Faculty of Computer Science, Dalhousie University *10-Oct-2018*

CSCI 2132 — Software Development

Lecture 15: Testing, Arrays in C

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Characters type (char)
- Type conversions: implicit and explicit
- typedef and sizeof keywords
- **Software Development Life Cycle (SDLC)**
- Waterfall model
- Rapid prototyping model

14.2 Software Testing

Software Testing and Debugging

- There will always be bugs (software errors)
    - obvious bugs, but also
    - sometimes nontrivial question:
      Is it a "bug" or a "feature" ?
- Testing: used to detect bugs
- Debugging: used to remove bugs

Software Testing

- Motivation
    - Ensuring robust software
    - Maintain reputation
    - Lower cost: Fixing a bug before release is always cheaper than after release
    - May be critical for security and privacy reasons, etc.
- There are job positions in testing
    - Software engineer in testing

What do We Test?

- Whether a program works
- In other words: whether it meets the specification
- Specification contains:
    - A description of input
    - A description of output
    - A set of conditions
    - Specifying what the output should be given input and conditions

How do We Test?

- Mindset
  - How to make the program fail?
- Typical test cases
  - Regular cases
  - Boundary cases
  - Error cases

Types of Testing

- White box testing
  - Use internal knowledge of implementation to guide the selection of test cases
  - To achieve maximum code coverage
- Black box testing
  - Use specification to guide the selection of test cases
  - To achieve maximum coverage of cases given in the specification

---

We distinguish *white box testing* and *black box testing* approaches. In white box testing, we use internal knowledge of an implementation to guide the selection of test cases. In black box testing, we do not use (or have) internal knowledge of an implementation, but use software specification to guide design of test cases. In white box testing, the tests are designed to provide a maximum code coverage, while in black box testing the tests are designed to provide a maximum coverage of specification requirements.

14.3 Software Debugging

Debugging

- **Debugging:** a methodical process of finding and reducing bugs, or defects, in a computer program
- The key step: Identifying where things go wrong
  - Track program state
    - Current location in the program
    - Current values of variables
    - Numbers of iterations through a loop
  - Find when expected program state does not match actual program state

Printf Debugging

- Idea: Use `printf` statement to print
  - Values of variables
  - Program location
- Example:

```
printf("Entering the second loop\n");
```

Strategies of printf Debugging

- The linear approach
  - Start at the beginning of the program adding `printf`'s
  - Until you reach the bug (state where your printout differs from what you expect)
- Binary search
  - Select half-way point
  - Determine if the bug has occurred

    - If yes, look in the first half
    - If no, look in the second half

Disadvantages of printf Debugging

- Time consuming for large programs
    - Modify program
    - Recompile
    - Rerun
- Possibly need to remove printf statements afterwards

Buffering issue with printf debugging. One issue to be aware of when using printf for debugging is that printf writes to the standard output, which is a buffered channel. It means that the content printed using printf is not necessarily printed right away, but printing may be delayed. The buffering of the standard output exists for efficiency reasons, but it makes printf sometimes unreliable as an indicator of how far a program got before a major problem happens. If we want to be sure that some output is printed as soon as the execution reaches that point, we can either flush the standard output, or use the standard error channel, which is normally flushed; i.e., not buffered. As an example, instead of using a printf statement like this one:

```
printf("x=%d a[%d]=%lf\n", x, i, a[i]);
```

we can use the standard error output as follows:

```
fprintf(stderr, "x=%d a[%d]=%lf\n", x, i, a[i]);
```

Sometimes a Better Approach: Use a Debugger

- Debugger — a tool that helps in debugging a program by running it in a controlled and transparent way
- Debugger usually provide ways to
    - step through the program
    - inspect variables
    - inspect wider program state (e.g., stack)
    - and some other functionallity
- Debuggers are frequently integrated into IDEs

GNU Project Debugger: gdb

- A symbolic, or source-level, debugger
- A program that allows programmer to
    - Access another program's state as it is running
    - Map the state to source code (variable names, line numbers, etc.: we need to compile with `-g` option)
    - View variable values
    - Set breakpoints

Breakpoints

- Internal pausing places in a program
- Breakpoints allow programmers to
    - Print values of variables
    - Step through code
    - Resume running the program until the next breakpoint

Commands

- Covered in more details in the Lab on gdb
- Notes about some commands
  - `break line_number`
  - `break function_name`
  - `next`: executes the next statement (function call = 1 statement)
  - `step`: executes the next statement, stepping into functions

Basic Operations

- Set breakpoints
- Examine variables at breakpoints or trace through code
- Until the bug is found
- Strategy: linear or binary search
- Advantage: No recompiling

15 Arrays in C

Reading: C book, Chapter 8.

