

How variables work

Comments and questions to [John Rowe](#).

Although it's not essential at this stage to know the precise details of how computers handle variables, having a rough idea can help avoid some common mistakes.

As always with C, things are **much simpler** than we might imagine, not more complicated.

Some Horrible History

In the very early days of computers, programs were written in an extremely low-level syntax called *machine code* which was different for each type of processor. These processors could only store a very limited amount of data and programmers had to manually keep track of what data was stored where.

The result was that programs consisted of rather inscrutable instructions such as:

```
Multiply the number stored in location 12 by the number stored in
location 7 and store the result in location 4.
```

Obviously, such programs were rather hard to understand!

(But on the other hand, it was extremely easy to understand exactly what the computer is doing.)

When machine-independent, programmer-friendly languages were invented they allowed a programmer to "declare a variable". The compiler would then allocate a previously-unused memory location (say 12) and from then on the programmer could use the name of that variable as a shorthand for "the number stored in location 12". This allowed the above instruction to be rewritten as something like:

```
force = mass * acceleration
```

In other words, the compiler allowed the programmer to **give programmer-friendly names to the individual memory locations**. The compiler also took care of the job of deciding which memory location was given which name to avoid the danger of the programmer accidentally using the same location twice.

This was obviously somewhat easier for the programmer to understand. The compiler then translated the programmer-friendly statements like this into instructions like the ones at the beginning which the computer actually executed.

The "location number" above is referred to as the variable's **address** and is a very important concept. If we know the address of a variable we can access and change its value, bypassing the programmer-friendly name altogether.

The computer stores the values of its variables in its memory

Although modern computers are rather different from the ones described above, the principles they work on are remarkably similar. Programs store their temporary, "running" data in their memory (RAM - for Random Access Memory). Memory is measured in bytes and we can think of it being arranged in one huge block with its bytes numbered 1, 2, 712, etc..

Typically it takes four bytes to store an `int`, so when the computer encounters a line like:

```
int i, j;
```

it reserves two chunks of four bytes each to store the values of these two variables:

| | | | | | | | | |
|--------------|-----------------|-----|-----|-----|-----------------|-----|-----|-----|
| Byte number: | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 407 |
| Used for: | <----- i -----> | | | | <----- j -----> | | | |

Here we have shown `i` and `j` as being stored next to each other, but the computer is under no obligation to do so.

The location where the variable is stored in memory is then the **address** of the variable. In the case of `i` that address is 400. There is no variable with address 401, but that doesn't matter, as the compiler has set aside bytes 400-403 inclusive to use for storing the value of `i`.

From now on, the compiler treats the "variable" `i` simply as a programmer-friendly name for "the four-byte integer stored starting at byte number 400", in exactly the same way as we described in the Horrible History. It goes through the rest of our code, replacing valid occurrences of the variable name `i` by "the four-byte integer stored starting at byte number 400". When it has finished compiling our code it throws away its own internal list of variable names and their addresses as it has no more use for it. The final program never even sees the list of

The final decision of where to store the variables in memory is actually only finalised when the enclosing `{ ... }` block is entered.

variable names, it operates directly on the memory locations.

After all, "programmer-friendly" names for memory locations aren't much use to a computer!

Which is why statements such as "i = i + j;" make sense:

| What we write | What the compiler translates it into |
|---------------|---|
| i = i + j; | Add the number stored in location 400 to the number stored in location 404 and store the result back in location 400. |

We can ask the compiler to keep the list variables and their addresses for debugging purposes, but it has no effect on the actual running program.

&i tells us where the variable i is stored in memory

C has an operator for (almost) everything and if we ever want to know where a particular variable is stored in memory then C provides the & operator and even a printf() format %p to go with it:

```
printf("i has the value %d and is stored at %p\n", i, &i);
```

In our example, the above code would print out the value of i followed by its address (location in memory) which is "400", although it would almost certainly print it out in hexadecimal. It's most unlikely you'll ever need to do this.

What you need to remember

1. A variable name is just a programmer-friendly name for the contents of a particular part of the computer's memory.
2. The & operator can tell us the **address** of a variable, i.e. where it is being stored, which allows us to bypass the "programmer-friendly" variable name altogether.

