

Twenty Half Notes of C

by Jared Schiffman

edited by Caleb Clark

First edition, Copyright HTH Learning, 2003

Examples Overview

The Left Pages

One of the best ways to learn to program is to start with little sample programs, and then change them to make them do what you want.

On these pages (the ones on the left), you will find a series of example programs. Each of these programs is complete and has been tested. They are intended to be followed through in order.

How to Proceed

The first step, naturally, is to try and type the programs, compile them, and see if they actually work. There's a good chance that they won't compile the first time around...and probably not the second time either. When beginning to program in C, it is very easy to make little mistakes that will prevent your programs from compiling. (See Syntax page on the right side).

Once the program is typed, compiles correctly and actually runs, you may begin to think about how it actually works. Do you understand what is really going on? If you don't (and that's normal), you may read the explanations under each program. These explanations describe in depth how each program works.

The explanations beneath each example program make reference to the other set of pages (the ones on the right). These are the reference pages, and you will probably be referring to them a lot. In fact, the very first program, "Hello World", makes reference to five separate reference pages.

Good luck and Code speed!

Reference Overview

The Right Pages

One of the best ways to learn to program is to read about all the different parts that make up a programming language and find out how they work.

On these pages (the ones on the right), you will find detailed explanations of most of the basic concepts needed to program in C. They are intended to be either read through in order, or referred to in the midst of a program.

While you may be tempted to only pay attention to the example programs, it is a worthwhile endeavor to read about the specifics of the language.

For First Time Programmers

Before we start, if you've never programmed, here's the basic idea.

Every piece of software on a computer was programmed by a human. How? Programming uses letters, numbers and other keyboard characters in strange sequences to tell computers what to do and how to react when we type or use the mouse. Why? Computers speak a very hard language called machine code that is basically lines and lines of 0s and 1s, and it drives humans crazy to try and learn to write it. And since computers don't know English (yet), we can't just say "Program a 3-D game in Iceland with blue snowmobiles..." Thus we have a middle language, in this case C, that we share. They made C enough like English so we don't go completely insane writing it, but also close enough to computer language so we can communicate well. The computer meets us half way with a thing called a "compiler" that takes our C code and turns it into the 0s and 1s that the computer can process.

While at times it may feel very, very, very strange to write a bunch of nonsensical text, when you compile your code and run your program, you will see and feel how powerful and fun programming can be.

You are about to leave the world of "User" and become a "Maker".

Hello World

Traditionally, the first C program that everyone writes is this one. This came about because it is one of the shortest examples of a full program. Also, it says “hello” to you when it is created, giving new programmers a sense of the power of programming to create things that were not there before.

```
#include <stdio.h>           // includes input/output library

int main()                   // begins “main” function
{
    printf( "hello world\n" ); // prints “hello world”
    return 0;                // returns 0 from function
}
```

When typing this program, be careful to type all the punctuation (See Syntax). You do not need to type the comments.

The first line of the program **#include <stdio.h>** includes the Standard Input & Output library. This library of code contains the code for the “printf” function. (See Functions and Printf).

The second line **int main()** sets up the “main” function. Every C program contains a function named “main” and this one is no exception. The “int” specifies that this function will return an integer value—in this case 0. The empty parentheses are for any inputs that the function may have. In this case, there are no inputs. (See Functions and Variables)

The lines with ‘{’ and the ‘}’ define the boundaries of the “main” function. (See Syntax and Functions).

The heart of the program is in the line **printf("hello world\n");**. The printf function helps up display the text “hello world” on the command line. The “\n” is the equivalent of pressing “enter” or “return” on the keyboard. (See Statements, Functions and Printf).

The last line inside the “main” function is **return 0;**. This line is here on a technicality. Many compilers require that “main” functions return an integer. We return 0 by default. (See Functions).

Syntax

Syntax is the grammar of a programming language. Just like every spoken language has its own grammar, every programming language has its own syntax.

The most common error in any C program is a **syntax error**. And the most common form of syntax error is incorrect punctuation.

