EECS 2031 3.0 A Software Tools

Week 5: October 3, 2018

Next 2 weeks

- Next week - reading week
  - No classes, no office hours, no labs
- Week after
  - Monday or Tuesday - lab test. Open book, no human contact, covers all course material up to and including lab 4 and today's lecture. Bring your Pi, it must be able to connect to the network.
  - Wednesday - 1 hr midterm (no quiz). Closed book. Scantron.

C program structure

- Functions and variables defined in files
- Keyword static can be used to limit scope to the file.
  - Note: static has another meaning when used for local variables.
- Non static global structures should be coordinated through include files
- Use of 'extern' for global variables shared across files.
- Functions cannot define inner functions, but can be recursive.

C pre-processor

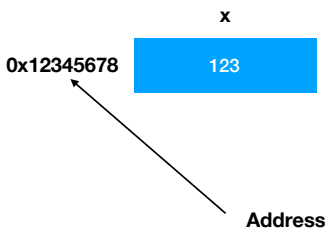
- # define allows you to define symbols (very useful for readability and consistency across files).
- Also allows for the definition of macros
  - #define foo(x) (x)->c
- Also allows for conditional inclusion
  - #ifdef FOO …. #endif

Software development

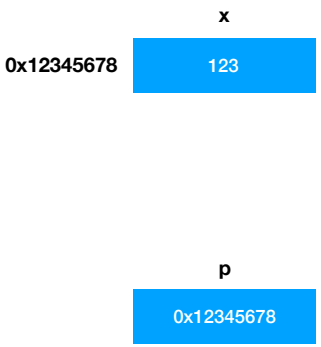
- Learn a good text-based editor (vi or emacs are obvious choices).
- These are tools that will work almost anywhere, and you can count on them.
- Very different philosophies as how to move from input to editing mode
  - vi - a/i input text … esc, :w :q to write and quit
  - emacs - every key is bound to something ^x^s to save ^x^q to quit
- You can have battles over which is better

How does a programming language map variables to memory?

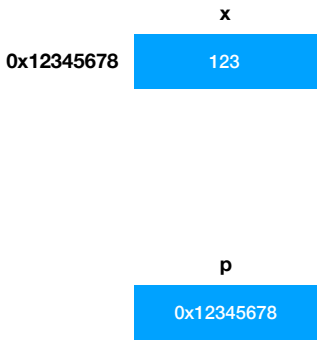
int x = 123;



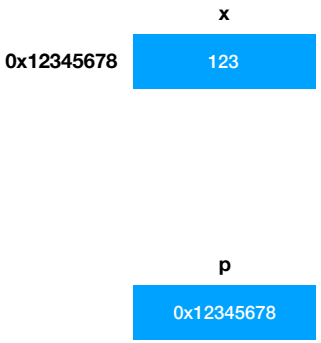
int *p = &x;



printf("0x%lx\n", p);

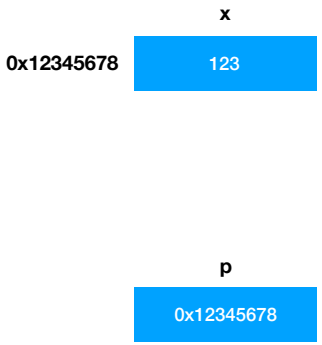


printf("0x%lx\n", p);



Prints 0x12345678

printf("%d\n", *p);



Prints 123

Pointers

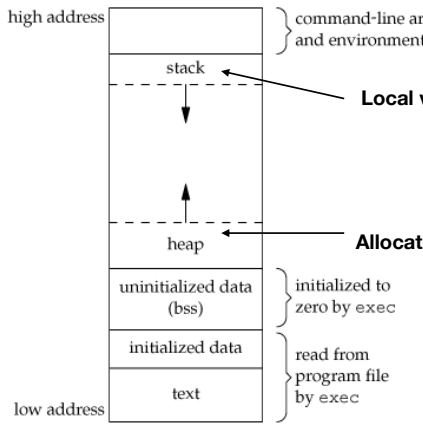
- int *p
  - Defines p to be a pointer of type int
- p++; p- -; p+=1; etc.
  - Increments p to point to the next (or previous) thing of this type. [pointers have a type]
- p=&x;
  - Sets p to the address of x.

