Variable Types

Whenever you use a variable in C, it expects you to first declare the type of variable that you want to use. We have already met variables of type `int` and `char`. There is one more basic kind of variable, other than these two, and all other types are relatives of these. The third basic type is the type `float` for storing floating point numbers.

You can think of a floating point number as one which has a sign, a decimal point and some exponent. The exponent can be used to conveniently specify large or small numbers. For example, the number 1237800000 can be written as 1.2378e9f. Here the `f` stands for floating point, and the `e9` stands for multiplication by 10^9 (which moves the decimal place 9 places to the right). Of course if the `e9` is replaced by an even larger exponent, then this number can be made even larger. An example of a small number is 0.0000000001243, which can be written as 1.243e-10f. If the exponent `e-10` (which moves the decimal point 10 places to the left) is made even more negative, then the number it is attached to will become tiny. Of course there is no need to specify an exponent at all with a floating point number. The numbers 0.1f, -1.246f and 12.0f are all perfectly good floating point numbers, and can be stored in a `float`.

The other C types allow you to specify variations on the three basic types. One variation is to specify types which are shorter (taking up less memory) or longer (for greater precision or for storing larger numbers).

The type `int` can be made shorter by prepending it
with the word \texttt{short}, and longer by prepending it with the word \texttt{long}. There is even a super long version where the word \texttt{long} is prepended twice.

To specify a floating point with higher precision, one can instead use the type \texttt{double} which as its name suggests, allows for floating point numbers with double precision (i.e. twice as many significant digits). In this case, one should not follow the number with the letter \texttt{f}, which is used solely for numbers of type \texttt{float}.

By default, all integer types in C are signed. Sometimes one wants to work with unsigned integers. When this is the case, one can prepend any of the integer types with the word \texttt{unsigned}. Initially it might not seem important to have separate unsigned and signed types. After all, if the largest whole number that can be saved as an \texttt{unsigned int} is 65535, then why not just specify that all integers above 32767 actually represent negative numbers, and all the others are positive. Actually this works if one is only going to add and subtract signed integers. However when multiplying and dividing signed integers, it fails. The computer must therefore treat signed and unsigned integers differently, thus the two different types.

Following are some examples of variable declarations in C. You will also observe that it is possible to assign values to variables at the same time as declaring their types. This is called initialization of the variables, and is always good programming practice. Some of the declarations also give examples of declaring arrays as we did in the previous program.

\begin{verbatim}
float radius = 2.0f;
long int a1 = 58683746;
short int b1 = 35;
char a, b, c;
unsigned long long int mylist[100];
\end{verbatim}
When making use of functions such as `printf` where a format string must be specified, each variable type has a code associated with it to be used in the format string, to specify that this particular type of variable will appear as a parameter. These codes specifying the variable types must be included in the format string in the same order that the variables appear as parameters.

The following code illustrates this.

```c
char alpha = 'd';
int num1 = 123;
float radius = 4.0f;
printf("The letter %c lives at %d North St. with the number %f.,",alpha,num1,radius)
```

The following table gives the format specifiers for the types we have introduced so far. In each case one can specify not only the type of the variable to be displayed, but the way that it will be displayed.

<table>
<thead>
<tr>
<th>Format Type Specifier</th>
<th>Variable Type</th>
<th>Displayed As</th>
</tr>
</thead>
<tbody>
<tr>
<td>d or i</td>
<td>int</td>
<td>decimal (base 10)</td>
</tr>
<tr>
<td>o</td>
<td>unsigned int</td>
<td>octal (base 8)</td>
</tr>
<tr>
<td>u</td>
<td>unsigned int</td>
<td>decimal (base 10)</td>
</tr>
<tr>
<td>x or X</td>
<td>unsigned int</td>
<td>hexadecimal (base 16)</td>
</tr>
<tr>
<td>f or F</td>
<td>float or double</td>
<td>decimal point but no exponent</td>
</tr>
<tr>
<td>e or E</td>
<td>float or double</td>
<td>decimal point and exponent</td>
</tr>
<tr>
<td>g or G</td>
<td>float or double</td>
<td>uses exponent if needed</td>
</tr>
<tr>
<td>a or A</td>
<td>float or double</td>
<td>hexadecimal (base 16) floating point</td>
</tr>
<tr>
<td>c</td>
<td>int or char</td>
<td>character</td>
</tr>
<tr>
<td>s</td>
<td>array of char's</td>
<td>character string</td>
</tr>
<tr>
<td>p</td>
<td>pointer</td>
<td>address in hexadecimal</td>
</tr>
</tbody>
</table>
We simplify somewhat here. Actually the format specifier \%c expects a variable of type int. However if one supplies a char this is automatically viewed as an int by C via a process called type casting which we will learn about later. Thus it does no harm at this stage to think of \%c as specifying a variable of type char. The usefulness of specifying an int is that you can display a character whose \textit{character code} is given by that int.