Punctuation

In C, almost every line of code ends in a **semicolon** ‘;’. *When in doubt, always end a line with a semicolon.* For example:

```
a = b + c;
```

The second most common form of punctuation are **parentheses** ‘(’ and ‘)’’. Parentheses are used mostly for functions (see Functions page). The key to using parentheses is that *for every left parenthesis... there must be a matching right parenthesis.* For example:

```
printf( "hello world\n" );
```

The third most important form of punctuation are **curly brackets** ‘{’ and ‘}’. Curly brackets (also called braces) are used to define multi-line blocks in C. A block is grouped set of lines of code. A function is a block. An if statement is a block. A while loop is a block, etc. *The first line of a block, interestingly, does not require a semicolon.* Once again, the key is: *for every left curly bracket... there must be a matching right curly bracket.* For example:

```
if ( a < 10 )
{
    b = b + 1;
}
```

There are many other forms of punctuation in C, but the three above are the most common. Every punctuation mark on the keyboard, except for @, #, and \$, is used at some point in the C syntax.

White Space

The syntax of C is very lenient when it comes to white space. White space includes spaces, tabs and blank lines. All white space is considered equivalent. A space can be replaced by ten spaces, a tab or even a whole blank line. You can also usually insert white space where there was none before, or even remove some white space and the program will still compile.

You could write a whole C program all on one continuous line, but it would be very hard to read. So, over the years programmers have come up with some standards for tabbing in and creating spaces between blocks of text, so they can read their own and other’s work easier. You’ll see this general idea first in the “Hello World” program and learn it over time just by reading lots of code.

Case Sensitivity

The C programming language is a “case sensitive” language. What this means is that *capitalization matters*. For example, “count” is not the same as “Count”. This is important when naming and using variables and functions.

Doctor, Doctor

This little program is good for seeing how basic variables work. It will print out variables of age, weight, height and blood type with values you define.

```
#include <stdio.h>

int main( )
{
    int age;                // declare an integer variables
    float height;          // declare a floating point variable
    char bloodType;        // declare a character variable

    age = 34;               // set age variable
    height = 68.5;          // set height variable
    bloodType = 'A';        // set bloodType variable

    printf( "You are %d years old.\n", age );    // prints age
    printf( "You are %f inches tall.\n", height ); // prints height
    printf( "You're blood type is %c.\n", bloodType ); // prints blood type
    return 0;
}
```

When typing this program, be careful to type all the punctuation (See Syntax). You do not need to type the comments.

Once again, we include the Standard Input & Output library. We also have a “main” function. (See Hello World, Functions).

This program has three distinct parts: variable declaration, variable assignment, and variable output (printing). (See Variables and Statements).

In the first section, three different types of variables are declared. In C, all variables must be declared at the top of the program and before being used. The age variable is stored as **int** (an integer number), the height is stored as a **float** (a real number with a decimal), and the blood type is stored as a **char** (a single character).

In the second section, each of the three variables is assigned a value. The age variable is assigned the integer value 34. The height variable is assigned the float value 68.5, and the blood type variable is assigned the value 'A'. (See Assignment in Statements).

In the final section of the program, the data is printed using the **printf** function. For each line that is printed, one of the variable values is inserted into the text. Notice the “%d” on the first printf line and the “age” variable at the end. When the program is run, printf will insert the age value where the %d is in the text. Likewise, the height will be inserted where the “%f” is on the next line. Finally, the blood type character will be inserted where the “%c” is on the next line. (See Printf).

The “%d” , “%f” and “%c” tell the printf function what type of variable to expect. For fun, try switching these conversion indicators with each other see what printf produces.

Variables

A variable can be thought of in many different ways. A variable can be a container in which values are stored. A variable can be a name for a little chunk of memory. A variable can be just a variable, like in algebra. Whichever makes sense, stick with that one.

There are several types of variables in C, each of which stores a different kind of value. Just like you wouldn't put a sandwich in a casserole dish, you wouldn't put an integer into a character variable.

The four most common types of variables are: int, float, char, and bool.

int

An ‘int’ variable stores an integer, which is a positive or negative number without any decimals. An int variable may be declared and used like this:

```
int A = 4;
```

float

A ‘float’ variable stores a floating point number, which is a positive or negative number with decimals. Floats can be very large or very small.

A float variable may be declared and used like this:

```
float A = 2.71828;
```

char

A ‘char’ variable stores a single character (anything on the keyboard).

A character variable may be declared and used like this:

```
char A = 'z';
```

bool

A ‘bool’ variable stores a single true or false value.