Reading in variables with scanf()

We have already encountered the printf() statement, which can be used to print the value of variables to the screen:

```
i = 1234;
j = 5678;
printf("The answers are: %d and %d\n", i, j);
```

C provides the scanf() function to do the opposite: to read in variables from the keyboard (or whatever the default input for our program is).

scanf() is a "user-friendly" input function, it skips over spaces and new lines so if we are required to input two integers we could type in the two integers either both on the same line, or one on each line. Unlike printf() it's not normal to put any text in the format string, if we want to read in two integers the format is just: "%d %d".

So to read in two integers i and j from the keyboard we write:

```
printf("Please enter two integers ");
scanf("%d %d", &i, &j);
```

Note the &s - see below.

The above code will allow us to type in two integers and will set the values of i and j to the two values we have just typed.

scanf(), and its siblings are probably the only time you will ever need to use the "&" operator.

Two things to look out for

1: Why the &?

By analogy with the printf() example above, our first attempt to use scanf() might have looked like this:

```
i = 1234;
j = 5678;
scanf("%d %d", i, j); /* Wrong! */
```

There's a big problem here, however. We stressed last week that functions take **arithmetic expressions** as their parameters and that even if that expression is a single variable name as above, that variables can't have its value changed by the function.

As written above, the `scanf()` function can no more change the values of `i` and `j` than it can change the values of the mathematical constants "1234" and "5678". All that `scanf()` knows is that it was called with two arithmetic expressions for its arguments, one with the value 1234 and the other 5678.

It's exactly as if we had written:

```
scanf("%d %d", 1234, 6789 - 1111); /* Wrong! */
```

which clearly has no chance of changing the values of `i` or `j`! Worse still, it will write four bytes of data into each of the "random" memory locations 1234 and 5678.

By using the `&` operator we are telling `scanf()` not the values of `i` and `j` but **where they are stored in the computer's memory**. In the example at the top, we are passing the values 400 and 404, not 1234 and 5678;

2: The format to read a double is "%lf"

This is one of the rare exceptions to C's helpful rule of treating all floating-point operations as `double`. It does this because it has no choice. Consider the following:

```
float littlevar; /* Four bytes */
double bigvar; /* Eight bytes */

scanf("%f", &littlevar);
scanf("%lf", &bigvar);
```

The first call to `scanf()` requires us to read in a number and then write that number into the four bytes starting at the address (memory location) of `littlevar`, the second writes eight bytes to the memory location of `bigvar`. Imagine we got these the wrong way round:

```
float littlevar; /* Four bytes */
double bigvar; /* Eight bytes */

scanf("%f", &bigvar); /* Wrong! */
scanf("%lf", &littlevar); /* Wrong! */
```

The first call will write four bytes into where `bigvar` is stored, leaving the other four bytes unchanged.

The second call is even worse: it will write eight bytes in to where `littlevar` is stored, but `littlevar` only has four bytes allocated to it. The result is that whatever is being stored immediately after `littlevar` will have its first four bytes over-written. (This may or not be the variable that was declared after `littlevar`).

These are particularly nasty bugs but the compiler will warn you so **pay particular attention to any Yellow Triangles of Peril next to `scanf()` statements**.

Remember: the compiler is very good at warning you if you have made one of these mistakes.

Checking for valid input

Often we need to read in some values that must be in a particular range. For example in a noughts and crosses program we may require that the coordinates of a square are in the range one to three. C provides a couple of convenient ways to do this.

Infinite loops and the break statement

The first way to do this is to use an infinite loop and to use the `break` statement to break out of it if the input data are valid. This gives the option of printing a helpful message rather than simply repeating the same one as before.

For example, consider a noughts and crosses program where we want to read in two integers, each in the range one to three:

```
/*
 * Demonstrate the break statement
 * This could be the start of a noughts and crosses program
 */
#include <stdio.h>

int main() {
    int x, y;

    printf("Welcome to the noughts and crosses program\n");
```

It always makes sense to check for valid input but if we were reading our data from a file rather than the keyboard it would be better to just quit the program, as the data file has no chance to correct itself.

```

while ( 1 == 1 ) {
    printf("Please enter the x and y coordinates in the range 1-3 ");
    scanf("%d %d", &x, &y);
    if ( x < 0 || x > 2 || y < 0 || y > 2 )
        printf("\nThe inputs are in the wrong range, please try again.\n\n");
    else
        break;
}

printf("Your move is: (%d, %d)\n", x, y);
return 0;
}

```

This slightly more complicated example uses a couple of related features of C that we haven't seen before.

