CSC 314
C and UNIX overview

Brief Unix/Linux Overview

- Log onto os.cs.siue.edu
  - From another Unix-like OS, open a terminal

    ssh <your_username>@os.cs.siue.edu
  - From windows, get PuTTY or, better yet, Cygwin
- Once logged in, you're faced with the shell
  - Bash by default

```
virgil:test icrk$ ssh icrk@os.cs.siue.edu
icrk@os.cs.siue.edu's password:
Last login: Tue Aug 20 23:49:42 2013 from ...
Expedience is the best teacher.
[icrk@os ~]$
```

Navigating the File System

```
[icrk@os ~]$ ls -l
total 263708
-rw------- 1 icrk profs    603773 Jan 14  2013 BCI2000Tutorial.odt
drwx------ 1 icrk profs      3864 Aug 24  2012 bkup
-rw-r----- 1 icrk profs      5949 Oct 18  2012 bkup_hw5.tgz
drwxr-x--- 1 icrk profs      3864 Oct 17  2012 cache
[icrk@os ~]$ touch filename
[icrk@os ~]$ cp filename newfilename
[icrk@os ~]$ mv filename otherfilename
[icrk@os ~]$ rm *filename
[icrk@os ~]$ cd bkup
[icrk@os bkup]$ ls -l
total 1144
drwxr-x--- 1 icrk profs    3864 Aug 24  2012 cs
-rw------- 1 icrk profs 1160150 Aug 24  2012 www.tgz
[icrk@os ~]$ pwd
/home/icrk
[icrk@os ~]$ cd -
/home/icrk/bkup
[icrk@os bkup]$
```

Secure copy

```
virgil:test icrk$ scp test.c icrk@os.cs.siue.edu:~/temp
icrk@os.cs.siue.edu's password:
test.c                                        100%  406     0.4KB

virgil:test icrk$ scp icrk@os.cs.siue.edu:~/temp .
icrk@os.cs.siue.edu's password:
temp                                          100%  406     0.4KB
```

man (as in manual)

- Look here first, if you know what you're looking for

```
[icrk@os ~]$ man bash
BASH(1)

NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [file]

[icrk@os ~]$ man 3 printf
PRINTF(3)                  Linux Programmer's Manual

NAME
       printf,   fprintf,   sprintf,   snprintf,  vprintf,  vfprintf
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       Etc etc
```

grep

```
[icrk@os src]$ grep -n "ProcessSchedule ()" *.c
process.c:158:ProcessSchedule ()
traps.c:142:      ProcessSchedule ();
traps.c:147:      ProcessSchedule ();
traps.c:155:      ProcessSchedule ();
traps.c:226:      ProcessSchedule ();
```

Process related

```
[icrk@os src]$ emacs dlx.h
```

In a running program, ctrl+z to send to background

```
[1]+  Stopped                 emacs dlx.h
[iicrk@os src]$ ps
  PID TTY          TIME CMD
10939 pts/0    00:00:00 bash
11169 pts/0    00:00:00 emacs
11184 pts/0    00:00:00 top
11198 pts/0    00:00:00 ps
[icrk@os src]$ jobs
[1]-  Stopped                 emacs dlx.h
[2]+  Stopped                 top
[icrk@os src]$ fg 1
emacs dlx.h
[icrk@os src]$ kill -9 11184
[icrk@os src]$ ps
  PID TTY          TIME CMD
10939 pts/0    00:00:00 bash
11169 pts/0    00:00:00 emacs
11203 pts/0    00:00:00 ps
[2]-  Killed                  top
```

Lists background jobs

fg resumes a stopped background job and brings to foreground, or brings a running background job to the foreground (use bg <jobnumber> to resume a stopped background job in the background (doesn't work on emacs).

