Week 8 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Allocating multidimensional arrays
- Memory allocation from the system perspective
- Random numbers

Questions?

Project 4

Quotes

…One had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs.

Maurice Wilkes

Father of EDSAC

The first fully operational computer with its own memory

Example

- Dynamically allocate an 8 × 8 array of `char` values
- Loop through each element in the array
  - With 1/8 probability, put a `'Q'` in the element, representing a queen
  - Otherwise, put a `' '` (space) in the element
- Print out the resulting chessboard
  - Use `|` and – to mark rows and columns
- Print out whether or not there are queens that can attack each other

Debugging

printf() debugging

- A time-honored technique for debugging is inserting print statements into the code

```
int i = 0;
int count = 0;
for (i = 1 ; i <= 100; ++i); // Mistake
{
    printf ("i: %d\n", i); // See what's up
    count += i;
}
printf ("%d\n", count);
```

Problems with `printf()`

- Using print statements can be a useful technique
- However
  - Be sure not to actually change the state of the program with an **`i++`** or other assignment inside the **`printf()`**
  - It may not be available in some GUI programs or in deep systems programming
  - It might mess up the output of your program
  - Remember to remove your debug statements before turning in your code

Another approach

- It turns out that there are two kinds of output to the terminal
  - **`stdout`** (where everything has gone so far)
  - **`stderr`** (which also goes to the screen, but can be redirected to a different place)
- The easiest way to use **`stderr`** is with **`fprintf()`**, which can specify where to print stuff

```
fprintf (stderr, "Going to stderr!\n");
printf ("Going to stdout!\n");
```

Redirecting streams

- When you redirect **`stdout`**, **`stderr`** still goes to the screen

```
./program > out.file
This stderr output still shows up.
```

- This will be incredibly useful for debugging Project 4
- If you want to redirect **`stderr`** to a file, you can do that as well with **2>**

```
./program > out.file 2> error.log
```

Newline

- Whether using **`stderr`** or **`stdout`**, it's **critical** that you use a newline (**`\n`**) to flush your output
  - Otherwise, the program crash might happen before your output is seen
- **`printf()`** uses a buffer, but the newline guarantees that the output will be put on screen

```c
int* pointer = NULL;
printf ("Made it here!"); // Not printed
*pointer = 42; // Crash!
```

GDB

GDB

- GDB (the GNU Debugger) is a debugger available on Linux and Unix systems
- It is a command line utility, but it still has almost all the power that the IntelliJ debugger does:
  - Setting breakpoints
  - Stepping through lines of code
  - Examining the values of variables at run time
- It supports C, C++, Objective-C, Java, and other languages

Prerequisites

- C doesn't run in a virtual machine
- To use GDB, you have to compile your program in a way that adds special debugging information to the executable
- To do so, add the **-ggdb** flag to your compilation

```
gcc -ggdb program.c -o program
```

- Note: You will **not** need to do this on this week's lab

Source code

- GDB can step through lines of source code, but it cannot magically reconstruct the source from the file
- If you want to step through lines of code, you need to have the source code file in the same directory as the executable where you're running GDB

Starting GDB

- The easiest way to run GDB is to have it start up a program
- Assuming your executable is called **`program`**, you might do it like this:

```
gdb ./program
```

- It is also possible to attach GDB to a program that is running already, but you have to know its PID
- You can also run GDB on a program that has died, using the core file (which is why they exist)

Basic GDB commands

Command	Shortcut	Description
`run`	`r`	Start the program running
`list 135`	`l`	List the code near line 135
`list function`	`l`	List the code near the start of `function()`
`print variable`	`p`	Print the value of an expression
`backtrace`	`bt`	List a stack trace
`break 29`	`b`	Set a breakpoint on line 29
`break function`	`b`	Set a breakpoint at the start of `function()`
`continue`	`c`	Start running again after stopping at a breakpoint
`next`	`n`	Execute next line of code, skipping over a function
`step`	`s`	Execute next line of code, stepping into a function
`quit`	`q`	Quit using GDB

GDB tips

- Set breakpoints before running the code
- The print command is absurdly powerful
  - You can type **`print x = 10`**, and it will set the value of **`x`** to **`10`**
  - This kind of manipulation will be key to solving the next lab
- For more information, use the **`help`** command in GDB
- You can also list your breakpoints by typing **`info breakpoints`**

Structs

Structs

- A struct in C is:
  - A collection of one or more variables
  - Possibly of different types
  - Grouped together for convenient handling.
- They were called records in Pascal
- They have similarities to classes in Java
  - Except all fields are public and there are no methods
- Struct declarations are usually global
  - They are outside of **`main()`** and often in header files

Anatomy of a struct

```
struct name
{
        type1 member1;
        type2 member2;
        type3 member3;

          . . .
};
```

Why should we bother?

- Some data is naturally grouped together
- For example, a roster of students where each student has a name, GPA, ID number
- You could keep an array of strings, **`double`** values, and **`int`** values that corresponded to each other
  - But then sorting by GPA would mean moving values in three different arrays
- Also, we'll need structs for linked lists and trees

Java examples

- In Java, a struct-like class would be used to group some data conveniently
- Examples:

A class to hold a point in space

```
public class Point
{
        private double x;
        private double y;
        // Constructor
        // Methods
}
```

A class to hold student data

```
public class Student
{
        private String name;
        private double GPA;
        private int ID;
        // Constructor
        // Methods
}
```

C examples

- The C equivalents are similar
  - Just remember to put a **semicolon** after the struct declaration
- A string can either be a **`char*`** (the memory for it is allocated elsewhere) or a **`char`** array with a maximum size
- Examples:

A struct to hold a point in space

```
struct point
{
        double x;
        double y;
};
```

A struct to hold student data

```
struct student
{
        char name[100];
        double GPA;
        int ID;
};
```

Declaring a struct variable

- Type:
  - **struct**
  - The name of the struct
  - The name of the identifier
- You have to put **struct** first!

```
struct student bob;
struct student jameel;
struct point start;
struct point end;
```

Accessing members of a struct

- Once you have a struct variable, you can access its members with dot notation (**`variable.member`**)
  - Members can be read and written

```
struct student bob;
strcpy(bob.name, "Bob Blobberwob");
bob.GPA = 3.7;
bob.ID = 100008;
printf("Bob's GPA: %f\n", bob.GPA);
```

Upcoming

Next time…

- More on structs
- String to integer conversion

Reminders

- Keep working on Project 4
- Read K&R chapter 6