COMP1917: Computing 1

8. Characters and Arrays

Reading: Moffat, Section 7.1-7.5

ASCII

- The ASCII table gives a correspondence between characters and numbers
- behind the scenes, a `char` behaves just like a small number
- the ASCII codes are really just:
  - a way of specifying numbers inside a program

    `ch = 'A';` is the same as `ch = 65;`
  - a format for scanning and printing values

```
printf( "ch = %c and %d\n", ch, ch );
ch = A and 65
```

Integers as Characters

- when an `int` is printed in character format, the high-order bytes are ignored and only the lowest-order byte is printed.

```
int x = -191; // -256 + 65

// print x in char, decimal, unsigned and Hex format
printf("x can print as %c, %d, %u or %X\n", x,x,x,x );
```

```
x can print as A, -191, 4294967105 or FFFFFF41
```

Character Input / Output Functions

C provides special library functions for reading and writing characters, which are often more convenient than `scanf()` and `printf()`:

`getchar()`	read a character from standard input
`putchar()`	print a character to standard output

The “End of File” Problem

Problem:

- we want to be able to read not just text files but also data files, which may use all 256 possible characters
- we also need to know when we have reached the end of the input
- if we reserve a special character for “end of file”, we would only have 255 characters for actual data

“End of File”

Solution:

- all 256 characters from 0 to 255 are used for data
- an additional, negative number (normally −1) is used to indicate `EOF` (“end of file”)
- each character is initially read into an `int` variable
  (which can distinguish between 255 and −1)
- after checking that it is not `EOF`, it can be copied into a `char`.
- for this reason, many character functions in the standard library use `int` rather than `char`

getchar()

```
#include <stdio.h>

int getchar( void );
```

Returns character read from standard input as an `int`,
or returns `EOF` on end of file.

putchar()

```
#include <stdio.h>

int putchar( int ch );
```

Writes the character `ch` to standard output

Returns the character written, or `EOF` on error.

Example of `getchar()` and `putchar()`

Here is a program which uses `getchar()` and `putchar()` to copy standard input to standard output.

```
int main( void )
{
    int ch;

    // Note parentheses around assignment!!
    while (( ch = getchar()) != EOF ) {
        putchar( ch );
    }
    return 0;
}
```

Character Functions

```
#include <ctype.h>
```

Function	Checks
`isalpha(ch)`	`'a'..'z', 'A'..'Z'`
`isdigit(ch)`	`'0'..'9'`
`isalnum(ch)`	`'a'..'z', 'A'..'Z', '0'..'9'`
`isspace(ch)`	whitespace (space, linefeed, tab, etc.)
`ispunct(ch)`	punctuation symbols

See your text for others.

Counting different types of Characters

```
while(( ch = getchar()) != EOF ) {
  if(( ch >= 'A' && ch <= 'Z' )||( ch >= 'a' && ch <= 'z' )){
      letters++;
  }
  else if( ch >= '0' && ch <= '9' ) {
      digits++;
  }
  else if( ch == ' ' ) {
      spaces++;
  }
  else { // treat everything else as "others"
      others++;
  }
}
```

Counting Individual Characters

Now suppose we want to count the number of A's, B's, C's etc.

We could declare 26 variables `countA`, `countB`, etc. but this is cumbersome.

Is there a better way of doing it?

Arrays

An array is a collection of variables of the same type.

```
int count[26];
```

This creates a group of 26 “variables” which are numbered from 0 to 25:

```
count[0], count[1], ... count[25]
```

The elements of the array are stored sequentially in memory.

Array Initialization

Arrays can be initialized when they are declared:

```
int power3[6] = {
   1, 3, 9, 27, 81, 243
};
```

If not all values are specified, the rest will be filled out with zeros:

```
int count[26] = { 0 };
```

Warning: if an array is not initialized, it will contain "garbage" values.

Array Example

```
int count[26] = { 0 }; // array to count occurrences
int ch;                // character
int i;                 // index

while(( ch = getchar()) != EOF ) {
  ch = toupper(ch);    // convert to upper case
  if( ch >= 'A' && ch <= 'Z' ) {
     i = ch - 'A';
     count[i]++;       // increment i'th array element
  }
}
```

Multi-Dimensional Arrays

C also allows for multi-dimensional arrays.

```
// create a 2-dimensional array with 10 rows and 20 columns
double table[10][20];

// scan numbers into the array row-by-row
for( i=0; i < 10; i++ ) {
  for( j=0; j < 20; j++ ) {
     scanf("%lf", &table[i][j] );
  }
}
```

Multi-Dimensional Array Initialization

Multi-Dimensional arrays can also be initialized:

```
int magic[4][4] = {
   { 16,  3,  2, 13 },
   {  5, 10, 11,  8 },
   {  9,  6,  7, 12 },
   {  4, 15, 14,  1 }
};
```

Array Pitfalls

Beware: C will not warn you about an array index out of bounds!

```
i = -5;
count[i] = 47;
```

This could over-write other memory locations, causing your program to behave in strange ways.

Note that an array of size n is always indexed from 0 to $n-1$.
If the array `count[]` has 26 elements, don't try to access `count[26]` !