Week 1

Now that we can enter and run a C program, lets look at the structure of a C program.

1.1 The Basics: Variables and Expressions

1.1.1 Variables

A **variable**, literally, is something that can change. In computer science, a variable is a place where you can store data. A variable can hold an integer, a real number (referred to as floating point numbers from now on), a character, or any of several types defined by the C programming language. It is even possible to define your own types.

Declarations

Unlike simpler languages like BASIC, you must tell the compiler about a variable before you use it. This process is called **variable declaration**. Let's say, for example, that you need to declare two integer variables, one character variable, and one floating point variable. Each variable must have a name, so call the integer variables "count" and "sum", the character "c", and the floating point variable "average". These names are arbitrary, but it is usually a good idea to give the variables a name that has something to do with their use. The declarations for the variables listed above is shown in figure 1.1.

```
int count, sum;
char c;
float average;
```

Figure 1.1: Sample variable declarations

The structure of declarations is always the same. First, you tell the compiler the **types** of the variables you are declaring, and then you give their names in a comma separated list, terminated by a semicolon. There is no limit to the number of variables you can have.

The placement of declarations is very important. Variable declarations MUST appear at the very top of a block of code. If a variable declaration appears anywhere else, the compiler will issue many cryptic errors.

Using variables

Once a variable has been declared, you can then use it. In order to assign a value to a variable, you simply type, `<variable name> = <value>;`. For example, to set the values of `sum` and `count` to zero, `c` to the letter a, and `average` to 1.5, you would use the code shown in figure 1.2.

```
sum = 0;
count = 0;
c = 'a';
average = 1.5;
```

Figure 1.2: Assigning values to variables

Note that to refer to a specific character in C, you enclose it in single quotes. If you assign an incompatible type to a variable, like assigning 4.5 to an integer variable, the compiler will usually round down and (maybe) issue a warning. In most cases, the compiler will do any conversion for you if it can. Sometimes, this can result in errors that are extremely hard to find.

Arithmetic operations

One thing you may want to do with variables is perform math with them. C defines several math operators. Some of the major ones are listed in figure 1.3.

There are no surprises as to the syntax of math in C, you simply write math expressions as you would on paper: `a = b + c` assigns the sum of b and c to a, and so on.

Integer division and type casting Remember that all variables have types? How then do we determine the type of an arithmetic operation? This is an extremely important point that gets glossed over far too often and I have seen far too many mistakes result from it.

In an assignment expression (`a = b + c`), the type of the assignment is determined by the variables on the *right* side of the equals sign. The compiler

/ Division

+ Addition

- Subtraction

* Multiplication

% Modulo

Figure 1.3: The major arithmetic operators

usually does the right thing; that is, if you add a `float` and an `int`, you will get a float.

Division, however, is a different matter. I cannot stress the following point enough except to put it in boldface: **If you divide two integers, you get an integer, not a float**. This point has been the main source of bugs in far too many programs for me to not emphasize it. Let's take a look at the code in figure 1.4.

```
float average;
int sum, count;

count = 20;
sum = 30;
average = sum/count;
```

Figure 1.4: Example of an integer division error

This is a perfectly reasonable piece of code that simply doesn't act as you expect. Whoever wrote this piece of code expected to divide the sum by the count and assign it to a *floating point* variable and get 1.5. However, what really happens is C performs integer division, and the right hand side of the equation evaluates to 1! Many times, what will happen is the user expects to divide two numbers and get a result between zero and one that represents some sort of percentage which will get used later. However, if you forget that C performs integer division, ALL the numbers you get will be zero, and you will get funny results. If you ever have a program that involves division and you get screwy results, suspect this first.

So how do you get around this? One solution is to make all your variables floating point. This works in practice, but it has two drawbacks. First, it won't make the graders happy because you're skirting the real problem, and second, it's a lot slower. Floating point operations are several times slower than integer operations and should be avoided if performance is an issue. So what's the real solution?

In C, it is possible to force a variable to act as another type during a computation. This is called **type casting**. The way this is performed is to precede the variable you want to cast with the type you want in parentheses. This is shown in figure 1.5.

```
average = (float) sum / (float) count;
```

Figure 1.5: Example of type casting

This will force both `sum` and `count` to behave as floating point numbers in this computation only, and will assign the correct value to `average`.