In fact, it can only store one of two values: true and false.

A bool variable may be declared and used like this:

```
bool A = true;
```

Variable Names

Variable names may be almost any combination of letters and numbers, as long as the first character is a letter. Names can be one to thirty-two characters long. Names cannot be the same as keywords in C like “for”, “if”, “while”, etc.

Unusual Variables

There are several other types of variables other than ints, floats, chars, and bools. Here is a list of a few others that you may encounter:

short	a short integer—stores a smaller total range of numbers
long	a long integer—stores a wider total range of numbers
double	a long float—stores a number with twice the precision
void	untyped—stores anything / nothing in particular

Mini-Calculator

This program implements a miniature calculator—that only does addition. The code for this mini-calculator includes a few basic statements in C.

```
#include <stdio.h>

int main( )
{
    int A, B, C;                // declare three int variables

    printf( "Enter the first number: " );    // print "enter first number"
    scanf( "%d", &A );                    // store user input in A

    printf( "Enter the second number: " );    // print "enter second number"
    scanf( "%d", &B );                    // store user input in B

    C = A + B;                    // store A+B in C
    printf( "A plus B = %d\n", C );        // print solution

    return 0;                    // return 0 from function
}
```

The basic idea behind this program is simple. Ask the user for two numbers. Add them together, and print the result. Let's go step by step.

Once again, we include the Standard Input & Output library. We also have a "main" function. (See Hello World, Functions).

On the first line of the function **int A, B, C;** three integer variables are declared. All variables in C must be declared at the top of the function before being used. (See Variables and Statements).

Now here is where the program really gets going. Printf is used to display the text "Enter the first number:". On the next line **scanf("%d", &A);** the number that the user entered is stored in the variable A. The "%d" indicates that the expected text will be an integer number. (as opposed to any other random text). The "&A" tells scanf where to store that number. (See Printf & Scanf).

The next two lines are essentially the same as the previous two lines. This time around, the second number is obtained from the user and stored in the variable B.

C = A + B; is where the computation takes place. This is an example of an assignment statement. The integer variables A and B are added together and the resulting sum is stored in the integer variable C. (See Statements).

Finally, the solution is printed for the user to see. Once again, the printf function is used in the line **printf("A plus B = %d\n", C);** but here things are a little different. Notice the "%d" and the variable C at the end. What printf does is insert the value of the variable C into the text in place of the "%d". (See Printf).

On your own, try to make this program add 3 numbers together. Then try to make it add the numbers and multiply them too.

Statements

A statement in C is just one line of code.

A statement is basically one "sentence" of C.

A statement always ends in a semicolon (see Syntax).

There are actually only three kinds of statements in C.

Variable Declaration

All variables in a C program must be declared first before being used.

A variable declaration simply tells C what kind of variable is being requested, and what the name of the variable is. For example:

```
int numAnts;    or    float antHeight;    or
char antID;     or    bool antHappy;
```

Assignment

An assignment statement is used to change a variable to another value.

The variable to the left of the equals takes on the value of whatever is to the right of the equals. This is called assignment. For example:

```
A = 3;    or    A = B + 2;    or    A = sqrt( 9 );
```

You may see a variable declaration combined with an assignment. For example:

```
int numAnts = 10    or    float antHeight = 1.3;
```

Function Calls

Another type of statement is a function call. (For a more elaborate discussion of functions, see the Functions page.)

A function is a grouped-together chunk of code that performs a certain function or procedure. Functions may be pre-defined or new functions may be defined by the user. Right now, we will simply discuss how one uses functions that are already defined. When one uses a function, it is called a function call. For example:

```
printf( "hello world\n" );    or    rewind( myFile );
```

Unusual Assignment

Occasionally, you may see statements that look a little strange. For example:

```
A += 3;    or    A *= 2;    or    A -= 5;
```

These unusual assignments can be interpreted as:

```
A = A+3;    or    A = A*2;    or    A = A-5;
```

This is used as shorthand when you want to change the value of a variable to a new value that is based on its original value.

The basic idea is to imagine the variable on the left being placed on the right and then followed by the mathematical operator. This works for division too.

