Tutorial on pointers, gdb and make utility

Topics Covered

- Quick recap of pointers
- Introduction to make
- Debugging programs using GDB

Avinash Gautam, CSIS, BITS Pilani

Quick recap of pointers

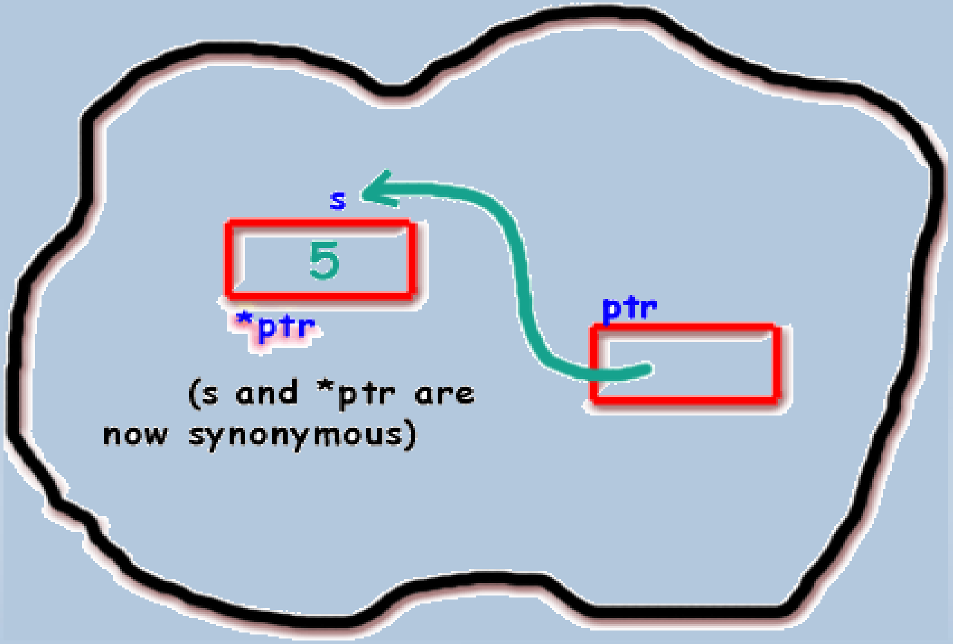
Quick recap of pointers

Objective

- Which pointer manipulations do most people readily understand?
- Which pointer manipulations do most people find challenging?
- Which errors do people most often make with pointers in their programming?
- How do people debug their programs?
- Do people even realize that there are bugs?

Quick recap of pointers

- What is a pointer?
  - Nothing but a memory address, more specifically the value of a pointer variable is always a memory address.



Pointer usage

- We can categorize the usage/ manipulation of pointers into three ways
  - Always safe
  - Dangerous
  - Never Safe

- We can put the pointer variable in four different states
  - Alive – The variable refers to memory that the storage management system has given to the program, i.e., the program “owns” that memory
  - Dead – The variable refers to the memory that the storage management system has never given to the program or has reclaimed from it, i.e., the program doesn't “own” that memory
  - Null – The variable refers to no memory locations at all
  - Out of scope – The variable is not in scope

Pointer usage (Contd.)

- Any pointer variable can be in one of the four states

  Alive, Dead, NULL, Out of scope
- Allowable transitions that a pointer (p) variable can undergo using different operators are as follows
- "declare" means the variable p is being declared
- "}" means the variable p is leaving the scope in which it was declared
- "*" means the variable p is being dereference, using either * or →
- "NULL" means the variable p is being assigned the constant NULL
- "malloc" and "free" are obvious

