UNIX and C Programming (COMP1000)

Lecture 4: Arrays and Strings

Updated: 19th August, 2019

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Chapter 7: Arrays**
  Note:
  - Chapter 7 also introduces the const keyword and enumerated types. Both are fairly simple, but are not covered in the lectures until lecture 9.
  - Section 7.10 deals with graphics again, which you can ignore completely.
- **Chapter 8: Strings**

This material **WILL** be assessed in the test!

Outline

- Arrays
- Arrays and Pointers
- 2D Arrays
- Array Parameters
- Strings
- Command Line
- Data Conversion

Arrays

- Like most languages (including Java), C has arrays.
- An arrays is a list of variables ("elements"), all having the same type, and a related purpose.
- Array elements are numbered, starting at zero.
- Array elements can be accessed by "indexing" the array.
- Once created, an array's length cannot be changed.

Declaring Fixed Arrays

- An array is declared like a normal variable, but with "[...]" after its name.
- In C, you create an array by declaring it (unlike in Java).
- You must provide the array length inside the square brackets.

Example

```
int intArray[10];
double doubleArray[100];
```

(In Java, you can also say "`int[] intArray;`". This is not allowed in C.)

Accessing Arrays

- Arrays in C are accessed one element at a time.
- You can access an array element just like a normal variable.
- Put the element to be accessed (the index) in square brackets after the array name.

Examples

```
intArray[2] = 10;    /* intArray[x] is an L-value */
```

```
intArray[3] = intArray[2] - 1;
```

```
int i;
for(i = 0; i < 10; i++) {
    intArray[i] = i * 2;
}
```

Array Length

- Keep track of the array length; e.g.:

```
#define NUM_ELEMENTS 10
...
int array[NUM_ELEMENTS];
int i;
for(i = 0; i < NUM_ELEMENTS; i++) { ... }
```

- Technically, you could do this:

```
int array[10];
int numElements = sizeof(array) / sizeof(int);
```

  - `sizeof(array)` gives the # of bytes in the array (40 or 80).
  - `sizeof(int)` gives the # of bytes in an `int` (4 or 8).
- Be careful! `sizeof` won't do this for *pointers* to arrays!

Fixed and Dynamic Arrays

- In C89, array lengths must be known before compilation and hard-coded.
- C99 introduces dynamic arrays.
  - Still fixed over the lifetime of an array.
  - However, the chosen length can be based on a variable, not known at compile time.

C89 Fixed Arrays

```
#define LENGTH 15
...
int array[LENGTH];
```

C99 Dynamic Arrays

```
int length;
scanf("%d", &length);
int array[length];
```

Bounds Checking

```
int array[10];
array[50] = ...; /* Out of bounds. */
array[-5] = ...; /* Out of bounds. */
```

- Newer languages (like Java) check your array indexes.
  - Must be 0 to length − 1.
- C does not check this.
- If you declare an array of 10 `int`s, C will not stop you accessing the 11th.
- This will access memory outside the array:
  - Possibly still *inside* your program – another variable – unpredictable effects!
  - Possibly *outside* your program – the OS will instantly kill your program – a "segmentation fault".

Array Initialisation

Often you want to initialise an array to all zeroes:

```
#define LENGTH 10
...
int array[LENGTH];
int i;
for(i = 0; i < LENGTH; i++) {
    array[i] = 0;
}
```

What if you want a particular set of values, rather than all zero?

```
int array[LENGTH];
array[0] = 23;
array[1] = 7;
...
array[9] = 349;
```

Array Initialisation (2)

There's a nicer way to initialise an array with pre-defined values:

```
#define LENGTH 10
...
int array[LENGTH] = {23, 7, 84, 34, 10,
                     14, 872, 332, 95, 349};
```

- Make sure that you give the right number of elements!
- This notation can *only* be used in the array declaration.
  - This is *not* the normal assignment operator.
  - Normally, the array as a whole cannot be assigned to.