Better Calculator

This program implements a better calculator—that can add AND subtract! You will build on the first calculator by using “if” statements to tell the computer to decide between addition and subtraction.

```
#include <stdio.h>

int main( )
{
    int A, B;                // declare two int variables
    char Oper;              // declare one char variable

    printf( "Type in a calculation (e.g: 5+10): " );
    scanf( "%d %c %d", &A, &Oper, &B );    // scanf captures both numbers
                                          // and the operator between them

    if ( Oper == '+' )      // if addition...
    {
        printf( "%d plus %d = %d\n", A, B, A+B );
    }

    if ( Oper == '-' )     // if subtraction...
    {
        printf( "%d minus %d = %d\n", A, B, A-B );
    }

    return 0;
}
```

Unlike in the Mini-Calculator in which each number was entered separately, the user of this program enters both numbers at the same time, and includes a mathematical operator (+ or -) in between. (See Mini-Calculator)

There are int variables (A and B) and a new character variable: **char Oper;** The Oper variable will store the one-character math operator ('+' or '-'). (See Variables).

The most complicated line in the whole program is:

```
scanf( "%d %c %d", &A, &Oper, &B );
```

This one line of code is able to capture data for three variable all in one stroke. The first “%d” captures the first number which is stored in A. The “%c” captures the character which is stored in Oper. Finally, the second “%d” captures the other number which is stored in B. (See Scanf).

What follows are two if blocks. The first one processes addition, and the second one processes subtraction. The line **if (Oper == '+')** tests if the character entered by the user is a plus symbol. The double equals (“==”) is an equality tester—not an assignment. If it is true (that Oper is really '+'), then the printf inside the if block is run. Otherwise, it is skipped over.

The line **if (Oper == '-')** works the same as the plus version. If it is true (that Oper is really '-'), then the printf inside the if block is run. Otherwise, it is skipped over. (See If...Else... and Printf).

Try to add to this program to enable multiplication and division as well.

If...Else...

Computers love to make decisions.

Using “if” and “else”, you can tell the computer how to make a decision.

If...

You might want the computer to print “you guessed it!” if the user guesses the magic number 10. (Assume the guess is already in the variable theGuess.) Here's the code you need:

```
if ( theGuess == 10 )
{
    printf( "you guessed it!\n" );
}
```

If the program reaches this point in the code, and theGuess variable does not equal 10, then it will not print “you guessed it!”. (See Printf page).

The double equals (“==”) is an equality tester. It is very different from a single equals (“=”) which is an assignment. *Be careful to always use the double equals (“==”) when using if.*

Here's how it all works. When the program reaches the if, it evaluates the conditional expression (the stuff in the parentheses). If the conditional expression is true (theGuess really does equal 10), the program will run whatever code is between the curly brackets. If the conditional expression is false (theGuess does not equal 10), the program will NOT run the code between the curly brackets.

Note that there is no semicolon after the conditional—this is correct.

If...Else...

Sometimes you want the computer to make a decision between two options. For example, you want the computer to print “you guessed it!” if the guess is correct, and otherwise, the computer should print “nope. try again.” For this, you use if and else:

```
if ( theGuess == 10 )
    printf( "you guessed it!\n" );
else
    printf( "nope. try again.\n" );
```

Conditional Operators

The double equals (“==”) equality tester is called a conditional operator. There are several other conditional operators which are quite useful.

```
A != B   true if A does not equal B, false otherwise
A < B    true if A is less than B, false otherwise
A > B    true if A is greater than B, false otherwise
A <= B  true if A is less than or equal to B, false otherwise
A >= B  true if A is greater than or equal to B, false otherwise
```

Optional Curly Brackets

Curly brackets are optional for any block (if, while, for) if the contents of the block are only one line long. It's still a good idea to indent.

This does not apply to functions.