The loop's controlling statement:

```

while ( 1 == 1 ) {
    ....
}

```

will obviously never be false. This is an example of an **infinite loop** and if you ever create one you had better be sure there is a way of getting out of it! (It's quite easy to create an infinite loop by accident.)

The **break** statement breaks out of the inner-most loop the statement is inside of. For fairly obvious reasons it's nearly always part of an `if ()` statement. For this reason `break` breaks out of a **loop** not an `if ()` statement, nor does it return from a function.

Notice the logic of the test preceding the `break` statement: we want to break out of the loop if the input data are valid.

The `do { ... } while();` loop

The second of C's loops, the `do { ... } while();`, behaves in exactly the same way as the more widely used `while() { ... }` loop except that the body of the loop is executed **before** the logical test, meaning that the loop always executes at least once.

Here is an example of the `do { ... } while();` loop doing a similar job to before.

```

/*
 * Demonstrate the do { ... } while; loop
 * This could be the start of a noughts and crosses program
 */
#include <stdio.h>

int main() {
    int x, y;

    printf("Welcome to the noughts and crosses program\n");

    do {
        printf("Please enter the x and y coordinates in the range 1-3 ");
        scanf("%d %d", &x, &y);
    } while ( x < 1 || x > 3 || y < 1 || y > 3);

    printf("Your move is: (%d, %d)\n", x, y);
    return 0;
}

```

The result is that the program will read in `x` and `y` and **then** check to see if either of them is outside on the range 1 to 3. The code will loop until the test condition that `x` or `y` is outside its allowed range is no longer true.

It's wise one of these two methods whenever you wish to read in from the keyboard some data that must have certain values.

We can use `break` to break out of **any** loop, not just an infinite one.

Note the position of the semi-colon at the very end of the statement.

As always, if the loop had contained just a single statement we could miss out the braces `{ }`.

Writing to, and reading from, files

So far, we have always written to the standard output or **stdout** for short which is usually the terminal window. Similarly `scanf ()` reads from standard input or **stdin** for short, usually the keyboard.

Reading from, or writing to, files on the computer is very similar once we have opened the file we wish to read from or write to.

The `fopen ()` function opens a file for reading ("r") or writing ("w").

The `fscanf ()` and `fprintf ()` functions read from or write to that file. They behave exactly like `scanf ()` and

`printf()` except for a "file" argument immediately before the format.

The `fclose()` function closes a file when we've finished with it. (All files are automatically closed for us when the program finishes.)

As an example, let us revisit our "Hello, world" program, this time making it write to a file:

```
/*
 * "Hello world" - to a file!
 */
#include <stdio.h>

int main() {
    FILE *outfile;

    /* Open a file for writing */
    outfile = fopen("helloworld.txt", "w");

    fprintf(outfile, "Hello, world\n");

    fclose(outfile);

    return 0;
}
```

The first thing we see is the funny **FILE** * variable called `outfile`. The precise meaning of `FILE *` varies from system to system and is defined in `stdio.h`; but whatever system we are using we can always use `FILE *` and it will mean the right thing for that system.

Don't forget the * !

We then open the file using `fopen()`. The first argument to `fopen()` is the name of the file we wish to open. **This is the only time we use the name of the file**, after that we just refer to it using the variable `outfile`. The second argument to `fopen()` tells the system how we want to use the file; "w" means to write to the file, over-writing any existing contents of the file, if any. We could also have used "a" to append our new data after the existing contents of the file. As mentioned above, "r" is used for reading.

We then just use `fprintf()` to write to the file. This is exactly the same as `printf()` except for the additional first argument which is the `FILE *` variable, `outfile`, not the file name.

Finally, we close the file, again using `outfile`, not the file name. This step is optional in this case, as the program is about to exit anyway.