Automatic Length

- You can omit the array length **if** you use the `{...}` notation[1].
- If you're using `#define` constants, this may be a **bad idea**!
- The following are equivalent:

```
int intArray[4] = {2, 4, 6, 8};
```

```
int intArray[] = {2, 4, 6, 8};
```

- The following will produce a compiler warning:

```
int intArray[2] = {2, 4, 6, 8};
```

[1] and you're declaring a *1D* array

Array Initialisation (3) — memset()

- Say we create this array: `int array[LENGTH];`
- Say we *do* want to initialise it to all zeros:

```
for(i = 0; i < LENGTH; i++) {
    array[i] = 0;
}
```

- We can *alternatively* use the `memset()` function:

```
#include <string.h>
...
memset(array, 0, LENGTH * sizeof(int));
```

- `memset()` sets all the *bytes* to a fixed value, usually zero.
- This effectively sets all the *elements* to zero too.
  - (Caution: this is really just an assumption that usually works!)
- "`string.h`" is where `memset()` lives; more on that later.

Arrays and Pointers (1)

- Array notation is an "add-on" to C.
- The name of the array is a pointer to the first element.
- `array[0]` is equivalent to `*array`.
- `&array[0]` is equivalent to `array`.

Example

```
int array[] = {10, 20, 30, 40, 50};
int* ptr = array;

printf("%d %d\n", array[2], ptr[2]);  /* Both 30 */
printf("%d %d\n", *array, *ptr);      /* Both 10 */
```

`array` and `ptr` both point to the first element of the array

Arrays and Pointers (2)

```
int array[] = {10, 20, 30, 40, 50};
int* ptr = array;
```

- However, an array only *looks* like a pointer variable.
- An array pointer is not stored (but simply calculated), so it can't be changed.
- `sizeof(ptr)` *does not* give the array length:
  - `sizeof(array) == 5 * sizeof(int)` (i.e. 20 or 40).
  - `sizeof(ptr)    == sizeof(int*)` (i.e. 4 or 8).
  - `sizeof` works at compile-time. In general, it can't possibly know what `ptr` actually points to.

Adding ints to Pointers

- Say you have `int i` and a pointer `p`.
- You can add `i` to `p`.
- The result is another memory address, `i` "elements" above `p`.
- An "element" is the data type pointed to by `p`, and may be several bytes long.

Example

```
double array[5] = {0.0, 1.1, 2.2, 3.3, 4.4};
double* ptr = array + 3;
printf("%f\n", *ptr);      /* Outputs 3.3 */
```

`array` by itself is a pointer. Adding 3 to it gives you a pointer to index 3.

Array Indexing With Pointers

- The square brackets are a short hand for two operations:
  - pointer arithmetic,
  - dereferencing.
- The following are equivalent:

```
someArray[i]
```

```
*(someArray + i)
```

  - Both expressions are L-values – they have a value plus a memory location to hold it.
  - Both can appear on the left of an assignment:

```
someArray[i] = ...;
*(someArray + i) = ...;  /* Equivalent */
```

Subtracting Two Pointers

- You can subtract two pointers to get their “distance” apart.
- Both pointers should point to elements of the same array.
- The result is an integer — the number of array elements separating the pointers.[2]

Example

```
double array[20];
double* x;
double* y;
int diff;
x = &array[2];
y = &array[10];
diff = y - x;        /* diff == 8 */
```

[2] The actual data type is technically `ptrdiff_t` — a type of integer defined by C solely for this purpose.

Malloc'd Arrays

- You can also create an array with `malloc`:

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

- This allocates a memory block *10 times the size of an int*.
- We keep track of it with an `int` pointer.
- It may not *look* like an array, but think about it:
  - We have space for 10 ints.
  - We have a pointer to the start; i.e. the first element.
  - Array indexing is just pointer manipulation.
- Once allocated, we can use this just like an ordinary array:

```
array[5] = array[4] + 2;
```

- For `malloc`'d arrays, we must clean up afterwards:

```
free(array);
```

Malloc'd Arrays — Why?

- Dynamic arrays in C89:
  - In C89, you must declare array sizes at *compile*-time.
  - We often don't know how big arrays should be until *run*-time.
  - `malloc()` gets around this — allocating an array without (strictly speaking) declaring one.
- Heap flexibility:
  - Malloc'd arrays remain on the heap until explicitly free'd.
  - This may occur in a completely different function, giving us flexibility in program design.
- Stack size limitations:
  - Fixed arrays are stack-based[3], but the stack is not meant for large amounts of data.
  - Large fixed arrays may overflow the stack.

