumentN]\n");
    }
    else {
        printf("The command-line arguments are:\n");
        for (i=1; i<argc; i++)
            printf("%s ", argv[i]);
        printf("\n");
    }

    return 0;
}

```

## Hour 19, “Understanding Structures”

### Quiz

1. The semicolon (;) should be included at the end of the structure declaration.
2. u, v, and w are three structure variables.
3. You can initialize the array of the automobile structure like this:

```

struct automobile {
    int year;
    char model[8]} car[2] = {
    {1997, "Taurus"},
    {1997, "Accord"}};

```

### Exercises

1. The following is one possible solution:

```

/* 19A01.c */
#include <stdio.h>

main(void)
{
    struct automobile {
        int year;
        char model[10];
        int engine_power;
        double weight;
    } sedan = {
        1997,
        "New Model",

```

```
        200,  
        2345.67});  
  
printf("year: %d\n", sedan.year);  
printf("model: %s\n", sedan.model);  
printf("engine_power: %d\n", sedan.engine_power);  
printf("weight: %6.2f\n", sedan.weight);  
  
return 0;  
}
```

2. The following is one possible solution:

```
/* 19A02.c */  
#include <stdio.h>  
  
struct employee {  
    int id;  
    char name[32];  
};  
  
void Display(struct employee s);  
  
main(void)  
{  
    /* structure initialization */  
    struct employee info = {  
        0001,  
        "B. Smith"  
    };  
  
    printf("Here is a sample:\n");  
    Display(info);  
  
    printf("What's your name?\n");  
    gets(info.name);  
    printf("What's your ID number?\n");  
    scanf("%d", &info.id);  
  
    printf("\nHere are what you entered:\n");  
    Display(info);  
  
    return 0;  
}  
  
/* function definition */  
void Display(struct employee s)  
{  
    printf("Employee Name: %s\n", s.name);  
    printf("Employee ID #: %04d\n\n", s.id);  
}
```

3. The following is one possible solution:

```
/* 19A03.c Use the -> operator */
#include <stdio.h>

struct computer {
    float cost;
    int year;
    int cpu_speed;
    char cpu_type[16];
};

typedef struct computer SC;

void DataReceive(SC *ptr_s);

main(void)
{
    SC model;

    DataReceive(&model);
    printf("Here are what you entered:\n");
    printf("Year: %d\n", model.year);
    printf("Cost: %6.2f\n", model.cost);
    printf("CPU type: %s\n", model.cpu_type);
    printf("CPU speed: %d MHz\n", model.cpu_speed);

    return 0;
}

void DataReceive(SC *ptr_s)
{
    printf("The type of the CPU inside your computer?\n");
    gets(ptr_s->cpu_type);
    printf("The speed(MHz) of the CPU?\n");
    scanf("%d", &(ptr_s->cpu_speed));
    printf("The year your computer was made?\n");
    scanf("%d", &(ptr_s->year));
    printf("How much you paid for the computer?\n");
    scanf("%f", &(ptr_s->cost));
}
```

4. The following is one possible solution:

```
/* 19L04.c Arrays of structures */
#include <stdio.h>

struct haiku {
    int start_year;
    int end_year;
```

```
char author[16];
char str1[32];
char str2[32];
char str3[32];
};

typedef struct haiku HK;

void DataDisplay(HK *ptr_s);

main(void)
{
    HK poem[2] = {
        { 1641,
          1716,
          "Sodo",
          "Leading me along",
          "my shadow goes back home",
          "from looking at the moon."
        },
        { 1729,
          1781,
          "Chora",
          "A storm wind blows",
          "out from among the grasses",
          "the full moon grows."
        }
    };

    /* define an array of pointers with HK */
    HK *ptr_poem[2] = {&poem[0], &poem[1]};
    int i;

    for (i=0; i<2; i++)
        DataDisplay(ptr_poem[i]);

    return 0;
}

void DataDisplay(HK *ptr_s)
{
    printf("%s\n", ptr_s->str1);
    printf("%s\n", ptr_s->str2);
    printf("%s\n", ptr_s->str3);
    printf("--- %s\n", ptr_s->author);
    printf("    (%d-%d)\n\n", ptr_s->start_year, ptr_s->end_year);
}
```

## Hour 20, "Understanding Unions"

### Quiz

1. The first statement is the declaration of a union with the tag name of `_union`. The second statement defines two union variables, `x` and `y`, with the `a_union` data type.
2. The semicolon (;) is missed in two places: at the end of the declaration of `char model[8]` and at the end of the declaration of the union.
3. The two union members have the same value, 1997.

### Exercises

1. The following is the modified version. The content of the `name` array is partially overwritten by the value assigned to the `double` variable `price`.

```
/* 20A01.c */
#include <stdio.h>
#include <string.h>

main(void)
{
    union menu {
        char name[23];
        double price;
    } dish;

    printf("The content assigned to the union separately:\n");
    /* access to name */
    strcpy(dish.name, "Sweet and Sour Chicken");
    /* access to price */
    dish.price = 9.95;
    printf("Dish Name: %s\n", dish.name);
    printf("Dish Price: %5.2f\n", dish.price);

    return 0;
}
```

2. The following is one possible solution:

```
/* 20A02.c */
#include <stdio.h>

union employee {
    int start_year;
    int dpt_code;
    int id_number;
};

void DataDisplay(union employee u);
```



```
main(void)
{
    union employee info;

    /* initialize start_year */
    info.start_year = 1997;
    DataDisplay(info);

    /* initialize dpt_code */
    info.dpt_code = 8;
    DataDisplay(info);

    /* initialize id */
    info.id_number = 1234;
    DataDisplay(info);

    return 0;
}

/* function definition */
void DataDisplay(union employee u)
{
    printf("Start Year:  %d\n", u.start_year);
    printf("Dpt. Code:  %d\n", u.dpt_code);
    printf("ID Number:  %d\n", u.id_number);
}
```

**OUTPUT**

The output of the program is

```
Start Year:  1997
Dpt. Code:   1997
ID Number:   1997
Start Year:   8
Dpt. Code:   8
ID Number:   8
Start Year:   1234
Dpt. Code:   1234
ID Number:   1234
```

3. The following is one possible solution:

```
/* 20A03.c */
#include <stdio.h>
#include <string.h>

struct survey {
    char name[20];
    union {
        char state[32];
        char country[32];
    } place;
};
```

```

void DataEnter(struct survey *s);
void DataDisplay(struct survey *s);

main(void)
{
    struct survey citizen;

    DataEnter(&citizen);
    DataDisplay(&citizen);

    return 0;
}

/* definition of DataDisplay() */
void DataDisplay(struct survey *ptr)
{
    printf("\nHere is what you entered: \n");
    printf("Your name is %s.\n", ptr->name);
    printf("You are from %s.\n", ptr->place.state);
    printf("\nThank you!\n");
}

/* definition of DataEnter() */
void DataEnter(struct survey *ptr)
{
    char is_yes[4];

    printf("Please enter your name:\n");
    gets(ptr->name);

    printf("Are you a U. S. citizen? (Yes or No)\n");
    gets(is_yes);
    if ((is_yes[0] == 'Y') ||
        (is_yes[0] == 'y')){
        printf("Enter the name of the state:\n");
        gets(ptr->place.state);
    } else {
        printf("Enter the name of your country:\n");
        gets(ptr->place.country);
    }
}

```

**OUTPUT**

The following is the output of the program on my machine:

Please enter your name:

**Tony**

Are you a U. S. citizen? (Yes or No)

**Yes**

Enter the name of the state:

**Texas**

```
Here is what you entered:  
Your name is Tony.  
You are from Texas.
```

```
Thank you!
```

4. The following is one possible solution:

```
/* 20A04.c */  
#include <stdio.h>  
#include <string.h>  
  
struct bit_field {  
    int yes: 1;  
};  
  
struct survey {  
    struct bit_field flag;  
    char name[20];  
    union {  
        char state[32];  
        char country[32];  
    } place;  
};  
  
void DataEnter(struct survey *s);  
void DataDisplay(struct survey *s);  
  
main(void)  
{  
    struct survey citizen;  
  
    DataEnter(&citizen);  
    DataDisplay(&citizen);  
  
    return 0;  
}  
  
/* function definition */  
void DataEnter(struct survey *ptr)  
{  
    char is_yes[4];  
  
    printf("Please enter your name:\n");  
    gets(ptr->name);  
  
    printf("Are you a U.S. citizen? (Yes or No)\n");  
    gets(is_yes);  
    if ((is_yes[0] == 'Y') ||  
        (is_yes[0] == 'y')){  
        printf("Enter the name of the state:\n");  
        gets(ptr->place.state);  
    }
```

```

        ptr->flag.yes = 1;
    } else {
        printf("Enter the name of your country:\n");
        gets(ptr->place.country);
        ptr->flag.yes = 0;
    }
}
/* function definition */
void DataDisplay(struct survey *ptr)
{
    printf("\nHere is what you've entered:\n");
    printf("Name: %s\n", ptr->name);
    if (ptr->flag.yes)
        printf("The state is: %s\n",
            ptr->place.state);
    else
        printf("Your country is: %s\n",
            ptr->place.country);
    printf("\nThanks and Bye!\n");
}

```

## Hour 21, “Reading and Writing with Files”

### Quiz

1. The first expression tries to open an existing binary file called `test.bin` for reading and writing. The second expression tries to open an existing text file called `test.txt` for appending. The last expression tries to create a text file, called `test.ini`, for reading and writing.
2. The `fopen()` function returns a null pointer when an error occurs during the procedure of opening a file. It's not legal to do any reading or writing with a null file pointer. Therefore, the code is wrong because it calls `fgetc()` when `fopen()` returns a null pointer.
3. The mode is set to read only, but the code tries to write a character to the opened file by calling the `fputc()` function.
4. The code still reads a text file by using the file pointer `fptr1`, even though the file pointer `fptr1` has been closed.

### Exercises

1. The following is one possible solution:

```

/* 21A01.c */
#include <stdio.h>

```

```
enum {SUCCESS, FAIL};

int CharRead(FILE *fin);

main(void)
{
    FILE *fptr;
    char filename[] = "haiku.txt";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "r")) == NULL){
        printf("Cannot open %s.\n", filename);
        reval = FAIL;
    } else {
        printf("\nThe total character number is %d.\n",
            CharRead(fptr));
        fclose(fptr);
    }

    return reval;
}

/* definition of CharRead() */
int CharRead(FILE *fin)
{
    int c, num;

    num = 0;
    while ((c=fgetc(fin)) != EOF){
        putchar(c);
        ++num;
    }

    return num;
}
```

2. The following is one possible solution:

```
/* 21A02.c */
#include <stdio.h>
#include <string.h>

enum {SUCCESS, FAIL, MAX_LEN = 80};

void LineWrite(FILE *fout, char *str);

main(void)
{
    FILE *fptr;
    char str[MAX_LEN+1];
    char filename[32];
    int reval = SUCCESS;
```

```

    printf("Please enter the file name:\n");
    gets(filename);

    printf("Enter a string:\n");
    gets(str);

    if ((fptr = fopen(filename, "w")) == NULL){
        printf("Cannot open %s for writing.\n", filename);
        reval = FAIL;
    } else {
        LineWrite(fptr, str);
        fclose(fptr);
    }

    return reval;
}

/* definition of LineWrite() */
void LineWrite(FILE *fout, char *str)
{
    fputs(str, fout);
    printf("Done!\n");
}

```

3. The following is one possible solution:

```

/* 21A03.c */
#include <stdio.h>

enum {SUCCESS, FAIL};

void CharWrite(FILE *fout, char *str);

main(void)
{
    FILE *fptr;
    char filename[] = "test_21.txt";
    char str[] = "Disk file I/O is fun.";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "w")) == NULL){
        printf("Cannot open %s.\n", filename);
        reval = FAIL;
    } else {
        CharWrite(fptr, str);
        fclose(fptr);
    }

    return reval;
}

```

```
/* function definition */
void CharWrite(FILE *fout, char *str)
{
    int i, c;

    i = 0;
    while ((c=str[i]) != '\0'){
        putchar(c);
        fputc(c, fout);
        i++;
    }
}
```

4. The following is one possible solution:

```
/* 21A04.c */
#include <stdio.h>
#include <string.h>

enum {SUCCESS, FAIL};

void BlkWrite(FILE *fout, char *str);

main(void)
{
    FILE *fptr;
    char filename[] = "test_21.txt";
    char str[] = "Disk file I/O is tricky.";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "w")) == NULL){
        printf("Cannot open %s.\n", filename);
        reval = FAIL;
    } else {
        BlkWrite(fptr, str);
        fclose(fptr);
    }

    return reval;
}

/* function definition */
void BlkWrite(FILE *fout, char *str)
{
    int num;

    num = strlen(str);
    fwrite(str, sizeof(char), num, fout);
    printf("%s\n", str);
}
```

## Hour 22, “Using Special File Functions”

### Quiz

1. Yes. The two statements are equivalent.
2. No. The two statements are not equivalent, unless the current file position indicator is indeed at the beginning of the file.
3. The `scanf()` function reads from the `test.txt` file, instead of the default input stream, because the `freopen()` function has redirected the input stream and associated it with the `test.txt` file.
4. The four double data items together are going to take 32 bytes in the binary file, if the size of the double data type is eight bytes long.

### Exercises

1. The following is one possible solution:

```
/* 22A01.c */
#include <stdio.h>

enum {SUCCESS, FAIL, MAX_LEN = 80};

void PtrSeek(FILE *fptr);
long PtrTell(FILE *fptr);
void DataRead(FILE *fptr);
int ErrorMsg(char *str);

main(void)
{
    FILE *fptr;
    char filename[] = "LaoTzu.txt";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "r")) == NULL){
        reval = ErrorMsg(filename);
    } else {
        PtrSeek(fptr);
        fclose(fptr);
    }

    return reval;
}

/* function definition */
void PtrSeek(FILE *fptr)
{
    long offset1, offset2, offset3;
```



```
offset1 = PtrTell(fp_ptr);
DataRead(fp_ptr);
offset2 = PtrTell(fp_ptr);
DataRead(fp_ptr);
offset3 = PtrTell(fp_ptr);
DataRead(fp_ptr);
printf("\nRe-read the paragraph:\n");

/* re-read the third sentence */
fseek(fp_ptr, offset3, SEEK_SET);
DataRead(fp_ptr);

/* re-read the second sentence */
fseek(fp_ptr, offset2, SEEK_SET);
DataRead(fp_ptr);

/* re-read the first sentence */
fseek(fp_ptr, offset1, SEEK_SET);
DataRead(fp_ptr);
}

/* function definition */
long PtrTell(FILE *fp_ptr)
{
    long reval;

    reval = ftell(fp_ptr);
    printf("The fp_ptr is at %ld\n", reval);

    return reval;
}

/* function definition */
void DataRead(FILE *fp_ptr)
{
    char buff[MAX_LEN];

    fgets(buff, MAX_LEN, fp_ptr);
    printf("%s", buff);
}

/* function definition */
int ErrorMsg(char *str)
{
    printf("Cannot open %s.\n", str);

    return FAIL;
}
```

2. The following is one possible solution:

```

/* 22A02.c */
#include <stdio.h>

enum {SUCCESS, FAIL, MAX_LEN = 80};

void PtrSeek(FILE *fptr);
long PtrTell(FILE *fptr);
void DataRead(FILE *fptr);
int ErrorMsg(char *str);

main(void)
{
    FILE *fptr;
    char filename[] = "LaoTzu.txt";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "r")) == NULL){
        reval = ErrorMsg(filename);
    } else {
        PtrSeek(fptr);
        fclose(fptr);
    }

    return reval;
}

/* function definition */
void PtrSeek(FILE *fptr)
{
    long offset1, offset2, offset3;

    offset1 = PtrTell(fptr);
    DataRead(fptr);
    offset2 = PtrTell(fptr);
    DataRead(fptr);
    offset3 = PtrTell(fptr);
    DataRead(fptr);
    printf("\nRe-read the paragraph:\n");
    /* re-read the third sentence */
    fseek(fptr, offset3, SEEK_SET);
    DataRead(fptr);
    /* re-read the second sentence */
    fseek(fptr, offset2, SEEK_SET);
    DataRead(fptr);
    /* re-read the first sentence */
    rewind(fptr); /* rewind the file position indicator */
    DataRead(fptr);
}

```

```
/* function definition */
long PtrTell(FILE *fptr)
{
    long reval;

    reval = ftell(fptr);
    printf("The fptr is at %ld\n", reval);

    return reval;
}

/* function definition */
void DataRead(FILE *fptr)
{
    char buff[MAX_LEN];

    fgets(buff, MAX_LEN, fptr);
    printf("%s", buff);
}

/* function definition */
int ErrorMsg(char *str)
{
    printf("Cannot open %s.\n", str);
    return FAIL;
}
```

3. On my machine, the data.bin binary file is 10 bytes. The following is one possible solution:

```
/* 22A03.c */
#include <stdio.h>

enum {SUCCESS, FAIL};

void DataWrite(FILE *fout);
void DataRead(FILE *fin);
int ErrorMsg(char *str);

main(void)
{
    FILE *fptr;
    char filename[] = "data.bin";
    int reval = SUCCESS;

    if ((fptr = fopen(filename, "wb+")) == NULL){
        reval = ErrorMsg(filename);
    } else {
        DataWrite(fptr);
        rewind(fptr);
        DataRead(fptr);
    }
```

```

        fclose(fptr);
    }

    return reval;
}

/* function definition */
void DataWrite(FILE *fout)
{
    double dnum;
    int inum;

    dnum = 123.45;
    inum = 10000;

    printf("%5.2f\n", dnum);
    fwrite(&dnum, sizeof(double), 1, fout);
    printf("%d\n", inum);
    fwrite(&inum, sizeof(int), 1, fout);
}

/* function definition */
void DataRead(FILE *fin)
{
    double x;
    int y;

    printf("\nRead back from the binary file:\n");
    fread(&x, sizeof(double), (size_t)1, fin);
    printf("%5.2f\n", x);
    fread(&y, sizeof(int), (size_t)1, fin);
    printf("%d\n", y);
}

/* function definition */
int ErrorMessage(char *str)
{
    printf("Cannot open %s.\n", str);
    return FAIL;
}

```

4. The following is one possible solution:

```

/* 22A04.c */
#include <stdio.h>

enum {SUCCESS, FAIL,
      MAX_NUM = 3,
      STR_LEN = 23};

void DataRead(FILE *fin);
int ErrorMessage(char *str);

```

```
main(void)
{
    FILE *fptr;
    char filename[] = "strnum.mix";
    int reval = SUCCESS;

    if ((fptr = freopen(filename, "r", stdin)) == NULL){
        reval = ErrorMsg(filename);
    } else {
        DataRead(fptr);
        fclose(fptr);
    }

    return reval;
}

/* function definition */
void DataRead(FILE *fin)
{
    int i;
    int miles;
    char cities[STR_LEN];

    printf("The data read:\n");
    for (i=0; i<MAX_NUM; i++){
        scanf("%s%d", cities, &miles);
        printf("%-23s  %d\n", cities, miles);
    }
}

/* function definition */
int ErrorMsg(char *str)
{
    printf("Cannot open %s.\n", str);
    return FAIL;
}
```

**B**

## Hour 23, "Compiling Programs: The C Preprocessor"

### Quiz

1. The semicolon (;) should not be included at the end of the macro definition because a macro definition ends with a newline, not a semicolon.
2. The value of 82 is assigned to `result` due to the assignment expression `result = 1 + 9 * 9`.

3. The message of Under `#else`. is printed out.
4. The message of Under `#ifdef`. is printed out.

## Exercises

1. The following is one possible solution:

```
/* 23A01.c */
#include <stdio.h>
/* main() function */
main()
{
    #define human      100
    #define animal     50
    #define computer   51
    #define SUN        0
    #define MON        1
    #define TUE        2
    #define WED        3
    #define THU        4
    #define FRI        5
    #define SAT        6

    printf("human: %d, animal: %d, computer: %d\n",
           human, animal, computer);
    printf("SUN: %d\n", SUN);
    printf("MON: %d\n", MON);
    printf("TUE: %d\n", TUE);
    printf("WED: %d\n", WED);
    printf("THU: %d\n", THU);
    printf("FRI: %d\n", FRI);
    printf("SAT: %d\n", SAT);

    return 0;
}
```

2. The following is one possible solution:

```
/* 23A02.c */
#include <stdio.h>

#define MULTIPLY(val1, val2) ((val1) * (val2))
#define NO_ERROR 0

main(void)
{
    int result;

    result = MULTIPLY(2, 3);
```

```
    printf("MULTIPLY(2, 3) produces value of %d.\n", result);

    return NO_ERROR;
}
```

3. The following is one possible solution:

```
/* 23A03.c */
#include <stdio.h>

#define UPPER_CASE 0
#define NO_ERROR 0

main(void)
{
    #if UPPER_CASE
        printf("THIS LINE IS PRINTED OUT,\n");
        printf("BECAUSE UPPER_CASE IS DEFINED.\n");
    #elif LOWER_CASE
        printf("This line is printed out,\n");
        printf("because LOWER_CASE is defined.\n");
    #else
        printf("This line is printed out,\n");
        printf("because neither UPPER_CASE nor LOWER_CASE is defined.\n");
    #endif

    return NO_ERROR;
}
```

4. The following is one possible solution:

```
/* 23A04.c: */
#include <stdio.h>

#define C_LANG 'C'
#define B_LANG 'B'
#define NO_ERROR 0

main(void)
{
    #if C_LANG == 'C'
        #if B_LANG == 'B'
            #undef C_LANG
            #define C_LANG "I know C language.\n"
            #undef B_LANG
            #define B_LANG "Also, I know BASIC.\n"
            printf("%s%s", C_LANG, B_LANG);
        #else
            #undef C_LANG
            #define C_LANG "I only know C language.\n"
            printf("%s", C_LANG);
        #endif
    #endif
}
```

```
        #endif
    #elif B_LANG == 'B'
        #undef B_LANG
        #define B_LANG "I only know BASIC.\n"
        printf("%s", B_LANG);
    #else
        printf("I don't know C or BASIC.\n");
    #endif

