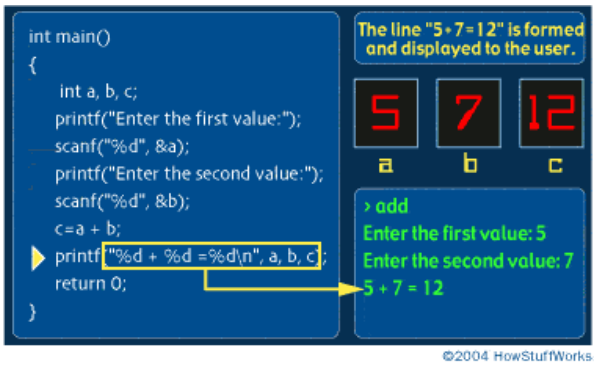
# C Programming Tutorial

**Introduction to How C Programming Works**

The C programming language is a popular and widely used programming language for creating **computer programs**. Programmers around the world embrace C because it gives maximum control and efficiency to the programmer.

If you are a programmer, or if you are interested in becoming a programmer, there are a couple of benefits you gain from learning C:

- You will be able to read and write code for a large number of platforms -- everything from microcontrollers to the most advanced scientific systems can be written in C, and many modern operating systems are written in C.
- The jump to the object oriented C++ language becomes much easier. C++ is an extension of C, and it is nearly impossible to learn C++ without learning C first.

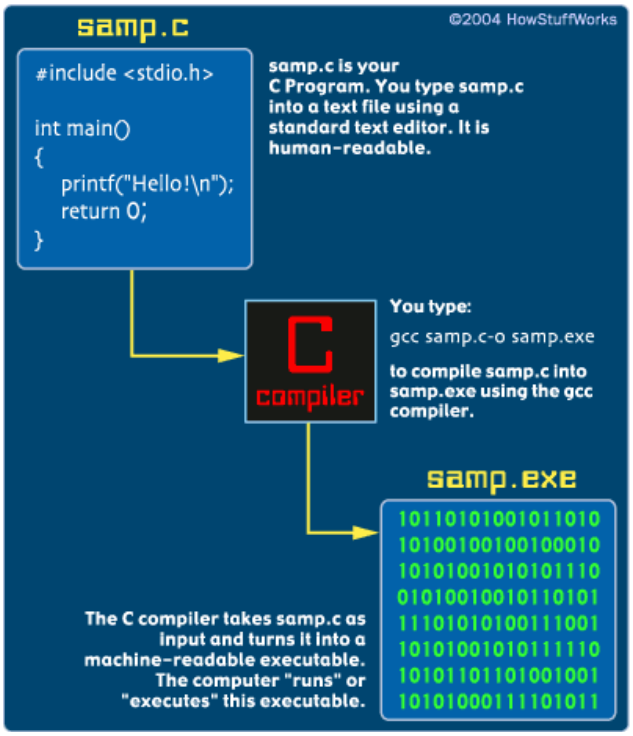



**This animation shows the execution of a simple C program. By the end of this article you will understand how it works!**

In this article, we will walk through the entire language and show you how to become a C programmer, starting at the beginning. You will be amazed at all of the different things you can create once you know C!

**What is C?**

C is a **computer programming language**. That means that you can use C to create lists of instructions for a computer to follow. C is one of thousands of programming languages currently in use. C has been around for several decades and has won widespread acceptance because it gives programmers maximum control and efficiency. C is an easy language to learn. It is a bit more cryptic in its style than some other languages, but you get beyond that fairly quickly.




C is what is called a **compiled language**. This means that once you write your C program, you must run it through a **C compiler** to turn your program into an **executable** that the computer can run (execute). The C program is the human-readable form, while the executable that comes out of the compiler is the machine-readable and executable form. What this means is that to write and run a C program, you must have access to a C compiler. If you are using a UNIX machine (for example, if you are writing CGI scripts in C on your host's UNIX computer, or if you are a student working on a lab's UNIX machine), the C compiler is available for free. It is called either "cc" or "gcc" and is available on the command line. If you are a student, then the school will likely provide you with a compiler -- find out what the school is using and learn about it. If you are working at home on a Windows machine, you are going to need to download a free C compiler or purchase a commercial compiler. A widely used commercial compiler is Microsoft's Visual C++ environment (it compiles both C and C++

programs). Unfortunately, this program costs several hundred dollars. If you do not have hundreds of dollars to spend on a commercial compiler, then you can use one of the free compilers available on the Web. See http://delorie.com/djgpp/ as a starting point in your search.

We will start at the beginning with an extremely simple C program and build up from there. I will assume that you are using the UNIX command line and gcc as your environment for these examples; if you are not, all of the code will still work fine -- you will simply need to understand and use whatever compiler you have available.

Let's get started!

**The Simplest C Program**

Let's start with the simplest possible C program and use it both to understand the basics of C and the C compilation process. Type the following program into a standard text editor (vi or emacs on UNIX, Notepad on Windows or TeachText on a Macintosh). Then save the program to a file named **samp.c**. If you leave off **.c**, you will probably get some sort of error when you compile it, so make sure you remember the **.c**. Also, make sure that your editor does not automatically append some extra characters (such as .txt) to the name of the file. Here's the first program:

```
#include <stdio.h>

int main()
{
    printf("This is output from my first program!\n");
    return 0;
}
```

When executed, this program instructs the computer to print out the line "This is output from my first program!" -- then the program quits. You can't get much simpler than that!

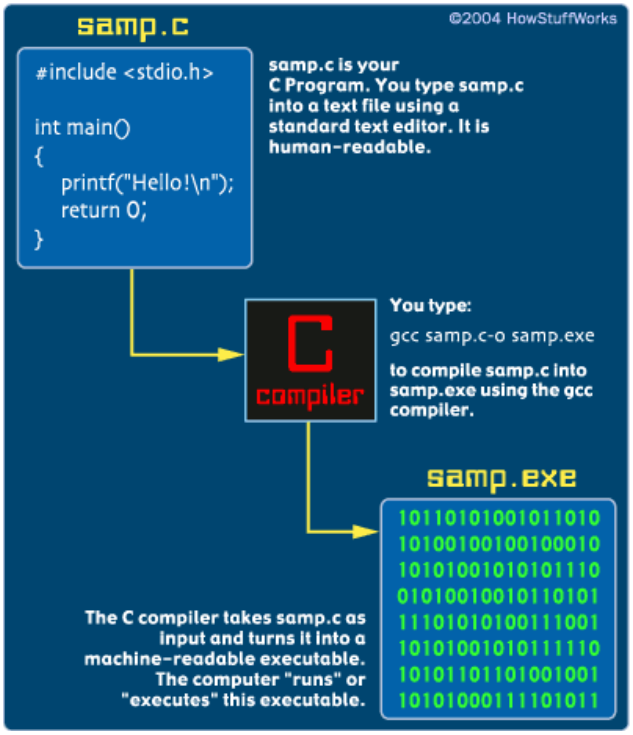
**Position**

When you enter this program, position **#include** so that the pound sign is in column 1 (the far left side). Otherwise, the spacing and indentation can be any way you like it. On some UNIX systems, you will find a program called **cb**, the C Beautifier, which will format code for you. The spacing and indentation shown above is a good example to follow.

To compile this code, take the following steps:

- On a UNIX machine, type **gcc samp.c -o samp** (if gcc does not work, try cc). This line invokes the C compiler called gcc, asks it to compile samp.c and asks it to place the executable file it creates under the name **samp**. To run the program, type **samp** (or, on some UNIX machines, **./samp**).
- On a DOS or Windows machine using DJGPP, at an MS-DOS prompt type **gcc samp.c -o samp.exe**. This line invokes the C compiler called gcc, asks it to compile samp.c and asks it to place the executable file it creates under the name **samp.exe**. To run the program, type **samp**.
- If you are working with some other compiler or development system, read and follow the directions for the compiler you are using to compile and execute the program.

You should see the output "This is output from my first program!" when you run the program. Here is what happened when you compiled the program:




If you mistype the program, it either will not compile or it will not run. If the program does not compile or does not run correctly, edit it again and see where you went wrong in your typing. Fix the error and try again.

**The Simplest C Program: What's Happening?**

Let's walk through this program and start to see what the different lines are doing:

- This C program starts with **#include <stdio.h>**. This line **includes** the "standard I/O library" into your program. The standard I/O library lets you read input from the keyboard (called "standard in"), write output to the screen (called "standard out"), process text files stored on the disk, and so on. It is an extremely useful library. C has a large number of standard libraries like stdio, including string, time and math libraries. A **library** is simply a package of code that someone else has written to make your life easier (we'll discuss libraries a bit later).
- The line **int main()** declares the main function. Every C program must have a function named **main** somewhere in the code. We will learn more about functions shortly. At run time, program execution starts at the first line of the main function.
- In C, the **{** and **}** symbols mark the beginning and end of a block of code. In this case, the block of code making up the main function contains two lines.
- The **printf** statement in C allows you to send output to standard out (for us, the screen). The portion in quotes is called the **format string** and describes how the data is to be formatted when printed. The format string can contain string literals such as "This is output from my first program!," symbols for carriage returns (\n), and operators as placeholders for variables (see below). If you are using UNIX, you can type **man 3 printf** to get complete documentation for the printf function. If not, see the documentation included with your compiler for details about the printf function.
- The **return 0;** line causes the function to return an error code of 0 (no error) to the shell that started execution. More on this capability a bit later.

**Variables**

As a programmer, you will frequently want your program to "remember" a value. For example, if your program requests a value from the user, or if it calculates a value, you will want to remember it somewhere so you can use it later. The way your program remembers things is by using **variables**. For example:

```
int b;
```

This line says, "I want to create a space called b that is able to hold one integer value." A variable has a **name** (in this case, b) and a **type** (in this case, int, an integer). You can store a value in b by saying something like:

```
b = 5;
```

You can use the value in b by saying something like:

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

In C, there are several standard types for variables:

- **int** - integer (whole number) values
- **float** - floating point values
- **char** - single character values (such as "m" or "Z")

We will see examples of these other types as we go along.

**Printf**

The **printf statement allows you to send output to standard out.** For us, standard out is generally the screen (although you can redirect standard out into a text file or another command).

Here is another program that will help you learn more about printf:

```
#include <stdio.h>

int main()
{
    int a, b, c;
    a = 5;
    b = 7;
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Type this program into a file and save it as **add.c**. Compile it with the line **gcc add.c -o add** and then run it by typing **add** (or **./add**). You will see the line "5 + 7 = 12" as output.

Here is an explanation of the different lines in this program:

- The line **int a, b, c;** declares three integer variables named **a**, **b** and **c**. Integer variables hold whole numbers.
- The next line initializes the variable named **a** to the value 5.
- The next line sets **b** to 7.
- The next line adds **a** and **b** and "assigns" the result to **c**.

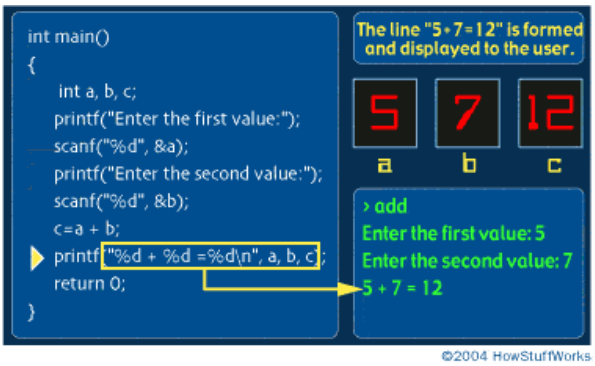
  The computer adds the value in **a** (5) to the value in **b** (7) to form the result 12, and then places that new value (12) into the variable **c**. The variable **c** is assigned the value 12. For this reason, the **=** in this line is called "the assignment operator."
- The **printf** statement then prints the line "5 + 7 = 12." The **%d** placeholders in the printf statement act as placeholders for values. There are three %d placeholders, and at the end of the printf line there are the three variable names: **a**, **b** and **c**. C matches up the first %d with a and substitutes 5 there. It matches the second %d with b and substitutes 7. It matches the third %d with c and substitutes 12. Then it prints the completed line to the screen: 5 + 7 = 12. The **+**, the **=** and the spacing are a part of the format line and get embedded automatically between the %d operators as specified by the programmer.

**Printf: Reading User Values**

The previous program is good, but it would be better if it read in the values 5 and 7 from the user instead of using constants. Try this program instead:

```
#include <stdio.h>

int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Here's how this program works when you execute it:




Make the changes, then compile and run the program to make sure it works. Note that scanf uses the same sort of format string as printf (type **man scanf** for more info). Also note the & in front of a and b. This is the **address operator** in C: It returns the address of the variable (this will not make sense until we discuss pointers). You must use the & operator in scanf on any variable of type char, int, or float, as well as structure types (which we will get to shortly). If you leave out the & operator, you will get an error when you run the program. Try it so that you can see what that sort of run-time error looks like.

Let's look at some variations to understand printf completely. Here is the simplest printf statement:

```
printf("Hello");
```

This call to printf has a format string that tells printf to send the word "Hello" to standard out. Contrast it with this:

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

The difference between the two is that the second version sends the word "Hello" followed by a carriage return to standard out.

The following line shows how to **output the value of a variable using printf**.

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

The **%d** is a placeholder that will be replaced by the value of the variable **b** when the printf statement is executed. Often, you will want to embed the value within some other words. One way to accomplish that is like this:

```
printf("The temperature is ");
printf("%d", b);
printf(" degrees\n");
```

An easier way is to say this:

```
printf("The temperature is %d degrees\n", b);
```

You can also use multiple %d placeholders in one printf statement:

```
printf("%d + %d = %d\n", a, b, c);
```

In the printf statement, it is extremely important that the number of **operators** in the format string corresponds exactly with the number and type of the variables following it. For example, if the format string contains three %d operators, then it must be followed by exactly three parameters and they must have the same types in the same order as those specified by the operators.

You can **print all of the normal C types with printf** by using different placeholders:

- **int** (integer values) uses **%d**
- **float** (floating point values) uses **%f**
- **char** (single character values) uses **%c**
- **character strings** (arrays of characters, discussed later) use **%s**

You can learn more about the nuances of printf on a UNIX machine by typing **man 3 printf**. Any other C compiler you are using will probably come with a manual or a help file that contains a description of printf.

**Scanf**

The **scanf function allows you to accept input from standard in**, which for us is generally the keyboard. The scanf function can do a lot of different things, but it is generally unreliable unless used in the simplest ways. It is unreliable because it does not handle human errors very well. But for simple programs it is good enough and easy-to-use.

The simplest application of **scanf** looks like this:

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

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b.

The scanf function uses the same placeholders as printf:

- **int** uses **%d**
- **float** uses **%f**
- **char** uses **%c**
- **character strings** (discussed later) use **%s**

You MUST put **&** in front of the variable used in scanf. The reason why will become clear once you learn about **pointers**. It is easy to forget the & sign, and when you forget it your program will almost always crash when you run it.