[3] unless they're inside a malloc'd struct (see lecture 6).

calloc() (An Alternative to malloc())

- calloc() combines the malloc() and memset() functions.
- It takes two parameters, which it multiplies together:
  - The number of “elements” to allocate.
  - The size of each element (in bytes).
- calloc() allocates the memory, then zeroes all the bytes.

Example

```
#define LENGTH 10
...
int* array;
array = (int*)calloc(LENGTH, sizeof(int));
```

(Note the comma in place of a multiplication.)

Copying Memory — `memcpy()`

- `memcpy()` (in `string.h`) copies one block of memory to another.
- Takes source and destination pointers, and a block size.
- Copies the specified number of bytes from the source to the destination.

Overlapping blocks

- `memcpy()` assumes that the two blocks *do not* overlap.
- `memmove()` is effectively the same, but slightly slower and does not make this assumption.

memcpy() — Example

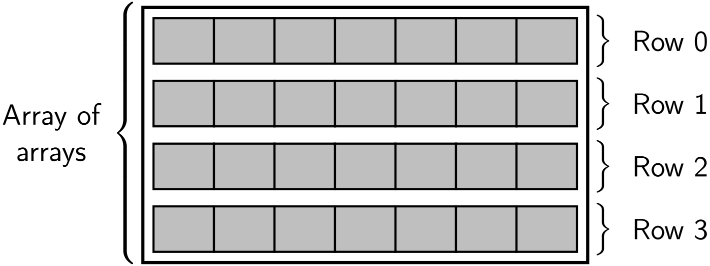
```
#define LEN 5

int main(void) {
    int stackArr[LEN] = {3, 6, 9, 12, 15};
    int* heapArr;
    heapArr = (int*)malloc(LEN * sizeof(int));

    /* Copy stackArr to heapArr */
    memcpy(heapArr, stackArr, LEN * sizeof(int));
    ...
}
```

2D Arrays

- Made up of rows and columns (effectively a matrix).
- Implemented as an array of arrays.
- Each "minor" array is one element of the "major" array.



Declaring 2D Arrays

- Use two sets of square brackets.
- Inside the first, put the number of rows.
- Inside the second, put the number of columns.

Example

```
#define ROWS 5
#define COLUMNS 8
...
int intMatrix[ROWS][COLUMNS];
```

Accessing 2D Arrays

- Again, use two sets of square brackets.
- Use both a row index and a column index.

Example

```
int intMatrix[ROWS][COLUMNS];
int i;
int j;
for(i = 0; i < ROWS; i++) {
    for(j = 0; j < COLUMNS; j++) {
        intMatrix[i][j] = i * j;
    }
}
```

2D Arrays in Memory

```
int intMatrix[ROWS][COLUMNS];
```

- What is "intMatrix[i]"?
- Remember that intMatrix is an array of arrays.
- Here, you're only indexing the major array.
- intMatrix[i] is the "name" of the 'i'th minor array; i.e. a pointer to its first element.
- intMatrix[i] is a pointer to the first element of row i.

2D Array Initialisation

- You can use the brace notation to initialise 2D arrays as well.
- Use extra braces around each row:

```
int intArray[2][3] = {{3, 4, 5}, {6, 7, 8}};
```

- The initialised 2D array will look like this:

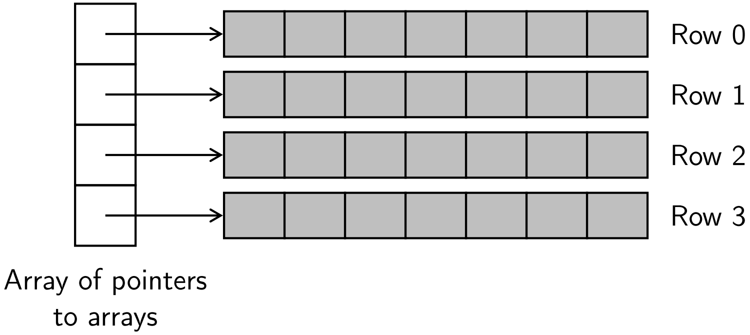
3	4	5
6	7	8

- 2D arrays are filled up row by row.
- You must always supply the number of rows and columns. The compiler won't try to guess the size here.
- However, you can omit the inner braces ("flat" as opposed to "fully-bracketted"):

```
int intArray[2][3] = {3, 4, 5, 6, 7, 8};
```

Arrays of Malloc'd Arrays (1)

- An alternative way of building a 2D array.
- We have an array of *pointers* to arrays (not an array of arrays).
- More flexible — we don't need to know the dimensions at compile time.



