Introduction to Computer Science
Lecture 10

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Agenda

- Global Variables
- Arrays as Function Parameters
- const
- Strings
- Variable-Length Arrays
- main()'s parameters

Global Variables

- A variable declared outside all functions, at the top level, is *global*.
- A global variable can be read and modified by any function in the program.
- Here, `a` is global, and `b` is local to `main()`.
- Global variables can be used to pass information between functions of a program.
  - Their use is however strongly discouraged, as they lead to high coupling (dependency) between functions.
  - Typically, it is significantly more convenient to communicate functions through arguments/parameters and return values.

```c
#include <stdio.h>

int a = 33; // global variable

int main() {

    int b = 1 + a;
    a = a + 1;
    printf("%d %d\n", a,b);

    return 0;
}
```

Abstraction by Parameterization

- Abstraction is crucial for dealing with complexity in software development.
- Functions and procedures allow us to better decompose problems into subproblems, and programs into subprograms, exploiting a form of abstraction known as abstraction by specification.
  - Each function hides its implementation details from its users/clients, which can use the function by just concentrating on what it does, rather than on how it does it.
- The effective definition of functions requires the use of abstraction by parameterization
  - Abstraction by parameterization abstracts from the concrete identity of the data a function operates on, replacing it by parameters.
  - For better exploiting abstraction by parameterization, we need to be able to parameterize any kind of data, not limited to simple data types.

Arrays as Function Parameters: Example

- Programming languages generally allow us to define parameters of functions/procedures, even if these are of structured types.
  - In C, in particular, we can have array parameters.

```
int sum(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + array[i];
    }
    return result;
}
```

Arrays as Function Parameters 1/2

- The specific mechanism for array parameters in C is subtle.
  - when an array is passed as an argument to a function, the **address** of the array is passed.
    - It maintains the “pass by value” approach of C, but the address of the array is passed by value, not the whole array information.
  - The array elements themselves are **not** copied.
  - The function can still access the array elements with the `a[i]` notation.

```c
int sum(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + array[i];
    }
    return result;
}
```

Arrays as Function Parameters 2/2

- In C, arrays are simply contiguous blocks of memory. A function **cannot know** the size of an array, just from the array variable itself.
  - Typically, when having array parameters in functions, the size of the arrays has to be passed as additional parameters.

```c
int sum(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + array[i];
    }
    return result;
}
```

Calling a function with an array parameter 1/2

- Suppose `main()` declares an array `v` of size 100.
- The usual way to call sum() from main with array v to sum 100 elements is:

```
sum(v, 100);
```

- Note that we pass the name `v` alone, without `[]` notation.
- This is because we do not pass an element of `v` (`v[i]`), we pass the address of `v`.

```
int sum(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + array[i];
    }
    return result;
}
```

Calling a function with an array parameter 2/2

- The size parameter that is typically defined together with an array parameter can be instantiated in different ways, giving alternative ways of calling functions on arrays in C.
- The following table illustrates some possibilities to call function `sum()` from `main()` with the `v` parameter

```
int sum(int array[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + array[i];
    }
    return result;
}
```

sum(v, 100)	v[0] + v[1] + … + v[99]
sum(v, 88)	v[0] + v[1] + … + v[87]
sum(&v[7], k-7)	v[7] + v[8] + … + v[k-1]

const in array parameters

- While the array address is passed by value, the array itself can be considered to be passed “by reference”.
  - Thus, one may (accidentally or not) modify the contents of an array parameter.
- The use of the type qualifier "const" before a parameter in the parameter list can prevent the modification of the parameter.
  - For arrays, it prevents us from modifying the contents of a parameter array. For instance, this function copies n elements from src[] into dst[]:

```
void copy(const int src[], int dst[], int n)
```

Strings

- Strings in C are just arrays of `char` elements.
- By convention, a string is terminated by the end-of-string sentinel `\0`, or null character. The null character's decimal value is zero.
- To be more explicit, we call such strings "zero-terminated strings".
- Zero-terminated strings enable functions to take a string parameter without a size, and to process them until a zero is met.
- For instance:
  - printf("This is a quite long string and I can handle it");
  - printf("This one too.");

Example

```
int strlen(char s[]) {
    int i = 0;
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

String Literals

- String constants are written between double quotes.
  - For example, `"abc"` is a character array of size 4, the last element being the null character `\0`.
- String constants are different from character constants.
  - "a" and 'a' are not the same.
  - Array `"a"` has two elements, the first with value `'a'` and the second with value `'\0'`.

String Initialization

- Character arrays have an alternate notation:

```
char s[] = "abc";
```

is equivalent to:

```
char s[] = {'a', 'b', 'c', '\0'};
```

Example: Input String Stored in Array

- gets() lets the user type a string in the standard input of the program, and stores it (as zero-terminated string) into the array given to it as parameter (str)
- gets() cannot know the size of str, so a long input string can go out of bounds!
- for "real" programs, `gets()` is not recommended, the alternative is:

```
fgets(str, 1000, stdin);
```

```c
int main() {

    char str[1000];

    printf("Input your name:\n");
    gets(str);
    printf("Your name is: %s\n", str);

    return 0;
}
```

Variable-Length Arrays

- Arrays can be created based on a parameter size.
  - In this way, the actual size of the array will be determined at run time.
  - The size of the array does not change during an execution, but for different executions we may have different sizes for the array.

Variable-Length Array Example

```c
#include <stdio.h>
#include <assert.h>

int fib(int n) {
    assert (n >= 0);
    if (n == 0) {
        return 1;
    }
    else {
        int fibs[n+1];
        fibs[0] = 1;
        fibs[1] = 1;
        for (int i = 2; i <= n; i++) {
            fibs[i] = fibs[i-1] + fibs[i-2];
        }
        return fibs[n];
    }
}

int main() {

    printf("fib(%d) is %d\n", 10, fib(10));
    printf("fib(%d) is %d\n", 100, fib(100));

    return 0;
}
```

Pointers, a short overview 1/2

- A variable is stored at a particular memory location, or address, in the computer.
- If `v` is a variable, then `&v` is the location, or address, in memory of its stored value.
- The declaration: `int * p;`

  declares `p` to be of type *pointer to* *`int`*. It's a variable that holds a memory address where an integer is stored.
- p = &i; means that we store in p the address of variable i.

Pointers, a short overview 2/2

- Operator `&` (reference, direction) gets the address of some variable.
- Operator `*` (dereference, indirection) gets the value pointed by a pointer,
- So, `(*p)` evaluates to the value stored in variable `i`. , assuming p has the memory address of i.

Arguments of `main()`

- argc provides a count of the number of command line arguments
- Second argument, `argv`, is an *array of pointers to char*, but think of it as an *array of zero-terminated strings*. These strings are the words that make up the command line.
- Now, our programs can read their commandline parameters!

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

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
       printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```