In general, it is best to use scanf as shown here -- to read a single value from the keyboard. Use multiple calls to scanf to read multiple values. In any real program, you will use the **gets** or **fgets** functions instead to read text a line at a time. Then you will "parse" the line to read its values. The reason that you do that is so you can detect errors in the input and handle them as you see fit.

The printf and scanf functions will take a bit of practice to be completely understood, but once mastered they are extremely useful.

### Try This!

- Modify this program so that it accepts three values instead of two and adds all three together:

```
#include <stdio.h>

int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

- Try deleting or adding random characters or words in one of the previous programs and watch how the compiler reacts to these errors.

  For example, delete the b variable in the first line of the above program and see what the compiler does when you forget to declare a variable. Delete a semicolon and see what happens. Leave out one of the braces. Remove one of the parentheses next to the main function. Make each error by itself and then run the program through the compiler to see what happens. By simulating errors like these, you can learn about different compiler errors, and that will make your typos easier to find when you make them for real.

### C Errors to Avoid

- Using the wrong character case - Case matters in C, so you cannot type **Printf** or **PRINTF**. It must be **printf**.
- Forgetting to use the **&** in scanf
- Too many or too few parameters following the format statement in printf or scanf
- Forgetting to declare a variable name before using it

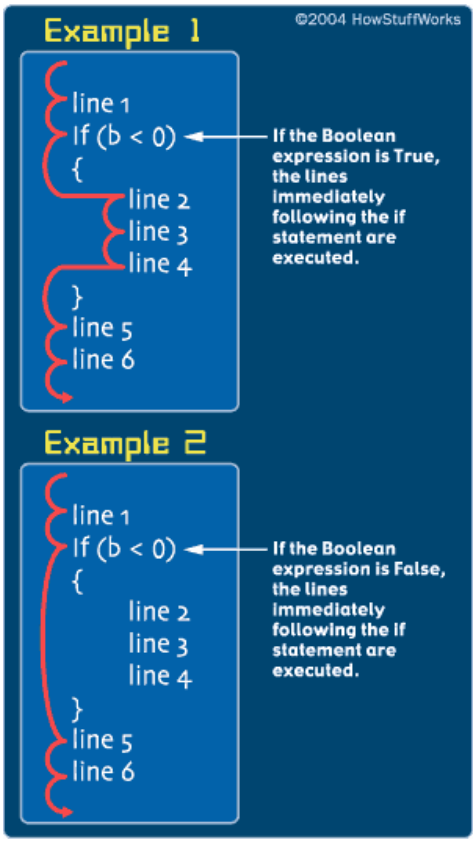
**Branching and Looping**

In C, both **if** statements and **while** loops rely on the idea of **Boolean expressions**. Here is a simple C program demonstrating an if statement:

```
#include <stdio.h>
```

```
int main()
{
    int b;
    printf("Enter a value:");
    scanf("%d", &b);
    if (b < 0)
        printf("The value is negative\n");
    return 0;
}
```

This program accepts a number from the user. It then tests the number using an if statement to see if it is less than 0. If it is, the program prints a message. Otherwise, the program is silent. The **(b < 0)** portion of the program is the Boolean expression. C evaluates this expression to decide whether or not to print the message. If the Boolean expression evaluates to **True**, then C executes the single line immediately following the if statement (or a block of lines within braces immediately following the if statement). If the Boolean expression is **False**, then C skips the line or block of lines immediately following the if statement.




Here's slightly more complex example:

```
#include <stdio.h>

int main()
{
    int b;
    printf("Enter a value:");
    scanf("%d", &b);
    if (b < 0)
        printf("The value is negative\n");
    else if (b == 0)
        printf("The value is zero\n");
    else
        printf("The value is positive\n");
    return 0;
}
```

In this example, the **else if** and **else** sections evaluate for zero and positive values as well.

Here is a more complicated Boolean expression:

```
if ((x==y) && (j>k))
    z=1;
else
    q=10;
```

This statement says, "If the value in variable x equals the value in variable y, and if the value in variable j is greater than the value in variable k, then set the variable z to 1, otherwise set the variable q to 10." You will use if statements like this throughout your C programs to make decisions. In general, most of the decisions you make will be simple ones like the first example; but on occasion, things get more complicated.

Notice that C uses **==** to **test for equality**, while it uses **=** to **assign a value** to a variable. The **&&** in C represents a Boolean AND operation.

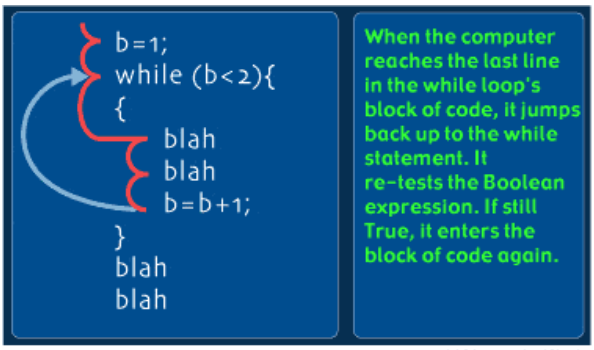
Here are all of the Boolean operators in C:

| | |
|---|---|
| **equality** | **==** |
| **less than** | **<** |
| **Greater than** | **>** |
| **<=** | **<=** |
| **>=** | **>=** |
| **inequality** | **!=** |
| **and** | **&&** |
| **or** | **\|\|** |
| **not** | **!** |

You'll find that **while** statements are just as easy to use as if statements. For example:

```
while (a < b)
{
    printf("%d\n", a);
    a = a + 1;
}
```

This causes the two lines within the braces to be executed repeatedly until **a** is greater than or equal to **b**. The while statement in general works like this:





C also provides a **do-while** structure:

```
do
{
    printf("%d\n", a);
    a = a + 1;
}
while (a < b);
```

The **for loop** in C is simply a shorthand way of expressing a while statement. For example, suppose you have the following code in C:

```
x=1;
while (x<10)
{
    blah blah blah
    x++; /* x++ is the same as saying x=x+1 */
}
```

You can convert this into a for loop as follows:

```
for(x=1; x<10; x++)
{
    blah blah blah
}
```

Note that the while loop contains an initialization step (**x=1**), a test step (**x<10**), and an increment step (**x++**). The for loop lets you put all three parts onto one line, but you can put anything into those three parts. For example, suppose you have the following loop:

```
a=1;
b=6;
while (a < b)
{
    a++;
```

```
    printf("%d\n",a);
}
```

You can place this into a for statement as well:

```
for (a=1,b=6; a < b; a++,printf("%d\n",a));
```

It is slightly confusing, but it is possible. The **comma operator** lets you separate several different statements in the initialization and increment sections of the for loop (but not in the test section). Many C programmers like to pack a lot of information into a single line of C code; but a lot of people think it makes the code harder to understand, so they break it up.

**= vs. == in Boolean expressions**

The **==** sign is a problem in C because every now and then you may forget and type just **=** in a Boolean expression. This is an easy mistake to make, but to the compiler there is a very important difference. C will accept either **=** and **==** in a Boolean expression -- the behavior of the program changes remarkably between the two, however.

Boolean expressions evaluate to integers in C, and integers can be used inside of Boolean expressions. The integer value 0 in C is False, while any other integer value is True. The following is legal in C:

```
#include <stdio.h>

int main()
{
   int a;

   printf("Enter a number:");
   scanf("%d", &a);
   if (a)
   {
      printf("The value is True\n");
   }
   return 0;
}
```

If **a** is anything other than 0, the printf statement gets executed.

In C, a statement like **if (a=b)** means, "Assign **b** to **a**, and then test **a** for its Boolean value." So if **a** becomes 0, the if statement is False; otherwise, it is True. The value of **a** changes in the process. This is not the intended behavior if you meant to type **==** (although this feature is useful when used correctly), so be careful with your **=** and **==** usage.

**Looping: A Real Example**

Let's say that you would like to create a program that prints a Fahrenheit-to-Celsius conversion table. This is easily accomplished with a for loop or a while loop:

```
#include <stdio.h>

int main()
{
    int a;
    a = 0;
    while (a <= 100)
    {
        printf("%4d degrees F = %4d degrees C\n",
            a, (a - 32) * 5 / 9);
        a = a + 10;
    }
    return 0;
}
```

If you run this program, it will produce a table of values starting at 0 degrees F and ending at 100 degrees F. The output will look like this:

```
   0 degrees F =  -17 degrees C
  10 degrees F =  -12 degrees C
```

```
  20 degrees F =   -6 degrees C
  30 degrees F =   -1 degrees C
  40 degrees F =    4 degrees C
  50 degrees F =   10 degrees C
  60 degrees F =   15 degrees C
  70 degrees F =   21 degrees C
  80 degrees F =   26 degrees C
  90 degrees F =   32 degrees C
 100 degrees F =   37 degrees C
```

The table's values are in increments of 10 degrees. You can see that you can easily change the starting, ending or increment values of the table that the program produces.

If you wanted your values to be more accurate, you could use **floating point** values instead:

```
#include <stdio.h>

int main()
{
    float a;
    a = 0;
    while (a <= 100)
    {
        printf("%6.2f degrees F = %6.2f degrees C\n",
            a, (a - 32.0) * 5.0 / 9.0);
        a = a + 10;
    }
    return 0;
}
```

You can see that the declaration for **a** has been changed to a float, and the **%f** symbol replaces the **%d** symbol in the printf statement. In addition, the %f symbol has some formatting applied to it: The value will be printed with six digits preceding the decimal point and two digits following the decimal point.

Now let's say that we wanted to modify the program so that the temperature 98.6 is inserted in the table at the proper position. That is, we want the table to increment every 10 degrees, but we also want the table to include an extra line for 98.6 degrees F because that is the normal body temperature for a human being. The following program accomplishes the goal:

```
#include <stdio.h>

int main()
{
    float a;
    a = 0;
    while (a <= 100)
    {
        if (a > 98.6)
        {
            printf("%6.2f degrees F = %6.2f degrees C\n",
                98.6, (98.6 - 32.0) * 5.0 / 9.0);
        }
        printf("%6.2f degrees F = %6.2f degrees C\n",
            a, (a - 32.0) * 5.0 / 9.0);
        a = a + 10;
    }
    return 0;
}
```

This program works if the ending value is 100, but if you change the ending value to 200 you will find that the program has a **bug**. It prints the line for 98.6 degrees too many times. We can fix that problem in several different ways. Here is one way:

```
#include <stdio.h>

int main()
{
    float a, b;
    a = 0;
    b = -1;
    while (a <= 100)
    {
        if ((a > 98.6) && (b < 98.6))
        {
            printf("%6.2f degrees F = %6.2f degrees C\n",
                98.6, (98.6 - 32.0) * 5.0 / 9.0);
        }
        printf("%6.2f degrees F = %6.2f degrees C\n",
            a, (a - 32.0) * 5.0 / 9.0);
        b = a;
```

```
        a = a + 10;
    }
    return 0;
}
```

### Try This!

- Try changing the Fahrenheit-to-Celsius program so that it uses scanf to accept the starting, ending and increment value for the table from the user.
- Add a heading line to the table that is produced.
- Try to find a different solution to the bug fixed by the previous example.
- Create a table that converts pounds to kilograms or miles to kilometers.

### C Errors to Avoid

- Putting = when you mean == in an if or while statement
- Forgetting to increment the counter inside the while loop - If you forget to increment the counter, you get an **infinite** loop (the loop never ends).
- Accidentally putting a **;** at the end of a for loop or if statement so that the statement has no effect - For example:

```
for (x=1; x<10; x++);
   printf("%d\n",x);
```

only prints out one value because the semicolon after the for statement acts as the one line the for loop executes.

**Arrays**

In this section, we will create a small C program that generates 10 random numbers and sorts them. To do that, we will use a new variable arrangement called an **array**.

An array lets you declare and work with a collection of values of the same type. For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

```
int a, b, c, d, e;
```

This is okay, but what if you needed a thousand integers? An easier way is to declare an array of five integers:

```
int a[5];
```

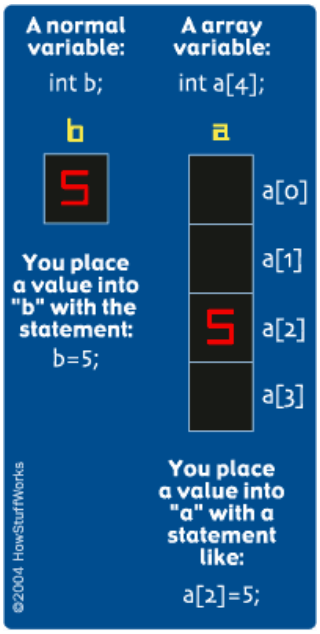
The five separate integers inside this array are accessed by an **index**. All arrays start at index zero and go to n-1 in C. Thus, **int a[5];** contains five elements. For example:

```
int a[5];

a[0] = 12;
a[1] = 9;
a[2] = 14;
a[3] = 5;
a[4] = 1;
```

One of the nice things about array indexing is that you can use a loop to manipulate the index. For example, the following code initializes all of the values in the array to 0:

```
int a[5];
int i;

for (i=0; i<5; i++)
    a[i] = 0;
```

The following code initializes the values in the array sequentially and then prints them out:



```
#include <stdio.h>

int main()
{
    int a[5];
    int i;

    for (i=0; i<5; i++)
        a[i] = i;
    for (i=0; i<5; i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

Arrays are used all the time in C. To understand a common usage, start an editor and enter the following code:

```
#include <stdio.h>

#define MAX 10

int a[MAX];
int rand_seed=10;

/* from K&R
   - returns random number between 0 and 32767.*/
int rand()
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}

int main()
{
    int i,t,x,y;

    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("%d\n",a[i]);
    }

    /* more stuff will go here in a minute */

    return 0;
}
```

This code contains several new concepts. The **#define** line declares a constant named **MAX** and sets it to 10. Constant names are traditionally written in all caps to make them obvious in the code. The line **int a[MAX];** shows you how to declare an array of integers in C. Note that because of the position of the array's declaration, it is global to the entire program.

The line **int rand_seed=10** also declares a global variable, this time named **rand_seed**, that is initialized to 10 each time the program begins. This value is the starting seed for the random number code that follows. In a real random number generator, the seed should initialize as a random value, such as the system time. Here, the **rand** function will produce the same values each time you run the program.

The line **int rand()** is a function declaration. The rand function accepts no parameters and returns an integer value. We will learn more about functions later. The four lines that follow implement the rand function. We will ignore them for now.

The main function is normal. Four local integers are declared, and the array is filled with 10 random values using a for loop. Note that the array **a** contains 10 individual integers. You point to a specific integer in the array using square brackets. So **a[0]** refers to the first integer in the array, **a[1]** refers to the second, and so on. The line starting with **/*** and ending with ***/** is called a **comment**. The compiler completely ignores the line. You can place notes to yourself or other programmers in comments.

Now add the following code in place of the **more stuff ...** comment:

```
/* bubble sort the array */
for (x=0; x < MAX-1; x++)
    for (y=0; y < MAX-x-1; y++)
        if (a[y] > a[y+1])
        {
            t=a[y];
            a[y]=a[y+1];
            a[y+1]=t;
        }
/* print sorted array */
printf("--------------------\n");
for (i=0; i < MAX; i++)
```

```
printf("%d\n",a[i]);
```

This code **sorts** the random values and prints them in sorted order. Each time you run it, you will get the same values. If you would like to change the values that are sorted, change the value of rand_seed each time you run the program.