Types

The basic types that we have learned so far are scalar types. A scalar type is a data type composed of a single element. In C, there are also aggregate types. An aggregate type is composed of multiple elements. The C arrays and structures are aggregate types.

15.1 One-Dimensional Arrays

Conceptually, a one-dimensional array is a fixed sequence of elements of the same type. Physically, a one-dimensional array is stored as consecutive bytes in memory.

To declare a one-dimensional array, we follow this syntax:

```
type name[size]
```

Here size must be an integer constant expression. For example, the following line of code declares an array named a that can store 40 int values:

```
int a[40];
```

When we declare an array, memory storage for the array is allocated. This is different from Java: In Java, we declare an array first without specifying its length, and we then use the new operator to allocate memory for it.

C arrays defined this way in a function are stored in stack memory. Recall that the memory space of a process is composed of four parts: code, data, stack and heap. As Java arrays are dynamically allocated, they are stored in heap memory.

Stack allocation is fast. In the stack memory, variable sizes are fixed and programmers cannot free any memory until the function returns. For heap memory, more management is required. That's why C arrays defined as above are allocated in the stack memory. We will learn more about how stack / heap memory is managed when we learn functions / pointers.

Note that in Java 6 (proposed in 2006, which was 11 years after Java was introduced), they introduced 'escape analysis' to speed up Java arrays. What this essentially does is to require the Java compiler to analyze whether a Java array can be allocated in stack or not. If it can, then it will be allocated in the stack. This improves the efficiency of Java programs significantly.

C programmers often define an array length as a macro. This way it is easy to modify the program. For example, we can define N to be 40:

```
#define N 40
```

After this, we can use N to declare arrays:

```
int a[N];
```

Array subscripting allows us to access each element of the array. For the array a defined above, its elements are `a[0],a[1],...a[N-1]`. When we use a subscript that is out of the range of indexes of an array (for example, if we attempt to access `a[40]` in this array), the behavior of the program will be undefined. This is because C does not require subscript bounds to be checked, as efficiency is important in C. In Java, this is checked, so a run-time error would be generated when we index an array out of range. This makes it easier to debug a Java program, but this sacrifices efficiency as checking this requires additional time.

The array name itself can be tought of as a variable that keeps track of the memory address where the first element of the array is. In this way, the index can be seen and the offset of how many entries we need to skip from the start to get to the appropriate array entry that we want. This is how the access is typically implemented in the machine code, and this is where the "index starts at 0" situation comes from.

An array can be given an initial value at the time it is declared, using an array initializer. For example, the following line of code declares an array of length 10 and stores the numbers 1, 2, 3, . . . , 10 in it:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Similar initializations are allowed for other basic types as well. The array length could be dropped if the initializer initializes all its elements. Hence the following line of code performs the same task:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The initializer can have fewer elements than the array length. The values in the initializer will be stored in the entries 0, 1, 2,... of the array, and the remaining array elements will be assigned 0. For example, the following line of code declares an array and stores the values 1, 2, 3, 0, 0, . . . , 0 in it:

```
int a[10] = {1, 2, 3};
```

We can take advantage of this to declare and initialize an array of all 0s:

```
int a[10] = {0};
```

When an initializer is not present, the array elements are not given initial values. This is different from Java, in which array elements are always assigned default values once an array is allocated.

The `sizeof` operator can be applied to an array to get the size of the array in bytes. If we want to find out the size of an array in the number of entries, this example illustrates how we can find it out:

```
int a[]={1,2,3};
int size = sizeof(a)/sizeof(a[0]);
```