C Notes

Relation Between Pointers and Arrays

Space for static arrays in C is allocated on the stack when the function that defines them is called and is automatically deleted when the function ends, just like any other local variable. Global static arrays are maintained for the entire lifetime of the program, like other global variables.

Space for dynamic arrays is allocated by a function call to a memory allocation function, and remains in use until it is explicitly deallocated. C does not have an automatic garbage collector as in Java, for example.

There is an important relation between pointers and arrays in C. The similarity gives the programmer two different notations to describe array related code. Although the notations are similar, they are not identical. It takes some practice to use both notations correctly.

In the following code

```
int x[10];
```

a static int array of size 10 is created using array notation.

In this code

```
int *x;
x = (int *) malloc(sizeof(int)*10);
```

an int pointer is created and space for an array of size 10 is allocated. Interestingly, in either case, x can now be treated in code as an array of size 10.

```
x[0] = 0;
x[1] = 53;
x[2] = 181;
...
```

In the case of the dynamic array, the space must be deallocated when the program is finished with it.

```
free(x);
```

Also in the case of the dynamic array, elements of the array can be named using pointer notation instead of array notation:

```
*(x+0) = 0;
*(x+1) = 53;
*(x+2) = 181;
```

So x[i] in array notation is equivalent to *(x+i) if x is a pointer to an int array.

Strings in C make heavy use of both array and pointer notation, except that the basic data type is char not int. It is also used in conjunction with initialization with string constants. Look at the following declarations:

```
char x[80] = "message one";
char *y = "message two";
```

Both x and y can now be used as strings.

```
sprint("%s",x);
sprint("%s",y);
```

So what is the difference? In the case of x, the programmer has explicitly created an array of size 80, initialized with the contents of the string constant "message one". In the case of y, the pointer points directly to the string constant "message two". Since x refers to actual storage created by the program, its content can be changed later. In the case of y, it points to a constant string which can’t be modified by the program.

```
x[0] = 'M'; // okay
y[0] = 'M'; // runtime error
```

Quick Overview of C Strings

C doesn't provide a fundamental datatype to represent strings directly, instead it uses array of char or pointer to char. Unlike Java, C strings don't have a built in length value, instead they have a special null character terminator. In order to determine the length of the string, code must start at the beginning of the string and count characters until reaching the null character at the end.

```
char s[10] = "pqr";
```

This code creates a char array named s, initialized to the contents of the string constant "pqr" (keeping in mind that every C string has a hidden extra null character at the end). Here's another code that duplicates the previous code:

```
char s[10];
s[0] = 'p';
s[1] = 'q';
s[2] = 'r';
s[3] = '\0';
```

Failing to terminate a C string with a null character '\0' will result in a malformed string that will not work correctly with the functions of the C string library.

Quick Overview of C String Library

The header file <string.h> introduces a dozen or so useful function to operate on strings. Common operations on scalar values like ints have their equivalents for strings using library functions.

Int Operation	String Equivalent

`int x = 5;`
`int y;`
`y = x;`
	`char x[80] = "abc";`
	`char y[80];`
	`strcpy(y,x);`

`if (x<y)` …	`if (strcmp(x,y)<0)` …

Use the **strlen()** function to get the length of a C string. The **strlen()** function looks for the null character at the end of the string but does not count it as part of the length.

```
char *s = "pqrstu";
int x = strlen(s);            // x is 6
```

This raises an issue that leads to mistakes for newbie programmers. If you want to allocate space big enough for a specific string, the space needs to be length + 1. The extra character is required for the null character.

```
void f(char *s) {
      char *t;
      t = (char *) malloc(strlen(s)+1);
      …
      strcpy(t,s);
}
```

The parameter s represents an arbitrary string, the function needs to create a copy and store in dynamic char array t. The space allocated for the copy in t must be the length of the string s plus 1 for the null character at the end.