The only easy way to truly understand what this code is doing is to execute it "by hand." That is, assume **MAX** is 4 to make it a little more manageable, take out a sheet of paper and pretend you are the computer. Draw the array on your paper and put four random, unsorted values into the array. Execute each line of the sorting section of the code and draw out exactly what happens. You will find that, each time through the inner loop, the larger values in the array are pushed toward the bottom of the array and the smaller values bubble up toward the top.

### Try This!

- In the first piece of code, try changing the for loop that fills the array to a single line of code. Make sure that the result is the same as the original code.
- Take the bubble sort code out and put it into its own function. The function header will be **void bubble_sort()**. Then move the variables used by the bubble sort to the function as well, and make them local there. Because the array is global, you do not need to pass parameters.
- Initialize the random number seed to different values.

### C Errors to Avoid

- C has no range checking, so if you index past the end of the array, it will not tell you about it. It will eventually crash or give you garbage data.
- A function call must include **()** even if no parameters are passed. For example, C will accept **x=rand;**, but the call will not work. The memory address of the rand function will be placed into **x** instead. You must say **x=rand();**.

**More on Arrays**

## Variable Types

There are three standard variable types in C:

- **Integer: int**
- **Floating point: float**
- **Character: char**

An int is a 4-byte integer value. A float is a 4-byte floating point value. A char is a 1-byte single character (like "a" or "3"). A string is declared as an array of characters.

There are a number of derivative types:

- **double** (8-byte floating point value)
- **short** (2-byte integer)
- **unsigned short** or **unsigned int** (positive integers, no sign bit)

## Operators and Operator Precedence

The operators in C are similar to the operators in most languages:

```
+ - addition
- - subtraction
/ - division
* - multiplication
% - mod
```

The **/** operator performs integer division if both operands are integers, and performs floating point division otherwise. For example:

```
void main()
{
    float a;
```

```
    a=10/3;
    printf("%f\n",a);
}
```

This code prints out a floating point value since **a** is declared as type **float**, but **a** will be 3.0 because the code performed an integer division.

**Operator precedence** in C is also similar to that in most other languages. Division and multiplication occur first, then addition and subtraction. The result of the calculation 5+3*4 is 17, not 32, because the * operator has higher precedence than + in C. You can use parentheses to change the normal precedence ordering: (5+3)*4 is 32. The 5+3 is evaluated first because it is in parentheses. We'll get into precedence later -- it becomes somewhat complicated in C once pointers are introduced.

## Typecasting

C allows you to perform type conversions on the fly. You do this especially often when using pointers. Typecasting also occurs during the assignment operation for certain types. For example, in the code above, the integer value was automatically converted to a float.

You do typecasting in C by placing the type name in parentheses and putting it in front of the value you want to change. Thus, in the above code, replacing the line **a=10/3;** with **a=(float)10/3;** produces 3.33333 as the result because 10 is converted to a floating point value before the division.

## Typedef

You declare named, user-defined types in C with the **typedef** statement. The following example shows a type that appears often in C code:

```
#define TRUE  1
#define FALSE 0
typedef int boolean;

void main()
{
    boolean b;

    b=FALSE;
    blah blah blah
}
```

This code allows you to declare Boolean types in C programs.

If you do not like the word "float" for real numbers, you can say:

```
typedef float real;
```

and then later say:

```
real r1,r2,r3;
```

You can place typedef statements anywhere in a C program as long as they come prior to their first use in the code.

## Structures

Structures in C allow you to group variable into a package. Here's an example:

```
struct rec
{
    int a,b,c;
    float d,e,f;
};

struct rec r;
```

As shown here, whenever you want to declare structures of the type **rec**, you have to say **struct rec**. This line is very easy to forget, and you get many compiler errors because you absent-mindedly leave out the **struct**. You can compress the code into the form:

```
struct rec
{
    int a,b,c;
    float d,e,f;
} r;
```

where the type declaration for **rec** and the variable **r** are declared in the same statement. Or you can create a typedef statement for the structure name. For example, if you do not like saying **struct rec r** every time you want to declare a record, you can say:

```
typedef struct rec rec_type;
```

and then declare records of type **rec_type** by saying:

```
rec_type r;
```

You access fields of structure using a period, for example, **r.a=5;**.

## Arrays

You declare arrays by inserting an array size after a normal declaration, as shown below:

```
int a[10];        /* array of integers */
char s[100];      /* array of characters
                     (a C string) */
float f[20];      /* array of reals */
struct rec r[50]; /* array of records */
```

**Incrementing**

```
Long Way     Short Way
i=i+1;       i++;
i=i-1;       i--;
i=i+3;       i += 3;
i=i*j;       i *= j;
```

### Try This!

- Try out different pieces of code to investigate typecasting and precedence. Try out **int**, **char**, **float**, and so on.
- Create an array of records and write some code to sort that array on one integer field.

### C Error to Avoid

- As described above, using the / operator with two integers will often produce an unexpected result, so think about it whenever you use it.

**Functions**

Most languages allow you to create functions of some sort. Functions let you chop up a long program into named sections so that the sections can be reused throughout the program. Functions accept **parameters** and **return** a result. C functions can accept an unlimited number of parameters. In general, C does not care in what order you put your functions in the program, so long as a the function name is known to the compiler before it is called.

We have already talked a little about functions. The **rand** function seen previously is about as simple as a function can get. It accepts no parameters and returns an integer result:

```
int rand()
/* from K&R
   - produces a random number between 0 and 32767.*/
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}
```

The **int rand()** line declares the function rand to the rest of the program and specifies that rand will accept no parameters and return an integer result. This function has no local variables, but if it needed locals, they would go right below the opening **{** (C allows you to declare variables after any **{** -- they exist until the program reaches the matching **}** and then they disappear. A function's local variables therefore vanish as soon as the matching **}** is reached in the function. While they exist, local variables live on the system stack.) Note that there is no **;** after the **()** in the first line. If you accidentally put one in, you will get a huge cascade of error messages from the compiler that make no sense. Also note that even though there are no parameters, you must use the **()**. They tell the compiler that you are declaring a function rather than simply declaring an int.

The **return** statement is important to any function that returns a result. It specifies the value that the function will return and causes the function to exit immediately. This means that you can place multiple return statements in the function to give it multiple exit points. If you do not place a return statement in a function, the function returns when it reaches **}** and returns a random value (many compilers will warn you if you fail to return a specific value). In C, a function can return values of any type: int, float, char, struct, etc.

There are several correct ways to call the **rand** function. For example: **x=rand();**. The variable **x** is assigned the value returned by rand in this statement. Note that you *must* use **()** in the function call, even though no parameter is passed. Otherwise, **x** is given the memory address of the rand function, which is generally not what you intended.

You might also call rand this way:

```
if (rand() > 100)
```

Or this way:

```
rand();
```

In the latter case, the function is called but the value returned by rand is discarded. You may never want to do this with rand, but many functions return some kind of error code through the function name, and if you are not concerned with the error code (for example, because you know that an error is impossible) you can discard it in this way.

Functions can use a void return type if you intend to return nothing. For example:

```
void print_header()
{
    printf("Program Number 1\n");
    printf("by Marshall Brain\n");
    printf("Version 1.0, released 12/26/91\n");
}
```

This function returns no value. You can call it with the following statement:

```
print_header();
```

You must include **()** in the call. If you do not, the function is not called, even though it will compile correctly on many systems.

C functions can accept parameters of any type. For example:

```
int fact(int i)
{
    int j,k;

    j=1;
    for (k=2; k<=i; k++)
        j=j*k;
    return j;
}
```

returns the factorial of **i**, which is passed in as an integer parameter. Separate multiple parameters with commas:

```
int add (int i, int j)
{
    return i+j;
}
```

C has evolved over the years. You will sometimes see functions such as **add** written in the "old style," as shown below:

```
int add(i,j)
    int i;
    int j;
{
    return i+j;
}
```

It is important to be able to read code written in the older style. There is no difference in the way it executes; it is just a different notation. You should use the "new style," (known as **ANSI C**) with the type declared as part of the parameter list, unless you know you will be shipping the code to someone who has access only to an "old style" (non-ANSI) compiler.

**Functions: Function Prototypes**

It is now considered good form to use **function prototypes** for all functions in your program. A prototype declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual declaration. To understand why function prototypes are useful, enter the following code and run it:

```
#include <stdio.h>

void main()
{
    printf("%d\n",add(3));
}

int add(int i, int j)
{
    return i+j;
}
```

This code compiles on many compilers without giving you a warning, even though **add** expects two parameters but receives only one. It works because many C compilers do not check for parameter matching either in type or count. You can waste an enormous amount of time debugging code in which you are simply passing one too many or too few parameters by mistake. The above code compiles properly, but it produces the wrong answer.

To solve this problem, C lets you place function prototypes at the beginning of (actually, anywhere in) a program. If you do so, C checks the types and counts of all parameter lists. Try compiling the following:

```
#include <stdio.h>

int add (int,int); /* function prototype for add */

void main()
{
    printf("%d\n",add(3));
}

int add(int i, int j)
{
    return i+j;
}
```

The prototype causes the compiler to flag an error on the **printf** statement.

Place one prototype for each function at the beginning of your program. They can save you a great deal of debugging time, and they also solve the problem you get when you compile with functions that you use before they are declared. For example, the following code will not compile:

```
#include <stdio.h>

void main()
{
    printf("%d\n",add(3));
}

float add(int i, int j)
{
    return i+j;
}
```

Why, you might ask, will it compile when add returns an int but not when it returns a float? Because older C compilers default to an int return value. Using a prototype will solve this problem. "Old style" (non-ANSI) compilers allow prototypes, but the parameter list for the prototype must be empty. Old style compilers do no error checking on parameter lists.

### Try This!

- Go back to the bubble sort example presented earlier and create a function for the bubble sort.
- Go back to earlier programs and create a function to get input from the user rather than taking the input in the main function.

**Libraries**

Libraries are very important in C because the C language supports only the most basic features that it needs. C does not even contain I/O functions to read from the keyboard and write to the screen. Anything that extends beyond the basic language must be written by a programmer. The resulting chunks of code are often placed in **libraries** to make them easily reusable. We have seen the standard I/O, or **stdio**, library already: Standard libraries exist for standard I/O, math functions, string handling, time manipulation, and so on. You can use libraries in your own programs to split up your programs into modules. This makes them easier to understand, test, and debug, and also makes it possible to reuse code from other programs that you write.

You can create your own libraries easily. As an example, we will take some code from a previous article in this series and make a library out of two of its functions. Here's the code we will start with:

```
#include <stdio.h>

#define MAX 10

int a[MAX];
int rand_seed=10;

int rand()
/* from K&R
   - produces a random number between 0 and 32767.*/
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}

void main()
```

```
{
    int i,t,x,y;

    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("%d\n",a[i]);
    }

    /* bubble sort the array */
    for (x=0; x < MAX-1; x++)
        for (y=0; y < MAX-x-1; y++)
            if (a[y] > a[y+1])
            {
                t=a[y];
                a[y]=a[y+1];
                a[y+1]=t;
            }

    /* print sorted array */
    printf("--------------------\n");
    for (i=0; i < MAX; i++)
        printf("%d\n",a[i]);
}
```

This code fills an array with random numbers, sorts them using a bubble sort, and then displays the sorted list.

Take the bubble sort code, and use what you learned in the previous article to make a function from it. Since both the array **a** and the constant MAX are known globally, the function you create needs no parameters, nor does it need to return a result. However, you should use local variables for x, y, and t.

Once you have tested the function to make sure it is working, pass in the number of elements as a parameter rather than using MAX:

```
#include <stdio.h>

#define MAX 10

int a[MAX];
int rand_seed=10;

/* from K&R
   - returns random number between 0 and 32767.*/
int rand()
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}

void bubble_sort(int m)
{
    int x,y,t;
     for (x=0; x < m-1; x++)
        for (y=0; y < m-x-1; y++)
            if (a[y] > a[y+1])
            {
                t=a[y];
                a[y]=a[y+1];
                a[y+1]=t;
            }
}

void main()
{
    int i,t,x,y;
    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("%d\n",a[i]);
    }
    bubble_sort(MAX);
    /* print sorted array */
    printf("--------------------\n");
    for (i=0; i < MAX; i++)
        printf("%d\n",a[i]);
}
```

You can also generalize the **bubble_sort** function even more by passing in **a** as a parameter:

```
bubble_sort(int m, int a[])
```

This line says, "Accept the integer array a of any size as a parameter." Nothing in the body of the **bubble_sort** function needs to change. To call bubble_sort, change the call to:

```
bubble_sort(MAX, a);
```

Note that **&a** has not been used in the function call even though the sort will change **a**. The reason for this will become clear once you understand pointers.

**Making a Library**

Since the rand and bubble_sort functions in the previous program are useful, you will probably want to reuse them in other programs you write. You can put them into a utility library to make their reuse easier.

Every library consists of two parts: a header file and the actual code file. The header file, normally denoted by a **.h** suffix, contains information about the library that programs using it need to know. In general, the header file contains constants and types, along with prototypes for functions available in the library. Enter the following header file and save it to a file named **util.h**.

```
/* util.h */
extern int rand();
extern void bubble_sort(int, int []);
```

These two lines are function prototypes. The word "extern" in C represents functions that will be linked in later. If you are using an old-style compiler, remove the parameters from the parameter list of **bubble_sort**.

Enter the following code into a file named **util.c**.

```
/* util.c */
#include "util.h"

int rand_seed=10;

/* from K&R
 - produces a random number between 0 and 32767.*/
int rand()
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}

void bubble_sort(int m,int a[])
{
    int x,y,t;
     for (x=0; x < m-1; x++)
        for (y=0; y < m-x-1; y++)
            if (a[y] > a[y+1])
            {
                t=a[y];
                a[y]=a[y+1];
                a[y+1]=t;
            }
}
```

Note that the file includes its own header file (**util.h**) and that it uses quotes instead of the symbols **<** and**>** , which are used only for system libraries. As you can see, this looks like normal C code. Note that the variable **rand_seed**, because it is not in the header file, cannot be seen or modified by a program using this library. This is called information hiding. Adding the word **static** in front of **int** enforces the hiding completely.

Enter the following main program in a file named **main.c**.

```
#include <stdio.h>
#include "util.h"

#define MAX 10

int a[MAX];

void main()
{
    int i,t,x,y;
    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("%d\n",a[i]);
    }
```

```
    bubble_sort(MAX,a);

    /* print sorted array */
    printf("--------------------\n");
    for (i=0; i < MAX; i++)
        printf("%d\n",a[i]);
}
```

This code includes the utility library. The main benefit of using a library is that the code in the main program is much shorter.

## Compiling and Running with a Library

To compile the library, type the following at the command line (assuming you are using UNIX) (replace gcc with cc if your system uses cc):

```
gcc -c -g util.c
```

The **-c** causes the compiler to produce an object file for the library. The object file contains the library's machine code. It cannot be executed until it is linked to a program file that contains a main function. The machine code resides in a separate file named **util.o**.