If machine is sluggish, check the output of 'top'. Kindly ask whoever is dragging down the machine to kill their processes.

tar

```
[icrk@os test]$ ls
execs  src
[icrk@os test]$ tar cfz lab1.tgz *
[icrk@os test]$ ls
execs  lab1.tgz  src
[icrk@os test]$ rm -rf src
[icrk@os test]$ rm -rf execs/
[icrk@os test]$ ls
lab1.tgz
[icrk@os test]$ tar xfz lab1.tgz
[icrk@os test]$ ls
execs  lab1.tgz  src
[icrk@os test]$
```

Scripts, pipes, redirection...

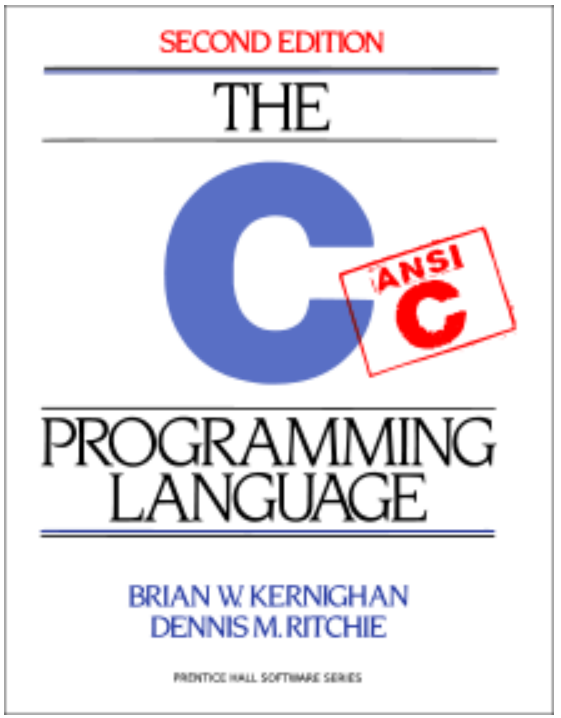
```
[icrk@os src]$ for file in `ls *.[ch]`; do if [ -e $file ]; then
echo $file >> diff_wc; diff $file ../../lab2/src/ | wc -l >> diff_
wc;  fi; done;
diff: ../../lab2/src/synch.working.h: No such file or directory
[icrk@os src]$ ls
diff_wc    Makefile         osend.s    synch.h          traps.c
dlx.h      Makefile.depend  process.c  synch.working.h  traps.h
dlxos.h    memory.c         process.h  synch.working.s  userprog.c
dlxos.s    memory.h         queue.c    syscall.h        usertraps.s
filesys.c  misc.c           queue.h    sysproc.c
filesys.h  misc.h           synch.c    trap_random.s
[icrk@os src]$ tail diff_wc
syscall.h
0
sysproc.c
0
traps.c
173
traps.h
23
userprog.c
67
[icrk@os src]$
```

C Overview

- UNIX: Designed and implemented at Bell Labs, 1969
  - Originally written in assembly
- C: designed by Dennis Ritchie and Ken Thompson at Bell Labs (AT&T) in the early 1970s
  - UNIX rewritten in C by 1973
- C influenced by:
  - ALGOL 60 (1960),
  - CPL (Cambridge, 1963),
  - BCPL (Martin Richard, 1967),
  - B (Ken Thompson, 1970)
- It's old. Why is it still around?
  - AT&T distributed UNIX and C to institutions worldwide
  - C spread and became ubiquitous in systems and application dev.

Classic

- For the classic and full treatment of the language features, see:
  - *The C Programming Language*
  - AKA the K&R book



Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89*
- As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)
- C++ and C
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write "*Clean C*" that conforms to both the C++ and C standards.

Elements of a C Program

- A C development environment includes
  - *System libraries* and *headers*: a set of standard libraries and their header files. For example see `/usr/include` and `glibc`.
  - *Application Source*: application source (.c) and header files (.h)
  - *Compiler*: converts source to object code for a specific platform (gcc)
  - *Linker*: resolves external references and produces the executable module (invoked by gcc, can disable with -c flag)
- User program structure
- There must be one main function where execution begins when the program is run. This function is `main`
    - `int main (void) { ... },`
    - `int main (int argc, char *argv[]) { ... }`
    - UNIX Systems have a 3rd way to define main(), though it is not POSIX.1 compliant

      `int main (int argc, char *argv[], char *envp[])`
  - Additional local and external functions and variables

A Simple C Program

- *Create* example file: `hello.c`
- *Edit* using vi, emacs, whatever
- *Compile* using gcc:
  `gcc -o hello hello.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./hello`
- Normal termination:
  `void exit(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

  or...

