Today

• Basic C programming

• Follow on to recitation
Structure of a C program

- A C program consists of a collection of C functions, structs, arrays, typedefs.

- One function must be called `main`:
  - `int main (int argc, char *argv[])`
  - `argc` is # of command-line args (>= 1)
  - `argv` is an array of `argc` “strings” (incl. program)

- There is no string type in C! These are “close”
  - `typedef char *string;
  - typedef char [] string;

- Optional or `char **argv`
Structure of a C program (cont’d)

- To run a program you simply type its executable name
  - To pass arguments you provide them on the command-line
- I have an executable program called mine
- In my login shell, I type:
  - `shell> mine -c 10 2.0 (./mine if paranoid)`

```
argv
| 0 | 'm' 'i' 'n' 'e' '\0' |
| 1 | '.' 'c' '\0' |
| 2 | '1' '0' '\0' |
...
```

- \0 null character
- C arrays start at 0
- argc = ?
Simple Exercise

• What will be the value of argc and argv inside the program argtest:
  
  ./argtest "-x $PRINTER a b c x"

  [run it, quote it]

• Why are command-line args useful?
  • Without them, how would you get inputs=>program?
  • Way to “automatically” pass parameters, i.e. a script

• Really useful call:
  
  x = atoi (argv[i]); // string to int
  y = x + 10;
Structure of a C program (cont’d)

• Functions may come from multiple source files and libraries or your own object modules (.o)
  • (e.g. /lib/gcc or /usr/lib/gcc)
  
  [run gcc -v]
  
  our compiler

• Types/constants/prototypes (signatures) are usually defined in header files (.h)

• Implementations go in (.c)

• Analogous to class defns & implementations in C++ or Java
Program Structure: Style #1

• A C program contains a set of “modules”
  • Separate files, separately compiled
  • Each contains functions
  • Common types, data-structures, function prototypes are in header files

```c
foo.h
#define MaxTokens 10
int sortit (char a[100]);
```

```c
foo.c
#include <foo.h> // like a macro
...
int sortit (char a[100]) {
    ...
}
```

```c
other.c
#include <foo.h>
int main () {
    ...
    y = sortit (...);
    ...
}
```

Link in foo.o (object file)
Program Scoping: Global

// allocated and available only to the file containing // this declaration
static int foo;

// allocated, global and exportable to any module
int bar;

// allocated elsewhere; declaration must be linked in
extern int baz;

Global variables get deallocated when?
Program Scoping: Local

```
int my_func (...) {
    int a;  // allocated new on the stack each call
    static int b=0; // allocated once, value stays!

    b++;
    b++;
    ...
    ...
}
```

Local variables get deallocated when?
What about statics?
Libraries and Include Files

• When you invoke a function, the compiler needs a prototype/signature for it
  • e.g. if you want to use `fopen`

```c
#include <stdio.h>
FILE *f;
F = fopen("/usr/jon/f.dat", "r");
```
Libraries and Include Files (cont’d)

• Function prototype is in `<stdio.h>`
• Usually functions themselves are in standard libraries, if NOT you must use:
  
  `-l<library-name>` when you compile

For example, `-lpthread`, `-lm

`stdio libraries (and others) linked in by default (libc.a)`
Compiling

• On most Unix systems, the compiler is gcc
  gcc -o foo foo.c
  • Compiles into a single executable named foo
  To run, shell> foo

• Multiple modules
  gcc -c foo1.c (produces foo1.o)
  gcc -c foo2.c (produces foo2.o)
  gcc -o foo foo1.o foo2.o -lpthread
  gcc -v -o foo foo1.o foo2.o // verbose
  gcc -o foo foo1.c foo2.c // ok, too
Error Handling: Style #2

#include <unistd.h>

// -1 returned if failure; sets errno (extern int)
int close (int fildes);

if (close (fildes) == -1)
    perror ("close failed ..."); // uses errno

GOOD style to check for errors in system calls!
The Ubiquity of 0

• In C and Unix, 0 is used a lot:
  • NULL is a synonym for 0
  • NULL often used to refer to a 0 pointer
• #define NULL 0