    return NO_ERROR;
}
```





# INDEX

## Symbols

- +=** (addition assignment operator), 93
- &** (ampersand)
  - address-of operator, 177-179
  - bitwise AND operator, 131
- <>** (angle brackets), 32
- >** (arrow operator), unions, 335, 351
- =** (assignment operator), 92
- \*** (asterisk)
  - deference operator, 182
  - determining meaning, 182
  - multiplication operator, 182
  - pointers, 180
- ~** (bitwise complement operator), 131
- |** (bitwise OR operator), 131
- ^** (bitwise XOR operator), 131
- { }** (braces), 45, 48
  - if statement, 156
  - if-else statement, 159
- [ ]** (brackets), 190
- \*/** (closing comment mark), 29
- ?:** (conditional operator), 135-136
- (decrement operator), 96-98
- /=** (division assignment operator), 93
- .** (dot operator), unions, 335-337, 351
- "** (double quotes), 32-33, 59
- ==** (equal to operator), 98
- \** (escape character), 59
- %%** format specifier, 79
- >** (greater than operator), 98
- >=** (greater than or equal to operator), 98
- ++** (increment operator), 96-98
- <<** (left-shift operator), 133-135
- <** (less than operator), 98
- <=** (less than or equal to operator), 98
- &&** (logical AND operator), 124-126
- !** (logical NEGATION operator), 128-129
- ||** (logical OR operator), 126-127
- \*=** (multiplication assignment operator), 93

**!= (not equal to operator), 98**  
**\0 (null character), 198**  
**/\* (opening comment mark), 29**  
**// (opening comment mark), 30**  
**() (parentheses)**  
     if statement, 156  
     placing around expressions, 419  
**% (remainder operator), 43**  
**%= (remainder assignment operator), 93**  
**>> (right-shift operator), 133-135**  
**;(semicolons), 28**  
**' (single quotes), 59**  
**-= (subtraction assignment operator), 93**  
**- (subtraction operator), 96**  
**- (unary minus operator), 95**  
**19L02.exe executable, 318**

## A

### access

    random  
         code example, 375-378  
         disk files, 374-377, 387  
         fseek() function, 374-378

        ftell() function, 374-378  
         sequential disk files, 374  
**accessing**  
     array elements, indexes, 190  
     arrays, via pointers, 264-266  
**addition assignment operator (+=), 93**  
**address variables. *See* pointers**  
**address-of operator (&), 177-179, 323**  
**addresses**  
     left value, 176  
     memory, 343  
**algorithms, implementing, 305**  
**aligning output, 83-84**  
**allocating memory**  
     calloc() function, 286-288  
     malloc() function, 280-283  
**American National Standards Institute. *See* ANSI**  
**angle brackets (<>), 32**  
**ANSI (American National Standards Institute), 15**  
     C standard, 15-16  
     header files, 439  
**applying static specifiers, 230-231**  
**argc arguments, 306**  
**argument lists, 47**  
**arguments**  
     argc, 306  
     argv, 306

    built-in  
         main() functions, 306  
         naming, 308  
         replacing, 308  
     command-line, 305  
         receiving, 306-308  
     passing to functions, 47-48, 305  
     variable, processing, 252-254  
**argv arguments, 306**  
**arithmetic assignment operators, 92-95**  
     addition assignment (+=), 93  
     division assignment (/=), 93  
     multiplication assignment (\*=), 93  
     remainder assignment (%=), 93  
     subtraction assignment (-=), 93  
**arithmetic expressions, #if directive, 406**  
**arithmetic operators, 43-44**  
**array data type, 424**  
**array element references, 191**  
**array subscript operator ([ ]), 190**  
**arrays, 190**  
     accessing via pointers, 264-266  
     character, 190, 210-211  
     displaying, 196-198  
     initializing, 208-209  
     declaring, 190

- elements, 190
  - accessing, 190
  - initializing, 191-192
  - integers, 190
- multidimensional
  - declaring, 199
  - displaying, 200-201
  - initializing, 199-201
- multidimensional
  - unsized
    - declaring, 201
    - initializing, 202-203
- listing, 325-326
- passing to functions, 266-267, 270-272
- pointers
  - declaring, 272
  - referencing with, 195-196
- strings, 272-274
- sizes
  - calculating, 192-194
  - specifying, 267
- of structures, 324-327
- unsized
  - calculating, 201
  - versus structures, 314
- unsized character, 209
- arrow (->) operators, 324, 335, 351**
- ASCII character codes, 57**
- asctime() function, 250-251**
- assert.h header file, 439**
- assessing structure members, 315**
- assigning**
  - character constants to pointers, 210
  - character strings to pointers, 210-211

- integers to structures, 315
- inter values to enum data types, 296, 300
- values
  - to enum data types, 299
  - to pointers, 180-181
- assignment operator (=), 92**
- asterisks (\*)**
  - deference operator, 182
  - determining meaning, 182
  - multiplication operator, 182
  - pointers, 180
- auto keyword, 56**
- auto specifier, variables, 229**
- avoiding**
  - duplication, structures, 322
  - goto statements, 168

## B

- \b (backspace character), 60**
- backslash. See escape character**
- big-endian format, 343**
- binary code, 14, 24**
- binary files**
  - reading, 378-381
  - writing, 378-381
- binary format, 13**
- binary numbers**
  - converting decimal numbers to, 129-130
  - negatives, 142

- binary operators, multiplication (\*), 182**
- binary streams, 356, 370**
- bit fields**
  - code example, 348-350
  - declaring, 347
  - defining, 347
- bit-manipulation operators, 130**
  - bitwise AND (&), 131
  - bitwise complement (~), 131
  - bitwise OR (|), 131
  - bitwise XOR (^), 131
  - left-shift (<<), 133-135
  - right-shift (>>), 133-135
- bits, 14, 58, 142**
- bitwise AND operator (&), 131**
- bitwise complement operator (~), 131**
- bitwise OR operator (|), 131**
- bitwise XOR operator (^), 131**
- block scope (variables), 224-225**
  - local variables, 225
  - nested, 225-226
  - program scope comparison, 227-229
- BlockReadWrite() function, 369**
- blocks**
  - commenting out, 31
  - statement, 45-46
- body, functions, 48-49**
- books (recommended reading), 434-435**

**Borland C++ compiler,**  
**21-24**  
     running, 23  
     starting, 21  
**bottom-up programming,**  
**255**  
**braces ({}), 45, 48**  
     if statement, 156  
     if-else statement, 159  
     switch statement, 165  
**brackets ([]), declaring**  
**arrays, 190**  
**break keyword, 56**  
**break statements, 155,**  
**164-165**  
     infinite loops, breaking,  
     166-167  
     listing, 164-165  
     location, 164  
     switch statements, exit-  
     ing, 164  
**breaking code lines, 28**  
**buffered I/O, 433**  
**buffers, 356**  
     flushing, 356  
     high-level I/O, 357  
     low-level I/O, 357, 387  
     setbuf() functions, 387  
     setvbuf() functions, 387  
**bugs. See debugging**  
**built-in arguments**  
     main() functions, 306  
     naming, 308  
     replacing, 308  
**bytes, 14, 58, 192-194**

## C

**%c (character) format**  
**specifier, 60-62, 74, 79**  
**.c filename extension, 28**  
**C**  
     advantages, 12-14  
     history, 12  
     portability, 13  
     programs  
         maintaining, 296  
         readability, improv-  
         ing, 296, 300  
     structured program-  
     ming, 169  
**C compiler, C preprocess-  
 or comparison, 392-  
 393, 405**  
**C preprocessor, 392**  
     # (pound sign), 392  
     C compiler comparison,  
     392-393, 405  
     #define directive, 393,  
     434  
         code example,  
         394-396  
         defining function-  
         like macros,  
         394-396  
         expressions, 396  
         nested macro defini-  
         tions, 396  
         syntax, 393  
     #elif directive, 401-402,  
     434  
     #else directive,  
     399-402, 434  
     #endif directive,  
     397-402, 406, 416  
         code example,  
         398-399  
         syntax, 397

    #if directive, 434  
         arithmetic expres-  
         sions, 406  
         code example,  
         401-402  
         macro definitions,  
         400  
         nested conditional  
         compilation,  
         402-404  
         syntax, 399  
     #ifdef directive,  
     397-399, 434  
     #ifndef directive,  
     397-399, 416  
     macro body, 392, 396  
     macro names, 392-393  
     macro substitution,  
     392-396  
     newline characters, 392  
     #undef directive,  
     393-394, 406, 434

## *C Programming Language, The, 15*

**C++, 14**  
**calculating array size,**  
**192-194, 201**  
**calendar time, date and**  
**time functions, 249**  
**calling**  
     functions, 49-51,  
     245-247  
         no arguments, 249  
         recursive, 303-304  
**calloc() function, 286-288**  
     listing, 287  
     malloc() functions,  
     compared, 292  
**case keyword, 56, 162**  
**case sensitivity, file-**  
**names, 32**

**cast operator, 101-102**

**central processing unit (CPU), 13**

**changing variable values via pointers, 183-184**

**char data type, 47, 57**

**char keyword, 56**

**character arrays. *See***

***also strings***

initializing, 208-209

string constants, initial-  
izing, 208-211

unsized, 209

**character codes (ASCII),  
57**

**character constants,**

**58-59, 422-423**

pointers, assigning, 210

string constants, com-  
pared, 209-212

**character data type. *See***

***char data type***

**character format speci-  
fier (%c), 60-62, 74, 79**

**character strings, 196,  
210-211**

**character variables, 58**

**characters**

arrays, 190, 196-198

converting numeric val-  
ues to, 61

null, 198. *See also*

*strings*

numeric values

converting, 61

showing, 63-64

printing, 60-62

reading

from standard input

stream, 215-217

gets() function,

215-217

writing

from standard output

stream, 217

puts() function,

215-217

to standard output

stream, 215-216

**CharReadWrite() func-  
tion, 362-363**

**checking command-line  
arguments, 307**

**classes, storage, 229**

**closing files, fclose() func-  
tion, 358-360, 371**

**closing comment mark  
(\*/), 29**

**code, 14. *See also* listings**

binary, 14

breaking lines, 28

comments, 29-31, 418

commenting out, 31

nested, 31

performance, 30

indentation, 28, 419

saving

Borland C++, 23

Visual C++, 19

source code file, 34

spacing, 419

spaghetti, 169

syntax, checking, 36

whitespace, 29

writing

Borland C++, 21

Visual C++, 18

**codes, executing if state-  
ment, 156**

**coding style, 418-419**

**collecting variables,  
groups, 314**

**combining declarations  
and definitions, 315**

**command-line argu-  
ments, 305**

listing, 306-307

number, checking, 307

receiving, 306-308

**comments, 29-31, 418**

commenting out code,  
31

nested, 31

program performance,  
30

**compiled languages, 14-  
15**

**compilers, 13**

accessing, 17

Borland C++, 21-24

C, C preprocessor com-  
parison, 392-393, 405

choosing, 17

compliance errors,  
280-281

Microsoft, 18-21

optimizers, turning off,  
235

**compliance error,  
280-281**

**conditional branching  
statements, 155,  
428-430**

break, 155, 164-165

continue, 155, 167-168

goto, 155, 168-169

if, 155-158

if-else, 155, 158-159

labels, 162

switch, 155, 161-164

**conditional compilation,  
397**

#elif directive, 401-402

#else directive, 399-402

- #endif directive,
  - 397-402, 406
  - code example,
    - 398-399
  - syntax, 397
- #if directive
  - arithmetic expressions, 406
  - code example,
    - 401-402
