Data Structures and Algorithms

Lecture 4

```
#define TRUE FALSE
//Happy debugging suckers
```

Debugging Tutorial

- The integrated debugger that ships with GCC is called GDB (GNU DeBugger)
- GDB is a command line tool, just like GCC
- There are several GUI interfaces for GDB: DDD, Netbeans, Eclipse, etc.
- Installing GCC will also install GDB
- To install DDD in Linux Mint, type `sudo apt-get install ddd` (Same for Netbeans/ Eclipse
- In order to be able to debug a C program, it needs to be compiled with debug symbols: `gcc -g test.c –o test`
- Running the debugger on a program means running the program in debug mode (this means it needs to compile first): `gdb test`

Debugging Tutorial (cont'd)

- Operations available in debug mode:
  - Run: runs the program
  - Breakpoint: stop the program at a specific line in the code, or when entering a specified function
  - Conditional breakpoint: stop the program at a specific line in the code, when a condition is met
  - Step over: executes a single line of code, stopping on the next line
  - Step into: executes a single line of code; if the line has a function call, it will stop on the first line of the function
  - Print: prints the value of a variable, if it is visible in the current context
  - Watch: watches a variable, updating its value every step
  - Continue: resumes the execution of the program until the first breakpoint, the program end or a segmentation fault
  - Backtrace: Prints the execution stack and the lines of code for every function call

" *If debugging is the process of removing software bugs, then programming must be the process of putting them in.* " - Edsger Dijkstra

Arrays

- The array is the simplest data structure – a sequence of elements of the same type
- The array variable name is a pointer to the first element in the array.
- There are two types of arrays:
  - Constant size, which are allocated either in program memory or on the stack
  - Variable size, which are only allocated manually, in the heap

<table>
<tr><th>Constant-Size Array</th><th>Variable-Size Array</th></tr>
<tr><td><pre>char g_string[10];

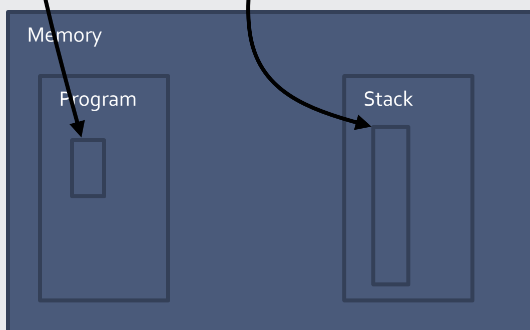
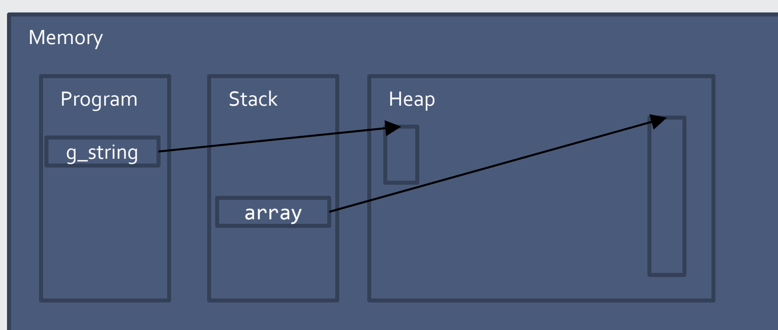
int main(){
    int some_array[20];
    return 0;
}</pre></td><td><pre>char *g_string;

int main(){
    g_string = (char*)malloc(10 * sizeof(char));
    int *some_array = (int*)malloc(2o * sizeof(int));
    //…
    free(g_string);
    free(some_array);
    return 0;
}</pre></td></tr>
</table>

Arrays (cont'd)

Constant-Size Array	Variable-Size Array

`char g_string[10];`
 	
`int main(){`
`    int some_array[20];`
`    return 0;`
`}`
	`char *g_string;`
	
	`int main(){`
	`    g_string = (char*)malloc(10 * sizeof(char));`
	`    int *array= (int*)malloc(2o * sizeof(int));`
	`    //…`
	`    free(g_string);`
	`    free(array);`
	`    return 0;`
	`}`

Arrays (cont'd)

Pros

- Allows constant time random access
- Data is in a continuous memory zone so when it is read sequentially, it is properly cached, optimizing access speeds

Cons

- Fixed size. Once an array is filled, a larger one needs to be allocated and data copied
- A continuous memory of the required size needs to be available in the memory
- Insertions and deletions are costly imply copying data back and forth

Arrays - Practice

1. Write a program that inputs a number N from the keyboard, creates an array of N elements of type int, initializes them with random values, then finds the maximum value and displays it on the standard output stream.

2. Write a program that inputs a number N from the keyboard, creates an array of N elements of type char, initializes them with random values, then computes the overall sum of all elements and displays it on the standard output stream.

3. Create an array of 52 char values, initializing them with values form 1 to 52. Write a function that will shuffle them randomly, then print the entire array on the standard output stream.

Arrays – Algorithms - Sorting

- Sorting is a fundamental problem in computer science, and it has countless real-life applications
- There are several sorting algorithms but only a few are actually used in practice due to having high performance
- We will study:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quicksort

Selection Sort

Algorithm: For each position in the array, find the minimum value of all elements between that position and the end of the array and swap that element with the one on the current position.

Performance:

- Worst case performance: $O(n^2)$
- Best case performance: $O(n^2)$
- Average case performance: $O(n^2)$

Practice:

Implement and run Selection Sort on a random generated array of data and measure execution time.

Bubble Sort

Algorithm: Compare each value with its next neighbor and swap if they are in the wrong order. Repeat until no swaps are made for a whole pass of the array.

Performance:

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$

Practice:

Implement and run Bubble Sort on a random generated array of data and measure execution time.

Insertion Sort

Algorithm: Iterate through each element in the input array and place it in the output array. Particularly useful for sorting "online" (i.e., can sort a list as it receives it).

Performance:

- Worst case performance: $O(n^2)$ comparisons, swaps
- Best case performance: $O(n)$ comparisons, $O(1)$ swaps
- Average case performance: $O(n^2)$ comparisons, swaps

Practice:

Implement and run Insertion Sort on a random generated array of data and measure execution time.

Merge Sort

Algorithm: Divide-et-impera approach consisting of splitting the array in two pieces, sorting the two pieces, then iterate through the two sorted sub-arrays and interleave them into a final sorted array. Particularly useful when merging two sorted arrays (thus the name Merge Sort).

Performance:

- Worst case performance: O(n log n)
- Best case performance:
  - O(n log n) typical,
  - O(n) natural variant
- Average case performance: O(n log n)

Practice:

Implement and run Merge Sort on a random generated array of data and measure execution time.

Quicksort

Algorithm: Divide-et-impera approach consisting of taking the first value of the array and use it as a pivot, moving it through several iterations so that it ends up in the correct spot in the array, having only smaller elements on the left and larger elements on the right. Then recursively apply quick sort on the two resulting segments.

Performance:

- Worst case performance: $O(n^2)$
- Best case performance:
  - $O(n \log n)$ (simple partition)
  - or $O(n)$ (three-way partition and equal keys)
- Average case performance: $O(n \log n)$
- Practice:

Implement and run Quicksort on a random generated array of data and measure execution time.

Thank you!

*Please and try and solve all practice exercises that haven’t been solved during the lecture.