To compile the main program, type the following:

```
gcc -c -g main.c
```

This line creates a file named **main.o** that contains the machine code for the main program. To create the final executable that contains the machine code for the entire program, link the two object files by typing the following:

```
gcc -o main main.o util.o
```

This links **main.o** and **util.o** to form an executable named **main**. To run it, type **main**.

Makefiles make working with libraries a bit easier. You'll find out about makefiles on the next page.

**Makefiles**

It can be cumbersome to type all of the **gcc** lines over and over again, especially if you are making a lot of changes to the code and it has several libraries. The **make** facility solves this problem. You can use the following makefile to replace the compilation sequence above:

```
main: main.o util.o
        gcc -o main main.o util.o
main.o: main.c util.h
        gcc -c -g main.c
util.o: util.c util.h
        gcc -c -g util.c
```

Enter this into a file named **makefile**, and type **make**to build the executable. Note that you *must* precede all **gcc** lines with a tab. (Eight spaces will not suffice -- it must be a tab. All other lines must be flush left.)

This makefile contains two types of lines. The lines appearing flush left are **dependency lines**. The lines preceded by a tab are executable lines, which can contain any valid UNIX command. A dependency line says that some file is dependent on some other set of files. For example, **main.o: main.c util.h** says that the file **main.o** is dependent on the files **main.c** and **util.h**. If either of these two files changes, the following executable line(s) should be executed to recreate **main.o**.

Note that the final executable produced by the whole makefile is main, on line 1 in the makefile. The final result of the makefile should always go on line 1, which in this makefile says that the file **main** is dependent on **main.o** and **util.o**. If either of these changes, execute the line **gcc -o main main.o util.o** to recreate **main**.

It is possible to put multiple lines to be executed below a dependency line -- they must all start with a tab. A large program may have several libraries and a main program. The makefile automatically recompiles everything that needs to be recompiled because of a change.

If you are not working on a UNIX machine, your compiler almost certainly has functionality equivalent to makefiles. Read the documentation for your compiler to learn how to use it.

Now you understand why you have been including stdio.h in earlier programs. It is simply a standard library that someone created long ago and made available to other programmers to make their lives easier.

**Text Files**

Text files in C are straightforward and easy to understand. All text file functions and types in C come from the **stdio** library.

When you need text I/O in a C program, and you need only one source for input information and one sink for output information, you can rely on **stdin** (standard in) and **stdout** (standard out). You can then use input and output redirection at the command line to move different information streams through the program. There are six different I/O commands in <stdio.h> that you can use with stdin and stdout:

- **printf** - prints formatted output to stdout
- **scanf** - reads formatted input from stdin
- **puts** - prints a string to stdout

- **gets** - reads a string from stdin
- **putc** - prints a character to stdout
- **getc**, **getchar** - reads a character from stdin

The advantage of stdin and stdout is that they are easy to use. Likewise, the ability to redirect I/O is very powerful. For example, maybe you want to create a program that reads from stdin and counts the number of characters:

```
#include <stdio.h>
#include <string.h>

void main()
{
    char s[1000];
    int count=0;
     while (gets(s))
        count += strlen(s);
    printf("%d\n",count);
}
```

Enter this code and run it. It waits for input from stdin, so type a few lines. When you are done, press CTRL-D to signal end-of-file (eof). The gets function reads a line until it detects eof, then returns a 0 so that the while loop ends. When you press CTRL-D, you see a count of the number of characters in stdout (the screen). (Use **man gets** or your compiler's documentation to learn more about the gets function.)

Now, suppose you want to count the characters in a file. If you compiled the program to an executable named **xxx**, you can type the following:

```
xxx < filename
```

Instead of accepting input from the keyboard, the contents of the file named **filename** will be used instead. You can achieve the same result using **pipes**:

```
cat < filename | xxx
```

You can also redirect the output to a file:

```
xxx < filename > out
```

This command places the character count produced by the program in a text file named **out**.

Sometimes, you need to use a text file directly. For example, you might need to open a specific file and read from or write to it. You might want to manage several streams of input or output or create a program like a text editor that can save and recall data or configuration files on command. In that case, use the text file functions in stdio:

- **fopen** - opens a text file
- **fclose** - closes a text file
- **feof** - detects end-of-file marker in a file
- **fprintf** - prints formatted output to a file
- **fscanf** - reads formatted input from a file
- **fputs** - prints a string to a file
- **fgets** - reads a string from a file
- **fputc** - prints a character to a file
- **fgetc** - reads a character from a file

**Text Files: Opening**

You use **fopen** to open a file. It opens a file for a specified mode (the three most common are r, w, and a, for read, write, and append). It then returns a file pointer that you use to access the file. For example, suppose you want to open a file and write the numbers 1 to 10 in it. You could use the following code:

```
#include <stdio.h>
#define MAX 10

int main()
{
    FILE *f;
    int x;
    f=fopen("out","w");
    if (!f)
        return 1;
    for(x=1; x<=MAX; x++)
        fprintf(f,"%d\n",x);
    fclose(f);
    return 0;
}
```

The **fopen** statement here opens a file named **out** with the w mode. This is a destructive write mode, which means that if **out** does not exist it is created, but if it does exist it is destroyed and a new file is created in its place. The fopen command returns a pointer to the file, which is stored in the variable f. This variable is used to refer to the file. If the file cannot be opened for some reason, f will contain NULL.

### Main Function Return Values

This program is the first program in this series that returns an error value from the main program. If the **fopen** command fails, f will contain a NULL value (a zero). We test for that error with the if statement. The if statement looks at the True/False value of the variable **f**. Remember that in C, 0 is False and anything else is true. So if there were an error opening the file, f would contain zero, which is False. The **!** is the NOT operator. It inverts a Boolean value. So the if statement could have been written like this:

```
if (f == 0)
```

That is equivalent. However, **if (!f)** is more common.

If there is a file error, we return a 1 from the main function. In UNIX, you can actually test for this value on the command line. See the shell documentation for details.

The fprintf statement should look very familiar: It is just like printf but uses the file pointer as its first parameter. The fclose statement closes the file when you are done.

**Text Files: Reading**

To read a file, open it with r mode. In general, it is not a good idea to use **fscanf** for reading: Unless the file is perfectly formatted, fscanf will not handle it correctly. Instead, use **fgets** to read in each line and then parse out the pieces you need.

The following code demonstrates the process of reading a file and dumping its contents to the screen:

```
#include <stdio.h>

int main()
{
    FILE *f;
    char s[1000];

    f=fopen("infile","r");
    if (!f)
        return 1;
    while (fgets(s,1000,f)!=NULL)
        printf("%s",s);
    fclose(f);
    return 0;
}
```

The fgets statement returns a NULL value at the end-of-file marker. It reads a line (up to 1,000 characters in this case) and then prints it to stdout. Notice that the printf statement does not include \n in the format string, because fgets adds \n to the end of each line it reads. Thus, you can tell if a line is not complete in the event that it overflows the maximum line length specified in the second parameter to fgets.

### C Errors to Avoid

- Do not accidentally type **close** instead of **fclose**. The close function exists, so the compiler accepts it. It will even appear to work if the program only opens or closes a few files. However, if the program opens and closes a file in a loop, it will eventually run out of available file handles and/or memory space and crash, because **close** is not closing the files correctly.

**Pointers**

Pointers are used everywhere in C, so if you want to use the C language fully you have to have a very good understanding of pointers. They have to become *comfortable* for you. The goal of this section and the next several that follow is to help you build a complete understanding of pointers and how C uses them. For most people it takes a little time and some practice to become fully comfortable with pointers, but once you master them you are a full-fledged C programmer.

C uses pointers in three different ways:

- C uses pointers to create **dynamic data structures** -- data structures built up from blocks of memory allocated from the heap at run-time.
- C uses pointers to handle **variable parameters** passed to functions.
- Pointers in C provide an alternative way to **access information stored in arrays**. Pointer techniques are especially valuable when you work with

strings. There is an intimate link between arrays and pointers in C.

In some cases, C programmers also use pointers because they make the code slightly more efficient. What you will find is that, once you are completely comfortable with pointers, you tend to use them all the time.

We will start this discussion with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Especially on this article, you will want to read things twice. The first time through you can learn all the concepts. The second time through you can work on binding the concepts together into an integrated whole in your mind. After you make your way through the material the second time, it will make a lot of sense.

**Pointers: Why?**

Imagine that you would like to create a **text editor** -- a program that lets you edit normal ASCII text files, like "vi" on UNIX or "Notepad" on Windows. A text editor is a fairly common thing for someone to create because, if you think about it, a text editor is probably a programmer's most commonly used piece of software. The text editor is a programmer's intimate link to the computer -- it is where you enter all of your thoughts and then manipulate them. Obviously, with anything you use that often and work with that closely, you want it to be just right. Therefore many programmers create their own editors and customize them to suit their individual working styles and preferences.

So one day you sit down to begin working on your editor. After thinking about the features you want, you begin to think about the "data structure" for your editor. That is, you begin thinking about how you will store the document you are editing in memory so that you can manipulate it in your program. What you need is a way to store the information you are entering in a form that can be manipulated quickly and easily. You believe that one way to do that is to organize the data on the basis of lines of characters. Given what we have discussed so far, the only thing you have at your disposal at this point is an array. You think, "Well, a typical line is 80 characters long, and a typical file is no more than 1,000 lines long." You therefore declare a **two-dimensional array**, like this:

```
char doc[1000][80];
```

This declaration requests an array of 1,000 80-character lines. This array has a total size of 80,000 characters.

As you think about your editor and its data structure some more, however, you might realize three things:

- Some documents are long lists. Every line is short, but there are thousands of lines.
- Some special-purpose text files have very long lines. For example, a certain data file might have lines containing 542 characters, with each character representing the amino acid pairs in segments of DNA.
- In most modern editors, you can open multiple files at one time.

Let's say you set a maximum of 10 open files at once, a maximum line length of 1,000 characters and a maximum file size of 50,000 lines. Your declaration now looks like this:

```
char doc[50000][1000][10];
```

That doesn't seem like an unreasonable thing, until you pull out your calculator, multiply 50,000 by 1,000 by 10 and realize the array contains 500 million characters! Most computers today are going to have a problem with an array that size. They simply do not have the RAM, or even the virtual memory space, to support an array that large. If users were to try to run three or four copies of this program simultaneously on even the largest multi-user system, it would put a severe strain on the facilities.

Even if the computer would accept a request for such a large array, you can see that it is an extravagant waste of space. It seems strange to declare a 500 million character array when, in the vast majority of cases, you will run this editor to look at 100 line files that consume at most 4,000 or 5,000 bytes. The problem with an array is the fact that **you have to declare it to have its maximum size in every dimension from the beginning.** Those maximum sizes often multiply together to form very large numbers. Also, if you happen to need to be able to edit an odd file with a 2,000 character line in it, you are out of luck. There is really no way for you to predict and handle the maximum line length of a text file, because, technically, that number is infinite.

Pointers are designed to solve this problem. With pointers, you can create **dynamic data structures**. Instead of declaring your worst-case memory consumption up-front in an array, you instead **allocate** memory from **the heap** while the program is running. That way you can use the exact amount of memory a document needs, with no waste. In addition, when you close a document you can return the memory to the heap so that other parts of the program can use it. With pointers, memory can be recycled while the program is running.

**Pointer Basics**

To understand pointers, it helps to compare them to normal variables.

A "normal variable" is a location in memory that can hold a value. For example, when you declare a variable **i** as an integer, four bytes of memory are set aside for it. In your program, you refer to that location in memory by the name **i**. At the machine level that location has a memory address. The four bytes at that address are known to you, the programmer, as **i**, and the four bytes can hold one integer value.

A pointer is different. A pointer is a variable that **points** to another variable. This means that a pointer holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. A pointer "points to" that other variable by holding a copy of its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That address points to a value. There is the pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get comfortable it becomes extremely powerful.

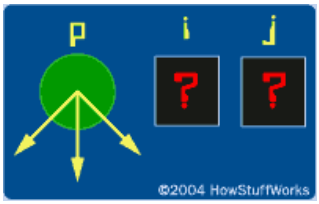
The following example code shows a typical pointer:

```
#include <stdio.h>

int main()
{
    int i,j;
    int *p;   /* a pointer to an integer */
    p = &i;
    *p=5;
    j=i;
    printf("%d %d %d\n", i, j, *p);
    return 0;
}
```

The first declaration in this program declares two normal integer variables named **i** and **j**. The line **int *p** declares a pointer named **p**. This line asks the compiler to declare a variable **p** that is a **pointer** to an integer. The * indicates that a pointer is being declared rather than a normal variable. You can create a pointer to anything: a float, a structure, a char, and so on. Just use a * to indicate that you want a pointer rather than a normal variable.

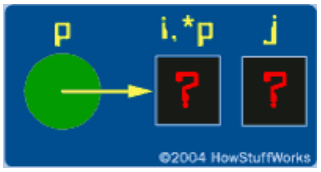
The line **p = &i;** will definitely be new to you. In C, **&** is called the **address operator**. The expression **&i** means, "The memory address of the variable **i**." Thus, the expression **p = &i;** means, "Assign to **p** the address of **i**." Once you execute this statement, **p** "points to" **i**. Before you do so, **p** contains a random, unknown address, and its use will likely cause a segmentation fault or similar program crash.

One good way to visualize what is happening is to draw a picture. After **i**, **j** and **p** are declared, the world looks like this:

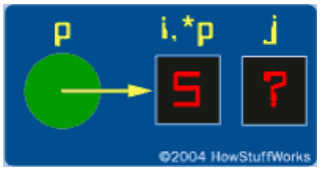



In this drawing the three variables **i**, **j** and **p** have been declared, but none of the three has been initialized. The two integer variables are therefore drawn as boxes containing question marks -- they could contain any value at this point in the program's execution. The pointer is drawn as a circle to distinguish it from a normal variable that holds a value, and the random arrows indicate that it can be pointing anywhere at this moment.

After the line **p = &I;**, **p** is initialized and it points to **i**, like this:




Once **p** points to **i**, the memory location **i** has two names. It is still known as **i**, but now it is known as ***p** as well. This is how C talks about the two parts of a pointer variable: **p** is the location holding the address, while ***p** is the location pointed to by that address. Therefore ***p=5** means that the location pointed to by **p** should be set to 5, like this:




Because the location ***p** is also **i**, **i** also takes on the value 5. Consequently, **j=i;** sets **j** to 5, and the **printf** statement produces **5 5 5**.

