CSCI 2132
Software Development

Lecture 15:

Testing, Arrays in C

Instructor: Vlado Keselj

Faculty of Computer Science

Dalhousie University

Previous Lecture

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

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

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

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

Arrays

- Reading: C book, Chapter 8
- Scalar types learned so far
  - composed of a single element
- Aggregate types:
  - composed of multiple elements
  - In C: arrays and structures

One-Dimensional Arrays

- One-dimensional array is
  - a fixed sequence of elements of the same type
- Syntax:

```
type name[size];
```

- Example:

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

- Unlike Java: cannot use `new` for dynamic allocation

Allocation of C Arrays

- Arrays allocation on stack
  - Remember process memory layout: code, data, stack, and heap
- In Java: arrays allocation in heap
- Java 6 (proposed in 2006) introduced ‘escape analysis’
- Effectively, compiler analyzes whether a Java array can be allocated on stack
- Efficiency reasons

Array Length

- Array length is frequently defined as a macro constant
- Example

```
#define N 40
```

- Then we declare the array:

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

- To access the elements of the array, we use:
  `a[0],a[1],...a[N−1]`

Array Boundaries not Checked in C

- Subscript out of range is not checked
- Example: defining array as

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

- and then accessing: `a[N]`
  - Leads to an error, which will go undetected by the compiler

Array Initialization

- Example:

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

- Size can be determined implicitly; e.g.:

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

- If initializer is shorter, the other elements get 0; e.g.:

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

- assigns 0 to the rest of elements
- Another useful example, to set all elements to 0:

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