Input and Output

The simplest I/O is input from the keyboard and output to the display constrained to be characters (ASCII or Unicode). All programming languages provide a variety of functions to process characters in input and output streams. Internally, character data may need to be converted to a more appropriate internal type such as numerical data. So the programming language library will also provide library functions to convert text to numerical for input and numerical to text for output. The former is called parsing, the latter is called formatting.

There are two main styles of handing I/O and parsing and formatting it as needed. The first is to use high level convenience functions that do most of the work for the programmer. This style is called auto parsing and auto formatting. The scanf() function reads in character data and auto-parses it into the desired type. The printf() function takes internal data and converts it to text for output.

The other style is manual parsing and manual formatting. In this style, I/O operations use low level functions that only move text data but don't parse or format it. For example, the C function gets() gets a string from the keyboard but stores it as a string and does no further processing. In this style, the programmer will have to add additional code to finish the processing.

The reason both styles are needed is that they serve different purposes. Auto parsing of input with scanf() is easy and convenient, and most programmers will use it by default. But a program using auto parsing of input is inflexible and has little to no ability to respond to errors in the input. Manual parsing is more work, but the programmer has more control over how input is processed. In the case of unexpected input, manual parsing can recognize many error patterns and correct them in some cases, or at least recognize them and respond in a more user-friendly way. For auto parsed input, an error in the input will usually cause the program to crash.

Auto-parsed Input and Auto-formatted Output
Using scanf() for input and printf() for output is all that is needed if you want the input to be parsed automatically during input and formatted automatically during output. Auto-parsing and auto-formatting is good because it is simple and easy for the programmer. The negative is that it is not flexible, any small mistake in the input can cause the auto-parsing to fail and the program to crash.

Internal Parsing and Formatting
The alternative is for the programmer to read only strings from the keyboard and parse them internally. This creates more work for the programmer, but it gives the programmer the power to respond to input errors without crashing the program. Output formatting is not as sensitive as input parsing, so auto-formatted output is sufficient for most applications.

Pick one Style or the other, Don't Mix
We will do examples of each, pick which one is the most natural for your application. But don't mix the two styles in the same application. The two styles account for special characters like newlines differently. Switching styles in the middle of the app will cause newlines to be mishandled.

Manual Parsing
Manual parsing consists of the following steps

- Capture one line of input as a single string
- Tokenize or split the string into multiple tokens based on the position of separator characters like white space.
- Convert (parse) individual tokens from text to numbers as needed

Below I will describe how to do each step in C.

Capture Input as String
If you use scanf() to read in a string from the input, auto-parsing rules apply and the string will be terminated by the first occurrence of white space.

```
char x[80];
…
scanf("%s",x);          // input: the quick brown fox
printf("%s",x);         // output: the
```

To capture a full line of input – whatever was typed on the keyboard up to the Enter key – as a single string, spaces and all, you must use either the gets() (get string) of fgets (file get string) library functions.

```
char x[80];
…
gets(x);                // input: the quick brown fox
printf("%s\n",x);       // output: the quick brown fox
```

The **gets()** function has a well-known problem that it can't limit the amout of input to fit in the char array that will hold it. Code that contains such calls creates a buffer overflow security vulnerability. Note that a buffer overflow by itself does not give a hacker control of a machine, but it is a very important first step in a chain of events that could lead to it. Try this code out perhaps with a smaller array size. The program will run as long as the input has no spaces and is no more than a few characters longer than the array length. Any longer and the program will crash as extra data begins to overwrite important data elsewhere or at least triggers an illegal memory access exception. An alternative function that limits input to a specified size is **fgets()**. It requires an extra parameter to indicate the input source.

```
char x[80];
…
fgets(x,79,stdin);      // input: the quick brown fox
printf("%s",x);         // output: the quick brown fox
```

In this example, the amount of actual input will be restricted to 79 characters. The char array holding the input has space for 80 characters, but fgets() will add the newline character at the end of the input to the array as well, so we have to save room for it. The fgets() function can be used to read from a file in addition to the keyboard. The final parameter stdin represents the standard input, which is normally attached to the keyboard.