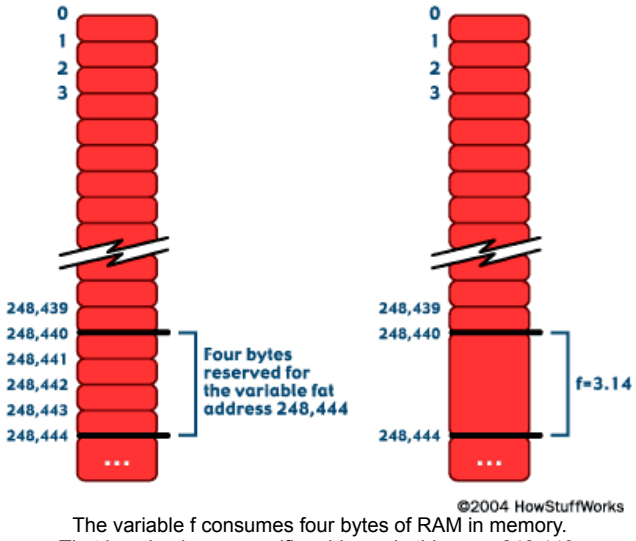
The main feature of a pointer is its two-part nature. The pointer itself holds an address. The pointer also points to a value of a specific type - the value at the address the point holds. The pointer itself, in this case, is **p**. The value pointed to is ***p**.

**Pointers: Understanding Memory Addresses**

All computers have **memory**, also known as **RAM** (random access memory). For example, your computer might have 16 or 32 or 64 megabytes of RAM installed right now. RAM holds the programs that your computer is currently running along with the data they are currently manipulating (their variables and data structures). Memory can be thought of simply as an array of bytes. In this array, every memory location has its own address -- the address of the first byte is 0, followed by 1, 2, 3, and so on. Memory addresses act just like the indexes of a normal array. The computer can access any address in memory at any time (hence the name "random access memory"). It can also group bytes together as it needs to to form larger variables, arrays, and structures. For example, a floating point variable consumes 4 contiguous bytes in memory. You might make the following global declaration in a program:

```
float f;
```

This statement says, "Declare a location named **f** that can hold one floating point value." When the program runs, the computer reserves space for the variable **f** somewhere in memory. That location has a fixed address in the memory space, like this:




The variable f consumes four bytes of RAM in memory.
That location has a specific address, in this case 248,440.

While you think of the variable **f**, the computer thinks of a specific address in memory (for example, 248,440). Therefore, when you create a statement like this:

```
f = 3.14;
```

The compiler might translate that into, "Load the value 3.14 into memory location 248,440." The computer is always thinking of memory in terms of addresses and values at those addresses.

There are, by the way, several interesting side effects to the way your computer treats memory. For example, say that you include the following code in one of your programs:

```
int i, s[4], t[4], u=0;

for (i=0; i<=4; i++)
{
        s[i] = i;
        t[i] =i;
}
printf("s:t\n");
for (i=0; i<=4; i++)
        printf("%d:%d\n", s[i], t[i]);
printf("u = %d\n", u);
```

The output that you see from the program will probably look like this:

```
s:t
1:5
2:2
3:3
4:4
5:5
u = 5
```

Why are **t[0]** and **u** incorrect? If you look carefully at the code, you can see that the for loops are writing one element past the end of each array. In memory, the arrays are placed adjacent to one another, as shown here:


Therefore, when you try to write to s[4], which does not exist, the system writes into t[0] instead because t[0] is where s[4] ought to be. When you write into t[4], you are really writing into u. As far as the computer is concerned, s[4] is simply an address, and it can write into it. As you can see however, even though the computer executes the program, it is not correct or valid. The program corrupts the array t in the process of running. If you execute the following statement, more severe consequences result:

```
s[1000000] = 5;
```

The location **s[1000000]** is more than likely outside of your program's memory space. In other words, you are writing into memory that your program does not own. On a system with protected memory spaces (UNIX, Windows 98/NT), this sort of statement will cause the system to terminate execution of the program. On other systems (Windows 3.1, the Mac), however, the system is not aware of what you are doing. You end up damaging the code or variables in another application. The effect of the violation can range from nothing at all to a complete system crash. In memory, i, s, t and u are all placed next to one another at specific addresses. Therefore, if you write past the boundaries of a variable, the computer will do what you say but it will end up corrupting another memory location.

Because C and C++ do not perform any sort of range checking when you access an element of an array, it is essential that you, as a programmer, pay careful attention to **array ranges** yourself and keep within the array's appropriate boundaries. Unintentionally reading or writing outside of array boundaries always leads to faulty program behavior.

As another example, try the following:

```
#include <stdio.h>

int main()
{
    int i,j;
    int *p;   /* a pointer to an integer */
    printf("%d %d\n", p, &i);
    p = &i;
    printf("%d %d\n", p, &i);
    return 0;
}
```

This code tells the compiler to print out the address held *in* **p**, along with the address *of* **i.** The variable **p** starts off with some crazy value or with 0. The address of **i** is generally a large value. For example, when I ran this code, I received the following output:

```
0   2147478276
2147478276   2147478276
```

which means that the address of **i** is 2147478276. Once the statement **p = &i;** has been executed, **p** contains the address of **i**. Try this as well:

```
#include <stdio.h>

void main()
{
    int *p;   /* a pointer to an integer */

    printf("%d\n",*p);
}
```

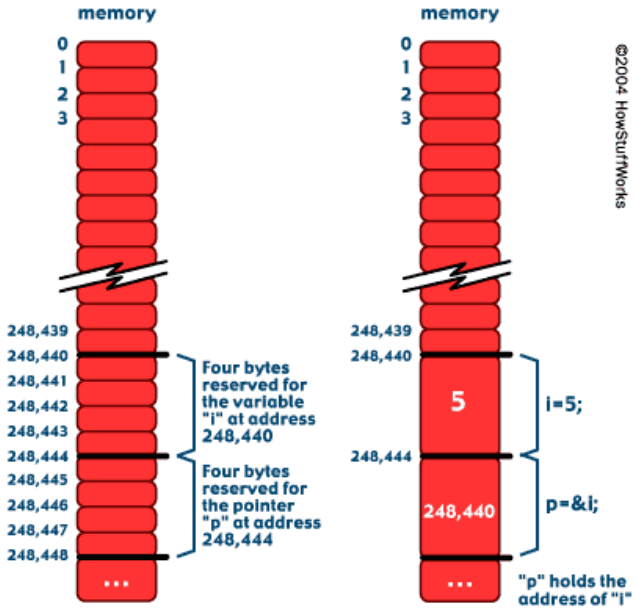
This code tells the compiler to print the value that **p** points to. However, **p** has not been initialized yet; it contains the address 0 or some random address. In most cases, a **segmentation fault** (or some other run-time error) results, which means that you have used a pointer that points to an invalid area of memory.

Almost always, an uninitialized pointer or a bad pointer address is the cause of segmentation faults.

Having said all of this, we can now look at pointers in a whole new light. Take this program, for example:

```
#include <stdio.h>

int main()
{
    int i;
    int *p;   /* a pointer to an integer */
    p = &i;
    *p=5;
    printf("%d %d\n", i, *p);
    return 0;
}
```

Here is what's happening:




The variable **i** consumes 4 bytes of memory. The pointer p also consumes 4 bytes (on most machines in use today, a pointer consumes 4 bytes of memory. Memory addresses are 32-bits long on most CPUs today, although there is a increasing trend toward 64-bit addressing). The location of i has a specific address, in this case 248,440. The pointer p holds that address once you say **p = &i;**. The variables ***p** and **i** are therefore equivalent.

The pointer **p** literally holds the address of **i**. When you say something like this in a program:

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

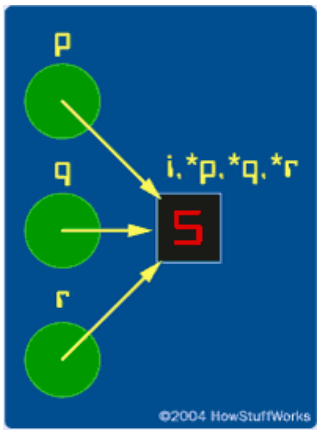
what comes out is the actual address of the variable **i**.

**Pointers: Pointing to the Same Address**

Here is a cool aspect of C: **Any number of pointers can point to the same address**. For example, you could declare p, q, and r as integer pointers and set all of them to point to i, as shown here:

```
int i;
int *p, *q, *r;

p = &i;
q = &i;
r = p;
```

Note that in this code, r points to the same thing that p points to, which is **i**. You can assign pointers to one another, and the address is copied from the right-hand side to the left-hand side during the assignment. After executing the above code, this is how things would look:




The variable i now has four names: i, *p, *q and *r. There is no limit on the number of pointers that can hold (and therefore point to) the same address.

**Pointers: Common Bugs**

**Bug #1 - Uninitialized pointers**
One of the easiest ways to create a pointer bug is to try to reference the value of a pointer even though the pointer is uninitialized and does not yet point to a valid address. For example:

```
int *p;

*p = 12;
```

The pointer **p** is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say ***p=12;**, the program will simply try to write a 12 to whatever random location **p** points to. The program may explode immediately, or may wait half an hour and then explode, or it may subtly corrupt data in another part of your program and you may never realize it. This can make this error very hard to track down. Make sure you initialize all pointers to a valid address before dereferencing them.

**Bug #2 - Invalid Pointer References**
An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block.

One way to create this error is to say **p=q;**, when **q** is uninitialized. The pointer p will then become uninitialized as well, and any reference to *p is an invalid pointer reference.

The only way to avoid this bug is to draw pictures of each step of the program and make sure that all pointers point somewhere. Invalid pointer references cause a program to crash inexplicably for the same reasons given in Bug #1.

**Bug #3 - Zero Pointer Reference**
A zero pointer reference occurs whenever a pointer pointing to zero is used in a statement that attempts to reference a block. For example, if **p** is a pointer to an integer, the following code is invalid:

```
p = 0;
*p = 12;
```

There is no block pointed to by p. Therefore, trying to read or write anything from or to that block is an invalid zero pointer reference. There are good, valid reasons to point a pointer to zero, as we will see in later articles. Dereferencing such a pointer, however, is invalid.

All of these bugs are fatal to a program that contains them. You must watch your code so that these bugs do not occur. The best way to do that is to draw pictures of the code's execution step by step.

**Using Pointers for Function Parameters**

Most C programmers first use pointers to implement something called **variable parameters** in functions. You have actually been using variable parameters in the scanf function -- that's why you've had to use the **&** (the address operator) on variables used with scanf. Now that you understand pointers you can see what has really been going on.

To understand how variable parameters work, lets see how we might go about implementing a **swap** function in C. To implement a swap function, what you would like to do is pass in two variables and have the function swap their values. Here's one attempt at an implementation -- enter and execute the following code and see what happens:

```
#include <stdio.h>

void swap(int i, int j)
{
    int t;

    t=i;
    i=j;
```

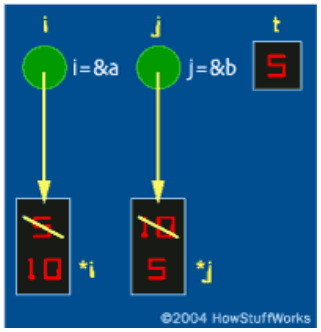
```
    j=t;
}

void main()
{
    int a,b;

    a=5;
    b=10;
    printf("%d %d\n", a, b);
    swap(a,b);
    printf("%d %d\n", a, b);
}
```

When you execute this program, you will find that no swapping takes place. The values of **a** and **b** are passed to swap, and the swap function does swap them, but when the function returns nothing happens.

To make this function work correctly you can use pointers, as shown below:

```
#include <stdio.h>

void swap(int *i, int *j)
{
    int t;
    t = *i;
    *i = *j;
    *j = t;
}

void main()
{
    int a,b;
    a=5;
    b=10;
    printf("%d %d\n",a,b);
    swap(&a,&b);
    printf("%d %d\n",a,b);
}
```

To get an idea of what this code does, print it out, draw the two integers **a** and **b**, and enter 5 and 10 in them. Now draw the two pointers **i** and **j**, along with the integer **t**. When **swap** is called, it is passed the *addresses* of **a** and **b**. Thus, **i** points to **a** (draw an arrow from **i** to **a**) and **j** points to **b** (draw another arrow from **b** to **j**). Once the pointers are initialized by the function call, ***i** is another name for **a**, and ***j** is another name for **b**. Now run the code in **swap**. When the code uses ***i** and ***j**, it really means **a** and **b**. When the function completes, a and b have been swapped.




Suppose you accidentally forget the **&** when the **swap** function is called, and that the **swap** line accidentally looks like this: **swap(a, b);**. This causes a segmentation fault. When you leave out the **&**, the *value* of **a** is passed instead of its *address*. Therefore, **i** points to an invalid location in memory and the system crashes when ***i** is used.

This is also why **scanf** crashes if you forget the **&** on variables passed to it. The **scanf** function is using pointers to put the value it reads back into the variable you have passed. Without the **&**, **scanf** is passed a bad address and crashes.

Variable parameters are one of the most common uses of pointers in C. Now you understand what's happening!

**Dynamic Data Structures**

Dynamic data structures are data structures that grow and shrink as you need them to by allocating and deallocating memory from a place called **the heap**. They are extremely important in C because they allow the programmer to exactly control memory consumption.

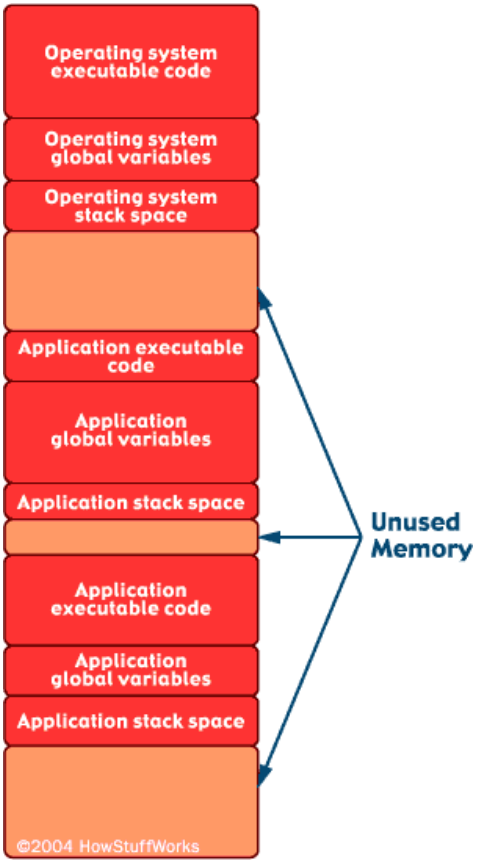
Dynamic data structures allocate blocks of memory from the heap as required, and link those blocks together into some kind of data structure using pointers. When the data structure no longer needs a block of memory, it will return the block to the heap for reuse. This recycling makes very efficient use of memory.

To understand dynamic data structures completely, we need to start with the heap.

**Dynamic Data Structures: The Heap**

A typical personal computer or workstation today has somewhere between 16 and 64 megabytes of RAM installed. Using a technique called **virtual memory**, the system can swap pieces of memory on and off the machine's hard disk to create an illusion for the CPU that it has much more memory, for example 200 to 500 megabytes. While this illusion is complete as far as the CPU is concerned, it can sometimes slow things down tremendously from the user's perspective. Despite this drawback, virtual memory is an extremely useful technique for "increasing" the amount of RAM in a machine in an inexpensive way. Let's assume for the sake of this discussion that a typical computer has a total memory space of, for example, 50 megabytes (regardless of whether that memory is implemented in real RAM or in virtual memory).