Magic Number

Let's make a little game. This is called the Magic Number game. With this game, you will learn about the wondrous while loop.

```
#include <stdio.h>

int main( )
{
    int magicNumber = 42;
    int numberOfGuesses = 0;
    int guess;

    printf( "Welcome to the Magic Number game!\n" );
    printf( "You have 8 tries to guess a number between 1 and 100.\n" );
    printf( "Enter your first guess: " );

    while( numberOfGuesses < 8)           // play for 8 rounds (loops)
    {
        scanf( "%d", &guess );           // capture the guess

        if ( guess < magicNumber )       // if too low...
            printf( "Too low. Try again: " ); // — curly brackets optional

        if ( guess > magicNumber )       // if too high...
            printf( "Too high. Try again: " ); // — curly brackets optional

        if ( guess == magicNumber )     // if just right..!
        {
            printf( "That's right!\n" );   // — curly brackets needed
            return 0;                     // exit program now
        }

        numberOfGuesses = numberOfGuesses + 1;
    }

    printf( "\nGame over. Better luck next time.\n" );
    return 0;
}
```

The major new thing in this program is the while loop. The while loop, which begins **while(numberOfGuesses < 8)**, repeats the same process up to 8 times. The variable “numberOfGuesses” has a starting value of 0. At the end of every loop, that number is incremented by one: **numberOfGuesses = numberOfGuesses + 1**; After 8 loops around, the numberOfGuesses variable is equal to 8, and hence is no longer less than 8. Thus, the loop is finished. (See While Loops).

There is one way, however, that the loop will not repeat 8 times. That is if the user guesses the magicNumber: **if (guess == magicNumber)**. When this occurs, the program will print “That's right!” and then run the next line of code: **return 0**; The return causes the program not only to jump out of while loop, but out of the whole program. The program is ended. (see If and Functions).

There are two simple ways that this game could be improved. First, it could tell the user how many guesses they have left after every guess. This is not so hard. A second improvement would be to make the Magic Number random. To make the Magic Number random you will need to add “#include <stdlib.h>” to the top of the program, and change “magicNumber = 42” to something else.

While Loops

If a computer can do something one time, it can do it a 100 times. Repetition like this is at the heart of programming in C.

A Simple Loop

The simplest way to repeat something is to use a while loop.

If you wanted the computer to print the word “repeat” 100 times, here's the code to do it:

```
int N = 0;
while ( N < 100 )
{
    printf( "repeat\n" )
    N = N+1;
}
```

This may look confusing at first. The basic idea is to count from 0 to 100, and at each count, print the word “repeat”. Let's go step by step.

1. The integer variable N is set to 0.
2. The conditional expression (N < 100) is evaluated.
3. If N is less than 100 (the expression is true), then...
 - a. It prints “repeat”.
 - b. It adds 1 to N.
 - c. It returns to step 2 for re-evaluation.
4. If N is NOT less 100 (the expression is false), do nothing.

First N = 0 and so it is less than 100. Then N = 1, which is also less than 100. Then N = 2, which is also less than 100, etc. As long as N is less than 100, the code in the curly brackets is repeatedly run.

If and While

You may have already realized that a while loop looks very similar to an if expression. In fact, it is almost identical. The only difference (besides the word “while”) is that a while loop repeatedly runs the code in the curly brackets, as long as the conditional expression (N < 100) is true.

Break

There is a special command called “break”, which instantly jumps out of a while loop at any point. For example:

```
char C;
while ( true )
{
    C = getChar();
    if ( C=='q' )
    {
        break;
    }
}
```

Since the conditional expression is always true, the loop repeats forever. That is, unless the user types in the letter 'q'. If the letter 'q' is typed, then the “break” command will be run, and the loop will be exited.

Multiplication Table

Here's a little program that prints the multiplication table for any number. Here, you will abandon while loops in favor of the exotic and compact for loop.

```
#include <stdio.h>

int main( )
{
    int baseNumber;
    int N;
    int product;

    printf( "Enter a number: " );           // ask for a base number
    scanf( "%d", &baseNumber );          // store int in baseNumber

    for( N = 0; N <=10; N++ )             // loop as N counts from 0 to 10
    {
        product = baseNumber * N;
        printf( "%d x %d = %d \n", baseNumber, N, product );
    }

    return 0;
}
```

This program is not very long, but it prints a lot. And with a small change (to the number 10), it could print even more.

The way that the for loop is used in this program is extremely common in C. In this case, the loop is repeated 11 times as the variable N steps from 0 to 10. (see For Loops).

The heart of the for loop is the line: **for(N = 0; N <=10; N++).** N is assigned the value 0 at the start (N = 0). Then it tests if N is less than or equal to 10 (N <= 10), which it is, so one loop is performed. The product is calculated and then printed. At the end of the loop, N is incremented by one (N++). N is now equal to 1, so N <= 10 is still true, and the loop continues. This repeats 11 times, until N is 11, and is no longer less than or equal to 10.