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
->	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
-	Unary minus (Negation)		
++	Increment		
--	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
-	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, +=, *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

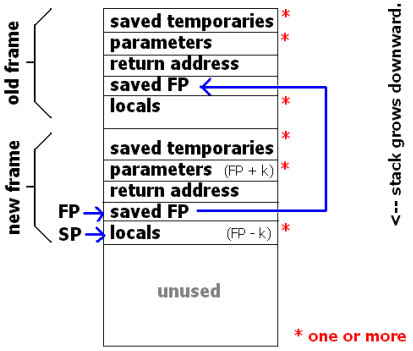
Table 5.1: Precedence and Associativity Table

++*p - increment what p points to
(*p)++ - increment what p points to
++p - increment p
***++p - increment p, then fetch value**

(parenthesis are your friend)




```
wandererecsyorkuca:foo jenkin$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```



<-- stack grows downward.

Stack holds
- **Parameters**
- **Local variables**
- **Reference to caller**
- **Return address**

Why does anyone want pointers anyway?

Java versus C

- Java has self-referential classes

```
public class Foo {
  public int value;
  public Foo next;
}
```

- C has pointers

```
struct foo {
  int value;
  struct foo *next;
}
```

Java versus C

- Java constructor

```
Foo z = new Foo();
z.next = null;
```

- C “constructor”

```
struct foo *z;
z = (struct foo *)
          malloc(sizeof(struct foo));
z->next = (struct foo *)NULL;
```

Java versus C

- Java destructor

```
z = NULL;
```

- C “destructor”

```
free(z);
z = (struct foo *)NULL;
```

Java versus C

- Java access element referred to by p
  - p.v;
- C access element referred to by p
  - p->v or (*p).v. First form is generally preferred

Java versus C

- Java printing out the list

```
for(Foo z=p;z!=null;z=z.next)
  System.out.println(z.value);
```

- C printing out the list

```
struct foo *z;
for(z=p;z!=(struct foo *)NULL;z=z->next)
  printf("%d\n", z->value);
```

If you don't like self-referential objects in Java…

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int x, *p;

   p=(int *)0;
   *p = 3;
   printf("got to here\n");
}
```

```
indigo 292 % ./foo
Segmentation fault (core dumped)
indigo 293 %
```

gdb

- Compile with -g, enables debugging information
- gdb foo
  - Runs gdb on foo
- Gdb has a very large number of commands.
  - Cannot review all of them here.

gdb commands

- list - lists (part of) your code
- break - sets a breakpoint
- info breakpoints - list breakpoints
- run - run the program
- delete - delete a breakpoint
- step - execute a step
- print - print a variable
- quit - quit

Learn gdb (or some other tool) before you need it.

```
#include <stdio.h>
#include <malloc.h>

struct Node {
  char v;
  struct Node *nxt;
};

struct Node *head = (struct Node *) NULL;


struct Node *add(struct Node *q, char v)
{
  struct Node *p = (struct Node *)malloc(sizeof(struct Node));
  p->v = v;
  p->nxt = q;
  return p;
}

void printList(struct Node *p)
{
  while(p != (struct Node *)NULL) {
    printf("%c\n",p->v);
    p = p->nxt;
  }
}


int main(int argc, char *argv[])
{
  head = add(head, 'a');
  head = add(head, 'b');
  head = add(head, 'c');
  printList(head);
  return 0;
}
```

Terrible code

```
gcc -g foo.c -o foo
pi@mypi:~/Documents/tmp$ gdb foo
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from foo...done.
(gdb)
```

Running gdb
Note: -g flag during compilation

```
(gdb) list 1,20
1       #include <stdio.h>
2       #include <malloc.h>
3
4       struct Node {
5         char v;
6         struct Node *nxt;
7       };
8
9       struct Node *head = (struct Node *) NULL;
10
11
12      struct Node *add(struct Node *q, char v)
13      {
14        struct Node *p = (struct Node *)malloc(sizeof(struct Node));
15        p->v = v;
16        p->nxt = q;
17        return p;
18      }
19
20      void printList(struct Node *p)
(gdb)
```

list

```
(gdb) break printList
Breakpoint 1 at 0x104d0: file foo.c, line 22.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000104d0 in printList at foo.c:22
```