The operating system on the machine is in charge of the 50-megabyte memory space. The operating system uses the space in several different ways, as shown here:




**The operating system and several applications, along with their global variables and stack spaces, all consume portions of memory. When a program completes execution, it releases its memory for reuse by other programs. Note that part of the memory space remains unused at any given time.**

This is, of course, an idealization, but the basic principles are correct. As you can see, memory holds the executable code for the different applications currently running on the machine, along with the executable code for the operating system itself. Each application has certain global variables associated with it. These variables also consume memory. Finally, each application uses an area of memory called **the stack**, which holds all local variables and parameters used by any function. The stack also remembers the order in which functions are called so that function returns occur correctly. Each time a function is called, its local variables and parameters are "pushed onto" the stack. When the function returns, these locals and parameters are "popped." Because of this, the size of a program's stack fluctuates constantly as the program is running, but it has some maximum size.

As a program finishes execution, the operating system unloads it, its globals and its stack space from memory. A new program can make use of that space at a later time. In this way, the memory in a computer system is constantly "recycled" and reused by programs as they execute and complete.

In general, perhaps 50 percent of the computer's total memory space might be unused at any given moment. The operating system owns and manages the unused memory, and it is collectively known as **the heap**. The heap is extremely important because it is available for use by applications during execution using the C functions **malloc** (memory allocate) and **free**. The heap allows programs to allocate memory exactly when they need it during the execution of a program, rather than pre-allocating it with a specifically-sized array declaration.

**Dynamic Data Structures: Malloc and Free**

Let's say that you would like to allocate a certain amount of memory during the execution of your application. You can call the malloc function at any time, and it will request a block of memory from the heap. The operating system will reserve a block of memory for your program, and you can use it in any way you like. When you are done with the block, you return it to the operating system for recycling by calling the free function. Then other applications can reserve it later for their own use.

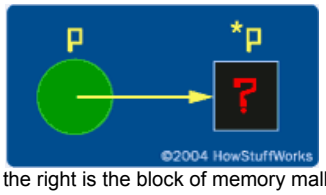
For example, the following code demonstrates the simplest possible use of the heap:

```
int main()
{
        int *p;

        p = (int *)malloc(sizeof(int));
        if (p == 0)
        {
                printf("ERROR: Out of memory\n");
                return 1;
        }
        *p = 5;
        printf("%d\n", *p);
        free(p);
        return 0;
}
```

The first line in this program calls the malloc function. This function does three things:

1. The malloc statement first looks at the amount of memory available on the heap and asks, "Is there enough memory available to allocate a block of memory of the size requested?" The amount of memory needed for the block is known from the parameter passed into malloc -- in this case, **sizeof(int)** is 4 bytes. If there is not enough memory available, the malloc function returns the address zero to indicate the error (another name for zero is NULL and you will see it used throughout C code). Otherwise malloc proceeds.
2. If memory is available on the heap, the system "allocates" or "reserves" a block from the heap of the size specified. The system reserves the block of memory so that it isn't accidentally used by more than one malloc statement.
3. The system then places into the pointer variable (p, in this case) the address of the reserved block. The pointer variable itself contains an address. The allocated block is able to hold a value of the type specified, and the pointer points to it.

The following diagram shows the state of memory after calling malloc:




The block on the right is the block of memory malloc allocated.

The program next checks the pointer p to make sure that the allocation request succeeded with the line **if (p == 0)** (which could have also been written as **if (p == NULL)** or even **if (!p)**. If the allocation fails (if p is zero), the program terminates. If the allocation is successful, the program then initializes the block to the value 5, prints out the value, and calls the free function to return the memory to the heap before the program terminates.

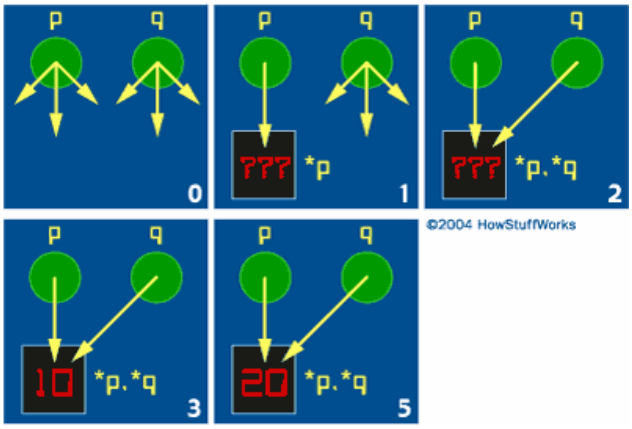
There is really no difference between this code and previous code that sets p equal to the address of an existing integer i. The only distinction is that, in the case of the variable i, the memory existed as part of the program's pre-allocated memory space and had the two names: **i** and ***p**. In the case of memory allocated from the heap, the block has the single name ***p** and is allocated during the program's execution. Two common questions:

- **Is it really important to check that the pointer is zero after each allocation?** Yes. Since the heap varies in size constantly depending on which programs are running, how much memory they have allocated, etc., there is never any guarantee that a call to malloc will succeed. You should check the pointer after any call to malloc to make sure the pointer is valid.
- **What happens if I forget to delete a block of memory before the program terminates?** When a program terminates, the operating system "cleans up after it," releasing its executable code space, stack, global memory space and any heap allocations for recycling. Therefore, there are no long-term consequences to leaving allocations pending at program termination. However, it is considered bad form, and "memory leaks" during the execution of a program are harmful, as discussed below.

The following two programs show two different valid uses of pointers, and try to distinguish between the use of a pointer and of the pointer's value:

```
void main()
{
        int *p, *q;

        p = (int *)malloc(sizeof(int));
        q = p;
        *p = 10;
        printf("%d\n", *q);
        *q = 20;
        printf("%d\n", *q);
}
```

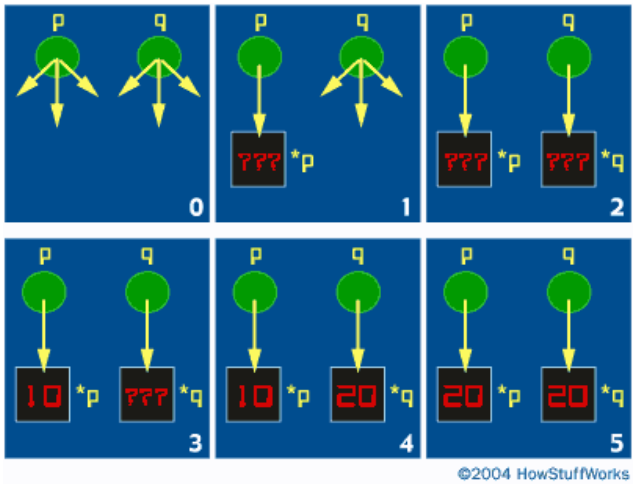
The final output of this code would be 10 from line 4 and 20 from line 6. Here's a diagram:




The following code is slightly different:

```
void main()
{
        int *p, *q;

        p = (int *)malloc(sizeof(int));
        q = (int *)malloc(sizeof(int));
        *p = 10;
        *q = 20;
        *p = *q;
        printf("%d\n", *p);
}
```

The final output from this code would be 20 from line 6. Here's a diagram:




Notice that the compiler will allow ***p = *q**, because *p and *q are both integers. This statement says, "Move the integer value pointed to by q into the integer value pointed to by p." The statement moves the values. The compiler will also allow **p = q**, because p and q are both pointers, and both point to the same type (if s is a pointer to a character, **p = s** is not allowed because they point to different types). The statement **p = q** says, "Point p to the same block q points to." In other words, the address pointed to by q is moved into p, so they both point to the same block. This statement moves the addresses.

From all of these examples, you can see that there are four different ways to initialize a pointer. When a pointer is declared, as in **int *p**, it starts out in the program in an uninitialized state. It may point anywhere, and therefore to dereference it is an error. Initialization of a pointer variable involves pointing it to a known location in memory.

1. One way, as seen already, is to use the malloc statement. This statement allocates a block of memory from the heap and then points the pointer at the block. This initializes the pointer, because it now points to a known location. The pointer is initialized because it has been filled with a valid address -- the address of the new block.
2. The second way, as seen just a moment ago, is to use a statement such as **p = q** so that p points to the same place as q. If q is pointing at a valid block, then p is initialized. The pointer p is loaded with the valid address that q contains. However, if q is uninitialized or invalid, p will pick up the same useless address.
3. The third way is to point the pointer to a known address, such as a global variable's address. For example, if i is an integer and p is a pointer to an integer, then the statement **p=&i** initializes p by pointing it to i.

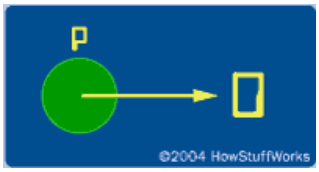
4. The fourth way to initialize the pointer is to use the value zero. Zero is a special values used with pointers, as shown here:

   ```
   p = 0;
   ```

   or:

   ```
   p = NULL;
   ```

   What this does physically is to place a zero into p. The pointer p's address is zero. This is normally diagrammed as:




Any pointer can be set to point to zero. When p points to zero, however, it does not point to a block. The pointer simply contains the address zero, and this value is useful as a tag. You can use it in statements such as:

```
if (p == 0)
{
        ...
}
```

or:

```
while (p != 0)
{
        ...
}
```

The system also recognizes the zero value, and will generate error messages if you happen to dereference a zero pointer. For example, in the following code:

```
p = 0;
*p = 5;
```

The program will normally crash. The pointer p does not point to a block, it points to zero, so a value cannot be assigned to *p. The zero pointer will be used as a flag when we get to linked lists.

The malloc command is used to allocate a block of memory. It is also possible to deallocate a block of memory when it is no longer needed. When a block is deallocated, it can be reused by a subsequent malloc command, which allows the system to recycle memory. The command used to deallocate memory is called **free**, and it accepts a pointer as its parameter. The free command does two things:

1. The block of memory pointed to by the pointer is unreserved and given back to the free memory on the heap. It can then be reused by later new statements.
2. The pointer is left in an uninitialized state, and must be reinitialized before it can be used again.

The free statement simply returns a pointer to its original uninitialized state and makes the block available again on the heap.

The following example shows how to use the heap. It allocates an integer block, fills it, writes it, and disposes of it:

```
#include <stdio.h>

int main()
{
    int *p;
    p = (int *)malloc (sizeof(int));
    *p=10;
    printf("%d\n",*p);
    free(p);
    return 0;
}
```

This code is really useful only for demonstrating the process of allocating, deallocating, and using a block in C. The **malloc** line allocates a block of memory of the size specified -- in this case, **sizeof(int)** bytes (4 bytes). The **sizeof** command in C returns the size, in bytes, of any type. The code could just as easily have said **malloc(4)**, since **sizeof(int)** equals 4 bytes on most machines. Using **sizeof**, however, makes the code much more portable and readable.

The **malloc** function returns a pointer to the allocated block. This pointer is generic. Using the pointer without typecasting generally produces a type warning from the compiler. The **(int *)** typecast converts the generic pointer returned by malloc into a "pointer to an integer," which is what **p** expects. The free statement in C returns a block to the heap for reuse.

The second example illustrates the same functions as the previous example, but it uses a structure instead of an integer. In C, the code looks like this:

```
#include <stdio.h>
```

```
struct rec
{
    int i;
    float f;
    char c;
};

int main()
{
    struct rec *p;
    p=(struct rec *) malloc (sizeof(struct rec));
    (*p).i=10;
    (*p).f=3.14;
    (*p).c='a';
    printf("%d %f %c\n",(*p).i,(*p).f,(*p).c);
    free(p);
    return 0;
}
```

Note the following line:

```
(*p).i=10;
```

Many wonder why the following doesn't work:

```
*p.i=10;
```

The answer has to do with the precedence of operators in C. The result of the calculation 5+3*4 is 17, not 32, because the * operator has higher precedence than + in most computer languages. In C, the **.** operator has higher precedence than *, so parentheses force the proper precedence.

Most people tire of typing **(*p).i** all the time, so C provides a shorthand notation. The following two statements are exactly equivalent, but the second is easier to type:

```
(*p).i=10;
p->i=10;
```

You will see the second more often than the first when reading other people's code.

**Advanced Pointers**

You will normally use pointers in somewhat more complicated ways than those shown in some of the previous examples. For example, it is much easier to create a normal integer and work with it than it is to create and use a pointer to an integer. In this section, some of the more common and advanced ways of working with pointers will be explored.

## Pointer Types

It is possible, legal, and beneficial to create pointer types in C, as shown below:

```
typedef int *IntPointer;
...
IntPointer p;
```

This is the same as saying:

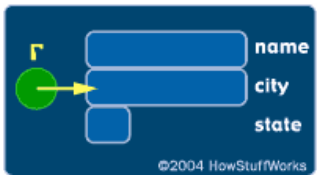
```
int *p;
```

This technique will be used in many of the examples on the following pages. The technique often makes a data declaration easier to read and understand, and also makes it easier to include pointers inside of structures or pass pointer parameters in functions.

**Pointers to Structures**

It is possible to create a pointer to almost any type in C, including user-defined types. It is extremely common to create pointers to structures. An example is shown below:

```
typedef struct
{
        char name[21];
        char city[21];
        char state[3];
} Rec;
typedef Rec *RecPointer;

RecPointer r;
r = (RecPointer)malloc(sizeof(Rec));
```




The pointer r is a pointer to a structure. Please note the fact that r is a pointer, and therefore takes four bytes of memory just like any other pointer. However, the malloc statement allocates 45 bytes of memory from the heap. ***r** is a structure just like any other structure of type **Rec**. The following code shows typical uses of the pointer variable:

```
strcpy((*r).name, "Leigh");
strcpy((*r).city, "Raleigh");
strcpy((*r).state, "NC");
printf("%s\n", (*r).city);
free(r);
```

You deal with *r just like a normal structure variable, but you have to be careful with the precedence of operators in C. If you were to leave off the parenthesis around *r the code would not compile because the "." operator has a higher precedence than the "*" operator. Because it gets tedious to type so many parentheses when working with pointers to structures, C includes a shorthand notation that does exactly the same thing:

```
strcpy(r->name, "Leigh");
```

The **r->** notation is exactly equivalent to **(*r).**, but takes two fewer characters.

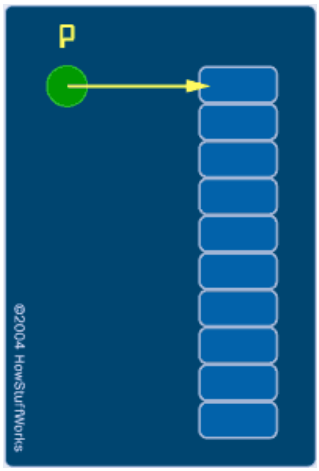
### Pointers to Arrays

It is also possible to create pointers to arrays, as shown below:

```
int *p;
int i;

p = (int *)malloc(sizeof(int[10]));
for (i=0; i<10; i++)
        p[i] = 0;
free(p);
```

or:

```
int *p;
int i;

p = (int *)malloc(sizeof(int[10]));
for (i=0; i<10; i++)
        *(p+i) = 0;
free(p);
```




Note that when you create a pointer to an integer array, you simply create a normal pointer to int. The call to malloc allocates an array of whatever size you desire, and the pointer points to that array's first element. You can either index through the array pointed to by p using normal array indexing, or you can do it using pointer arithmetic. C sees both forms as equivalent.