Study this for loop carefully. It is worth knowing well.

One could easily change this program to create a nice conversion table. For example, try making a program that generates a fahrenheit to celsius conversion table. Or feet to meters. Or pounds to kilograms, etc. You may want to switch from ints to floats.

For a challenge, try making a full multiplication table (with rows and columns). You can do this by placing a for loop inside another for loop. When doing so, be careful to use different variables for each for loop.

For Loops

A for loop is a more compact (and more complex) version of a while loop. Most C programmers always use for loops, and rarely use while loops.

A Simple Loop (Again)

Here is the same code that prints "repeat" 100 times, but this time it is expressed using a for loop:

```
int N;
for( N=0; N<100; N=N+1 )
{
    printf( "repeat\n" );
}
```

Certainly, you have already noticed how much more complex the first line is. There are three parts to the first line of a for loop, separated by semicolons.

1. N=0 The initializer. This is run only once at the very beginning.
2. N<100 The conditional expression. This is run before every loop. If it is true, the loop continues. If it is false, the loop stops.
3. N=N+1 The incrementer. This is run at the end of every loop.

The same logic that applies to the while loop applies to the for loop. First N=0, then N=1, then N=2, and so on until N=100. Every time around the loop, it prints the word "repeat".

Break

The break command works in exactly the same way in a for loop as it does in a while loop. (See While page for an explanation).

Increment & Decrement

Often times you will see "N++" as the third of the three parts in a for loop. "N++" means "N=N+1". This is called an increment operator. It adds 1 to N.

You may also see a decrement operator. "N--" means "N=N-1". It subtracts 1 from N. The only reason for using the increment and decrement operators is to save typing.

(The object-oriented version of C is called C++ because of this).

Loops in Loops

Once in a while (if you're feeling loopy), you may want to put a loop in a loop. This is often used when dealing with a two-dimensional table or grid. For example, here's how you would print a multiplication table.

```
int X,Y;
for( X=0; X<10; X++ )
{
    for( Y=0; Y<10; Y++ )
    {
        printf( "%d x %d = %d ", X, Y, X*Y );
    }
}
```

Encoder/Decoder

This is a simple numeric encoder / decoder program. In this program, the encoding and decoding are done by a pair of functions.

```
#include <stdio.h>

char encode( char N )           // the encode function
{
    char E = N + 5;
    return E;                   // return encoded letter
}

char decode( char N )          // the decode function
{
    char D = N - 5;
    return D;                   // return decoded letter
}

int main( )
{
    char Original = 'A';        // declare variables
    char Enc, Dec;

    printf( "Original is: %c \n", Original );

    Enc = encode( Original );   // use encode function
    printf( "Encoded is: %c \n", Enc );

    Dec = decode( Enc );        // use decode function
    printf( "Decoded is: %c \n", Dec );
    return 0;
}
```

The heart of this program is in the two functions: “encode” and “decode”. This is the first program we have seen with a function other than “main”. (See Functions).

At the top of the program, both functions are defined. The functions must be defined before being used in “main”.

The first line of each function is called the signature: **char encode(char N)**. has three parts. The first “char” specifies that a char value will be returned by this function. The “encode” part is simply the name of the function. The second “char” inside the parentheses specifies that the function takes a char as an input argument. The “N” is the name of the char variable to which the input value is bound. This “N” variable is how information is transferred between functions.

The encoding works like this. Every char in C is actually a number. For example. 'A' is 65' and 'Z' is 90. After the char is passed to the encode function and bound to the variable N, the program adds 5 to the character. So, an 'A' (65) becomes an 'F' (70). The encode function returns the converted number.

When the converted number is given to the decoder, the decoder subtracts 5 from the letter, thus returning the letter to its original letter.

With a knowledge of arrays, string and loops, one could create a general purpose encoder/decoder for whole text strings.

Functions

Functions allow code that is used more than once to be grouped together. Functions can process several inputs to produce one output. Functions provide the organizational basis for large programs.

The Process of Functions

Every C program has at least one function. It is named “main”, and it is where every C program begins. Main may call on other functions, which in turn, may call on other functions themselves. Whenever a function's job is completed, the program returns to the calling (or parent) function.