  - macro definitions, 400
  - nested conditional compilation,
    - 402-404
  - syntax, 399
- #ifdef directive,
  - 397-399
- #ifndef directive,
  - 397-399
- conditional expressions, switch statement, 161-162**
- conditional operator (:?), 135-136**
- conditions, evaluating (if statement), 156**
- consolidating data types, 300**
- const keyword, 56**
- const modifier (variables), 234-235**
- constants, 42**
  - character, 58-59,
    - 209-212, 422-423
  - EOF (end-of-file), 73
  - floating-point, 423
  - integer, 422
  - macro body, 392
  - named, versus numeric, 419
  - named integer, declaring, 296

- numeric, versus named, 419
- string, 209-212, 423
- content, memory, 177**
- continue keyword, 56**
- continue statement, 155, 167-168**
  - listing, 167
  - loops, 167, 171
- control flow, 426**
  - conditional branching statements, 155, 428-430
    - break, 164-165
    - continue, 167-168
    - goto, 168-169, 419
    - if, 156-158
    - if-else, 158-159
    - switch, 161-164
  - looping statements, 155, 427-428
- Convert2Upper() function, 303**
- converting**
  - data types, cast operator, 101-102
  - dates, asctime() function, 250-251
  - numbers
    - decimal to binary, 129-130
    - decimal to hex, 129-130
  - time, asctime() function, 250-251
  - to uppercase, 302-303
- copying strings, 213-215**
- cos() function, 149-150**
- CPU (central processing unit) register, 233**
- CPUs (central processing units), 13**

- creating**
  - declarations, 299
  - expressions, 299
  - names for data types, 300-302
- ctype.h header file, 439**

## D

- %d (integer) format specifier, 63-64, 79**
- data**
  - formatted
    - fprintf() function, 381-384, 388
    - fscanf() function, 381-384
  - stack, overwriting, 305
- data items, separating, 317**
- data modifiers**
  - long, 145-147
  - short, 145
  - signed, 142-143
  - unsigned, 143-145
- data structures, linked lists, 410**
  - advantages, 410
  - creating, 410-418
- data types, 423. *See also* variables**
  - array, 424
  - char, 47, 57
  - consolidating, 300
  - converting, cast operator, 101-102
  - creating, 426
  - defining, variable lists, 424
  - double, 67

- enum, 296, 424
  - declaring, 296
  - defining variables, 296
- float, 47, 64
- int, 46, 62
- names, 300-302
- pointers, moving, 260-262
- size, changing, 145-147
- sizes, measuring, 122-123
- struct, 424-425
  - bit fields, 347-350
- structures, 314
- union, 425-426
- void, function declarations, 248-249
- DataDisplay() function, 326, 346**
- DataEnter() function, 346**
- DataRead() function, 377, 381, 384**
- DataReceive() function, 321**
- DataWrite() function, 381, 384**
- date and time functions, 249**
- dates, converting, 250-251**
- daylight savings time, 249**
- debugging, 37**
  - bugs, 420
  - checking syntax, 36
  - error messages, 36
- decimal numbers**
  - converting to binary, 129-130
  - converting to hex, 79-81, 129-130
- decisions, unlimited (switch statements), 161**
- declaring. *See also* defining; prototyping functions**
  - arrays, 190
    - [ ] (brackets), 190
    - of pointers, 272
    - of structures, 324-327
  - bit fields, 347-350
  - creating declarations, 299
  - definitions
    - combining, 315
    - compared, 244, 432
  - enum data types, 296
  - functions, 244-249
    - prototypes, 245
    - specifying return types, 244
  - getchar() function, 248
  - global variables, 229
  - main() functions, 306
  - members, structures, 314
  - multidimensional arrays, 199
  - multidimensional unsized arrays, 201
  - named integer constants, 296
  - nested structures, 327
  - pointers, 180-181, 274-275
  - structures, 314-315
  - synonyms, 300
  - unions, 334
  - unsized character arrays, 209
  - variables, 48, 177, 244
    - floating-point numbers, 64-65
    - integers, 62-63
- decrement operator (--), 96-98**
- default keyword, 56**
- default values**
  - enum data types, 298
  - integers, 296
- deference operator (\*), 182**
- #define directive, 296, 393, 434**
  - code example, 394-396
  - defining function-like macros, 394-396
  - expressions, 396
  - nested macro definitions, 396
  - syntax, 393
- defining. *See also* declaring**
  - bit fields, 347-350
  - data types, variable lists, 424
  - declarations
    - combining, 315
    - compared, 244, 432
  - #define directive, 296, 393, 434
    - code example, 394-396
  - defining function-like macros, 394-396
  - expressions, 396
  - function-like macros, 394-396
  - nested macro definitions, 396
  - syntax, 393
  - functions, 244-247
  - macros, #if directives, 400
  - structure variables, 315

- synonyms, 321
- variable lists, enum data types, 296
- variables, 244
  - enum data types, 296
  - unions, 334-335
- dereferenced pointers, 323**
- directives (preprocessor)**
  - `#define`, 296, 393
    - code example, 394-396
  - defining function-like macros, 394-396
  - expressions, 396
  - nested macro definitions, 396
  - syntax, 393
  - `#elif`, 401-402
  - `#else`, 399-402
  - `#endif`, 397-402, 406
    - code example, 398-399
  - syntax, 397
  - `#if`
    - arithmetic expressions, 406
    - code example, 401-402
    - macro definitions, 400
    - nested conditional compilation, 402-404
    - syntax, 399
  - `#ifdef`, 397-399
  - `#ifndef`, 397-399
  - `#include`, 31
  - `#undef`, 393-394, 406
- disabling buffering, 387**

- displaying**
  - arrays of characters, 196-198
  - multidimensional arrays, 200-201
- division, truncation, 100**
- division assignment operator (`/=`), 93**
- do keyword, 56**
- do-while loops, 107-109**
- dot operator (`.`), 315-317**
  - unions, 335-337, 351
- double data type, 67**
- double keyword, 56**
- double quotes (`"`), 32-33, 59**
- duplication, avoiding (structures), 322**
- durations (variables), 229**

## E

- `%e` (scientific notation) format specifier, 67, 79**
- elements (arrays), 190-192**
- `#elif` directive, 401-402, 434**
- `#else` directive, 434**
  - code example, 401-402
  - syntax, 399
- `else` keyword, 56**
- end-of-file (EOF), 73**
- `#endif` directive, 397-402, 406, 416**
  - code example, 398-399
  - syntax, 397
- ending outputs, null characters, 198-199**
- enum data type, 296, 424**
  - declarations, 296
  - declaring, 296
  - defining, listing, 297
  - defining variables, 296
  - integer values, assigning, 296, 300
  - listing, 298-299
  - values, 298-299
  - variable lists, 296
- enum keyword, 56**
- enumerated data type. *See* enum data type**
- EOF (end-of-file) constant, 73**
- equal to operator (`==`), 98**
- `errno.h` header file, 439**
- error messages, 36**
- errors**
  - compliance (compilers), 280-281
  - pointers, uninitialized, 262
- escape character (`\`), 59**
- evaluating conditions, if statement, 156**
- `.exe` filename extension, 29**
- executable files, 24, 34-35, 318**
- executing**
  - codes, if statement, 156
  - programs, 305
  - statements, switch, 164
- exhausting stack resources, 305**
- `exit()` function, 34**
- exiting switch statements, 164**
- exponents, 67**



**expressions, 42-43, 426.**

*See also operators*

arithmetic, 406

conditional, switch

statement, 161-162

creating, 299

#define directive, 396

in for loops, 113-115

macro body, 396

placing parentheses

around, 419

return values in, 47

**extern keyword, 56, 234**

## F

**%f (floating-point) format specifier, 65-67, 79**

**\\f (form-feed character), 60**

**fclose() function**

code example, 359-360

files, closing, 358-360, 371

syntax, 359

**feof() function, 367**

code example, 368-369

syntax, 367

**fgetc() function**

code example, 361-363

files, reading, 360-363

syntax, 361

**fgets() function**

code example, 364-366

files, reading, 363-366

gets() function comparison, 364-366, 371

reading keyboard input, 366

syntax, 363

**file pointers, 357, 371**

**file position indicators.**

*See FILE structure*

**file scope (variables), 232**

**file streams, 433**

**FILE structure, 357**

file position indicators, 357

fseek() function, 374

ftell() function, 375

rewind() function, 378

**filename extensions, 28-29**

**files, 433**

binary

reading, 378-381

writing, 378-381

defined, 356

executable, 24, 34-35

fclose() function

closing files, 358-360, 371

code example, 359-360

syntax, 359

feof() function, 367-369

fgetc() function

code example, 361-363

reading files, 360-363

syntax, 361

fgets() function

code example, 364-366

gets() function comparison, 364-366, 371

reading files, 363-366

reading keyboard

input, 366

syntax, 363

file pointers, 357, 371

FILE structure, 357, 374-375, 378

fopen() function, 357, 381

code example,

359-360

formatting, 388

modes, 357-358, 388

opening files, 358

syntax, 357

fputc() function,

360-363

fputs() function,

364-366

fread() function, 381

code example,

368-369

reading files, 366-369

syntax, 366

fseek() function,

374-378

ftell() function, 374-378

fwrite() function, 381

code example, 368-369

syntax, 367

writing files,

366-369

header, 32

ANSI, 439

stddef.h, 301

stdio.h, 32

stdlib.h, 302

string.h, 302

naming, case sensitivity, 32

- object, 34
- random access, 374-378, 387
- rewind() function, 378
  - code example, 379-384
  - syntax, 378
- sequential access, 374
- source code, 34
- streams comparison, 356
- float data type, 47, 64**
- float keyword, 56**
- float.h header file, 439**
- floating data type. *See* float data type**
- floating-point constants, 423**
- floating-point format specifier (%f), 65-67, 79**
- floating-point numbers, 64**
  - calculations, 152
  - declaring, 64-65
- flushing buffers, 356**
- fopen() function, 381**
  - code example, 359-360
  - files, opening, 357-358
  - formatting new files, 388
  - modes, 357-358, 388
  - syntax, 357
- for keyword, 56**
- for loops, 109-112**
  - complex expressions in, 113-115
  - infinite, 166
  - null statements, 112-113
- form-feed character (\f), 60**
- format specifiers**
  - %%, 79
  - %c (character), 60-62, 79
  - %d (integer), 63-64, 79
  - %e (scientific notation), 67, 79
  - %f (floating-point), 65-67, 79
  - fprintf() function
    - adding h to, 147-148
    - adding l to, 147-148
  - %G (uses %f or %E), 79
  - %i (integer), 79
  - minimum field width, 81-83
  - %n, 79
  - %o (unsigned octal), 79
  - %p, 79
  - precision specifiers, 84-85
  - printf() function, 78-79
    - adding h to, 147-148
    - adding l to, 147-148
  - %p, 179
  - %s, 217
  - %s (string), 79
  - scanf() function, 217
  - %u (unsigned integer), 79
  - %X (unsigned hexadecimal), 79
- formatted data**
  - fprintf() function
    - code example, 382-384
  - printf() function
    - comparison, 381, 388
  - syntax, 382
- fscanf() function
  - code example, 382-384
  - scanf() function
    - comparison, 381
  - syntax, 381
- formatting files, fopen(), 388**
- forms, switch statement, 161**
- fprintf() function**
  - code example, 382-384
  - format specifiers
    - adding h to, 147-148
    - adding l to, 147-148
  - printf() function comparison, 381, 388
  - syntax, 382
- fputc() function**
  - code example, 361-363
  - files, writing, 360-363
  - syntax, 361
- fputs() function**
  - code example, 364-366
  - files, writing, 363-366
  - puts() function comparison, 364
  - syntax, 364
- fread() function**
  - binary files, 381
  - code example, 368-369
  - files, reading, 366-369
  - syntax, 366
- fRecur() recursive function, 304**
- free() function, 283-286**
- freopen() function**
  - code example, 385-386
  - modes, 385-386