(Note: this is also how a 2D array works in Java.)

Arrays of Malloc'd Arrays (2)

These need to be constructed piece by piece:

1. Declare a double pointer to keep track of the array:

```
int** array;
```

2. `malloc` the array of pointers:

```
array = (int**)malloc(ROWS * sizeof(int*));
```

3. `malloc` each row array.
   - Use your algorithmic skills to figure this out!
   - Hint: you'll need a loop.

- Once constructed, you access this array just like a 2D array:

```
array[i][j] = 14;
```

Higher-Dimensional Arrays

- Follow the same pattern for 2D arrays.
- For a 3D array, use three pairs of square brackets when declaring/accessing.

```
#define ROWS 10
#define COLUMNS 15
#define BANANAS 20

float array[ROWS][COLUMNS][BANANAS];
...
array[i][j][k] = 25.0;
```

- Alternatively, you can use `malloc` as before.
- In practice, higher-dimensional arrays are hardly ever used.

Passing Arrays to Functions

- An array cannot be passed by value — only by reference.
- For 1D arrays, the following are exactly equivalent:

```
void func(float array[], int length) { ... }
```

```
void func(float* array, int length) { ... }
```

- You can pass *both* fixed *and* malloc'd arrays using *either* notation!
- In both cases, inside `func()`, `array` is a real pointer variable.
- Here, the `[]` notation actually creates a pointer, not an array.
- Always pass the array length, along with the array itself.

Passing Fixed 2D Arrays

- For 2D arrays, the malloc'd and non-malloc'd arrays differ!
- For fixed 2D arrays, use square bracket notation only.
- However, you must specify a fixed number of *columns*.

```
#define COLS 15
...
void func(float array[][COLS], int rows) { ... }
```

- You can optionally specify a fixed number of *rows* as well

```
#define ROWS 10
#define COLS 15
...
void func(float array[ROWS][COLS]) { ... }
```

- Here, `array` is a single pointer to a 2D array, not a double pointer!

Passing Malloc'd 2D Arrays

- For malloc'd 2D arrays, use double pointers only.
- Pass the dimensions as parameters.

```
void func(float** array, int rows, int cols) {
    ...
}
```

Passing Multidimensional Arrays to Functions

- For fixed multidimensional arrays, specify a fixed length for all dimensions except the first:

```
#define X 10
#define Y 15
#define Z 20
...
void func(float array[][X][Y][Z], int sizeW) {
    ...
}
```

- The first dimension is "open-ended" — the function can accept arrays having any size for the first dimension.
- All other dimensions must be fixed at compile time.
- For malloc'd multidimensional arrays, just add a "*" and a size parameter for each dimension.

C99 Smart-arsery

In C99, the compiler allows other parameters to determine the dimensions:

Example

```
int func(int rows, int cols, int arr[rows][cols]) {
    ...
}
```

- This allows you to pass fixed-size multi-dimensional arrays without knowing their size.
- Great, but not available in C89.

Strings as `char` Arrays

- In other languages (like Java), strings are a distinct data type.
- In C, "string" is just a name given to an array of `char`s.
- When you see "`char*`", it's (almost) always a string.

Null Termination

- C doesn't keep track of array lengths, so how does it know where a string ends?
- All C strings have a "null terminator".
- This is an extra character on the end, representing the end of the string.
- The character used is the "null" character.

The null character

- Has a value of zero.
- Is represented by `'\0'` (backslash-zero):

```
char nullCharacter = '\0';
```

- Cannot occur inside a string, only at the end.
- Should not be confused with a `NULL` pointer.

String Initialisation

- The following are equivalent:

```c
char s[] = {'H', 'e', 'l', 'l', 'o', ' ',
            'w', 'o', 'r', 'l', 'd', '\0'};
```

```c
char s[] = "Hello world";
```

Both create a character array `s` containing "Hello world".