  `return 0;`

  What's the difference?

```c
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   exit(0);
}
```

Source and Header files

- Just as in C++, place related code within the same module (i.e. file).
- Header files (`.h`) export interface definitions
  - function prototypes, data types, macros, inline functions and other common declarations
- Definitions shouldn't go into a header file with a few exceptions:
  - inlined code
  - const definitions
- *C preprocessor* (`cpp`) is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

Another Example C Program

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file.
`#define` directs preprocessor to define macros.

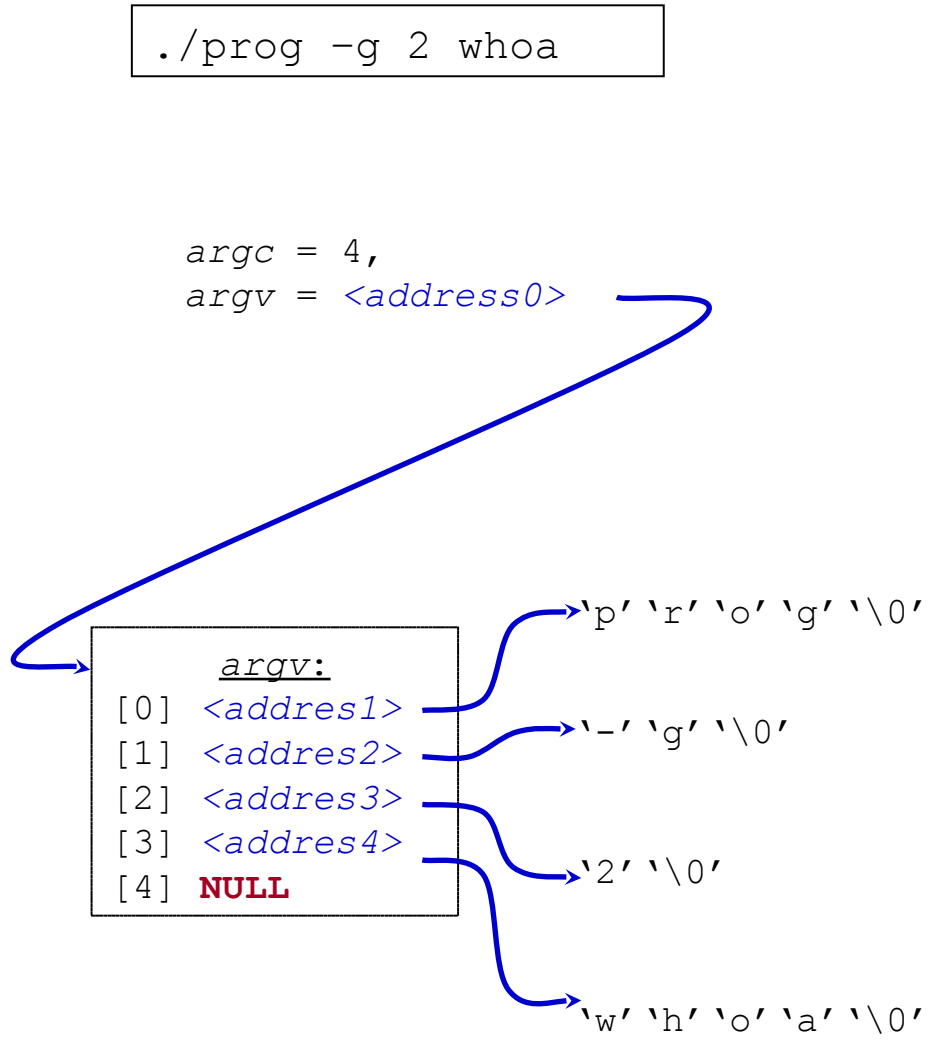
example.c

```
/* this is a C-style comment
 * You generally want to palce
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
  printf("Hello, Prog name = %s\n",
              argv[0]);
  exit(0);
}
```

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.