  - streams, redirecting, 384-388
  - syntax, 384

**fscanf() function**

code example, 382-384  
 scanf() function comparison, 381  
 syntax, 381

**fseek() function**

code example, 375-378  
 random access, 374-378  
 syntax, 374

**ftell() function**

code example, 375-378  
 random access, 374-378  
 syntax, 375

**function scope (variables), 226-227****functions, 46, 432-433.***See also statements*

arrays, passing,  
   266-267, 270-272  
 asctime(), 250-251  
 beginning, 48  
 BlockReadWrite(), 369  
 body, 48-49  
 calling, 49-51, 245-249  
 calloc(), 286-288  
 CharReadWrite(),  
   362-363  
 complexity, 49  
 Convert2Upper(), 303  
 cos(), 149-150  
 DataDisplay(), 326, 346  
 DataEnter(), 346  
 DataRead(), 377, 381,  
   384  
 DataReceive(), 321  
 DataWrite(), 381, 384  
 date and time, 249  
 declaring, 244-247, 432  
   prototypes, 245  
   specifying return  
   types, 244  
 void data type,  
   248-249

defining, 245-247, 432

ending, 48

exit(), 34

fclose()

closing files,  
   358-360, 371

code example,  
   359-360

syntax, 359

feof(), 367

code example,  
   368-369

syntax, 367

fgetc()

code example,  
   361-363

reading files,  
   360-363

syntax, 361

fgets()

code example,  
   364-366

gets() function comparison,  
   364-366,  
   371

reading files,  
   363-366

reading keyboard  
 input, 366

syntax, 363

fopen(), 381

code example,  
   359-360

formatting new files,  
   388

modes, 357-358, 388  
 opening files,  
   357-358

syntax, 357

fprintf()

code example,  
   382-384  
 printf() function  
 comparison, 381,  
   388

syntax, 382

fputc()

code example,  
   361-363

syntax, 361

writing files,  
   360-363

fputs()

code example,  
   364-366

puts() function comparison,  
   364

syntax, 364

writing files,  
   363-366

fread()

binary files, 381

code example,  
   368-369

reading files,  
   366-369

syntax, 366

free(), 283-286

freopen()

code example,  
   385-386

modes, 385-386

redirecting streams,  
   384-388

syntax, 384

fscanf()

code example,  
   382-384

scanf() function  
 comparison, 381

syntax, 381

- fseek()
  - code example, 375-378
  - random access, 374-378
  - syntax, 374
- ftell()
  - code example, 375-378
  - random access, 374-378
  - syntax, 375
- fwrite()
  - binary files, 381
  - code example, 368-369
  - syntax, 367
  - writing files, 366-369
- getc(), 72-74
- getchar(), 74-75, 248-249
- gets()
  - fgets() function comparison, 364-366, 371
  - syntax, 215
- InfoDisplay(), 329
- InfoEnter(), 329
- LineReadWrite(), 365
- localtime(), 250
- longjmp(), 440
- low-level I/O, 387
- main(), 29, 33, 305-306
- malloc(), 280-283, 292
- naming, 47, 418
- passing arguments to, 47-48
- pointers to
  - declaring, 274-275
  - passing, 268-270
- pow(), 150-152

- printf(), 78-79, 217, 386
  - format specifiers, 78-79
  - fprintf() function
    - comparison, 381, 388
- prototyping, 245-247, 432
  - fixed number of arguments, 251
  - no arguments, 248-249
  - variable number of arguments, 251-252
- PtrSeek(), 377
- PtrTell(), 377
- putc(), 75-76
- putchar(), 77-78
- puts()
  - fputs() function comparison, 364
  - syntax, 215
- realloc(), 288-291
- recursive, 303-305
  - calling, 303-304
- fRecur(), 304
- running, 305
- rewind(), 378
  - code example, 379-384
  - syntax, 378
- scanf(), 217-219
  - fscanf() function
    - comparison, 381
  - syntax, 217
- setbuf(), 387
- setjmp(), 440
- setlocale(), 440
- setvbuf(), 387
- sin(), 149-150
- sqrt(), 150-152
- strcpy(), 213-215, 336

- strlen(), 212-213
- StrPrint, 386
- structures, passing, 319-321
- tan(), 149-150
- time(), 250
- types, 46-47
- variable declarations, 48
- va\_end(), 252
- fwrite() function**
  - binary files, 381
  - code example, 368-369
  - files, writing, 366-369
  - syntax, 367

## G

- %g format specifier, 79**
- getc() function, 72-74**
- getchar() function, 74-75, 248-249**
- gets() function, 215-217, 364-366, 371**
- global variables, 227, 233**
  - declaring, 229
  - versus local, 418
- goto statement, 56, 155, 168-169, 419**
  - avoiding, 168
  - labels, location, 169
  - spaghetti code, 169
- greater than operator (>), 98**
- greater than or equal to operator (>=), 98**
- grouping variables with structures, 314**
- groups, 314**

## H

### h

adding to fprintf format specifiers, 147-148  
adding to printf format specifiers, 147-148

**hardware requirements, 16**

**header files, 32**

ANSI, 439  
stddef.h, 301  
stdio.h, 32  
    gets() function, 215  
    puts() function, 215  
stdlib.h, 302  
string.h, 302

**hex numbers, converting decimal numbers to, 79-81, 129-130**

**high-level I/O, 357, 433**

**high-level programming languages, 12-14**

**history of C, 12**

## I

**%i (integer) format specifier, 79**

**I/O, 433-434**

buffered, 433  
high-level, 433  
streams, 72  
user input, 72  
    getc() function, 72-74  
    getchar() function, 74-75

writing output, 75  
    printf() function, 78-79  
    putc() function, 75-76  
    putchar() function, 77-78

**IDE (integrated development environment)**

Borland C++, 21  
Visual C++, 18

**identifiers, 44**

**#if directive, 434**

arithmetic expressions, 406  
code example, 401-402  
macro definitions, 400  
nested conditional compilation, 402-404  
syntax, 399

**if statement, 56, 155-158**

braces, 156  
codes, executing, 156  
conditions, evaluating, 156  
listing, 157  
nesting, 160-161

**if-else statement, 155, 158-159**

**#ifdef directive, 397, 434**

code example, 398-399  
syntax, 397

**#ifndef directive, 397-399, 416**

code example, 398-399  
syntax, 398

**illegal characters (identifiers), 44**

**implementing algorithms, 305**

**improving readability, C programs, 296, 300**

**#include directive, 31**

**increasing program**

**portability, 234**

**increment operator (++), 96-98**

**indentation (in code), 28, 419**

**indexes (arrays), 190**

**indirection operator. *See* deference operator**

**infinite loops, 166-167**

**InfoDisplay() function, 329**

**InfoEnter() function, 329**

**information hiding (modular programming), 419**

**initializers, 317**

**initializing**

array elements, 191  
character arrays, 208-211  
elements, 192  
multidimensional arrays, 199-201  
multidimensional unsized arrays, 202-203  
strings, 208-211  
structures, 317-319  
unions, 337-339, 351

    memory sharing  
        code example, 338-339

    structures, 338

    unsized character arrays, 209

**input data, reading, 217-219**

**input/output. *See* I/O**

**input, user, 72**

    getc() function, 72-74  
    getchar() function, 74-75

**int data type, 46, 62**

**int keyword, 56**

**integer constants, 422**

**integer format specifiers**

    %d, 63-64, 79

    %i, 79

**integer values**

    default, 296

    enum data types,  
        assigning, 296, 300

**integers**

    adding, 304

    arrays, 190

    declaring, 62-63

    negatives, 95

    pointers, 259

    structures, assigning,  
        315

**integrated development  
environment. See IDE**

**interpreted programming  
languages, 15**

**interpreters. See compilers**

**iteration. See loops**

## J-K

**jumping statements. See  
conditional branching  
statements**

**justifying. See aligning**

**”K & R”, 15**

**keyboards, input, 366**

**keywords, 56**

    auto, 56

    break, 56

    case, 56, 162

    char, 56

    const, 56

    continue, 56

    default, 56

    do, 56

    double, 56

    else, 56

    enum, 56

    extern, 56

    float, 56

    for, 56

    goto, 56

    if, 56

    int, 56

    list of, 420-421

    long, 56

    register, 56, 233

    reserved, 45

    return, 56

    short, 56

    signed, 56

    sizeof, 56

    static, 56

    struct, 56, , 348-350

    switch, 56

    typedef, 57, 300-301,  
        321, 426

    union, 57

    unsigned, 57

    void, 57, 305

    volatile, 57

    while, 57

## L

### I

    adding to fprintf format  
        specifiers, 147-148

    adding to printf format  
        specifiers, 147-148

**labels**

    conditional branching  
        statements, 162

    location, goto state-  
        ment, 169

**languages. See program-  
ming languages**

**left values, 176**

    defined, 187

    obtaining, 178

**left-hand operands, 92**

**left-justifying output,  
83-84**

**left-shift operator (<<),  
133-135**

**legal characters (identi-  
fiers), 44**

**lengths of strings, mea-  
suring, 212-213**

**less than operator (<), 98**

**less than or equal to  
operator (<=), 98**

**levels (programming lan-  
guages), 12**

**libraries, 14**

**limits.h header file, 439**

**line numbers, 28**

**LineReadWrite() func-  
tion, 365**

**lines (code)**

    breaking, 28

    indenting, 28

**linked lists, 410**

    advantages, 410

    creating, 410

        adding nodes, 414

        calling functions in  
            module file,  
            416-418

        header file, 415-416

        module program

        example, 410-415

**linkers, 34-36****listings**

- accessing arrays via pointers, 264-265
- adding integers with a function, 49
- aligning output, 83
- arithmetic assignment operators, 94
- arrays of structures, 325-326
- binary files, reading/writing, 379-380
- bit fields, 348-349
- bitwise operators, 132
- break statement, 164-165
- calculating an addition and printing results to the screen, 50
- calculating array size, 193
- calling functions after they are declared and defined, 245
- calling recursive functions, 303-304
- calloc() function, 287
- cast operator, 101
- changing variable values via pointers, 183
- command-line arguments, 306-307
- conditional operator, 135
- continue statement, 167
- converting numeric values back to characters, 61
- copying strings, 213

- declaring and assigning pointer values, 180
- #define directive, 394-395
- defining enum data types, 297
- do-while loop example, 108
- #elif directive, 401
- #else directive, 401
- #endif directive, 398
- ending output at null character, 198
- enum data types, 298-299
- fclose function, 359-360
- feof() function, 368-369
- fgetc() function, 361-362
- fgets() function, 364-365
- floating-point format specifier, 65-66
- fopen() function, 359-360
- for loops, 110, 114-115
- fprintf() function, 382-383
- fputc() function, 361-362
- fputs() function, 364-365
- fread() function, 368-369
- freopen() function, redirecting streams, 385-386
- fscanf() function, 382-383
- fwrite() function, 368-369

- gets() function, 216
- %hd, %lu, and %ld format specifiers, 147-148
- hex numbers, converting to, 80
- #if directive, 401
- if statement in decision making, 157
- if-else statement, 159
- #ifdef directive, 398
- #ifndef directive, 398
- increment and decrement operators, 97
- infinite loops, breaking, 166
- initializing arrays, 191
- initializing strings, 210-211
- initializing structures, 318
- initializing unsized arrays, 202
- integer format specifier, 63
- linked lists
  - calling functions in module file, 416-418
  - header file, 415
  - module program, 410-413
- logical AND operator, 125
- logical negation operator (!), 128
- logical OR operator, 126
- malloc() and free() functions, 283-285
- malloc() function, 281-282

- measuring string lengths, 212
  - minimum field width specifiers, 82
  - moving pointers, different data types, 260-261
  - multiple pointers, 185
  - nested conditional compilation, 403-404
  - nested if statements, 160
  - nested loops, 116