• NULL character that terminates a string: '\0' has ascii value of 0

• If a system calls takes an int flag, 0 is usually a safe default

• Don’t like 0 for logical NOT …
  • #define FALSE 0
  • #define TRUE 1
(Most) Programs shown in class?

http://usp.cs.utsa.edu/usp/programs.html
Memory Allocation

• The primary dynamic allocation function
  • `void *malloc (size_t size)`
  • Allocates size bytes, returns ptr (address) or NULL if memory not available

```
ptr1 = malloc (5);
ptr2 = (my_t *) malloc (sizeof (my_t));
```

Release allocated memory
```
void free (void *ptr_var);
```

VERY error-prone!
Memory Allocation (cont’d)

• The heap

<table>
<thead>
<tr>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global data</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
</tr>
</tbody>
</table>
void *ptr;
char *aptr;

// void can be casted to ANY pointer type and vice-versa

ptr = (void *) aptr;
aptr = (char *) ptr;

// void type means no return value or no args
void my_func (void);
Memory Leakage

- Your program **leaks** if its memory usage grows w/o bound
  - For what kind of program is this a problem?
  - Server: while (forever) { do something; }

- Happens if you forget to free memory not needed anymore

- Moral: don’t lose ptr to allocated memory!
  
a = malloc (100000);
a = 10;

- On program exit, OS reclaims memory
Buffer Overflow (Attack)

- Buffer overflow

```c
void func (char *buffer, ...) {
    char local[5];
    ...
    // string copy ... copies until \0
    strcpy (local, buffer);
    ...
}
```

You Bad guy calls it with a big string:

```c
func ("sjfh28&54NASTY_CODEw992385jsdh8");
```
Buffer Overflow (cont’d)

• You will clobber the stack
  • This will overwrite local variables and possibly the return address of the call!
  • If you are lucky the program just dies

• It can get worse ... attack
  • Suppose overwrite causes return address to be a location that now contains NASTY_CODE!

• Solutions?
  • Use strncpy and/or check length
  • Use non-ptr language (Java)
Why C?

Is it more fun to drive an *automatic* (your mom’s oldsmobile)

Or

A *manual* stick shift?
C crashes

• C program crash
• Segmentation violation
  • Program attempts to access memory outside its boundary
    ```c
    int a[10], *b;
    A[10] = 3;       // maybe cause an error
    A[-2] = 5;      // maybe cause an error
    *b = 6;         // for sure
    ```

To catch this you can run splint
[run crash and splint]

• Illegal instruction
  • Program attempted to execute an undefined or privileged machine instruction
    [usually a very bad memory overwrite]
Debugging

• Debugging 101: the `printf` and debugging levels

```c
#ifdef DEBUG
    printf (stderr, "A=%d\n", A);
#endif
```

```
gcc -o foo foo.c -DDEBUG
```

Can set multiple levels: `DEBUG1`, `DEBUG2`, ...

Several preprocessor directives:

```c
#include, #define, #ifdef, #ifndef
```
Time Permitting ...
Unix/C tools: Debugging

• Use **gdb**: GNU debugger
  • There are many others
  • Set breakpoints, look at vars, step, trace
  • Recommend you learn this!

```
gcc -g -o crash crash.c
```

[run gdb]
Unix/C tools: System Monitoring

- **ps**: tells you the state of a running program
  - **R**: running, if always R, maybe an infinite loop
- **ps -lu <uid>**
  - Shell commands have many flag options
- **top**: shows complete information and dynamically updates

[top -u jon + loop]
Unix/C tools: Makefiles

- Make builds programs by processing a dependency tree
  - It is a set of rules that describes dependencies and how to resolve them
  - Uses time-stamps
    
    ```diff
    foo.o: foo.c foo.h
    gcc -c foo.c
    ```

- Each action line begins with a TAB
- Default makefile is called makefile
**Make is your friend**

- Be aggressive with recompilation

- Strange bugs can be resolved by recompilation

```make
make clean
make all

clean:
    rm *.o

call:
    gcc ..
    gcc ...
```