For the hexadecimal display formats, the difference between the lower and upper case format specifiers is that any letters in the hexadecimal number are displayed in lower or upper case respectively.

Again, all of the floating point types actually expect double arguments, however a variable of type float is automatically type cast by C to a double.

For the floating point types, computations such as 0 divided by 0 do not result in a number. The result of such a computation is displayed as \texttt{nan} or \texttt{NAN}, depending on whether a lower or upper case format specifier was used. Similarly if a large number is divided by a small number, the result may be too large to store and is treated as infinity. This is displayed as \texttt{inf} or \texttt{INF}, again depending on the case of the format specifier.

Sometimes when displaying or outputting data, one wishes to align the display of the data so it looks visually appealing. Since numbers may be of varying lengths, it is useful to know exactly how many characters have been output so far by the current output statement. This is done with the format specifier \%n. Instead of expecting a variable to be output, the number of characters displayed so far is put into the corresponding variable, which is expected to be of type int.

Of course if an int is short, long or long long you must tell C this too. One simply puts an h in front of the letter from the table above to specify short. (The
letter \( h \) is used since it is the second letter of the
word \textit{short}, and \( s \) is already used for something
else.)

One specifies \textit{long} and \textit{long long} in a similar way by
prepending \texttt{l} or \texttt{ll} respectively.

The inverse operation to displaying a character with
a given integer character code, is to display the
integer character code of a given character. This is
done by prepending \texttt{hh} to one of the format
specifiers which expects an \texttt{int}. This causes it to
expect a \texttt{char} instead, whose character code is then
displayed as an integer.

We summarize in a table.

<table>
<thead>
<tr>
<th>Format Size Specifier</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{h}</td>
<td>short</td>
</tr>
<tr>
<td>\texttt{l}</td>
<td>long</td>
</tr>
<tr>
<td>\texttt{ll}</td>
<td>long long</td>
</tr>
<tr>
<td>\texttt{hh}</td>
<td>(convert char to int code)</td>
</tr>
</tbody>
</table>

Of course C would not be very flexible if it did not
allow you to specify the number of digits that should
be displayed when outputing a number, or the
number of characters to display if outputing a
string. This is called the \textit{precision}. So far we have
merely shown how to construct format specifiers of
the following kind:

\[
\% \, \texttt{size} \, \texttt{type}
\]

However, numerous other optional details can be
specified, such as the precision. Along with all
optional components, a format specifier may look as
follows:

\[
\% \, \texttt{flags} \, \texttt{width} \, .\texttt{precision} \, \texttt{size} \, \texttt{type}
\]

The \textit{precision} is simply a whole number giving the
number of decimal places which should be displayed
for format types \texttt{a, A, e, E, f, F}, the maximum
number of significant digits for types \texttt{g, G, the}
minimum number of characters for types \(d, i, o, u,\) \(x, X\) or the maximum number of characters to be printed for type \(s\). Leading zeroes are added if an integer falls short of the specified minimum. The default precision for integer types is 1, whilst for floating point types except \(a, A\) it is 6. For the two exceptions it is taken to be sufficiently large to distinguish values of type \(double\) except when the exponent is a power of 2, in which case an exact value can be and is output.

If one does not wish to specify once and for all what precision to use in a particular output statement, one can specify the precision with a variable of type \(int\). Instead of specifying the precision with a number, one simply uses an asterisk "*" in its place and includes an \(int\) in the list of parameters of the output function, corresponding to that position in the format string.

