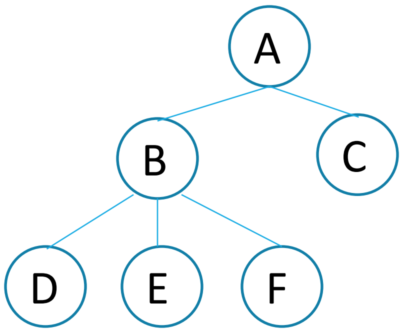
CSci 4061
Introduction to Operating Systems

Programs in C/Unix:

Chapter 2 (R&R)

Operating System Concepts: Process



- Process is an executing program: container for computing resources (abstraction)

Structure of a C program

- A C program consists of a collection of C functions, types and variable declarations, e.g. `structs, [], typedefs`

all main args are optional
`char **argv`

- One functions 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 name)

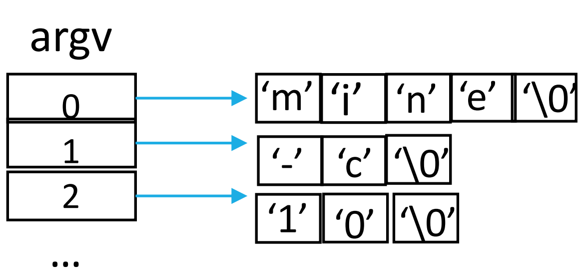
- There is no string type in C! These are “close”
  - `typedef char *string;                // VERY careful`
  - `typedef char [] string = “abc…”; // string literal`
  - `typedef char [MaxLength] string; // VERY careful`

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` (or `./mine`)

whitespace important

argc = ?



‘\0’ null character = string termination, C arrays start at 0

Type conversions

- Really useful call:

```
int x, y, i;
…
x = atoi (argv[i]); // string to int
y = x + 10;
```

Back to Command-line arguments

```
./argtestS 10 2 jon
```

- Why are command-line args useful?

Structure of a C program (cont'd)

- Functions may come from multiple source files and libraries or your own object modules (.o)
  - (e.g. `/usr/lib/gcc/… libgcc.a`)
    ls /usr/lib/gcc/x86_64-linux-gnu/9
    [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

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

sort.c
```
#include <sort.h> // like a macro
…
int sortit (char a[100]) {
   int B[MaxTokens];
   …
}
```

main.c
```
#include <sort.h>
int/void main (<options>) {
   …
   y= sortit (…);
   …
}
```

Link in sort.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; *allocation* (`int baz`) must be linked in eventually

```
extern int baz;
```

Global variables get de-allocated when?

Global Scope

sort.c

```
#include <sort.h>
static int foo = 4;
int bar = 5;
int sortit (char a[100]) {
   int B[MaxTokens];
   …
}
```

main.c

```
#include <sort.h>
extern int bar;
extern int foo;

int main () {
   …
   y= sortit (…);
   …
   bar = 10; // cool
   foo = 20; // NOPE
}
```

Program Scoping: Local

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

   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`

```
#include <stdio.h>
FILE *f;
F = fopen ("/usr/weiss039/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 (`libgc.a, libgcc.a`)

Compiling

- On most Unix/Linux systems, the compiler is gcc

  `gcc -o foo foo.c` (only 1 `main`)

  - Compiles into a single executable named foo

  To run, shell>`foo` (or `./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
- 0 is logical not: `if (0) will_never_do_this;`
- Don't like 0 for logical NOT …
  - `#define FALSE 0`
  - `#define TRUE 1`

(Most) Programs shown in class?

[Book programs]()

Pointers = Memory address

```
int x;
int *y;

y = &x;
*y = 10; // awesome
```

Parameter Passing

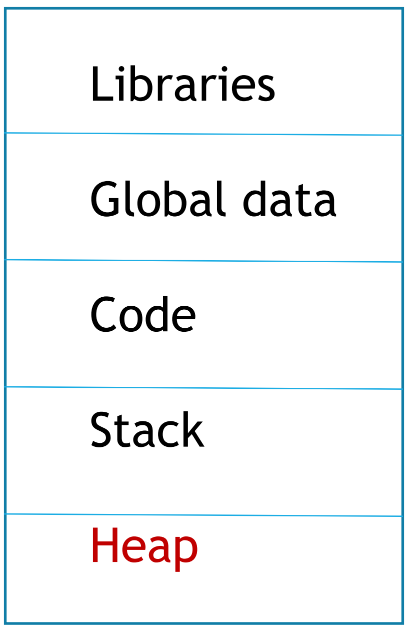
- By value …

```
int func (int x, int *y) {
   int a, *b;
   a= x;
   b = *y;
   *y = 10;  // on RHS, output parameter
 }

int main () {
   int *y;
   func (3, y);
}
```

Memory Allocation

- The heap



Memory Allocation (cont'd)

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

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

Casting: keeps compiler happy

Handy! Returns size of a variable or type in bytes

Release allocated memory

```
void free (void *ptr_var);
```

VERY error-prone!

`malloc`: underneath the `new` operator in C++ or Java

VOID

```
void *vptr;
char *aptr, *iptr;
// void can be casted to ANY pointer type
// and vice-versa
iptr = (int *) aptr;
aptr = vptr; //void cast not needed
vptr = iptr; //void cast not needed


// void type means no return value or no args
void my_func (void); // same as
void my_func ();
```

Unix/C tools: System Monitoring

- top: basic info on your processes
  - `top -u <uid>`
  - `top -p <pid>`

- top: shows complete information and dynamically updates
  - R: running, if always R, maybe an infinite loop
  - VIRT: virtual memory

Memory Leakage

- Your program leaks if its memory usage grows w/o bound
  - For what kind of program is this a problem?

- Happens if you forget to free memory not needed anymore

- Moral: don't lose ptr to allocated memory!

```
a = malloc (100000);
a = 10;
free (a); // oops
```

- On program exit, OS reclaims memory

C crashes

- C program crash
- Segmentation violation
  - Program attempts to access memory outside its boundary

```
int *b, a[10];
A[13] = 99;         // maybe cause an error
A[-3] = 88;         // maybe cause an error
*b = 10;            // for sure
```

To catch this you can run `valgrind` or `splint`

Can gcc catch stuff … maybe, [run man gcc]

Debugging

- Debugging 101: the `printf` and debugging levels

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

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

Can set multiple levels: DEBUG1, DEBUG2, …

Several preprocessor directives:

`#include, #define, #ifdef, #ifndef`

Unix/C tools: Debugging

- Use gdb: GNU debugger
  - There are many others
  - Set breakpoints, look at vars, step, trace
  - Recommend you learn this if no IDE

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

[run gdb]

Buffer Overflow/Stack Smashing (Attack)

- Buffer overflow

```
void func (char *buffer, …) {
      char local[5];
      …
      // string copy … copies until '\0'
      strcpy (local, buffer);
      // local[0] = buffer [0];
      // … local [i] = buffer [i];

      …
}
```

Bad guy calls it with a big string:

```
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
- Solutions?

Next time

- Processes!
- Chapter 3