  - nested structures, 327-329
  - obtaining left values, 178
  - passing arrays to functions, 266-267
  - passing functions with pointers, 322-323
  - passing multidimensional arrays to functions, 270-271
  - passing pointers to functions, 268-269
  - passing structures to functions, 320
  - pointing to functions, 274-275
  - pow90 and sqrt() functions, 151
  - precision specifiers, 84-85
  - printing array of characters, 196
  - printing characters, 60
  - printing two-dimensional arrays, 200
  - printing variables, different scope levels, 225
  - processing variable arguments, 253-254
  - put() function, 216
  - random access to files, 375-376
  - realloc() function, 289-290
  - referencing arrays with pointers, 195
  - referencing structure members, 316
  - relational operators, 99
  - relationship between program and block scope, 227
  - scanf() function with various format specifiers, 218
  - shift operators, 134
  - short and long data modifiers, 146
  - signed and unsigned data modifiers, 144
  - simple C program, 28
  - sizeof operator, 122-123
  - static specifier, 230
  - subtracting pointers, 263
  - switch statement, 162
  - trigonometry functions, 149
  - typedef definitions, 301-302
  - unions
    - measuring size, 339-340
    - memory sharing, 338-339
    - nesting in structures, 344-345
    - referencing, 335-336
    - referencing memory locations, 341-342
  - user input
    - getc() function example, 73
    - getchar() function, 74-75
  - using array, pointer to character strings, 272-273
  - void in function declarations, 248-249
  - while loop example, 106
  - writing output, putc() function, 76
- lists**
- linked, 410
    - advantages, 410
    - creating, 410-418
    - variables, enum data types, 296
- little-endian format, 343**
- local scope. *See* block scope**
- local time, 249**
- local variables, 225, 418**
- locale.h header file, 440**
- localtime() function, 250**
- locations (memory), multiple pointers, 185-186**
- logical operators, 124**
- AND (&&), 124-126
  - NEGATION (!), 128-129
  - OR (||), 126-127
- long data modifier, 145-147**
- long keyword, 56**
- longjmp() functions, 440**



**loops, 105, 427-428**

- continue statement, 171
- control flow, 155
- do-while, 107-109
- for, 109-112
  - complex expressions in, 113-115
  - null statements, 112-113
- infinite, 166-167
- nesting, 116-117
- skipping, continue statement, 167
- while, 106-107

**low-level I/O, 357, 387**

## M

**macro body, 392, 396****macro names, 392-393****macro substitution, 392**

- code example, 394-396
- #define directive, 393

**macros**

- #define directive, 393
  - code example, 394-396
  - defining function-like macros, 394-396
  - expressions, 396
  - nested macro definitions, 396
  - syntax, 393
- #if directive, 400
- #undef directive, 393-394, 406
- va\_arg(), 252
- va\_start(), 252

**main() functions, 29, 33, 305**

- arguments, 305-306
- declaring, 306

**maintaining C programs, 296****malloc() function, 280-283**

- calloc() function, compared, 292
- listings, 281-285

**manipulating variables indirectly, 176****mantissas, 67****math.h header file, 440****mathematical functions, 148**

- cos(), 149-150
- pow(), 150-152
- sin(), 149-150
- sqrt(), 150-152
- tan(), 149-150

**measuring**

- size
  - data, 122-123
  - structures, 339-341
  - unions, 339-341
- string lengths, 212-213

**members, structures, 314**

- assessing, 315
- declaring, 314
- referencing, 315-319

**memory**

- addresses, 343
- allocated, releasing, 283-286
- allocating
  - calloc() function, 286-288
  - malloc() function, 280-283

**buffers, 356**

- flushing, 356
- high-level I/O, 357
- low-level I/O, 357, 387
- setbuf() function, 387
- setvbuf() function, 387

**locations, multiple**

- pointers, 185-186

**reallocating, realloc() function, 288-291****stack, 305****unions, 334, 425-426**

- arrow operator (->), 335, 351
- declaring, 334
- defining variables, 334-335
- dot operator (.), 335-337, 351
- initializing, 337-339, 351
- memory sharing
  - code example, 338-339
- nesting, 343-346
- referencing, 335-337, 351
- referencing memory locations, 341-343, 352
- size, 339-341
- structures comparison, 351
- tag names, 334
- variables, 176-177

**memory locations, temporary, 229**

**Microsoft compiler, 18-21**

- running, 19

- starting, 18

**minimum field width**

- specifiers, 81-83

**modes**

- fopen() function,

- 357-358, 388

- freopen() function,

- 385-386

**modifiers (variables),**

- 234-235. *See also* data

- modifiers

**modular programming,**

- 419-420

**modules, 419****modulus operator (%),**

- 43

**moving**

- file position indicators

- fseek() function, 374

- rewind() function,

- 378

- pointers, 260-262

**multidimensional arrays**

- declaring, 199

- displaying, 200-201

- initializing, 199-201

**multidimensional unsized**

- arrays, 201-203

**multiple pointers,**

- 185-186

**multiplication assign-**

- ment operator (\*=), 93

**multiplication operator**

- (\*), 182

**N****%n format specifier, 79****\n (newline character), 33****named constants, versus**

- numeric, 419

**named integer constants,**

- declaring, 296

**names**

- data types, 300-302

- macro names, 392-393

- tag, unions, 334

**naming**

- built-in arguments, 308

- functions, 47, 418

- identifiers, 44

- variables, 68, 418

**negative numbers, 95,**

- 142

**nested block scope (vari-**

- ables), 225-226

**nested comments, 31****nested conditional compi-**

- lation, 402-404

**nested macro definitions**

- (#define directives), 396

**nested structures**

- declaring, 327

- listings, 327-329

**nesting**

- if statements, 160-161

- loops, 116-117

- unions

- code example,

- 344-345

- in structures,

- 343-346

**newline character (\n), 33****nibbles, 14****nodes (linked lists),**

- adding, 414

**not equal to operator**

- (!=), 98

**null characters, 198-199****null pointers, 183****null statements, 112-113****numbers**

- binary, negatives, 142

- decimal

- converting to binary,

- 129-130

- converting to hex,

- 79-81, 129-130

- floating-point, calcula-

- tions, 152

- negatives, 95

- scientific notation, 67

**numeric constants, ver-**

- sus named, 419

**numeric values (of char-**

- acters)

- converting, 61

- showing, 63-64

**O****%o (unsigned octal) for-**

- mat specifier, 79

**object files, 34****objects**

- bit fields

- code example,

- 348-350

- declaring, 347

- defining, 347

- pointing to, 431-432

**obtaining left values, list-**

- ing, 178

**opening files (fopen()**

- function), 357-360, 381

**opening comment marks,  
29-30****operands, 92****operators**

address-of (&),  
177-179, 323  
arithmetic, 43-44  
arithmetic assignment,  
92-95  
    addition assignment  
    (+ =), 93  
    division assignment  
    (/ =), 93  
    multiplication  
    assignment (\* =),  
    93  
    remainder assign-  
    ment (% =), 93  
    subtraction assign-  
    ment (- =), 93  
array subscript ([ ]),  
190  
arrow (->), 324, 335,  
351  
assignment (=), 92  
binary, multiplication  
(\*), 182  
bit-manipulation, 130  
    bitwise AND (&),  
    131  
    bitwise complement  
    (tilde), 131  
    bitwise OR (|), 131  
    bitwise XOR (^),  
    131  
    left-shift (<<),  
    133-135  
    right-shift (>),  
    133-135  
cast, 101-102  
cautions, 419

conditional (:?),  
135-136  
decrement (—), 96-98  
deference (\*), 182  
dot (.), 315-317  
    unions, 335-337, 351  
increment (++), 96-98  
logical, 124  
    AND (&&), 124-126  
    NEGATION (!),  
    128-129  
    OR (||), 126-127  
precedence, 98-99  
relational, 98-100  
    equal to (==), 98  
    greater than (>), 98  
    greater than or equal  
    to (>=), 98  
    less than (<), 98  
    less than or equal to  
    (<=), 98  
    not equal to (!=), 98  
sizeof, 122-123  
table of, 421-422  
unary, deference opera-  
tors (\*), 182

**optimizers, turning off,  
235****output**

aligning, 83-84  
format specifiers, 79  
    %c, 60-62  
    %d, 63-64  
    %E, 67  
    %f, 65-67  
minimum field width  
specifiers, 81-83  
precision specifiers,  
84-85

writing, 75  
    printf() function,  
    78-79  
    putc() function,  
    75-76  
    putchar() function,  
    77-78

**outputs, null characters,  
198-199****overwriting data in stack,  
305****P****%p format specifier, 79,  
179****parameters, command-  
line arguments, 305****parentheses**

if statement, 156  
placing around expres-  
sions, 419

**passing**

arguments  
    to functions, 47-48  
    to main () functions,  
    305  
arrays, to functions,  
266-267  
functions, with pointers,  
322-323  
multidimensional  
    arrays, to functions,  
    270-272  
pointers, 268-270, 322  
structures, to functions,  
319-321

**performance, comments,  
30**

**Perl, 14****pointers, 176, 430**

- arrays
  - accessing, 264-266
  - declaring, 272
  - referencing with, 195-196
  - strings, 272-274
- character constants, assigning, 210
- character strings, assigning, 210-211
- declaring, 179-181
- dereferenced, 323
- file pointers, 357, 371
- to functions, declaring, 274-275
- integers, 259
- moving, 260-262
- multiple, 185-186
- null, 183
- passing, 268-270, 322
- pointing to objects, 431-432
- size, 260
- structures, referencing, 322-324
- subtracting, 263-264
- types, 180
- uninitialized, errors, 262
- values, assigning, 180-181
- variable values, changing, 183-184

**portability, 13, 36, 234****post-decrement operator (--), 96****post-increment operator (++), 96****pow() function, 150-152****pre-decrement operator (--), 96****pre-increment operator (++), 96****precedence (of operators), 43, 98-99****precision specifiers, 84-85****preprocessor. *See* C preprocessor****preprocessor directives. *See* directives****printf() function, 78-79, 386**

- format specifiers, 78-79
  - adding h to, 147-148
  - adding L to, 147-148
  - %c, 60-62
  - %d, 63-64
  - %e, 67
  - %f, 65-67
  - %p, 179
  - %s, 217

- fprintf() function comparison, 381, 388

**printing characters, 60-62****processing variable arguments, 252-254****program portability, increasing, 234****program scope (variables), 227, 233**

- block scope comparison, 227-229
- global variables, 227

**programming**

- modular, 419-420
- structured
  - bottom-up, 255
  - C, 169
  - top-down, 255

**programming languages**

- C++, 14
  - compiled, 14
  - high-level, 12-14
  - interpreted, 15
  - levels, 12
- Perl, 14

**programming style, 418-419****programs**

- C preprocessor, 392
  - # (pound sign), 392
  - C compiler comparison, 392-393, 405
  - directives. *See* directives
  - macro body, 392, 396
  - macro names, 392-393
  - macro substitution, 392-396
  - newline characters, 392
- executing, 305
- file names, .c extension, 28
- performance, comments, 30
- running
  - Borland C++, 23
  - Visual C++, 20

**prototyping functions, 245-247, 432**

- fixed number of arguments, 251
- no arguments, 248-249
- variable number of arguments, 251-252

**PrtSeek() function, 377****PtrTell() function, 377****putc() function, 75-76**

**putchar() function, 77-78**

**puts() function, 215-217**

fputs() function comparison, 364  
requirements, `stdio.h`  
header file, 215  
syntax, 215

## Q-R

**qualifiers. *See* modifiers (variables)**

**\r (return character), 60**

**random access**

code example, 375-378  
disk files, 374-377, 387  
fseek() function, 374-378  
ftell() function, 374-378