- The next one gives you a read-only string:

```c
char* t = "Hello world";
```

  - Allocate read-only *global storage* for "Hello world".
  - Make `t` point to it.
  - `t` is *only* a pointer. There's no freely-modifiable array here.

Char Literals

- Enclosed in single quotes: `'A'`, `'3'`, etc.
- Really just an 8-bit integer; e.g. `'A' == 65`, `'a' == 97`, etc.
- The numeric equivalent is called the ASCII code.
  - A fixed standard mapping between symbols and numbers.
- Some special characters:

`'\n'`	New line
`'\r'`	Carriage return
`'\t'`	Tab
`'\e'`	Escape character
`'\''`	Single quote character: `'`
`'\\'`	Backslash character: `\`
`'\`*nnn*`'`	Octal character *nnn*, where $0 \leq n \leq 7$.
`'\x`*nn*`'`	Hexadecimal character *nn*, where $0 \leq n \leq \mathrm{F}$.

String Literals

- Enclosed in double quotes: `"Hello world"`.
- This is still a *pointer* to an array of `char`s.
- Can contain special characters:

```
"\tExample string with\n\"special\" characters"
```

- If you output the above string:

```
        Example string with
"special" characters
```

String Input/Output

- `printf()` can output strings using "%s".
- `scanf()` can input strings using "%*n*s" (e.g. %15s)
  - *n* is the maximum number of characters to read
  - `scanf()` will only read a single word at a time
  - Make sure to leave space for the null terminator!
- Why is the *n* important for `scanf()`?

Example

```
char input[21];

printf("Enter a word: ");
scanf("%20s", input);
printf("You entered '%s'\n", input);
```

You don't need "`&input`" for `scanf()` because `input` is a pointer.

The C String Library

- Includes functions for getting information from and manipulating strings.
- To use these functions, you need:

```
#include <string.h>
```

String Length — `strlen()`

- To count the number of characters in a string, use `strlen()`.
- Takes one parameter — the string.

Example

```
char* string1 = "";
char* string2 = "Hello world";

int len1 = strlen(string1);  /* len1 == 0 */
int len2 = strlen(string2);  /* len2 == 11 */

/* Why not use sizeof? Because... */
int x = sizeof(string2);  /* x == size of a pointer
                                (4 or 8) */
```

- *But* you don't need this to loop through a string!
- Just loop until you see the null terminator.

String Comparison — `strcmp()` (1)

- In C, the == operator does not compare strings.
  - "`str1 == str2`" checks whether two *pointers* are equal, not the strings they point to.
  - Something similar happens in Java, which uses the `equals()` method.
- Use `strcmp()` instead to compare strings.
- Takes two string parameters.
- *Does not* return true or false (as you might expect)!
- Instead, `strcmp()` returns:
  - a negative value, if the first string is "less than" the second (i.e. the first comes before the second in dictionary order);
  - zero, if the two strings are equal;
  - positive, if the first is "greater than" the second.
- (Since zero is considered `FALSE`, strcmp may appear to produce the opposite of the expected result.)

String Comparison — strcmp() (2)

Example

```
char input[21];

printf("Enter a word: ");
scanf("%20s", input);

if(strcmp(input, "Hello") == 0)
{
    printf("You said 'hello'");
}
```

The "== 0" tests if the strings are equal.

String Copying — `strncpy()`

- Sometimes you need copies of strings, if you modify them.
  - (Note: sometimes copying the pointer will suffice!)
- `strncpy()` takes three parameters:
  - A destination string (`char*`).
  - A source string (`char*`).
  - A maximum length (`int`), including the null terminator.
- Copies the second string (source) into the first (destination).
- Stops when it hits the maximum length.
  - You must set this to the amount of space available.

`strncpy()` vs. `memcpy()`

- Both copy blocks of memory.
- `memcpy()` copies *exactly* a given number of bytes.
- `strncpy()` copies *up to* a given number of bytes, ending in a null character.