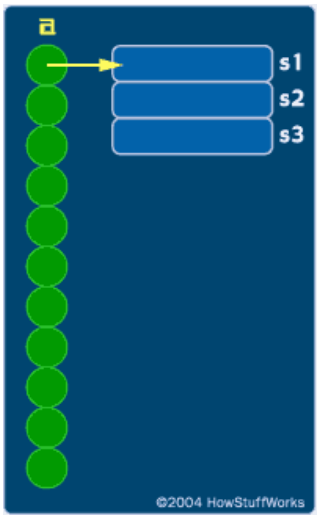
This particular technique is extremely useful when working with strings. It lets you allocate enough storage to exactly hold a string of a particular size.

### Arrays of Pointers

Sometimes a great deal of space can be saved, or certain memory-intensive problems can be solved, by declaring an array of pointers. In the example code below, an array of 10 pointers to structures is declared, instead of declaring an array of structures. If an array of the structures had been created instead, 243 *

10 = 2,430 bytes would have been required for the array. Using the array of pointers allows the array to take up minimal space until the actual records are allocated with malloc statements. The code below simply allocates one record, places a value in it, and disposes of the record to demonstrate the process:

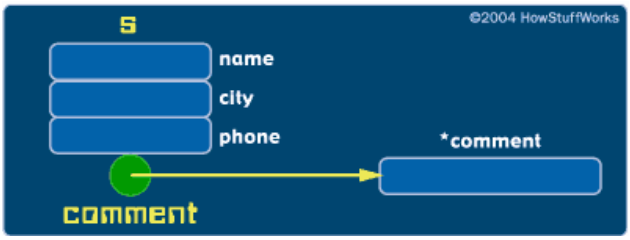
```
typedef struct
{
        char s1[81];
        char s2[81];
        char s3[81];
} Rec;
Rec *a[10];

a[0] = (Rec *)malloc(sizeof(Rec));
strcpy(a[0]->s1, "hello");
free(a[0]);
```




**Structures Containing Pointers**

Structures can contain pointers, as shown below:

```
typedef struct
{
        char name[21];
        char city[21];
        char phone[21];
        char *comment;
} Addr;
Addr s;
char comm[100];

gets(s.name, 20);
gets(s.city, 20);
gets(s.phone, 20);
gets(comm, 100);
s.comment =
(char *)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s.comment, comm);
```



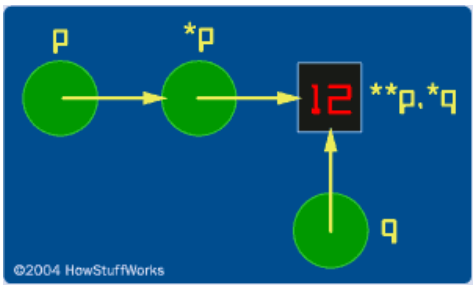

This technique is useful when only some records actually contained a comment in the comment field. If there is no comment for the record, then the comment field would consist only of a pointer (4 bytes). Those records having a comment then allocate exactly enough space to hold the comment string, based on the length of the string typed by the user.

**Pointers to Pointers**

It is possible and often useful to create pointers to pointers. This technique is sometimes called a **handle**, and is useful in certain situations where the

operating system wants to be able to move blocks of memory on the heap around at its discretion. The following example demonstrates a pointer to a pointer:

```
int **p;
int *q;

p = (int **)malloc(sizeof(int *));
*p = (int *)malloc(sizeof(int));
**p = 12;
q = *p;
printf("%d\n", *q);
free(q);
free(p);
```



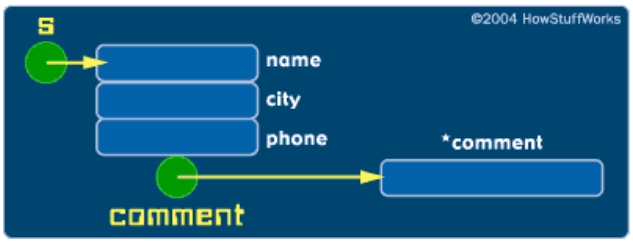

Windows and the Mac OS use this structure to allow memory compaction on the heap. The program manages the pointer p, while the operating system manages the pointer *p. Because the OS manages *p, the block pointed to by *p (**p**) can be moved, and *p can be changed to reflect the move without affecting the program using p. Pointers to pointers are also frequently used in C to handle pointer parameters in functions.

### Pointers to Structures Containing Pointers

It is also possible to create pointers to structures that contain pointers. The following example uses the **Addr** record from the previous section:

```
typedef struct
{
        char name[21];
        char city[21];
        char phone[21];
        char *comment;
} Addr;
Addr *s;
char comm[100];

s = (Addr *)malloc(sizeof(Addr));
gets(s->name, 20);
gets(s->city, 20);
gets( s->phone, 20);
gets(comm, 100);
s->comment =
(char *)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s->comment, comm);
```

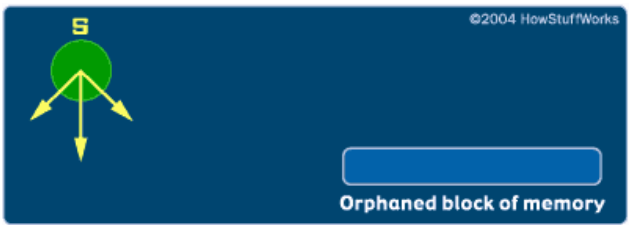
The pointer s points to a structure that contains a pointer that points to a string:




In this example, it is very easy to create lost blocks if you aren't careful. For example, here is a different version of the AP example.

```
s = (Addr *)malloc(sizeof(Addr));
gets(comm, 100);
s->comment =
(char *)malloc(sizeof(char[strlen(comm)+1]));
strcpy(s->comment, comm);
free(s);
```

This code creates a lost block because the structure containing the pointer pointing to the string is disposed of before the string block is disposed of, as shown below:

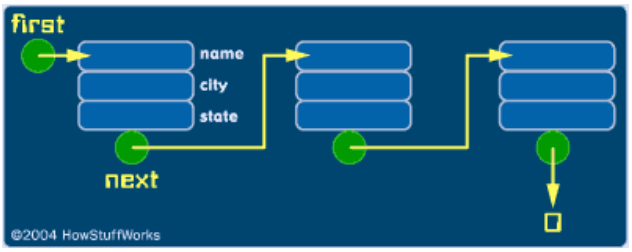



### Linking

Finally, it is possible to create structures that are able to point to identical structures, and this capability can be used to link together a whole string of identical records in a structure called a linked list.

```
typedef struct
{
        char name[21];
        char city[21];
        char state[21];
        Addr *next;
} Addr;
Addr *first;
```

The compiler will let you do this, and it can be used with a little experience to create structures like the one shown below:




**A Linked Stack Example**

A good example of dynamic data structures is a simple stack library, one that uses a dynamic list and includes functions to init, clear, push, and pop. The library's header file looks like this:

```
/* Stack Library - This library offers the
   minimal stack operations for a
   stack of integers (easily changeable) */

typedef int stack_data;

extern void stack_init();
/* Initializes this library.
   Call first before calling anything. */

extern void stack_clear();
/* Clears the stack of all entries. */

extern int stack_empty();
/* Returns 1 if the stack is empty, 0 otherwise. */

extern void stack_push(stack_data d);
/* Pushes the value d onto the stack. */

extern stack_data stack_pop();
/* Returns the top element of the stack,
   and removes that element.
   Returns garbage if the stack is empty. */
```

The library's code file follows:

```
#include "stack.h"
#include <stdio.h>

/* Stack Library - This library offers the
   minimal stack operations for a stack of integers */

struct stack_rec
{
    stack_data data;
    struct stack_rec *next;
```

```
};

struct stack_rec *top=NULL;

void stack_init()
/* Initializes this library.
   Call before calling anything else. */
{
    top=NULL;
}

void stack_clear()
/* Clears the stack of all entries. */
{
    stack_data x;

    while (!stack_empty())
    x=stack_pop();
}

int stack_empty()
/* Returns 1 if the stack is empty, 0 otherwise. */
{
    if (top==NULL)
        return(1);
    else
        return(0);
}

void stack_push(stack_data d)
/* Pushes the value d onto the stack. */
{
    struct stack_rec *temp;
    temp=
  (struct stack_rec *)malloc(sizeof(struct stack_rec));
    temp->data=d;
    temp->next=top;
    top=temp;
}

stack_data stack_pop()
/* Returns the top element of the stack,
   and removes that element.
   Returns garbage if the stack is empty. */
{
    struct stack_rec *temp;
    stack_data d=0;
    if (top!=NULL)
    {
        d=top->data;
        temp=top;
        top=top->next;
        free(temp);
    }
    return(d);
}
```

Note how this library practices information hiding: Someone who can see only the header file cannot tell if the stack is implemented with arrays, pointers, files, or in some other way. Note also that C uses **NULL**. NULL is defined in **stdio.h**, so you will almost always have to include **stdio.h** when you use pointers. NULL is the same as zero.

### Try This!

- Add a **dup**, a **count**, and an **add** function to the stack library to duplicate the top element of the stack, return a count of the number of elements in the stack, and add the top two elements in the stack.
- Build a driver program and a makefile, and compile the stack library with the driver to make sure it works.

### C Errors to Avoid

- Forgetting to include parentheses when you reference a record, as in **(*p).i** above
- Failing to dispose of any block you allocate - For example, you should not say **top=NULL** in the stack function, because that action orphans blocks that need to be disposed.
- Forgetting to include **stdio.h** with any pointer operations so that you have access to NULL.

**Using Pointers with Arrays**

Arrays and pointers are intimately linked in C. To use arrays effectively, you have to know how to use pointers with them. Fully understanding the relationship between the two probably requires several days of study and experimentation, but it is well worth the effort.

Let's start with a simple example of arrays in C:

```
#define MAX 10

int main()
{
    int a[MAX];
    int b[MAX];
    int i;
    for(i=0; i<MAX; i++)
        a[i]=i;
    b=a;
    return 0;
}
```

Enter this code and try to compile it. You will find that C will not compile it. If you want to copy **a** into **b**, you have to enter something like the following instead:

```
for (i=0; i<MAX; i++)
    b[i]=a[i];
```

Or, to put it more succinctly:

```
for (i=0; i<MAX; b[i]=a[i], i++);
```

Better yet, use the **memcpy** utility in **string.h**.

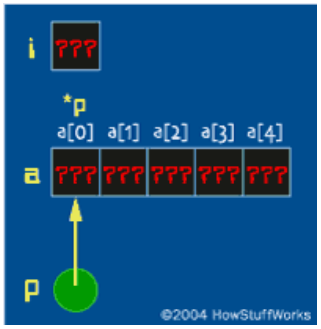
Arrays in C are unusual in that variables **a** and **b** are not, technically, arrays themselves. Instead they are permanent pointers to arrays. **a** and **b** permanently point to the first elements of their respective arrays -- they hold the addresses of **a[0]** and **b[0]** respectively. Since they are **permanent** pointers you cannot change their addresses. The statement **a=b;** therefore does not work.

Because **a** and **b** are pointers, you can do several interesting things with pointers and arrays. For example, the following code works:

```
#define MAX 10

void main()
{
    int a[MAX];
    int i;
    int *p;

    p=a;
    for(i=0; i<MAX; i++)
        a[i]=i;
    printf("%d\n",*p);
}
```

The statement **p=a;** works because **a** is a pointer. Technically, **a** points to the address of the 0th element of the actual array. This element is an integer, so **a** is a pointer to a single integer. Therefore, declaring **p** as a pointer to an integer and setting it equal to **a** works. Another way to say exactly the same thing would be to replace **p=a;** with **p=&a[0];**. Since **a** contains the address of **a[0]**, **a** and **&a[0]** mean the same thing.

The following figure shows the state of the variables right before the for loop starts executing:




Now that **p** is pointing at the 0th element of **a**, you can do some rather strange things with it. The **a** variable is a permanent pointer and can not be changed, but **p** is not subject to such restrictions. C actually encourages you to move it around using *pointer arithmetic* . For example, if you say **p++;**, the compiler knows that **p** points to an integer, so this statement increments **p** the appropriate number of bytes to move it to the next element of the array. If **p** were pointing to an array of 100-byte-long structures, **p++;** would move **p** over by 100 bytes. C takes care of the details of element size.

You can copy the array **a** into **b** using pointers as well. The following code can replace **(for i=0; i<MAX; a[i]=b[i], i++);** :

```
p=a;
q=b;
for (i=0; i<MAX; i++)
{
    *q = *p;
    q++;
    p++;
}
```

You can abbreviate this code as follows:

```
p=a;
q=b;
for (i=0; i<MAX; i++)
    *q++ = *p++;
```

And you can further abbreviate it to:

```
for (p=a,q=b,i=0; i<MAX; *q++ = *p++, i++);
```

What if you go beyond the end of the array **a** or **b** with the pointers **p** or **q**? C does not care -- it blithely goes along incrementing **p** and **q**, copying away over other variables with abandon. You need to be careful when indexing into arrays in C, because C assumes that you know what you are doing.

You can pass an array such as **a** or **b** to a function in two different ways. Imagine a function **dump** that accepts an array of integers as a parameter and prints the contents of the array to stdout. There are two ways to code **dump**:

```
void dump(int a[],int nia)
{
    int i;
     for (i=0; i<nia; i++)
        printf("%d\n",a[i]);
}
```

or:

```
void dump(int *p,int nia)
{
    int i;
     for (i=0; i<nia; i++)
        printf("%d\n",*p++);
}
```

The **nia** (number_in_array) variable is required so that the size of the array is known. Note that only a pointer to the array, rather than the contents of the array, is passed to the function. Also note that C functions can accept variable-size arrays as parameters.

**Strings**

Strings in C are intertwined with pointers to a large extent. You must become familiar with the pointer concepts covered in the previous articles to use C strings effectively. Once you get used to them, however, you can often perform string manipulations very efficiently.

A string in C is simply an array of characters. The following line declares an array that can hold a string of up to 99 characters.

```
char str[100];
```

It holds characters as you would expect: **str[0]** is the first character of the string, **str[1]** is the second character, and so on. But why is a 100-element array

unable to hold up to 100 characters? Because C uses *null-terminated strings*, which means that the end of any string is marked by the ASCII value 0 (the null character), which is also represented in C as **'\0'**.

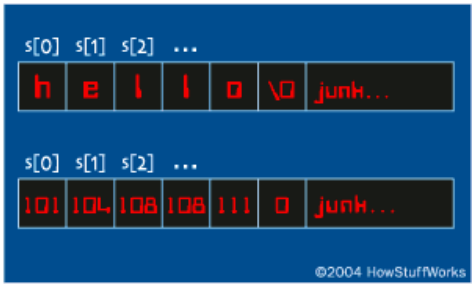
Null termination is very different from the way many other languages handle strings. For example, in Pascal, each string consists of an array of characters, with a length byte that keeps count of the number of characters stored in the array. This structure gives Pascal a definite advantage when you ask for the length of a string. Pascal can simply return the length byte, whereas C has to count the characters until it finds **'\0'**. This fact makes C much slower than Pascal in certain cases, but in others it makes it faster, as we will see in the examples below.

