Arrays and Structs and Pointers, Oh My!

Programming in C

- Input and output
  - Using printf
  - Standard input and output
- Pointers
- Arrays
- Structures
- Combining these things together…
Printing in C

- C has no “built-in” function to print out values
- Instead, printing is done via library calls
  - A library is a collection of function calls that someone else wrote and has provided
    - Similar to standard classes in Java
    - Standard C library is called libc
- Standard printing function in C is **printf**
  - Prints everything—numbers, strings, etc.
  - May be complex to use…

Using printf

- Printf takes a variable number of arguments
- First argument is *formatting string*
  - Tells printf what values to expect
- Remaining arguments correspond to specifications in formatting string
- Specifiers include
  - %d : integer (as decimal)
  - %s : string
  - %c : single character
  - %f : floating point number
- Specifiers may be modified
  - %10d : decimal using at least 10 characters
  - %-20s : string using at least 20 characters, left justified

```c
int x = 10;
char *s = "Ethan"
printf ("Hello, %s!\n", s);
printf ("x is %d\n", x);
printf ("%8s, x = %d\n", s, x);
printf ("%-8s on left.\n", s);
```

Hello, Ethan!
x is 10.
’ Ethan’, x = 10
‘Ethan ‘ on left.
More on printf

-Printf doesn’t automatically produce newlines
  -Must use \n to insert a newline
  -This can be done anywhere in a string
-Printf doesn’t check the types of arguments
  -Example:
    ```c
    char *s = "Hello";
    printf ("value is \%d\n", s);
    may produce
    value is 39640
    ```
  -Output might vary depending on where string is stored…
-Printf doesn’t check the number of arguments
  -Example: printf("Value is \%d") will not cause an error
  but won’t do anything useful, either

Single character I/O

-Printf is nice, but we often want to read and write single characters
  -Solution: getchar() and putchar(char c)
-Getchar gets a single character from standard input
  -Returns -1 if no characters remaining
  -Problem: must read character into an int rather than char
  to distinguish reading 0xFF from end-of-file
- Putchar writes a single character to standard output
- What are standard input and output?
Standard input & output

- In Unix, there’s a standard input stream and standard output stream
- By default
  - Standard input is keyboard
  - Standard output is terminal screen
- These can be changed!
  - Change standard input with `<inputfile`
  - Change standard output with `>outputfile`
  - May do both
- Example:
  - `asgn4.pl < plaintext > compressedfile`
  - This uses the file `plaintext` rather than keyboard for input
  - This sends standard output to `compressedfile`

Pointers

- C has many basic types
  - Integers: `char`, `short`, `int`, `long`
  - Floating point: `float`, `double`
- Each value has an `address`
  - Location in memory where the value is actually stored
- C allows the program to get this value with a `pointer`
  - Similar to a reference in Java
- Pointers are declared like this:
  ```c
  int *x;
  ```
  - This says that `x` is a pointer to an integer
- Pointers can be assigned like this:
  ```c
  int value = 5;
  x = &value;
  ```
- Pointers can be used (dereferenced) like this:
  ```c
  n = *x; /* will set n = number stored at location x */
  ```
More about pointers

- May declare a pointer to \textit{any} type in C
  - Includes built-in types!
- May do pointer arithmetic
  \begin{verbatim}
  int *p = &x;
  y = *(p+1)
  \end{verbatim}
  - This sets \( y = \) value in integer following \( x \) in memory
  - \( *(p+1) \) means “add the size of what \( p \) points to to \( p \), and dereference it”
  - No type checking…
- May assign to dereferenced pointer
  \( *p = z; \)
  - Doesn’t modify \( p! \)
  - Modifies what \( p \) points to


<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1004</td>
<td>17</td>
</tr>
<tr>
<td>1008</td>
<td>0</td>
</tr>
<tr>
<td>1012</td>
<td>53</td>
</tr>
<tr>
<td>1016</td>
<td>17</td>
</tr>
<tr>
<td>1020</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1012</td>
</tr>
<tr>
<td>1028</td>
<td>0</td>
</tr>
<tr>
<td>1032</td>
<td>53</td>
</tr>
</tbody>
</table>

What about arrays?

- Arrays are declared:
  \begin{verbatim}
  int arr[8];
  \end{verbatim}
  - This declares an array of 8 integers
  - Memory allocated for it, but not initialized
- Arrays are referenced:
  \begin{verbatim}
  arr[4] = 5;
  z = arr[4];
  \end{verbatim}
  - No bounds checking; \( arr[51] \) won’t signal an error
  - Not easy to find the length of an array
Arrays and pointers

- Arrays are implemented with pointers
  - `&(arr[0]) == arr == address of the first element of the array`
  - `arr[j] == *(arr + j)`
- Any integer can be added to a pointer
  - Negative ones, too!
  - As before, C doesn’t check to see if it looks OK

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accessing arrays

- Common to access arrays in loops
- Two forms of loops are common
  - Array index changes
  - Pointer to element changes (incremented)
- Both forms are OK (and equivalent)

```c
int arr[50];
int j;
int *p;
int sum = 0;

for (j = 0; j < 50; j++) {
    sum += arr[j];
}

for (j = 0, p = arr;
     j < 50; j++, p++) {
    sum += *p;
}
```
Strings in C

- Strings in C are simply arrays of characters
  - Strings must be terminated by NUL (\0)
  - This makes it impossible to include '\0' in a string…
- Since arrays can’t be resized, nor can strings!
  - They can be copied onto one another, though
- Strings can be declared like this:
  ```c
  char s[1][50];
  char *s2 = "test";
  ```
  - Now, s2 points to an array of at least 5 characters (including the trailing \0)
  - Some smart compilers may share storage for multiple strings initialized to “test”…
    - Don’t modify strings initialized the second way!

Structures in C

- A structure is a mechanism for grouping values together
  - Similar to class in Java
  - No associated methods!
  - May contain builtin types as well as pointers and other structures
- Refer to elements of a structure with “.” notation
  - Different notation for referring to elements of a structure being pointed to
- Useful for building complex data structures
  - Pointers may refer to memory used for a structure
  - How is memory allocated?

```c
struct treenode {
    int strLength;
    char s[8];
    struct treenode *l;
    struct treenode *r;
};

struct treenode root;
root.strLength = 5;
// Copy 5 characters into s
bcopy (value, s, 5);
```
Pointers and structures

- Pointers may point at structures
  - `nodep = &root;`
- Elements referred to with "->" notation
  - `(*p).strLength` is equivalent to `p->strLength`
- How is space allocated?
  - Library call `malloc`
  - Compute necessary memory size using `sizeof` function
    - Works on built-in types, too
    - May modify the value

```c
struct treenode {
  int strLength;
  char s[8];
  struct treenode *l;
  struct treenode *r;
};

struct treenode root;
struct treenode *nodep;
root.strLength = 5;
// Copy 5 characters into s
bcopy (value, s, 5);
p->strLength = 6;
bcopy (nValue, p->s, 6);
nodep = (struct treenode *) malloc (sizeof (struct treenode));
root.l = nodep;
```