Simple Input/Output

Each C program comes with three I/O streams:

$$\begin{tikzcd}
\text{stdin} \arrow[r] & \boxed{\text{program}} \arrow[r] \arrow[dr] & \text{stdout} \\
 & & \text{stderr}
\end{tikzcd}$$

The input stream is called standard-input (stdin), the usual output stream is called standard-output (stdout), and the side stream of output characters for errors is called standard-error (stderr). Internally they occupy file descriptors 0, 1 and 2 respectively.

This convention permits C programs to be connected together so that the output stream from one program can be filtered into the input stream for another via the pipe operator **|**. For example:

```
ls * | sort –r
```

That is, the standard output from "**ls ***" is being fed into the standard input for "**sort –r**" and the standard output from there appears on the output device (your monitor).

Sometimes these command line calculations can be involved:

```
cat infile | tr '[A-Z]' '[a-z]' | tr ' ' '\012' | sort
```

Translates the words in a file to lower case, then separates them into one word per line, finally sorting all the words.

getchar and putchar

The simplest I/O is to read and write one character at a time.

```
#include  <stdio.h>      /* getchar prototype in here  /*
#define BLANK ' '         /* there is a space between ' ' */

int c;

while ( (c = getchar() ) != EOF ) {
  if ( c != BLANK ) {
    putchar(c);
  }
}
```

Important points (King p. 121, 498/9, K&R p. 247):
One would expect the simple input/output functions (getchar/putchar) to be prototyped in the header as:

```
extern int getchar (void);
extern int putchar (int c);
```

From this, and King Appendix D, we can deduce that:
(a). getchar() and putchar() are both functions, returning an int
(b). extern tells us that both have been pre-compiled.
(c). void says that getchar() takes no parameters.
(d). putchar () takes a single parameter of type integer.
Not so clear here is that putchar() will accept a 'char' as its parameter, and that it returns either the output character, or EOF in case of error. Also not clear is that getchar() returns EOF if the read fails. See King.

The function getchar uses "out of band signalling" to return as EOF a value that cannot possibly be a legitimate character. Since all 8-bit characters are legitimate, the character is returned in a bigger field (e.g. as a 16-bit or 32-bit integer). Thus getchar returns an int whose right-most 8 bits hold the character and whose left-most 8-bits provide the "out-of-band signalling".

The function putchar returns the character just sent to stdout. Although this may seem unusual, it provides consistency with getchar and is also a recognition that in C there are no procedures, just functions. It can also be used as an error indicator, should putchar fail.

When is EOF sensed? After the I/O operation is tried.

scanf and printf

If we want to manipulate more than just one character at a time, we need to use the structured input/output routines.

```
extern int scanf (const char* format, ... &addrs);
extern int printf (const char* format, ... values);
```

From King Appendix D,
scanf() returns the number of items read. EOF is returned if the read fails, or if an end-of-file is detected before any data items are read.
const char* format means that the first parameter is a pointer to a "constant" array of chars. That is, a null-terminated literal string.

printf() returns the number of characters (bytes) written. If an error occurs a negative integer is returned.

Here is a program to read integers and add them up.

```
#include <stdio.h>
#ifndef TRUE
   #define TRUE 1
#endif

int main (void) {
    int cur_value;
    int sum = 0;
    int rcode;
    while( TRUE ) {
        rcode = scanf ("%d", &cur_value);
        if ( rcode == 0 ) {
            fprintf (stderr,
                "Received an invalid input from scanf\n");
            return (1); /* or exit (1); */
        }
        if ( rcode == EOF ) break;
        sum = sum + cur_value;  // sum += cur_value;
    } /* the break comes here */
    printf ("The sum is %d\n", sum);
    return (0);
}
```

These routines have much more complicated parameter processing - scanf generates a return code, and also retrieves an actual integer value. To understand parameter passing in functions we must understand pointers and how memory is referenced in C.