**readability of code, improving, 296, 300, 419**

**reading**

characters, 215-217  
files, 360  
binary, 378-381  
fgetc() function, 360-363  
fgets() function, 363-366, 371  
fread() function, 366-369, 381  
random access, 374-377, 387  
sequential access, 374  
formatted data  
fprintf() function, 381-384, 388  
fscanf() function, 381-384

input

getc() function, 72-74  
getchar() function, 74-75  
scanf() function, 217-219  
keyboard input, fgets() function, 366  
strings, 217

**real numbers. *See* floating-point numbers**

**realloc() function, 288-291**

**reallocating memory, 288-291**

**recommended reading, 434-435**

**recursive functions, 303-305**

calling, 303-304  
fRecur(), 304  
running, 305

**redirecting streams, freopen() function, 384-388**

**referencing**

arrays, with pointers, 195-196  
memory locations, unions, 341-343, 352  
structure members, 315-319  
structures  
with arrow operators, 324  
with pointers, 322-324  
unions, 335, 351  
arrow operator (->), 335, 351  
code example, 335-337

dot operator (.), 335-337, 351

**regions (variables), 229**

**register keyword, 56, 233**

**registers, 233**

**relational operators, 98-100**

equal to (==), 98  
greater than (>), 98  
greater than or equal to (>=), 98  
less than (<), 98  
less than or equal to (<=), 98  
not equal to (!=), 98

**releasing allocated memory, 283-286**

**remainder assignment operator (%=), 93**

**remainder operator (%), 43**

**replacing built-in arguments, 308**

**requirements**

`stdio.h` header file, 215  
system requirements, 16-17

**reserved words. *See* keywords**

**resetting file position indicators, 378**

**resources, 305, 434-435**

**return character (\r), 60**

**return keyword, 34, 56**

**return types, specifying, 244**

**return values, 47**

**returning times, 250**

**reusability, 14**

**rewind() function, 378**

code example, 379-384  
syntax, 378

**right value (variables),**  
 177, 187  
**right-hand operands, 92**  
**right-justifying output,**  
 83-84  
**right-shift operator (>>),**  
 133-135  
**Ritchie, Dennis, 12**  
**rounding (division), 100**  
**running**  
   compilers  
     Borland C++, 23  
     Visual C++, 19  
   programs  
     Borland C++, 23  
     Visual C++, 20  
   recursive functions, 305

## S

**%s (string format specifier), 79**  
   printf() function, 217  
   scanf() function, 217  
**saving code**  
   Borland C++, 23  
   Visual C++, 19  
**scanf() function, 217-219**  
   format specifiers, 217  
   fscanf() function comparison, 381  
   syntax, 217  
**scientific notation, 67**  
**scientific notation format specifier (%E), 67, 79**  
**scopes (variables)**  
   block, 224-225  
   file, 232  
   function, 226-227  
   nested block, 225-226  
   program, 227, 233  
   semicolons (;), 28  
   separating data items, 317  
   sequential access, disk files, 374  
   setbuf() function, 387  
   setjmp() function, 440  
   setjmp.h header file, 440  
   setlocale() function, 440  
   setting null pointers, 183  
   setvbuf() function, 387  
   shift operators, 133-135  
   short data modifier, 145  
   short keyword, 56  
   sign bits, 142  
   signal.h header file, 440  
   signed data modifier, 142-143  
   signed keyword, 56  
   sin() function, 149-150  
   single quotes ('), 59  
   size  
     pointers, 260  
     structures, 339-341  
     unions, 339  
       measuring, 339-341  
       structure size comparison, 339-341  
   sizeof keyword, 56  
   sizeof operator, 122-123  
   sizes, arrays  
     calculating, 192-194  
     specifying, 267  
     unsized arrays, 201  
   skipping loops, 167  
   software requirements, 16-17  
   software engineering, 420  
   source code. *See* code  
   spacing (in code), 419  
   spaghetti code, 169

**spatial regions (variables), 229**  
**specifiers, storage classes**  
   auto, 229-231  
   extern, 234  
   register, 233  
   static, 230  
**specifying**  
   sizes, arrays, 267  
   return types, 244  
**sqrt() function, 150-152**  
**stack, 305**  
**standard input stream**  
   reading characters from, 215-217  
   reading strings from, 217  
**standard input-output header file, 32**  
**standard output stream**  
   writing characters from, 217  
   writing characters to, 215-216  
**standards, 15-16**  
**starting compilers**  
   Borland C++, 21  
   Visual C++, 18  
**statement blocks, 45-46**  
**statements, 45, 426. *See also* functions; loops**  
   break, 155, 164-165  
   conditional branching, 155  
   continue, 155, 167-168  
   control flow, 426  
   goto, 155, 168-169  
   if, 155-158  
     braces, 156  
     evaluating conditions, 156

- executing codes, 156
  - nesting, 160-161
- if-else, 155, 158-159
- looping. *See* loops
- null, 112-113
- return, 34
- switch, 155, 161-164
- static keyword, 56**
- static specifier, 230-231**
- static specifiers, applying, 231**
- stdarg.h header file, 440**
- stddef.h header file, 301, 440**
- stderr file stream, 433**
- stderr stream, 72**
- stdin file stream, 433**
- stdin stream, 72**
- stdio.h file, 32**
- stdio.h header file, 440**
- stdlib.h header file, 302, 440**
- stdout file stream, 433**
- stdout stream, 72**
- storage, memory, 343**
- storage classes, 229**
  - auto specifier, 229-231
  - extern specifier, 234
  - modifiers, 234-235
  - register specifier, 233
  - static specifier, 230
- storing variables, registers, 233**
- strcpy() function, 213-215, 336**
- streams, 72, 433**
  - binary, 356, 370
  - defined, 356
  - device independence, 356
  - files comparison, 356
- freopen() function, 384-388
  - code example, 385-386
  - modes, 385-386
  - syntax, 384
- high-level I/O, 357
- low-level I/O, 357
- setbuf() function, 387
- setvbuf() function, 387
- stderr, 72
- stdin, 72
- stdout, 72
- text, 356, 370
- string constants, 208, 423**
  - character arrays, initializing, 208-211
  - character constants, compared, 209-212
- string format specifier (%s), 79**
- string.h header file, 302, 440**
- strings, 208**
  - copying, 213-215
  - initializing, 208-211
  - lengths, measuring, 212-213
  - pointers, arrays, 272-274
  - reading, 217
- strlen() function, 212-213**
- StrPrint() function, 386**
- struct data type, 424-425**
  - bit fields
    - code example, 348-350
    - declaring, 347
- struct keyword, 56, 347**
- structure members, referencing, 315-319**
- structured programming**
  - bottom-up programming, 255
  - C, 169
  - top-down programming, 255
- structures**
  - arrays
    - arrays of structures, 324-327
    - compared, 314
  - arrow operators, referencing, 324
  - data types, 314
  - declaring, 314-315
  - definitions, 315
  - duplication, avoiding, 322
  - FILE, 357, 374-375, 378
  - functions, passing, 319-321
  - initializing, 317-319
  - integers, assigning, 315
  - members, 314-315
  - nested, declaring, 327
  - nesting unions, 343-346
  - pointers, referencing, 322-324
  - size, 339-341
  - tag names, 314
  - union comparison, 351
  - union initialization, 338
  - variables, 314-315
- style (programming), 418-419**
- subtracting integers, pointers, 259, 263-264**
- subtraction assignment operator (-=), 93**
- subtraction operator (-), 96**

**switch statement, 56, 155, 161-164**  
braces, 165  
case keyword, 162  
conditional expressions, 161-162  
exiting break statements, 164  
form, 161  
listing, 162  
statement execution, 164  
unlimited decisions, 161

**switches, 13**

**synonyms**  
data type names, 300  
declaring, 300  
defining, 321

**system requirements, 16-17**

## T

**\t (tab character), 60**

**tag names**  
structures, 314  
unions, 334

**tan() function, 149-150**

**temporal regions (variables), 229**

**temporary memory locations, 229**

**text editors, requirements, 17**

**text streams, 356, 370**

**time, converting, 250-251**

**time() function, 250**

**time.h header file, 440**

**top down programming, 255**

**transistors, 13**

**trigonometry functions, 149-150**

**troubleshooting. *See* debugging**

**truncation (division), 100**

**turning off optimizers, 235**

**typedef keyword, 57, 300, 321, 426**  
advantages, 300-301  
listing, 301-302  
updating, 301

## U

**%u (unsigned integer) format specifier, 79**

**unary deference operator (\*), 182**

**unary minus operator (-), 95**

**#undef directive, 393-394, 406, 434**

**uninitialized pointer errors, 262**

**union data type, 425-426**

**union keyword, 57**

**unions, 334**  
arrow operator (->), 335, 351  
declaring, 334  
dot operator (.), 335-337, 351  
initializing, 337-339, 351  
memory sharing  
code example, 338-339  
structures, 338

**nesting**  
code example, 344-345  
in structures, 343-346  
referencing, 335-337, 351  
referencing memory locations, 341-343, 352  
size, 339-341  
structures comparison, 351  
tag names, 334  
variables, defining, 334-335

**UNIX systems, requirements, 17**

**unsigned data modifier, 143-145**

**unsigned hexadecimal format specifier (%x), 79**

**unsigned integer format specifier (%u), 79**

**unsigned keyword, 57**

**unsigned octal format specifier (%o), 79**

**unsized arrays**  
character arrays, 209  
declaring, 201  
initializing, 202-203  
sizes, calculating, 201

**updating definitions, 301**

**uppercase, converting to, 302-303**

**user input, 72**  
getc() function, 72-74  
getchar() function, 74-75



## V

### values

- constants, 42
- enum data types, 298-299
- integers, default, 296
- left, 187
- pointer, assigning, 180-181
- right, 187
- variables, 42, 183-184

### variable arguments, processing, 252-254

### variables, 42. *See also*

#### data types

- assigning return values to, 47
- auto specifier, 229
- character, 58
- const modifiers, 234-235
- declaring, 48, 177, 244
  - floating-point numbers, 64-65
  - integers, 62-63
- defining, unions, 334-335
- extern specifier, 234
- global, 227, 233
  - declaring, 229
  - versus local, 418
- groups, collecting, 314
- indirectly manipulating, 176
- left value, 176
- lists, defining data types, 424
- local, 225, 418
- naming, 68, 418

- pointers, 176, 430-432
- register specifier, 233
- registers, storing, 233
- right value, 177
- scope
  - block, 225
  - block scope, 224
  - file, 232
  - function, 226-227
  - nested block scope, 225-226
  - program, 227
  - program and block scope comparison, 227-229
- spatial regions, 229
- static specifier, 230-231
- storage classes, 229
- structures, 314-315
- temporal regions, 229
- values, 42, 183-184
- volatile modifiers, 235

### va\_arg() macro, 252

### va\_end() function, 252

### va\_start() macro, 252

### viewing file position indicator value, 375

### Visual C++, 17

- compiler, 18-21
  - running, 19
  - starting, 18
- IDE, 18

### void data type, function declarations, 248-249

### void keyword, 57, 305

### volatile keyword, 57

### volatile modifier (variables), 235

## W-Z

### while keyword, 57

### while loops, 106-107, 166

### whitespace, 29

### writing

- characters
  - from standard output stream, 217
- puts() function, 215-217
- to standard output stream, 215-216

### code

- Borland C++, 21
- Visual C++, 18

### files, 360

- binary, 378-381
- fputc() function, 360-363
- fputs() function, 363-366
- fwrite() function, 366-369, 381
- random access, 374-377, 387
- sequential access, 374

### output, 75

- printf() function, 78-79
- putc() function, 75-76
- putchar() function, 77-78

### %x (unsigned hexadecimal) format specifier, 79