What if we can't open the file? - NULL and exit()

Our program has a weakness in that when it tries to open the file "helloworld.txt" it doesn't check to see if the call to `fopen()` failed. This might be because the disk was full, for example, or because we are trying to create a file inside a directory (or "folder" in Mac-speak) we don't have permission to write to.

This raises two questions: "How do we know if `fopen()` failed?" and "what do we do if it does?".

If `fopen()` fails it returns the special value **NULL**, defined in `stdio.h`. We can then test for failure using:

```
if ( outfile == NULL ) ... /* We could not open the file */
```

What we choose to do then this will vary according to our program but the simplest thing is just to print a warning message and exit from the program. We can do this from inside any function, not just `main()`, by calling the `exit()` function with an integer argument. This is defined inside the `#include` file `stdlib.h`; the first time we have used this file. Just as when we `return` from `main()`, the convention is to use non-zero for failure.

`return` returns from that function, `exit()` exits from the whole program.

We demonstrate this by making our code into a function:

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

/*
 * Write "Hello world" - to a file, exiting on failure
 */
void writehello() {
    FILE *outfile;

    outfile = fopen("helloworld.txt", "w");
    if ( outfile == NULL ) {
        printf("I cannot open the file\n");
        exit(1);
    }
}
```

```

    fprintf(outfile, "Hello, world\n");

    fclose(outfile);
}

```

Reading from a file with fscanf()

This works in exactly the same way:

```

/*
 * Read an integer from a file
 */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *infile;
    int input;

    infile = fopen("input.txt", "r");
    if ( infile == NULL ) {
        printf("I cannot open the file\n");
        exit(1);
    }

    fscanf(infile, "%d", &input);
    printf("The value is %d\n", input);

    /* Do something interesting */

    fclose(infile);
    return 0;
}

```

It's worth noticing here that since we are inside `main()` we could have used either `exit()` or `return`. Also notice how we have been slightly paranoid by printing out the value of `input` immediately after we have read it. This is a good idea!

How do we know when we have run out of input?

When reading data from a file, we may not know in advance exactly how many data values there are in there. Fortunately C makes it easy for us as `scanf()` and `fscanf()` both return the number of values they have successfully read in or a special value called `EOF` if they have reached the end of the file. This allows us to write simple loops such as:

```

while ( fscanf(infile, "%d", &i) == 1 ) {
    ... /* Do something interesting with i */
}

```

Had we been reading in two data values we would have checked that `fscanf()` had returned the value 2:

```

while ( fscanf(infile, "%d %d", &i, &j) == 2 ) {
    ... /* Do something even more interesting with i and j */
}

```

This tends to be a lot easier than, say, writing the number of data values at the start of the file. It's much less useful when reading from the screen as `scanf()` just assumes it's waiting for the user to type the numbers in.

The three defaults: `stdin`, `stdout` and `stderr`

We have briefly mentioned standard output (`stdout`) and standard input (`stdin`) above. Specifically, C sets up three default channels for input and output.

`stdin` is used for input

`scanf("%d", &i)` is identical to `fscanf(stdin, "%d", &i)`

`stdout` is used for output

`printf("Hello, world\n")` is identical to `fprintf(stdout, "Hello, world\n")`

`stderr` is used for errors, warnings and diagnostics

Thus it's good practice to use `fprintf(stderr, ...)` for error messages, rather than `printf(...)`. For example:

```

if (infile == NULL ) {
    fprintf(stderr, "I cannot open the file\n");
    exit(1);
}

```

is better than using `printf("I cannot open the file\n")`.

Normally `stdout` and `stderr` both go to the terminal window but they can be sent to different places.

We won't penalise you for sending all your error messages to `stdout` using `printf()` but you should be aware that it's better practice to use `stderr`.

Summary

1. A variable is just a programmer-friendly name for the value stored in a particular location in the computer's memory.
2. The `&` operator tells us where the variable is stored, its **address**.
3. `scanf()` reads from the keyboard.
 - Don't forget to use the `&` in front of the variable names you want to read in.
 - The input format for a double is `"lf"` not `"f"`
 - The compiler is good at warning about errors in `scanf()` - make sure you check them!
4. If possible, check input is valid either by using `do { ... } while();` or an infinite loop with a `break` for valid input.
5. `fopen()` opens a file, usually with `"r"`, `"w"` or `"a"` (read, write or append).
6. `fopen()` returns `NULL` if it cannot open a file.
 - Always check and either try to recover from the error or print a message and call `exit()`.
7. `fscanf()` and `fprintf()` are just like `scanf()` and `printf()` but with an additional first argument.
8. `fscanf()` returns the number of items it read in which is an easy test for end of input data.

Appendix: a larger example for you to study

Let's pull a few of these concepts together by writing a program that reads some integers from the screen, stopping when it reads a value of zero, and for each value prints that value and the square of that value to a file.

The program then opens that file for reading and checks that it does indeed contain a list of integers, each followed by its square.

Since writing and reading the file are separate, self-contained tasks we make each a separate functions.

The original is a single file but we split it into three parts for readability.

The program header and the main() function