C Standard Header Files

- Standard Headers you should know about:
  - `stdio.h` – file and console (also a file) IO: *`perror`*, *`printf`*, *`open`*, *`close`*, *`read`*, *`write`*, *`scanf`*, etc.
  - `stdlib.h` - common utility functions: *`malloc`*, *`calloc`*, *`strtol`*, *`atoi`*, etc
  - `string.h` - string and byte manipulation: *`strlen`*, *`strcpy`*, *`strcat`*, *`memcpy`*, *`memset`*, etc.
  - `ctype.h` – character types: *`isalnum`*, *`isprint`*, *`isupport`*, *`tolower`*, etc.
  - `errno.h` – defines *`errno`* used for reporting system errors
  - `math.h` – math functions: *`ceil`*, *`exp`*, *`floor`*, *`sqrt`*, etc.
  - `signal.h` – signal handling facility: *`raise`*, *`signal`*, etc
  - `stdint.h` – standard integer: *`intN_t`*, *`uintN_t`*, etc
  - `time.h` – time related facility: *`asctime`*, *`clock`*, *`time_t`*, etc.

The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

- Commands begin with a '#'. Abbreviated list:
  - `#define` : defines a macro
  - `#undef` : removes a macro definition
  - `#include` : insert text from file
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#ifndef` : conditional based on whether macro is not defined
  - `#else` : alternative
  - `#elif` : conditional alternative
  - `defined()` : preprocessor function: 1 if name defined, else 0

```
#if defined(__NetBSD__)
```

Preprocessor: Macros

- Using macros as functions, exercise caution:
  - flawed example: `#define mymult(a,b) a*b`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = i - 1 * j + 5;`
  - better: `#define mymult(a,b) (a)*(b)`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = (i - 1)*(j + 5);`
- Be careful what you ask for
  - Macro: `#define mysq(a) (a)*(a)`
  - Flawed usage:
    - Source: `k = mysq(++i)`
    - Post preprocessing: `k = (++i)*(++i)`
- Alternative is to use inline'ed functions
  - `inline int mysq(int a) {return a*a};`
  - `mysq(++i)` works as expected in this case.

Preprocessor: Conditional Compilation

- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts

```
#define DEFAULT_SAMPLES  100
#ifdef __linux
 static inline int64_t
   gettime(void) {...}
#elif defined(sun)
 static inline int64_t
   gettime(void) {return (int64_t)gethrtime()}
#else
 static inline int64_t
   gettime(void) {... gettimeofday()...}
#endif
```

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

  ```
  int x[5]; // an array of 5 4-byte ints.
  ```

- All arrays begin with an index of 0
- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x (remember that size of int is 4 bytes typically)!!

Pointers

- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, **&** creates a pointer (reference to)
  - ```
    int i = 3; int *j = &i;
    *j = 4; printf("i = %d\n", i); // prints i = 4
    ```
  - ```
    int myfunc (int arg);
    int (*fptr)(int) = myfunc;
    i = fptr(4); // same as calling myfunc(4);
    ```
- Generic pointers:
  - Traditional C used (char *)
  - Standard C uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use ***NULL*** or ***0***. *It is a good idea to always initialize pointers to NULL.*

Pointers in C

```
Step 1:
int main (int argc, argv) {
 int  x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int  a[4] = {1, 2, 3, 4};
...
```

