CS 240 Laboratory 6
Pointers and Introduction to gdb/valgrind

• Predict results of pointer code
• Write some pointer code
• Analyze incorrect code
• Start to use GNU debugger **gdb**
  - see what is going on “inside” a program while it executes
  - display values of variables and examine contents of memory
  - understand the effect of your programs on the hardware of the system
• Start to use **Valgrind** memory error detection tool to indicate problems with memory allocation/deallocation and access
**Pointers**

A *pointer* is a variable that contains the address of another variable.

Since a pointer contains the address of an item, it is possible to access the item “indirectly” through the pointer. For example,

```c
int x;
int* px;
px = &x;
```

means $px$ contains the address of $x$, or “points” to $x$.

Similarly,

```c
int y = *px;
```

means that $y$ gets the value stored at the address in $px$ (the value $px$ “points” to).

**Pointer Arithmetic**

If $p$ is a pointer, then $p++$ increments $p$ to point to the next element of whatever kind of object $p$ points to. So, the actual number by which $p$ gets increments is a multiple of the size in bytes of the object pointed to.

```c
int *p;
p++; 
```

results in $p$ being incremented by the size of an integer in bytes on the particular machine on which the operation is performed. If the word size is 32 bits, $p$ is incremented by 4. If the word size is 64 bits, $p$ is incremented by 8.
Multiple Dereferencing and Memory Models
The following declaration allocates space in memory for an array of pointers (specifically, 3 pointers to chars):

```c
char* commandA[3];
```

You can also dereference more than once with the use of multiple operators (remember that arrays and pointer can be used interchangeably). For example:

```c
char** commandPtr = commandA;
```

If the following statements were executed to initialize some strings (arrays of characters):

```c
commandA[0] = "emacs";
commandA[1] = "strings.c";
```

You could use the following diagram to model the data (the directed arrows indicate a pointer, or address):
Another way to understand how memory is organized here is to use our model of memory from lecture:
GNU Debugger (gdb)

Commands
Can be shortened to a single letter, or repeated by entering <return> at the prompt):

- Compile C program with -g option to create debugging information
- Run the program under gdb

$ gdb testprog

(gdb) run

- Set breakpoints

  (gdb) break main

- Step/next statement by statement through your program

  (gdb) step
  (gdb) next
  (gdb) cont -- continue execution
• Display/print code or values of variables and arguments

(gdb) list
(gdb) print x
(gdb) info locals
(gdb) info args

• (gdb) quit or Ctrl-d -- to exit.

• To find a bug:

  1. Set breakpoints at the start of every function
  2. Restart the program and step line-by-line until you locate the problem exactly.
  3. If program is stuck (infinite loop) Ctrl-c terminates the action of any gdb command that is in progress and returns to the gdb prompt.

• Execute statements/expressions during execution to tweak program execution state

  (gdb) set var i = 2

• Display/print binary and hexadecimal representation of variables and arguments

  (gdb) print /x result -- uses hex representation
  (gdb) print /t result -- uses binary representation