Setting a breakpoint at printList (and verifying it using info)

```
(gdb) run
Starting program: /home/pi/Documents/tmp/foo

Breakpoint 1, printList (p=0x22028) at foo.c:22
22        while(p != (struct Node *)NULL) {
(gdb) list 20,24
20      void printList(struct Node *p)
21      {
22        while(p != (struct Node *)NULL) {
23          printf("%c\n",p->v);
24          p = p->nxt;
(gdb)
```

```
(gdb) list 20,24
20      void printList(struct Node *p)
21      {
22        while(p != (struct Node *)NULL) {
23          printf("%c\n",p->v);
24          p = p->nxt;
(gdb) print p
$1 = (struct Node *) 0x22028
(gdb) print *p
$2 = {v = 99 'c', nxt = 0x22018}
(gdb) print p->v
$3 = 99 'c'
(gdb) print p->nxt
$4 = (struct Node *) 0x22018
```

```
(gdb) list 20,24
20      void printList(struct Node *p)
21      {
22        while(p != (struct Node *)NULL) {
23          printf("%c\n",p->v);
24          p = p->nxt;
(gdb) print p
$11 = (struct Node *) 0x22028
(gdb) print *p
$12 = {v = 99 'c', nxt = 0x22018}
(gdb) print p->v
$13 = 99 'c'
(gdb) print p->nxt
$14 = (struct Node *) 0x22018
(gdb)
```

```
(gdb) list 20,24
20      void printList(struct Node *p)
21      {
22        while(p != (struct Node *)NULL) {
23          printf("%c\n",p->v);
24          p = p->nxt;
(gdb) print p
$11 = (struct Node *) 0x22028
(gdb) print *p
$12 = {v = 99 'c', nxt = 0x22018}
(gdb) print p->v
$13 = 99 'c'
(gdb) print p->nxt
$14 = (struct Node *) 0x22018
(gdb) print p->nxt->nxt
$15 = (struct Node *) 0x22008
(gdb) print p->nxt->nxt->nxt
$16 = (struct Node *) 0x0
(gdb)
```

So lets use the debugger to find a bug

```
#include <stdio.h>
#include <malloc.h>

struct Node {
  char v;
  struct Node *nxt;
};

struct Node *head = (struct Node *) NULL;


struct Node *add(struct Node *q, char v)
{
  struct Node *p = (struct Node *)malloc(sizeof(struct Node));
  p->v = v;
  p->nxt = q;
  return p;
}

void printList(struct Node *p)
{
  while(1) {
    printf("%c\n",p->v);
    p = p->nxt;
  }
}


int main(int argc, char *argv[])
{
  head = add(head, 'a');
  head = add(head, 'b');
  head = add(head, 'c');
  printList(head);
  return 0;
}
```

```
gcc -g bar.c -o bar
pi@mypi:~/Documents/tmp$ ./bar
c
b
a
Segmentation fault
pi@mypi:~/Documents/tmp$
```

Where did we die?

```
#include <stdio.h>
#include <malloc.h>

struct Node {
  char v;
  struct Node *nxt;
};

struct Node *head = (struct Node *) NULL;


struct Node *add(struct Node *q, char v)
{
  struct Node *p = (struct Node *)malloc(sizeof(struct Node));
  p->v = v;
  p->nxt = q;
  return p;
}

void printList(struct Node *p)
{
  while(1) {
    printf("%c\n",p->v);
    p = p->nxt;
  }
}


int main(int argc, char *argv[])
{
  head = add(head, 'a');
  head = add(head, 'b');
  head = add(head, 'c');
  printList(head);
  return 0;
}
```

```
(gdb) run
Starting program: /home/pi/Documents/tmp/bar
c
b
a

Program received signal SIGSEGV, Segmentation fault.
0x000104d4 in printList (p=0x0) at bar.c:23
23          printf("%c\n",p->v);
(gdb)
```

```
(gdb) list 20,30
20      void printList(struct Node *p)
21      {
22        while(1) {
23          printf("%c\n",p->v);
24          p = p->nxt;
25        }
26      }
27
28
29      int main(int argc, char *argv[])
30      {
(gdb) print p
$1 = (struct Node *) 0x0
```