Allowable pointer operations

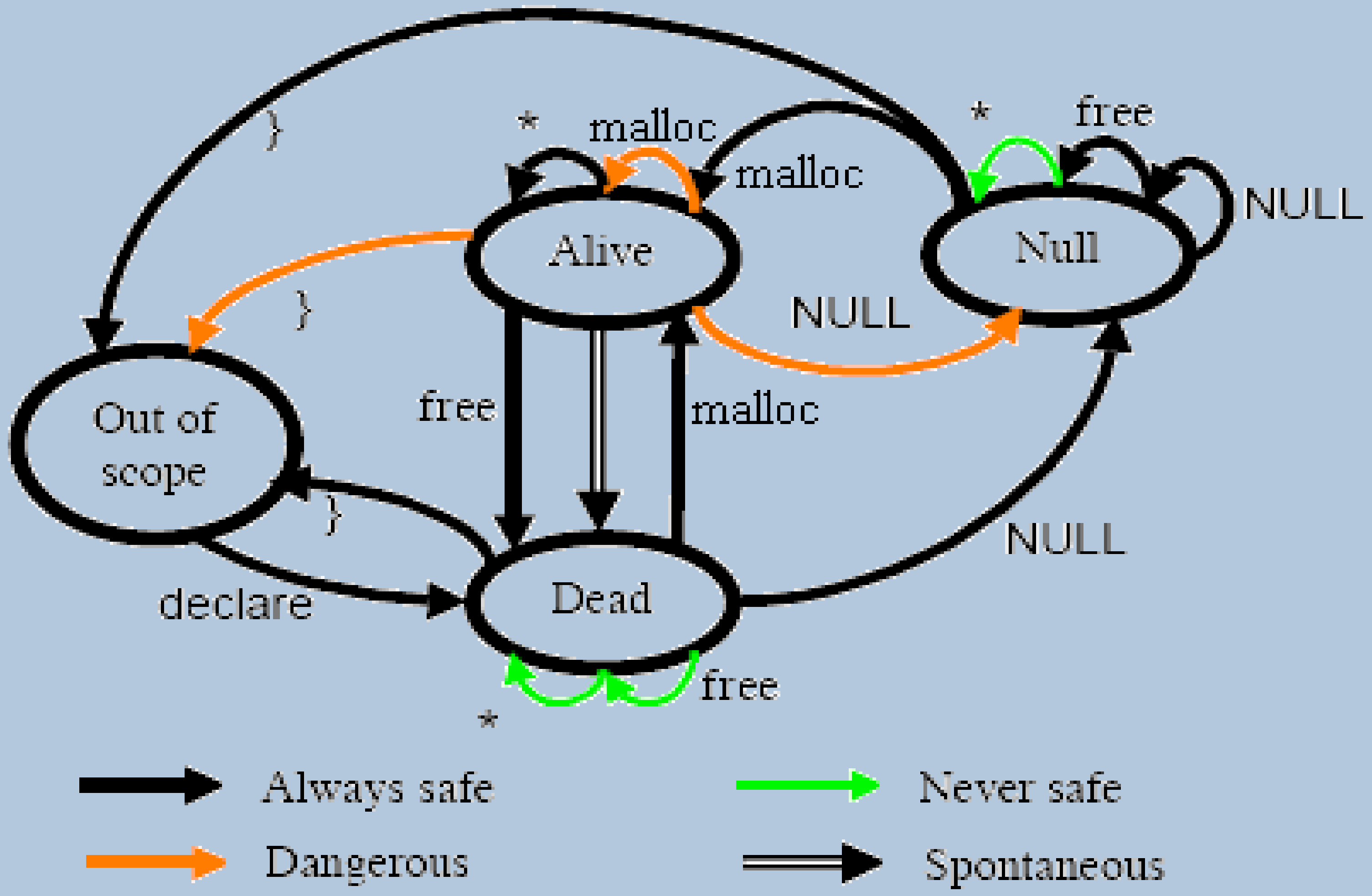
- Unary operations

  p = malloc(), free(p), *p, p->, p++, ++p, p--, --p

- Binary operations

  p = q (assignment), p == q, p != q

Pointer state transition



Pointer state transition (Contd.)

- Two more ways that a pointer variable “p” can change state
  - First, a variable “p” that is in scope can be assigned a pointer “q” that is in a different state, in which case the state of “p” changes to the state of “q”
  - Second, if “p” is alive and is aliased to another pointer “q” that transitions from alive to dead, then “p” also becomes dead.
  - This is the only spontaneous transition, i.e., the only possible change to the state of “p” that can occur as a result of a statement that doesn’t syntactically mention “p”; it captures an interesting effect of aliasing.

BEWARE OF SPONTANEOUS TRANSITIONS

Pointer state transition (Contd.)