One final difference between gets() and fgets() is that fgets() also preserves the Enter key as a new line and adds it to the end of the string. If you don't want the new line char at the end of your string you'll have to manually remove it or overwrite it.

Here's a program that allows you to try out multiple input command options based on a command line parameter:

```
#include <stdio.h>

#define XSIZE 5

                        // declare main to accept cmd line args
                        // argc is the arg count (int)
                        // argv is the arg vector (array of string)

int main(int argc, char *argv[]) {

      char x[XSIZE];    // create string of size XSIZE
      int option;

                        // if the command appears on cmd line by itself
                        // argc = 1
                        // if there is an option value following,
                        // argc = 2

                        // if there is an option, we assume it is
                        // an int and use atoi() to parse it

      if (argc==2) option = atoi(argv[1]);
      else option = 1;

      if (option==1)      scanf("%s",x);
      else if (option==2) gets(x);
      else if (option==3) fgets(x,XSIZE,stdin);

      printf("%s\n",x);

      return 0;
}
```

Note in particular the following:

- Option 1 (scanf) only reads input up to the first white space
- Option 2 (gets) reads all input on the line even if it overflows x.
- Option 3 (fgets) truncates the input so that it fits in the indicated space

Tokenize Input

The C library provides a tokenize or string splitter function called **strtok**() in the string.h function library. This function must be called repeatedly on a string until all tokens have been found. Since strings in C are represented as an array of characters, a series of strings must be stored as a 2D array of characters. Each row represents a string, and the elements in a row represent the individual characters in a single string.

```
#define XSIZE 80
char x[XSIZE];          // string to hold one line of input
fgets(x,XSIZE,sdin);    // read in line of input
```

The tokenizing process will modify the string being tokenized, so it is recommended to make a copy of the string and tokenize the copy.

```
char y[XSIZE];
strcpy(y,x);            // note order of params: copies x into y
```

The strtok function is called repeatedly to tokenize the string. The initial call uses one pattern and the follow-on calls use another. Each time it is called, it returns a pointer to the next token. When the returned token is set to NULL, the tokenization is complete.

```
char *p;
p = strtok(y," ");      // tokenize y using " " (space) as separator
while (p != NULL) {
      printf("next token is %s\n",p);
      p = strtok(NULL," ");
}
```

Note the difference between the initial call and the follow-on call. For the initial call the first param is the string being tokenized, but for the follow-on call inside the loop the first parameter is NULL.

The program shown finds and prints out the tokens. A more common task is to store the tokens for future processing. Since each token is a string and a string is stored as an array of chars, what we need to store an array of strings is a 2D array of chars.

```
#define TSIZE 20
#define TCOUNT 10
char tokens[TCOUNT][TSIZE];
```

This is a char array with 10 rows of 20 chars each. One row represents a single string buffer that can hold up to 19 chars (save one char at the end for the null char). There are 10 rows so it can hold 10 strings. Use this kind of array to store all the tokens in one array. If the string contains fewer than 19 chars then the remaining chars on its row are wasted. If there are fewer than 10 strings then the extra rows are wasted. Creating an array of tokens that wastes no space is possible but will require more work: multiple passes through the input, dynamically allocated space, try it as an exercise.

To copy p into a specific row of the array, just refer to the array with only a first index but not a second. For example:

```
strcpy(tokens[0],p);
```

copies p into row 0 of the table.

Parse

Once we have captured a single token, it can be converted to its equivalent int or double value, assuming the characters in the string are restricted to the proper set of characters for that conversion. Use the library functions

- atoi(): ascii-to-int
- atof(): ascii-to-float, which, oddly enough, produces a double value

Example

```
char *p = "5437";
int x = atoi(p);

char *q = "3.14159";
double y = atof(q);
```

These functions perform a similar role as the Integer.parseInt() and Double.parseDouble() methods from the Java library.