Ascii Values and Characters

Internally, characters are stored as integers. For example, each char is stored using 8 bits (0s and 1s). Technically, we could store values that ranged from 0 to 255 this way, or -128 to 127 if we chose to store negative numbers.

However, of these values, only some correspond to printable characters. The standard printable characters have ascii values ranging from 32 to 126. (See the table on page 609 of the text.)

To print the integer equivalent of a character, we can do the following:

```
printf("%d", 'a');
```

This will print 97, the ascii code for a lowercase a.

On the otherhand, if we have an integer and want to print the character with its associated ascii code, we can do:

```
printf("%c", 97);
```

We can also actually do arithmetic between characters. Consider the following (assuming that val is an integer, and ch is a character storing a lower case letter):

```
val = ch - 'a';
```

If ch were 'k', val would be set to 10. In general, this system would determine integer equivalents for all the letters, starting with a=0, b=1, ..., z=25. Somewhat surprisingly, this little "trick" has many uses when dealing with characters.

Escape Sequences

The special characters we have talked about before are ones with escape sequences. When denoted in string literals, we always start these with a backslash. Consider the list on page 176 of the text. You should try some of these out, but the most important are probably:

backslash, double quote, tab, newline, and the null character (this will be come very important once we start talking about strings later).

getchar() and putchar()

These two functions read and write a single character from the keyboard and to the screen, respectively. getchar takes no parameters and returns a char while putchar takes in a single character as a parameter and is void. When processing input character by character, these functions are preferred to using printf and scanf. In particular, the getchar function will not ignore any character, whereas the scanf may do so.

Macros in ctype.h

When processing characters, some pieces of information may be useful such as:

Is a character a letter?
Is a character a digit?
Is a character a lower case letter?
Is a character a white space character?
etc.

The ctype.h library provides some macros that are prewritten that make these computations.

You can essentially think of these macros as functions, but what makes them different is that they are fully "calculated" before run-time. They work very much like a #define, which simply replaces one piece of text with another.

Let's consider the following example that exchanges all lower case characters with their uppercase equivalents and vice versa:

```
while (c == getchar() != EOF) {
  if (isupper(c))
    putchar(tolower(c));
  else if (islower(c))
    putchar(toupper(c));
}
```

Common Errors and System Considerations

When dealing with character processing, it is better to declare the variable(s) used to read in characters as ints instead of char, due to portability reasons.

Also, in order to test the programs we have viewed today, since we haven't introduced files yet, we must pipe a file as input into our program. This can be done as follows on olympus:

progexec < inputfile

Similarly, if we'd like to pipe the output of our program to a file, we can do that as follows on olympus:

progexec > outputfile

Keep in mind, in order to do this, we must put a filename that does NOT exist on the right hand side of the > sign.

Keeping files for input and output and using the > and < signs can help speed up testing for any program over olympus. (jGRASP apparently does not allow you to pipe in a file as input or output.)