Because C provides no explicit support for strings in the language itself, all of the string-handling functions are implemented in libraries. The string I/0 operations (gets, puts, and so on) are implemented in <stdio.h>, and a set of fairly simple string manipulation functions are implemented in <string.h> (on some systems, <strings.h> ).

The fact that strings are not native to C forces you to create some fairly roundabout code. For example, suppose you want to assign one string to another string; that is, you want to copy the contents of one string to another. In C, as we saw in the last article, you cannot simply assign one array to another. You have to copy it element by element. The string library (<string.h> or <strings.h> ) contains a function called **strcpy** for this task. Here is an extremely common piece of code to find in a normal C program:

```
char s[100];
strcpy(s, "hello");
```

After these two lines execute, the following diagram shows the contents of s:




The top diagram shows the array with its characters. The bottom diagram shows the equivalent ASCII code values for the characters, and is how C actually thinks about the string (as an array of bytes containing integer values).

The following code shows how to use **strcpy** in C:

```
#include <string.h>
int main()
{
    char s1[100],s2[100];
    strcpy(s1,"hello"); /* copy "hello" into s1 */
    strcpy(s2,s1);      /* copy s1 into s2 */
    return 0;
}
```

**strcpy** is used whenever a string is initialized in C. You use the **strcmp** function in the string library to compare two strings. It returns an integer that indicates the result of the comparison. Zero means the two strings are equal, a negative value means that **s1** is less than **s2**, and a positive value means **s1** is greater than **s2**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[100],s2[100];
     gets(s1);
    gets(s2);
    if (strcmp(s1,s2)==0)
        printf("equal\n");
    else if (strcmp(s1,s2)<0)
        printf("s1 less than s2\n");
    else
        printf("s1 greater than s2\n");
    return 0;
}
```

Other common functions in the string library include **strlen**, which returns the length of a string, and **strcat** which concatenates two strings. The string library contains a number of other functions, which you can peruse by reading the man page.

To get you started building string functions, and to help you understand other programmers' code (everyone seems to have his or her own set of string functions for special purposes in a program), we will look at two examples, **strlen** and **strcpy**. Following is a strictly Pascal-like version of **strlen**:

```
int strlen(char s[])
```

```
{
    int x;
    x=0;
    while (s[x] != '\0')
         x=x+1;
    return(x);
}
```

Most C programmers shun this approach because it seems inefficient. Instead, they often use a pointer-based approach:

```
int strlen(char *s)
{
    int x=0;
    while (*s != '\0')
    {
        x++;
        s++;
    }
    return(x);
}
```

You can abbreviate this code to the following:

```
int strlen(char *s)
{
    int x=0;
    while (*s++)
        x++;
    return(x);
}
```

I imagine a true C expert could make this code even shorter.

When I compile these three pieces of code on a MicroVAX with gcc, using no optimization, and run each 20,000 times on a 120-character string, the first piece of code yields a time of 12.3 seconds, the second 12.3 seconds, and the third 12.9 seconds. What does this mean? To me, it means that you should write the code in whatever way is easiest for you to understand. Pointers generally yield faster code, but the **strlen** code above shows that that is not always the case.

We can go through the same evolution with **strcpy**:

```
strcpy(char s1[],char s2[])
{
    int x;
    for (x=0; x<=strlen(s2); x++)
        s1[x]=s2[x];
}
```

Note here that **<=** is important in the **for** loop because the code then copies the **'\0'**. Be sure to copy **'\0'**. Major bugs occur later on if you leave it out, because the string has no end and therefore an unknown length. Note also that this code is very inefficient, because **strlen** gets called every time through the **for** loop. To solve this problem, you could use the following code:

```
strcpy(char s1[],char s2[])
{
    int x,len;
     len=strlen(s2);
    for (x=0; x<=len; x++)
        s1[x]=s2[x];
}
```

The pointer version is similar.

```
strcpy(char *s1,char *s2)
{
    while (*s2 != '\0')
    {
        *s1 = *s2;
        s1++;
        s2++;
    }
}
```

You can compress this code further:

```
strcpy(char *s1,char *s2)
{
    while (*s2)
        *s1++ = *s2++;
}
```

If you wish, you can even say **while (*s1++ = *s2++);**. The first version of **strcpy** takes 415 seconds to copy a 120-character string 10,000 times, the second version takes 14.5 seconds, the third version 9.8 seconds, and the fourth 10.3 seconds. As you can see, pointers provide a significant performance boost here.

The prototype for the **strcpy** function in the string library indicates that it is designed to return a pointer to a string:

```
char *strcpy(char *s1,char *s2)
```

Most of the string functions return a string pointer as a result, and **strcpy** returns the value of **s1** as its result.

Using pointers with strings can sometimes result in definite improvements in speed and you can take advantage of these if you think about them a little. For example, suppose you want to remove the leading blanks from a string. You might be inclined to shift characters over on top of the blanks to remove them. In C, you can avoid the movement altogether:

```
#include <stdio.h>
#include <string.h>

int main()
{
    char s[100],*p;
     gets(s);
    p=s;
    while (*p==' ')
        p++;
    printf("%s\n",p);
    return 0;
}
```

This is much faster than the movement technique, especially for long strings.

You will pick up many other tricks with strings as you go along and read other code. Practice is the key.

**Special Note on Strings**

**Special Note on String Constants**
Suppose you create the following two code fragments and run them:

```
Fragment 1

{
    char *s;

    s="hello";
    printf("%s\n",s);
}

Fragment 2

{
    char s[100];

    strcpy(s,"hello");
    printf("%s\n",s);
}
```

These two fragments produce the same output, but their internal behavior is quite different. In fragment 2, you cannot say **s="hello";**. To understand the differences, you have to understand how the *string constant table* works in C.

When your program is compiled, the compiler forms the object code file, which contains your machine code and a table of all the string constants declared in the program. In fragment 1, the statement **s="hello";** causes **s** to point to the address of the string **hello** in the string constant table. Since this string is in the string constant table, and therefore technically a part of the executable code, you cannot modify it. You can only point to it and use it in a read-only manner.

In fragment 2, the string **hello** also exists in the constant table, so you can copy it into the array of characters named **s**. Since s is not a pointer, the statement **s="hello";** will not work in fragment 2. It will not even compile.

**Special Note on Using Strings with malloc**
Suppose you write the following program:

```
int main()
{
    char *s;

    s=(char *) malloc (100);
    s="hello";
    free(s);
    return 0;
```

```
}
```

It compiles properly, but gives a segmentation fault at the **free** line when you run it. The **malloc** line allocates a block 100 bytes long and points s at it, but now the **s="hello";** line is a problem. It is syntactically correct because s is a pointer; however, when **s="hello";** is executed, s points to the string in the string constant table and the allocated block is orphaned. Since s is pointing into the string constant table, the string cannot be changed; free fails because it cannot deallocate a block in an executable region.

The correct code follows:

```
int main()
{
    char *s;
    s=(char *) malloc (100);
    strcpy(s,"hello");
    free(s);
    return 0;
}
```

### Try This!

- Create a program that reads in a string containing a first name followed by a blank followed by a last name. Write functions to remove any leading or trailing blanks. Write another function that returns the last name.
- Write a function that converts a string to uppercase.
- Write a function that gets the first word from a string and returns the remainder of the string.

### C Error to Avoid

Losing the **\0** character, which is easy if you aren't careful, can lead to some very subtle bugs. Make sure you copy \0 when you copy strings. If you create a new string, make sure you put \0 in it. And if you copy one string to another, make sure the receiving string is big enough to hold the source string, including \0. Finally, if you point a character pointer to some characters, make sure they end with \0.

**Operator Precedence**

C contains many operators, and because of the way in which operator precedence works, the interactions between multiple operators can become confusing.

**x=5+3*6;**

X receives the value 23, not 48, because in C multiplication and division have higher precedence than addition and subtraction.

**char *a[10];**

Is **a** a single pointer to an array of 10 characters, or is it an array of 10 pointers to character? Unless you know the precedence conventions in C, there is no way to find out. Similarly, in E.11 we saw that because of precedence statements such as ***p.i = 10;** do not work. Instead, the form **(*p).i = 10;** must be used to force correct precedence.

The following table shows the precedence hierarchy in C. The top line has the highest precedence.

```
Operators                                   Associativity
( [ - .                                     Left to right
! - ++  -{-  + *  &  (type-cast)  sizeof    Right to left
(in the above line, +, - and * are the unary forms)
*  / %                                      Left to right
+  -                                        Left to right
<<  >>                                      Left to right
<  <=  >  >=                                Left to right
==  !=                                      Left to right
&                                           Left to right
^                                           Left to right
|                                           Left to right
&&                                          Left to right
||                                          Left to right
?:                                          Left to right
=  +=  -=  *=  /=  %=  &=  ^=  |=  <<= >>= Right to left
,                                           Left to right
```

Using this table, you can see that **char *a[10];** is an array of 10 pointers to character. You can also see why the parentheses are required if **(*p).i** is to be

handled correctly. After some practice, you will memorize most of this table, but every now and again something will not work because you have been caught by a subtle precedence problem.

**Command Line Arguments**

C provides a fairly simple mechanism for retrieving command line parameters entered by the user. It passes an **argv** parameter to the main function in the program. **argv** structures appear in a fair number of the more advanced library calls, so understanding them is useful to any C programmer.

Enter the following code and compile it:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int x;

    printf("%d\n",argc);
    for (x=0; x<argc; x++)
        printf("%s\n",argv[x]);
    return 0;
}
```

In this code, the main program accepts two parameters, argv and argc. The argv parameter is an array of pointers to string that contains the parameters entered when the program was invoked at the UNIX command line. The argc integer contains a count of the number of parameters. This particular piece of code types out the command line parameters. To try this, compile the code to an executable file named **aaa** and type **aaa xxx yyy zzz**. The code will print the command line parameters xxx, yyy and zzz, one per line.

The **char *argv[]** line is an array of pointers to string. In other words, each element of the array is a pointer, and each pointer points to a string (technically, to the first character of the string). Thus, **argv[0]** points to a string that contains the first parameter on the command line (the program's name), **argv[1]** points to the next parameter, and so on. The argc variable tells you how many of the pointers in the array are valid. You will find that the preceding code does nothing more than print each of the valid strings pointed to by argv.

Because argv exists, you can let your program react to command line parameters entered by the user fairly easily. For example, you might have your program detect the word **help** as the first parameter following the program name, and dump a help file to stdout. File names can also be passed in and used in your fopen statements.

**Binary Files**

Binary files are very similar to arrays of structures, except the structures are in a disk file rather than in an array in memory. Because the structures in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

Binary files have two features that distinguish them from text files:

- You can jump instantly to any structure in the file, which provides random access as in an array.
- You can change the contents of a structure anywhere in the file at any time.

Binary files also usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

C supports the file-of-structures concept very cleanly. Once you open the file you can read a structure, write a structure, or seek to any structure in the file. This file concept supports the concept of a **file pointer**. When the file is opened, the pointer points to record 0 (the first record in the file). Any **read operation** reads the currently pointed-to structure and moves the pointer down one structure. Any **write operation** writes to the currently pointed-to structure and moves the pointer down one structure. **Seek** moves the pointer to the requested record.

Keep in mind that C thinks of everything in the disk file as blocks of bytes read from disk into memory or read from memory onto disk. C uses a file pointer, but it can point to any byte location in the file. You therefore have to keep track of things.

The following program illustrates these concepts:

```
#include <stdio.h>

/* random record description - could be anything */
struct rec
{
    int x,y,z;
};

/* writes and then reads 10 arbitrary records
   from the file "junk". */
int main()
{
    int i,j;
    FILE *f;
    struct rec r;
```

```
    /* create the file of 10 records */
    f=fopen("junk","w");
    if (!f)
        return 1;
    for (i=1;i<=10; i++)
    {
        r.x=i;
        fwrite(&r,sizeof(struct rec),1,f);
    }
    fclose(f);

    /* read the 10 records */
    f=fopen("junk","r");
    if (!f)
        return 1;
    for (i=1;i<=10; i++)
    {
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
    }
    fclose(f);
    printf("\n");

    /* use fseek to read the 10 records
       in reverse order */
    f=fopen("junk","r");
    if (!f)
        return 1;
    for (i=9; i>=0; i--)
    {
        fseek(f,sizeof(struct rec)*i,SEEK_SET);
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
    }
    fclose(f);
    printf("\n");

    /* use fseek to read every other record */
    f=fopen("junk","r");
    if (!f)
        return 1;
    fseek(f,0,SEEK_SET);
    for (i=0;i<5; i++)
    {
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
        fseek(f,sizeof(struct rec),SEEK_CUR);
    }
    fclose(f);
    printf("\n");

    /* use fseek to read 4th record,
       change it, and write it back */
    f=fopen("junk","r+");
    if (!f)
        return 1;
    fseek(f,sizeof(struct rec)*3,SEEK_SET);
    fread(&r,sizeof(struct rec),1,f);
    r.x=100;
    fseek(f,sizeof(struct rec)*3,SEEK_SET);
    fwrite(&r,sizeof(struct rec),1,f);
    fclose(f);
    printf("\n");

    /* read the 10 records to insure
       4th record was changed */
    f=fopen("junk","r");
    if (!f)
        return 1;
    for (i=1;i<=10; i++)
    {
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
    }
    fclose(f);
    return 0;
}
```

In this program, a structure description **rec** has been used, but you can use any structure description you want. You can see that **fopen** and **fclose** work exactly as they did for text files.

The new functions here are **fread**, **fwrite** and **fseek**. The fread function takes four parameters:

- A memory address
- The number of bytes to read per block
- The number of blocks to read
- The file variable

Thus, the line **fread(&r,sizeof(struct rec),1,f);** says to read 12 bytes (the size of **rec**) from the file **f** (from the current location of the file pointer) into memory address **&r**. One block of 12 bytes is requested. It would be just as easy to read 100 blocks from disk into an array in memory by changing 1 to 100.

The **fwrite** function works the same way, but moves the block of bytes from memory to the file. The **fseek** function moves the file pointer to a byte in the file. Generally, you move the pointer in **sizeof(struct rec)** increments to keep the pointer at record boundaries. You can use three options when seeking:

- SEEK_SET
- SEEK_CUR
- SEEK_END

**SEEK_SET** moves the pointer **x** bytes down from the beginning of the file (from byte 0 in the file). **SEEK_CUR** moves the pointer **x** bytes down from the current pointer position. **SEEK_END** moves the pointer from the end of the file (so you must use negative offsets with this option).

Several different options appear in the code above. In particular, note the section where the file is opened with **r+** mode. This opens the file for reading and writing, which allows records to be changed. The code seeks to a record, reads it, and changes a field; it then seeks back because the read displaced the pointer, and writes the change back.