Use of type casting is something of a religious debate among computer scientists. Some people feel that it masks potential errors and can make your code unreadable and ugly. Others feel that it is appropriate to exploit all features of the language, and that type casting can allow you to do powerful things. Personally, I think it should be avoided most of the time, but like most things, there are times when it is totally appropriate and makes a lot of sense. So use it as you see fit, and if someone tells you that type casting is inherently evil, yell at them for being closed minded.

Incremental operators

In addition to the standard arithmetic operators, C provides corresponding operators that allow you to modify the value in a variable. For example, you might want to increase the value of a variable by 7, without knowing what is in the variable. Most other languages would require you to use a construct like `sum = sum + 7`, and this will work in C also. However, it is possible to use an *incremental* operator, formed by appending an equals sign to the arithmetic operator. That is, to add 7 to the value of sum, you would simply write `sum += 7;`. Note that the `/=` operator suffers from the integer division problem as well — if sum has a value of 2 and you write `sum /= 3;`, sum will be assigned a value of 0.

Finally, C provides special operators to increment or decrement the value of a variable by one. These are the `++` and `--` operators. The statement `sum++;` will add one to the value of sum, and `sum--;` will decrement it by one. You can also write `++sum;` or `--sum;`. These have virtually the same effect; the difference is subtle and will be discussed in the precepts.

Other types

There are other basic types that can be used in the C language. Some of them are too complex to discuss at this point, but some of the basic ones are shown in figure 1.6.

The most interesting thing to note about C types is that the **char** type, which is normally used to hold a character, can be used in arithmetic expressions just like an integer. Therefore, if you have a character variable `c` that you *know* to be between 'a' and 'z', you can write `c - 'a'` to get its position in the alphabet between 0 and 25! This can be extremely useful in certain situations, some of which I'm sure you will encounter.

The second most interesting thing to note about the types listed in figure 1.6 is that the "signed" and "unsigned" designations I've given to them is totally non-portable. That is, different machines, even different C compilers, have their own notion about whether or not a `char` is signed or not. Don't write **any** code that depends on the signedness of a `char` or an `int`. If you *absolutely* need something that is signed, you can put the `signed` keyword in front of the type, like: `signed int i;`. You probably won't need to do this *ever*, but it should be in the back of your mind in case you're making assumptions that aren't valid.

int Signed integer. Can represent numbers from -2^{31} to 2^{31} on the Sparcs.

unsigned Unsigned integer. Can represent numbers from 0 to 2^{32} on the Sparcs

long Long precision integer. Same as int on the Sparcs.

char Signed character. Can hold values from -127 to 128.

unsigned char Unsigned character. Can hold values from 0 to 255.

float Floating point.

double Double precision floating point. Can represent much larger/smaller numbers.

Figure 1.6: Sample C types

1.1.2 Digression: `printf` and `scanf`

For now, our C programs will use very simple ways to input and output data. We use the `printf` and `scanf` functions. If there is an integer n in the program whose value you (the user) want to set while the program is running, then the program has to contain the following line at the appropriate place: `scanf("%d", &n);`. The use of & is weird, but will become clearer later in the course.

Printing variables

To tell printf to output a variable, you put a percent sign (%) in the string, followed by a "format specifier". The format specifier simply tells printf what type your variable is. So, for example, if you want to print out the value of the integer variable `sum` and the floating point variable average, you would type:

```
printf ("The sum is %d and the average is %f",sum,average);
```

Note that the variable names appear in a comma separated list after the string. Later we will look at functions and see why this is so, but for now just remember what a call to `printf` looks like. The format specifiers for common variable types are shown in figure 1.7.

int `%d`

long `%ld`

char `%c`

float `%f`

double `%lf`

Figure 1.7: Format specifiers for `printf`

Note: In order to print an actual percent sign, you must type two percent signs: %%. You `will` need to print percent signs if you ever have to write a program that outputs PostScript code (which you will), so bear this in mind. It may look right on the screen, but go to print it and it won't look anything like you expect.

Printing special characters

`printf` gives you the ability to format your output text as well. For example, if you want to print "Hello, world" on two separate lines, you might insert a **newline** character into the string, telling `printf` to break the line there. Special characters are indicated by following the special character \ by the control character you want. This is shown in figure 1.8.

Some of the more important special characters are shown in figure 1.9.

1.1.3 Conditionals

No computer program would be useful if it couldn't make decisions. **Conditionals** are ways for your program to decide what block of code to execute based

```
printf ("Hello,\nworld");
```

Figure 1.8: Using a special character with `printf`

\n Newline

\t Tab

**** Backslash

\b Backspace

Figure 1.9: Special characters for `printf`

on a test of some sort. They allow your program to take different branches, or, to use the technical term, they modify your program's flow of execution.