```

/*
 * Demonstrate writing to, and reading from a file.
 * We do this by reading some integers from the keyboard
 * and writing them, and their squares, to a file.
 * After closing the file, we read integers and their squares,
 * back in again, checking the squares are correct
 */
#include <stdio.h>
#include <stdlib.h>

void write_data();
void read_and_check_data();

int main() {
    write_data();
    read_and_check_data();

    return 0;
}

```

The only things to notice here are the inclusion of `stdlib.h` for `exit()`, and the function prototypes.

Reading input from the terminal and writing the file

```

/*
 * Read integers from standard input and write the
 * integers and their squares to a file.
 * Exit the program on failure.
 */
void write_data() {
    FILE *outfile;

    outfile = fopen("mydata.txt", "w");
    if ( outfile == NULL ) {
        fprintf(stderr, "I cannot write to the data file\n");
        exit(1);
    }

    printf("Please enter one integer per line. Enter zero to stop\n");

    /* loop until the input value is zero */
    while( 1 == 1 ) {
        int i;

        printf("Next value? (zero quits) ");
        scanf("%d", &i);

        if ( i == 0 )
            break;

        fprintf(outfile, "%d %d\n", i, i * i);
    }

    fclose(outfile);
}

```

First we open the output file, exiting the program if that fails, with a warning to `stderr`. Notice we exit with a non-zero value (1) to indicate the program failed to run successfully.

Then we have an infinite loop, breaking out of it when the input is zero and writing each input integer and its square to the output file. Although it's not critical, we declared the variable `i` inside the loop's `{ ... }` block as it is never used outside of it.

Finally we close the output file using `fclose()`. This is important as otherwise the data might not be written as C is allowed to buffer output for efficiency (i.e. to save up output until a reasonable amount needs to be written and then write it all at once).

Read in the file and check it is correct

```

/*
 * Read integers and their squares, checking the squares are correct
 * Exit the program on failure.
 */
void read_and_check_data() {
    int i, isquared;
    FILE *infile;

    infile = fopen("mydata.txt", "r");
    if ( infile == NULL ) {
        fprintf(stderr, "I cannot read from the data file\n");
        exit(2);
    }

    /* Carry on as long as we can read two integers from the file */
    while ( fscanf(infile, "%d %d", &i, &isquared) == 2 ) {
        if ( isquared != i * i )

```

C provides the "file flush" function `fflush(outfile)` to flush any buffered data to disk even before the file is closed.

```
    while ( isquared != i * i ) {
        fprintf(stderr, "The ivalues %d and %d disagree!\n",
                i, isquared);
        exit(3);
    }

    printf("%d squared is %d\n", i, isquared);
}
fclose(infile);
}
```

First we open for reading the file we have just written. Notice we use "r" as the second argument to `fopen()` not "w". As before, if the open fails we write a message to `stderr` and `exit`, using a different non-zero value (2).

We then enter a classic loop for reading from a file: we try to read in two integers, `i` and `isquared`. The call to `fscanf()` is **inside** the control expression of the `while` loop: the value returned by `fscanf()`, which is equal to the number of data values it has successfully read in, is compared to see if it equal to two. If it isn't, the loop quits.

If the data values disagree we print an error message and `exit`, this time with the value 3.

One subtle point that's worth noticing is that we print a **positive confirmation** that we have read in some valid input. It is much better to print a message when we are sure something is correct than when we are sure something is wrong. It's always possible for a program to find new ways of going wrong!

Finally, we close the input file.