Here are some examples making use of a format string with a precision specified:

```c
float scale = 1.57f;
int num2 = 123;
int prec = 4;
printf("Scale is %.5a\n",scale);     //Scale is 0xc.8f5c3p-3
printf("Scale is %.5E\n",scale);     //Scale is 1.57000525E+00
printf("Scale is %.5f\n",scale);     //Scale is 1.57000
printf("Scale is %.5g\n",scale);     //Scale is 1.57
printf("Scale is %.*f\n",prec,scale); //Scale is 1.5700
printf("Num2 is %.5d\n",num2);       //Num2 is 00123
printf("Num2 is %.50\n",num2);       //Num2 is 00173
printf("Num2 is %.5x\n",num2);       //Num2 is 0007b
```

The \(width\) specifies the minimum number of characters to be output for this format specification. If it falls short, spaces are used to pad the left hand side. If the width is a negative number, any padding is added to the right. Specifying the width is useful for producing output appearing in tables or tabbed rows. Again an asterisk can be used to specify that the width is given by a variable of type \(int\).

Here are some examples making use of a width:
Finally one has various flags to modify other behaviour. The minus sign of a negative width is already one such flag. It specifies that the data for the field should be left aligned instead of right aligned. The other flags are:

+ specifies for a number, that a sign should always be included, even if it is positive.

a space specifies that a space should be prepended to the output of a number if the sign is positive. Of course this flag is ignored if the previous flag is also used.

# specifies that a leading zero be used for an octal number (format specifiers o and O), a leading 0x or 0X be used for a hexadecimal number (format specifiers x and X) or that a decimal point always be included for the floating point types.

0 specifies that leading zeroes should be used to pad a number rather than spaces. Of course a "-" flag or a specified precision for an integer type trumps a 0 flag.

Here are some examples:

```
printf("Scale is %#.0f\n",scale); //Scale is 2.
printf("%5.d!\n",num2);  //Num2 is 123 !
printf("%+i\n",num2);     //Num2 is +123
printf("%#o\n",num2);     //Num2 is 0173
```

Finally we note that to output a % sign, one simply puts %% in the format string:

```
printf("Here is a %% sign.\n"); //Here is a % sign.
```

---

**Input Format Strings**

When using certain functions for input, such as `scanf`, one requires an input format string specifying the types of variables which data will go into, along
with any extra information the user must type to format the data.

The format specifiers that go into these format strings are very similar to those that are used for output format strings. A complete input format specifier looks as follows:

\[
% \ast \text{width size type}
\]

Of these fields, only the \% and type are compulsory.

The size and type fields are used in almost exactly the same way as for output format strings (see the tables above). This time, of course, they specify the type and size of the incoming data.

Two of the type specifiers are different for input however:

c specifies that the number of characters specified by the width field should be read. If the width field is missing, it is taken to be 1. Of course, to store a number of characters, one must use an array rather than a simple variable of type char. However, recall that with functions like scanf one must supply pointers to the variables used by appending ampersands to their names or by giving the name of an array. Thus the parameter requirements for functions like scanf are already in effect demanding that an array be used. One peculiarity with this format specifier is that no end of string, ",\"\0", character is added. Also, any whitespace which is input is also stored in the array, which is not the same as with the a format specifier which only considers non-whitespace characters to be input which is to be recorded in the corresponding array.

\ specifies that the input must only use characters from the set of characters following it (and before a closing \) to delineate the end of the set). The parameter used must again be an array of char's.
[^ specifies that the input must only use characters not in the set of characters following it (and before the closing square bracket).

In order to include a ] in the set of allowed/disallowed characters, just add a ] immediately after the opening square bracket followed by the other allowed/disallowed characters then the usual closing ].

Here are some examples of input strings illustrating these differences with format strings for output:

```c
char buffer1[10];
scanf("%10c",buffer1);
scanf("[\[][123,]\",buffer1);  //allows characters "[", "]", "1", "2", "3" and ","
```

The width field, when it is used, specifies the maximum number of characters which the user will enter for the corresponding variable or array.

The asterisk is used to specify that the variable corresponding to the current part of the format string should not actually be filled with the data which the user enters. This is basically used to ignore data of a specified type, coming from the user. This is useful when used in combination with the type specifiers [ and [^].