- Never safe transitions also compile but produce difficult to debug code
  - Dereferencing a pointer that is NULL
  - Dereferencing or freeing a pointer that is a dead pointer

- Dangerous transitions may or may not result in memory leaks
  - If a pointer variable “p” is alive and it leaves scope, then whether there is a memory leak depends on whether the memory referenced by “p” is also reachable via another alive pointer that is aliased to “p”; similarly for statements p = malloc() and p = NULL;

Introduction to Make

Multiple File Example

Suppose you have three files: main.c, func1.c, func2.c

Compilation

```
$ gcc –c main.c                [gives a.o]
$ gcc –c func1.c               [gives b.o]
$ gcc –c func2.c               [gives c.o]
```

Linking

```
$ gcc –o execute main.o func1.o func2.o
```

[gives executable file with the name execute]

Remember if you change the source code in any of the files you must recompile the source code and re-link to produce the new executable

Multiple File Example

Have a shell script doCompilation containing the commands

```
gcc –c main.c
gcc –c func1.c
gcc –c func2.c
gcc –o execute main.o func1.o func2.o
```

Every time you change any of the files run the script doCompilation and a.out will be produced

Multiple File Example

Say main.c takes twenty minutes to compile
Say func1.c takes one minute to compile
Say func2.c takes 30 seconds to compile

First build takes 21.5 minutes to compile by doCompilation

Now let’s say you make changes in func2.c. Now you will run doCompilation again and that will hold you for 21.5 minutes again since the whole compilation would be redone !

You really needed to compile only c.c again – you already had the .o’s for a.c and b.c !

Multiple File Example - MakeFiles

- Solution: utility called “make”
  - The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them
  - make is not limited to programs
    - It can be used to do any task where some files must be updated automatically from others whenever the others change
  - make uses a text file called “Makefile”, which tells it what to do
  - makefile describes the relationships among files and provides commands for updating each file

Multiple File Example - MakeFiles

- Make utility
  - ➢ The make utility uses the makefile and the last-modification times of the files to decide which of the files need to be updated
  - ➢ For each of those files, it issues the commands recorded in the makefile

- Structure of a Makefile
  - ➢ A simple Makefile consists of Rules, which typically look as follows:

```
target: prerequisites dependencies
   commands
```

Multiple File Example - MakeFiles

- Targets
  - Targets are to be achieved/built by the make
  - A target is usually the name of a file that is generated by a program; examples of targets are executable or object files
  - A target can also be the name of an action to carry out, such as ‘clean’

- Prerequisites or Dependencies
  - Each target depends on other entities, A prerequisite is a file that is used as input to create the target. A target often depends on several files
  - Prerequisites may also be targets in a different rule

Multiple File Example - MakeFiles

- Commands
  - Commands to be executed to achieve the targets
  - A command is an action that make carries out
  - make invokes shell to execute rules:
  - They are executed by invoking a new sub shell for each command line
  - A rule may have more than one command, each on its own line
  - Please note: you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Multiple File Example - MakeFiles

```
execute : main.o func1.o func2.o
    gcc –o execute main.o func1.o func2.o

main.o : main.c
    gcc –c main.c

func1.o : func1.c
    gcc –c func1.c

func2.o : func2.c
    gcc –c func2.c
```

[Suppose the above file is named myMake]

```
$ make  –f  myMake
```

Multiple File Example - MakeFiles

- It checks for timestamps on targets
- Based on timestamps it finds out if the target is to be rebuilt or not
- Eg. If you change the source code of only func2.c, the timestamp of func2.c will be more recent than timestamp of c.o thus only c.c will be rebuilt resulting into fresh func2.o
- The above will change the timestamp of func2.o which will become more recent than “execute” thus “execute” will be rebuilt

Another Example - MakeFile

```
myexecutable: main.o x.o
    gcc -o myexecutable  main.o x.o

x.o: x.c
    gcc -c x.c

main.o: main.c
    gcc -c -I../include main.c

clean:
    rm *.o
    rm myexecutable
```

Another Example - MakeFile

Default target

- First Target is the default target

When you fire the make command *$make*

- It tries to build the first target defined in the “makefile”
- In order to build the first target, the make utility may need to build the dependencies first if they are also defined as targets in the makefile
- When you fire the command *$make clean* using the previous “makefile”, make would execute following commands

rm *.o

rm myexecutable

Macros in makefile

- A macro is a makefile variable
- We name a macro and assign a value to it. When make reads and executes the makefile, the macro is expanded to its assigned value
- Example:

```
CC = gcc
CFLAGS = -c –Wall
a.out: a.o b.o
	$(CC) a.o b.o
a.o: a.c
	$(CC) $(CFLAGS) a.c
b.o: b.c
	$(CC) $(CFLAGS) b.c
```

THANK YOU