Booleans

A **boolean** is an expression that is either true or false. In C, anything that has a value can be used as a boolean, with zero being false and anything non-zero being true. C provides operators to test certain relationships between variables. For example, if we have our two integer variables `sum` and `count`, the expression `sum < count` is a boolean expression that will evaluate to 1 if `sum` is less than `count`, and zero otherwise. C provides many of these operators; some are listed in figure 1.10.

Remember that the test for equality is a DOUBLE equals sign, not a single one. You will get no compiler error if you use a single equals sign, because it is a valid expression that has a value!

Since C has no specific type for booleans but rather uses zero and nonzero to indicate true and false, sometimes you will see people omitting the conditional test and writing code like that in figure 1.11.

This is merely shorthand for testing if the variable is not equal to zero. This is the most confusing C idiom for beginners; I suggest you don't begin using it because you may look back at code you wrote a couple of weeks ago and have no idea what you wrote. It's just a couple of extra keystrokes to type out the whole expression.

== Is equal to

!= Is not equal to

< Is less than

<= Is less than or equal to

> Is greater than

>= Is greater than or equal to

Figure 1.10: Relational operators in C

```
if (variable_name)
{
  do_something;
}
```

Figure 1.11: A common C idiom

Logical operators

It is possible to combine several simple booleans into a more complicated one using logical operators. In english, these would be the words "and", "or", and "not". In C, the operators are, respectively, &&, ||, and !. Therefore, the statement in figure 1.12 is a valid boolean expression.

```
(!(a < b) && ((c >= d) || (d == a)))
```

Figure 1.12: Compound boolean statements

The parentheses are not necessary in this example, but they are included to show the precedence of the logical operators.

If-Then

The most common conditional is the **If-Then** statement. This simply says, "If this boolean expression is true, then execute this block of code." If the expression is not true, the block of code will be skipped. As an example, let's say you have computed some floating point value in a variable `density`, and you know that this variable MUST be between zero and one. Due to experimental error, it is possible that the value may be slightly higher than one or slightly lower than zero. You would like to "clamp" this variable; that is, if it is outside the range, you would like to set it equal to the endpoint of the range. A simple application of the If-Then test is shown in figure 1.13.

```
if (density > 1.0)
  density = 1.0;

if (density < 0.0)
  density = 0.0;
```

Figure 1.13: An example of the If-Then construction

These four lines of code will clamp the variable as desired. Note the structure of the If-Then statement: the boolean expression is enclosed in parentheses, and the block of code follows the if statement. This example also points out something about code blocks: they can be one line long. If this is the case, the curly braces are not necessary. The example above could have easily been written as in figure 1.14 but it is not necessary.

```
if (density > 1.0)
{
  density = 1.0;
}

if (density < 0.0)
{
  density = 0.0;
}
```

Figure 1.14: Using braces in an If-Then block

It is important to note that if you want the block of code to be more than one line long, the braces are essential. I see far too often people who think they only want one line of code as their conditional block, but then realize that they really need two lines, so they just add a line without remembering the curly brace. The result? If the statement is false, the second line will get executed anyway.

If-Then-Else

The logical next step from If-Then constructs is a construct to allow you to execute a block of code if the boolean expression is false. That is just what the **If-Then-Else** construct allows you to do. A simple one is shown in figure 1.15.

```
if (density < 0.5)
  printf ("The density was less than 0.5\n");
else
  printf ("The density was more than 0.5\n");
```

Figure 1.15: A simple If-Then-Else construction

Note, also, that If-Then-Else constructs can be nested, as shown in figure 1.16.

Goto

The **goto** statement is an unconditional jump to some other point in your program.

Don't use it. Period. It's terrible programming style, and it won't win you any points with the graders.

```
if (density < 0.25 )
  printf ("The density was less than 0.25\n");
else if (density < 0.5 )
  printf ("The density was less than 0.5\n");
else if (density < 0.75 )
  printf ("The density was less than 0.75\n");
else
  printf ("The density was less than 1.\n");
```

Figure 1.16: Nested If-Then-Else constructions

Having said that, there `are` certain situations in which the alternatives are much worse. If you think you've encountered such a situation, either ask a TA if there's really no better way to do it, or write a comment next to your goto explaining why you think it's necessary.

Goto statements are so rarely a good idea that I'm not even going to document their syntax here – you'll have to look it up in K&R.