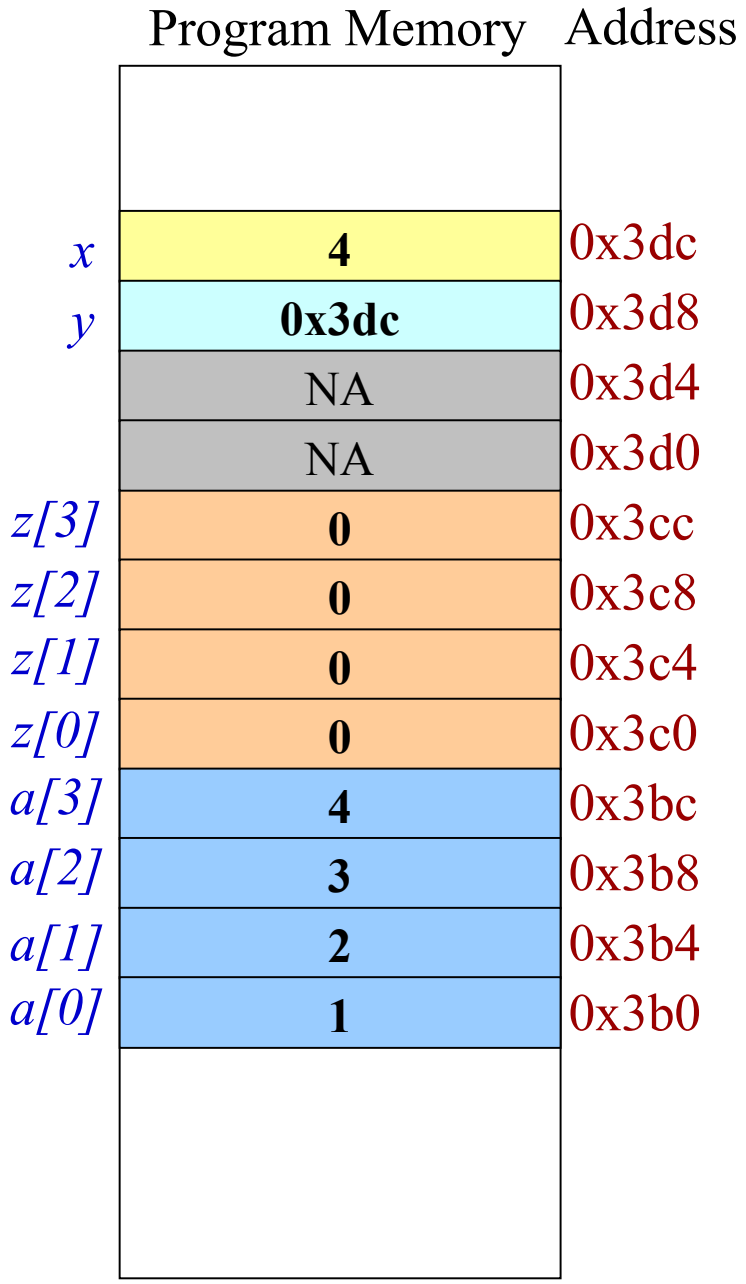
Note: The compiler converts z[1] or *(z+1) to
Value at address (*Address of z* + *sizeof(int)*);