String Concatenation — `strncat()`

- C does not understand "`string1 + string2`".
  - In other languages, this joins strings together.
  - In C, you're adding two pointers, which is meaningless.
- Use `strncat()` instead; same parameters as `strncpy()`.
- Appends to the destination, instead of overwriting it.
- The "maximum length" parameter refers to the *source* string.
- The destination must have this much *extra* space (not total space).

Example

```
char dest[12] = "Hello ";
char source[] = "world";  /* Six bytes */
strncat(dest, source, 6);
```

(Conceptually the same as "string1 = string1 + string2".)

String Searching — strstr()

- Locates a substring within a string.
- Takes two strings; searches for the second inside the first.
- Returns a pointer to the substring, if found.
- Returns `NULL` if no match occurs.

Example

```
char* bigString = "Hello world";
char* smallString = "wo";
char* substring;

substring = strstr(bigString, smallString);
```

Use pointer subtraction to get the index of the substring:
"`substring - bigString`" (6 in this case).

String Tokenising — `strtok()` (1)

- Tokenising breaks a string down into "tokens".
- Tokens are separated by single characters called "delimiters".
- In C, you can use `strtok()` to do this.

The right tool for the job?

- `sscanf()` or `fscanf()` are simpler and more powerful, if:
  - You know how many tokens to expect, and
  - You don't expect multi-word tokens (containing whitespace).
- `sscanf()` is discussed later in this lecture.
- `fscanf()` is discussed in the IO lecture (next).
- Use `strtok()` only when these conditions are not met.

String Tokenising — `strtok()` (2)

- Assuming `strtok` is the right choice...
- It takes two parameters — a string and a delimiter.
- Designed to be called multiple times:
  - The first time, you supply the string and delimiter.
  - Each subsequent time, you supply `NULL` and delimiter.

  (When `strtok()` is given `NULL`, it continues with the previous string.)
- Overwrites the delimiters in the original string with null terminators.
  - Destroys the original string, breaking it up into tokens.
  - Returns a pointer to the start of each token.
- Use `strncpy` to preserve the original string, if necessary.

Command Line Parameters

- Common practice to provide parameters to a program/command on the command line
- Virtually every UNIX command (`ls`, `cd`, `cp`, etc.) accepts parameters
- These parameters are strings, which are supplied to the program
- The first “parameter” is the name of the executable file

argc and argv

To access command-line arguments (parameters) from within your program, declare `main()` as follows:

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

```
int main(int argc, char** argv) {...} /* Equivalent */
```

- argc — argument count:
  - # of command-line arguments, plus 1 for the executable name.
- argv — argument vector:
  - An array of strings of length argc.
  - `argv[0]` is the executable name.
  - `argv[1]` is the 1st argument.
  - `argv[2]` is the 2nd argument.
  - ...
  - `argv[argc - 1]` is the last argument.

Example

Command-line

```
[user@pc]$ ./yourprogram eggs bananas pasta
```

yourprogram.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    for(i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Example Output

Based on the previous slide, the output would be:

```
./yourprogram
eggs
bananas
pasta
```

(Yes, "./yourprogram" is part of the output!)

Data Conversion

- Programs often deal with a lot of string/character data.
- You need to embed data into strings — "formatting".
- You need to extract data from strings — "parsing".
- Typecasting *cannot* do this.
- `printf()` and `scanf()` (and some variants thereof) are very useful for this.
- So far, we've barely touched on their capabilities.

Formatting With `printf()`

- The string you pass to `printf()` is the "format string".
- It contains ordinary characters to output, and also "conversion specifications": `%d`, `%f`, `%c`, `%s`, etc.
- Each specification is replaced with a parameter value:

```
printf("%s is %d years old.\n", name, age);
```

This will print (depending on `name` and `age`):

```
Fred is 108 years old.
```

- "%%" will output a single "%" sign.
- Specifications can also contain formatting information, encoded between the "%" and letter.

`printf()` Conversion Specifications — Example (1)

The following is a single `printf()` conversion specification:

```
%-+10.4f
```

The specification can be read as follows (from right to left):

`f`	The value is a floating-point number.
`.4`	The "precision" (no. decimal places) is 4.
`10`	The "field width" (minimum no. characters to output) is 10. The output is padded with spaces if necessary.
`+`	The output is always given a "sign" ("−" if negative, "+" if positive or either if zero)
`-`	The output is left-aligned inside the field width (by default, it is right-aligned).