Switch

The **switch** statement is like a multi-valued conditional statement. You can think of it as sort of a glorified goto; but don't be afraid of it — it has enough niceties to make clumsy code into elegant code. Often, you will have a variable that can have one of several discrete values, and you want a different block of code to be executed for each value. For example, let's say you have a character variable `gene` that represents a part of a genetic sequence. `gene` can be one of either 'a', 'c', 'g', or 't'. You could enumerate every possibility with If-Then-Else statements, but since the values are discrete, switch is much better. An example of the `switch` construction is shown in figure 1.17.

This should be self explanatory. The values that follow the `case` statements must be either integers or characters (actually, there's no difference). The only interesting thing to note here is the use of the `break` statement. The `break` statement tells the switch statement to, literally, break out of the switch code block and resume execution after the last curly brace. If you forget the break statement, execution will "fall through" to the next case statement. Sometimes this is what you want! Imagine a program where the user presses a key and you respond to it in some way. If the user presses a lowercase 'a', you do one thing, and if the user presses an uppercase 'A', you do something else first and then do the lowercase 'a' thing. This would look like the code in figure 1.18.

```
switch(gene)
{
 case ´a´:
  Do_The_A_Thing();
  break;
 case ´c´:
  Do_The_C_Thing();
  break;
 case ´g´:
  Do_The_G_Thing();
  break;
 case ´t´:
  Do_The_T_Thing();
  break;
 default:
  printf ("Error: Unknown gene sequence %c\n",gene);
  break;
}
```

Figure 1.17: Using the switch statement

```
switch(input_character)
{
 case ´A´:
  Capital_A_Thing();
 case ´a´:
  Lowercase_A_Thing();
  break;
 case ´B´:
  ...
}
```

Figure 1.18: An example of “falling through” a case statement

1.2 Loops

Many times, you will want to execute a block of code more than once. You may have some variable that tells you how many times to execute a block of code. Or, you may want to execute a block of code until some boolean expression is false. All of these things are possible with loops. Loops are additional flow control structures that allow you to repeatedly execute a block of code until some specified condition happens. You will find that they are one of the most commonly used features of the C programming language. For those of you familiar with programming in general, you may be surprised to know that some languages do not have loops at all! C has three kinds of loops: While loops, Do-While loops, and For loops.

While loops

`while` loops are the second most common type of loops. They execute their block of code while their boolean expression is true. Let's say you have a variable called `num_times` that told you how many times you wanted the block of code to execute. You could do this with a while loop as shown in figure 1.19.

```
int i;

i = 0;

while(i < num_times)
{
  do_some_really_interesting_thing();
  i++;
}
```

Figure 1.19: A simple While loop

This loop will increment `i` by one each time the block of code executes. Once `i` is equal to `num_times`, the loop will terminate, and execution will resume after the closing curly brace of the loop.

It is important to realize that the boolean expression gets tested *before* each execution of the block of code. Therefore, if the boolean expression is false to begin with, the loop will not execute the block of code at all!

Do-While loops

Do-While loops are very rare. They basically are the same as While loops, but the test gets evaluated AFTER the block of code. This means that the block of code will always get executed at least once. Sometimes this is what you want. A simple Do-While loop is shown in figure 1.20.

```
do
{
  Some_Interesting_Block_Of_Code();
} while(Some_Boolean_Expression);
```

Figure 1.20: A simple Do-While loop

In general, you should avoid these loops unless you can't express what you want any other way. They are usually clumsy and don't get used all that much.

For loops

These are by far the most common type of loop. In fact, all While loops can be written as For loops and most of the time it makes sense to do so[1]. The theory behind For loops is that every loop has some initialization statement, a running condition, and an increment statement. In the while loop example shown in figure 1.19, we initialize the variable `i` to zero, the running condition is that `i` be less than `num_times`, and the increment happens right at the end of the loop. For loops allow you to combine all these ideas into one compact representation. A generic for loop is shown in figure 1.21.

```
for ( INITIALIZATION ; RUNNING CONDITION ; INCREMENT )
{
  Do_Something();
}
```

Figure 1.21: A generic for loop

The while loop example is rewritten in figure 1.22. As you can see, it is much more natural and compact.

[1]You can also rewrite all For loops as While loops, although why one would want to do that is beyond me.

```
for (i=0; i < num_times ; i++)
{
  do_some_really_interesting_thing();
}
```

Figure 1.22: A better way to write a loop