Pointers and Addresses

C has a simple memory model. Blocks of memory are organized as a sequence of bytes which can be manipulated individually or in contiguous groups. Each byte of memory has an address. The basic unit of storage is the 8-bit byte, and it is usually safe to assume that a char is exactly one byte.

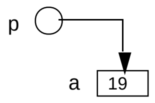
An address is stored in a pointer. This is an **unsigned int.**

Each datatype requires one or more bytes to store its value. Typically a character requires one byte (for the moment, but there are cases when you need more, for instance Japanese and Chinese Characters), an integer usually uses between 2 and 8 bytes, and so on. Thus not every byte address is a legitimate address of a data object (and this is true in all computers).

An example:

A pointer, p, to an integer data object, a, whose initial value is 19 is declared as follows:

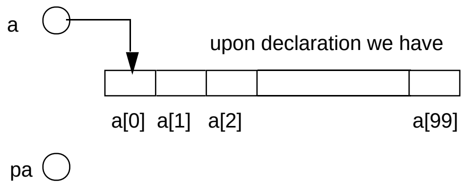
```
int a = 19;
int* p = &a;
```



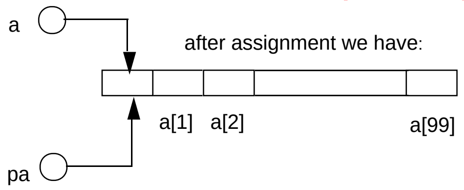
Arrays and Pointers

An array name is really a pointer to the first element of the array. For example:

```
int a[100];
int* pa;
```



```
pa = a;                 // or pa = &a[0];
```



Both pa and a point to the first element of the array of 100 integers. Thus both the following reference the same array element (the i'th element):

a[i] and *(pa+i) and even *(a+i) are all the same

The expression pa+i takes the value of the pointer pa and adds **i elements** to it. Thus pa+i points to the i'th element of the array a.

When we say "adds i elements", we mean adds the size of i objects!

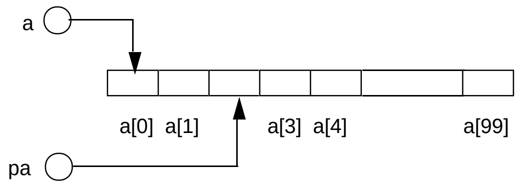
The * operator treats its operand as a pointer and retrieves the value at that address, thus *(pa+i) first computes the address of element i in array a, and then retrieves its value

The & operator is used to compute the address of a variable, for example:

Clearly a is the same as &a[0], and we can assign the address of any array element to a variable, e.g.

```
pa = &a[2];
```

stores the address of the 2'nd element of a in pa



So the expression

```
*(pa+2)
```

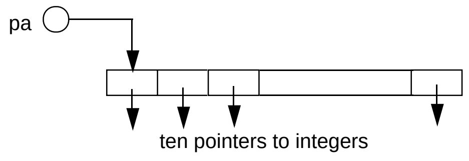
will now retrieve the value of a[4]

There is an important distinction between array and pointer declarations.

- An array declaration makes space for the whole array.
- **But a pointer declaration only allocates enough space to hold the pointer itself, no storage for the value pointed to is allocated**

We can of course have an array of pointers, it is declared in the following way

```
int* pa[10];
```



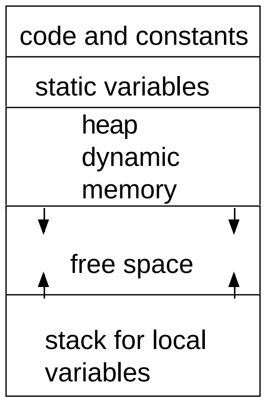
This produces an array containing 10 pointers to integers. See King P. 410-411 for "deciphering complex declarations"

Thus each pointer can have a value in the same range as an unsigned int, that is, in the range 0 to 2^32 –1.

The C and Unix Memory Model

Memory is a sequence of bytes, each byte with a specific address. The general large scale organization of memory for a C program running under Unix is as follows:

small addresses



large addresses

- When the stack and heap collide you are out of memory.
- Static variables remain unaltered between each function call.