Functions as Robots

Think of a function as a robot. There is one master robot called “main”. “Main” may call on a sub-robot to perform a sub-task (or multiple sub-robots to perform multiple sub-tasks). Whenever a sub-robot's task is completed, it returns to its master (or parent) robot. Large programs may involve hundreds or thousands of robots (i.e. functions).

Inputs and Outputs

Every function may process several inputs and may produce at most one output. For example, the inputs may be two numbers and the output may be their sum. These inputs are called **arguments**. The output is called the **result**. Here is the code for a sum function.

```
int mySum( int A, int B )
{
    int C = A + B;
    return C;
}
```

This certainly looks complicated. Let's go part by part:

- “int” — the first int states that the function returns an int as output
- “mySum” — this is the name of the function
- “(int A, int B)” — these are the two arguments, ints A and B
- “int C = A + B;” — this simply sets C to equal A + B
- “return C” — this returns C as the output of the function

Return

Return is a command which makes the program instantly jump out of the function and return to the parent function. It also brings back a value to the parent function. (Think of a child returning from the store with milk). *The type of variable returned must match the type at the start of the function.*

Function Calls

Once a function is defined (like the mySum function above), it may be used anywhere else in a program. Use of an existing function is called a function call. Here's how one would use the mySum function:

```
int result;           or       int X = 4, Y = 3, Z;
result = mySum( 2, 4 );   Z = mySym( X, Z );
```

The best way to learn more about functions is to study example functions.

List Maker, List Maker

This program lets you make a list of numbers and calculates the average. List Maker utilizes a powerful array variable to remember many things.

```
#include <stdio.h>

int main( )
{
    int numberList[5];           // declare a 5 integer array
    int N;
    float sum = 0;               // declare and set sum to 0
    float average;

    numberList[0] = 18;         // set 1st array value
    numberList[1] = 32;         // set 2nd array value
    numberList[2] = 16;         // set 3rd array value
    numberList[3] = 10;         // set 4th array value
    numberList[4] = 42;         // set 5th array value

    for( N = 0; N < 5; N++ )    // loop as N counts from 0 to 4
    {
        printf( "%d. %d\n", N, numberList[N] ); // print all numbers in list
        sum = sum + numberList[N];           // add each number to sum
    }

    average = sum / 5;          // calculate average by dividing
    printf( "The average is: %f \n", average ); // print average

    return 0;
}
```

The list in the List Maker program is an array of integers. Five of them, in fact. The five integer array is declared with the line **int numberList[5];**

The numberList array is then filled out, one number at a time. The first item in the array (item 0) is set to 18 with the line **numberList[0] = 18;**. Notice that even though the array is 5 numbers long, the last index is 4. This is because in C, you start counting with 0, not 1. (See Arrays)

The for loop that follows serves two purposes: it prints out every number in the list, and it adds all the number into one sum. As the counter variable N steps from 0 to 4, one list item is printed with each step. The sum, which begins at 0, increases every time the line **sum = sum + numberList[N];** is run. In this case, the array is indexed with the variable N, instead of a constant. (See For Loops).

The way in which the for loop is used here to run through all the values in the numberList array is very common. For loops and arrays go hand in hand.

The last section of code computes the average by dividing the sum by the total number of items in the list, which is 5. Both sum and average are floats (instead of ints), since the average may have a decimal part.

One challenging change to make to this program would be to allow the user to enter the numbers in the list. An even greater challenge would be to have a variable number of items in the list. With a little work, you could make your own spreadsheet program.

Arrays

An array is a list of variables.

An array can be a list of any kind of variable.

Array Declaration

Imagine that you want to record not just the age of one person, but the ages of 100 people. You could create 100 variables as follows:

```
int age1, age2, age3, age4, age5....age100;
```

Obviously, this is not practical. Arrays make this much easier to do.

Here is the variable declaration for an array of 100 integers.

```
int ages[100];
```

With this one little variable declaration, 100 ints have been created, all of them accessible via the int array variable "ages".

Indexing

Anywhere a variable can be used, an array value can be used instead.