All these components are optional, but the order is important.

`printf()` Conversion Specifications — Example (2)

Code	Output
`printf("%d\n",   97);`	97
`printf("%+d\n", 97);`	+97
`printf("%5d\n",  97);`	␣␣␣97
`printf("%05d\n", 97);`	00097
`printf("%-5d\n", 97);`	97␣␣␣
`printf("%-+5d\n", 97);`	+97␣␣
`printf("%f\n",    97.0);`	97.000000
`printf("%.2f\n",  97.0);`	97.00
`printf("%5f\n",   97.0);`	97.000000
`printf("%12f\n",  97.0);`	␣␣␣97.000000
`printf("%8.2f\n", 97.0);`	␣␣␣97.00

(Note: this is not exhaustive.)

Printing to Strings — `sprintf()`

- A variant of `printf()`.
- Does not display anything.
- Stores the formatted text as a string in memory.
- It takes an extra parameter — the location to store the string.

```
char full[100];
char* first = "Joe";
char* last = "Smith";
char middle = 'A';

sprintf(full, "%s %c. %s", first, middle, last);

/* 'full' now contains "Joe A. Smith"! */
```

(We could also have used `strcat()` several times, but this is cleaner and more flexible.)

Parsing Input With `scanf()`

- `scanf()` also accepts a format string, with text and conversion specifications (just like `printf()`).
- This string specifies what `scanf()` should *expect*, and in what order.
- This may include integers, reals, characters and strings.
- You can also tell `scanf()` to expect specific literal characters at specific points in the input.
- Thus, `scanf()` can read fairly complex data.

Spaces

- `scanf()` skips any spaces preceeding a conversion specifier.
  - . . . *except* for “%c”!
- If you put a space in the format string, `scanf()` will skip over *all* whitespace (if any) at that point.

scanf() — Example

- Say you want the user to enter a date, like "18-05-2017".
- You can use scanf() to "parse" this as follows:

```
int day, month, year;
scanf("%d-%d-%d", &day, &month, &year);
```

The format string "%d-%d-%d" tells scanf() to expect an integer, then a dash, then another integer, then another dash, then another integer.

- What about a complex number, expressed as "13.5 + 4.75i"?

```
double real, imag;
scanf("%lf + %lfi", &real, &imag);
```

(We must use %lf to read doubles.)

A Curious %lf Inconsistency

- `%lf` stands for "long float"; i.e. `double`.
- For `scanf`, `%f` reads `float`s only, while `%lf` reads `double`s.
- For `printf`, `%f` prints both `float`s and `double`s.

Why? (Just out of interest!)

- The inconsistency is due to subtle function call mechanics.
- `printf` and `scanf` are "variadic" functions — they don't have a fixed list of parameters.
- So, C has to *guess* the parameter datatypes.
- C converts ("promotes") any passed-in `float`s to `double`s.
- If you give `printf` a `float`, it actually receives a `double`.
- `scanf` takes *pointers*, and `float*` cannot be converted to `double*`.
- So, `scanf` has to distinguish between them.

Parsing Strings With `sscanf()`

- A variant of `scanf()` that reads from a string in memory, not the keyboard.
- Adapting the previous example:

```
char* date = "18-05-2012";
int day, month, year;

sscanf(date, "%d-%d-%d", &day, &month, &year);
```

  This *does not* read input, but parses the date as stored in a string.
- Why? Perhaps your input comes from the command line!

Parsing Single Numbers

- Say you have a string-representation of a single number.
- There are some simpler alternatives to `sscanf()`:
  - For integers — `atoi()`, `atol()` and `strtol()`.
  - For real numbers — `atof()`, `strtof()` and `strtod()`.
- They each take a string, and return an `int`, `long`, `float` or `double`.
- The `strtoX()` functions provide an error-checking mechanism:
  - They take an extra parameter — a `char` pointer passed by reference (i.e. a double `char` pointer).
  - On return, this points to the first non-numeric character of the string.
  - What does it mean if it points to the start of the string?

Coming Up

- Test 1 is next! (based on the first four lectures and tutorials.)
- The next lecture after the test will discuss input and output in more detail, including how to read and write files.