p is null

So lets watch it happen

```
(gdb) list 20,30
20     void printList(struct Node *p)
21     {
22       while(1) {
23         printf("%c\n",p->v);
24         p = p->nxt;
25       }
26     }
27
28
29     int main(int argc, char *argv[])
30     {
(gdb) break 23
Breakpoint 1 at 0x104d0: file bar.c, line 23.
```

```
 (gdb) break 23
 Breakpoint 1 at 0x104d0: file bar.c, line 23.
 (gdb) run
 Starting program: /home/pi/Documents/tmp/bar

 Breakpoint 1, printList (p=0x22028) at bar.c:23
 23         printf("%c\n",p->v);
 (gdb) print p
 $1 = (struct Node *) 0x22028
 (gdb) print *p
 $2 = {v = 99 'c', nxt = 0x22018}
 (gdb)
```

So lets watch it happen

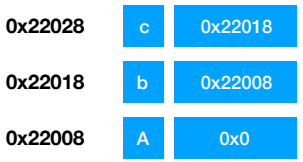
```
(gdb) list 20,30
20     void printList(struct Node *p)
21     {
22       while(1) {
23         printf("%c\n",p->v);
24         p = p->nxt;
25       }
26     }
27
28
29     int main(int argc, char *argv[])
30     {
(gdb) break 23
Breakpoint 1 at 0x104d0: file bar.c, line 23.
```

```
 (gdb) continue
 Continuing.
 c

 Breakpoint 1, printList (p=0x22018) at bar.c:23
 23         printf("%c\n",p->v);
 (gdb) print p
 $3 = (struct Node *) 0x22018
 (gdb) print *p
 $4 = {v = 98 'b', nxt = 0x22008}
 (gdb)
```

So lets watch it happen

```
(gdb) list 20,30
20     void printList(struct Node *p)
21     {
22       while(1) {
23         printf("%c\n",p->v);
24         p = p->nxt;
25       }
26     }
27
28
29     int main(int argc, char *argv[])
30     {
(gdb) break 23
Breakpoint 1 at 0x104d0: file bar.c, line 23.
```

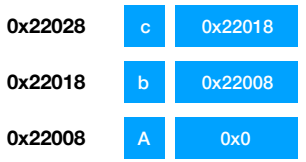
```
 (gdb) continue
 Continuing.
 b

 Breakpoint 1, printList (p=0x22008) at bar.c:23
 23         printf("%c\n",p->v);
 (gdb) print p
 $5 = (struct Node *) 0x22008
 (gdb) print *p
 $6 = {v = 97 'a', nxt = 0x0}
 (gdb)
```



So lets watch it happen

```
(gdb) list 20,30
20     void printList(struct Node *p)
21     {
22       while(1) {
23         printf("%c\n",p->v);
24         p = p->nxt;
25       }
26     }
27
28
29     int main(int argc, char *argv[])
30     {
(gdb) break 23
Breakpoint 1 at 0x104d0: file bar.c, line 23.
```

```
 (gdb) continue
 Continuing.
 a

 Breakpoint 1, printList (p=0x0) at bar.c:23
 23         printf("%c\n",p->v);
 (gdb) print p
 $7 = (struct Node *) 0x0
 (gdb) print *p
 Cannot access memory at address 0x0
 (gdb)
```



Summary

- C allows you to create variables that contain the address of another variable.
- It then provides mechanisms to manipulate this address including referencing the value stored at that memory address.
- Allows for construction of linked data structures (self-referential structures) as well as passing references to other data to functions.

Summary

- All (almost all?) environments provide some sort of debugger to allow you to investigate the structure of running code.
- Learn how to use one, well. Before you need it.
- Gdb is a standard, as are dbx and adb. They all work (more or less) the same way and have slightly different syntaxes.

Summary

- Reading week next week.
  - No classes, labs, or office hours
- Following week: lab test in your lab, midterm in class starting at 8:30
- Midterm: Scantron, pencil, closed book
- Labtest: You will need you PI, and be able to connect to the network.