Accessing the ints inside the array is called **indexing** and is done as follows:

```
ages[0] = 12;           // this sets the first age to 12
ages[23] = 55;         // this sets the 24th age to 55
ages[99] = 42;         // this sets the last age to 42

int D = ages[23];      // this sets D to the 24th age
```

Notice that ages[23] represents the 24th age, and the 100th age is ages[99]. This is because array indexing begins with 0 instead of 1.

Indexing Caution

When indexing an array, be very careful not to exceed the boundaries of the array. For example, never use -1 or any negative number as an index. Also, never use a number that is greater than or even equal to the size of the array. Both of these problems will result in serious memory errors which could cause a program to crash.

A String is an Array

Arrays of characters are called strings. For example, "code" is a string. Here are two ways to make and print the string "code".

```
char myString[5];      or      char myString[5];

myString[0] = 'c';     sprintf( myString, "code" );
myString[1] = 'o';
myString[2] = 'd';     printf( "%s \n" , myString);
myString[3] = 'e';
myString[4] = 0;

printf( "%s \n" , myString);
```

Notice something strange? The last letter in the array is assigned the value 0. *Every string in C must end with a character that is equal to 0 (not '0').*

Checksum

A checksum is a number which is used to check if a large chunk of data has been transferred correctly. They are used in CD players to correct for scratches, and by web browsers for file downloading. The basic idea is to add all the little pieces together before the transfer. This is called the checksum. Then, after the data has been transferred, the sum is computed again. If the new sum and the checksum do not match, the data did not transfer perfectly.

This program demonstrates the use of character arrays—also known as strings.

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

int main( )
{
    char myString[64];
    int length, N;
    int checksum = 0;

    printf( "Enter a name: " );           // ask user for a string
    scanf( "%s", myString );             // put string into myString
    length = strlen( myString );          // use strlen to find string length

    for( N = 0; N < length; N++ )        // add each letter to checksum
    {
        checksum = checksum + myString[N];
    }

    printf( "The checksum for %s is %d. \n", myString, checksum );
    return 0;
}
```

The list in the List Maker program is an array of integers. Five of them, in fact. The five integer array is declared with the line **int numberList[5];**

The numberList array is then filled out, one number at a time. The first item in the array (item 0) is set to 18 with the line **numberList[0] = 18;**. Notice that even though the array is 5 numbers long, the last index is 4. This is because in C, you start counting with 0, not 1.

Printf & Scanf

The functions printf and scanf provide output to and input from a text interface. These are the two of the most used functions in C.

Printf

Printf prints text on a text (command line) interface, such as a terminal. Printf may be used to print a simple string (a set of chars, see Array page):

```
printf( "hello world\n" );    or    printf( "C is great!\n" );
```

You are probably wondering what the “\n” is doing at the end. This is a way of specifying a “newline” character. A “newline” character is just like pressing “return” or “enter” on the keyboard. (Oftentimes, if this is not included, the program will not print out the line of text.)

Formatted Text

The ‘f’ at the end of printf stands for “formatted.” Printf can also format text. For example, you may want to print out the value of a variable. Here’s how:

```
int age = 12;
printf( "you are %d years old.\n", age );
```

Obviously, something looks strange in the middle of the string—the “%d” part. Also notice the “age” variable tacked on to the end, after the string. These two things are connected. When this program is run, it will print the following text: “you are 12 years old.” Here’s another example:

```
int age = 9;
float height = 1.4;
printf( "you are %d years old and %f meters tall.", age, height );
```

This program will print the text: “you are 9 years old and 1.4 meters tall.” First, the age is inserted for “%d”, and then the height is inserted for the “%f”. “%d” is for formatting ints. “%f” is for formatting floats. Here’s a brief list of some of the formatting options for printf.

%d	for inserting and formatting ints
%f	for inserting and formatting floats
%c	for inserting and formatting chars
%s	for inserting and formatting strings (see Arrays page)

Scanf

Scanf works exactly like printf but in reverse. Rather than printing out text for the user, scanf collects inputted text from the user. Here’s an example:

```
int age;
printf( "enter your age: " );
scanf( "%d", &age );
printf( "you are %d years old\n", age );
```

This program asks for the user’s age. Then waits for the user to type it in. Then it confirms what the user typed. Notice the “&” before “age” in scanf. The “&” allows scanf to store the entered number into the variable “age”. *Don’t forget the “&” before all variables when using scanf (see Addresses page).*