In C you would write the byte address as:

```
(char *)z + sizeof(int);
```

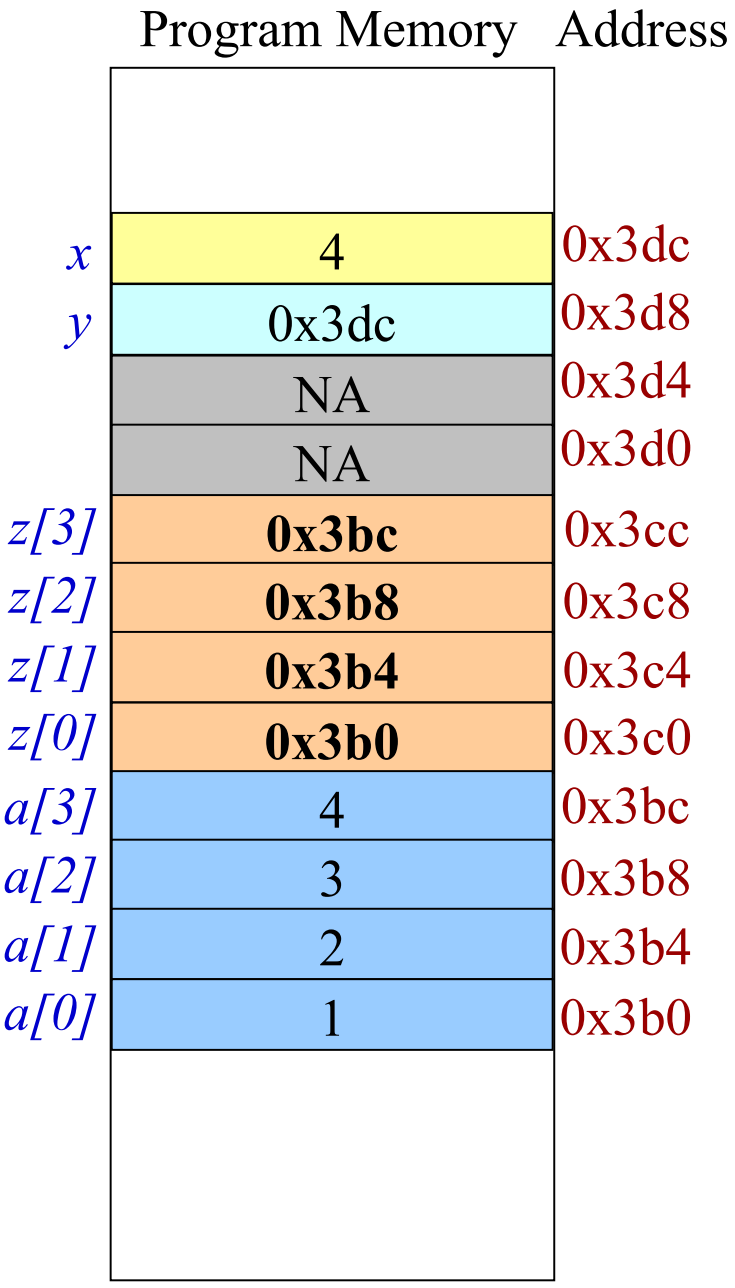
or letting the compiler do the work for you

```
(int *)z + 1;
```



Pointers Continued

```
Step 1:
int main (int argc, argv) {
  int  x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int  a[4] = {1, 2, 3, 4};
Step 2: Assign addresses to array Z
  z[0] = a;     // same as &a[0];
  z[1] = a + 1;     // same as &a[1];
  z[2] = a + 2;     // same as &a[2];
  z[3] = a + 3;     // same as &a[3];
```



Functions

- Prototypes should appear in .h, else functions should be defined ahead of invocation

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- Can pass by value or “by reference” (just as in C++, pass the pointer)
  - in reality, there is no pass by reference in C :)

Basic Types and Operators

- Basic data types
  - Types: *char, int, float and double*
  - Qualifiers: *short, long, unsigned, signed, const*
- Constant: 0x1234, 12, “Some string”
- Enumeration:
  - Names in different enumerations must be distinct
  - ```
    enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
    enum WeekendDay_t {Sat = 0, Sun = 4};
    ```
- Arithmetic: +, -, *, /, %
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ||
- Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)
    ```

Operator Precedence (from "C a Reference Manual", 5th Edition)

Tokens	Operator	Class	Precedence	Associates
names, literals	simple tokens	primary	16	n/a
`a[k]`	subscripting	postfix		left-to-right
`f(...)`	function call	postfix		left-to-right
`.`	direct selection	postfix		left-to-right
`->`	indirect selection	postfix		left to right
`++ --`	increment, decrement	**postfix**		left-to-right
`(type){init}`	compound literal	postfix		left-to-right
`++ --`	increment, decrement	**prefix**	15	right-to-left
`sizeof`	size	unary		right-to-left
`~`	bitwise not	unary		right-to-left
`!`	logical not	unary		right-to-left
`- +`	negation, plus	unary		right-to-left
`&`	address of	unary		right-to-left
`*`	indirection (*dereference*)	unary		right-to-left

Tokens	Operator	Class	Precedence	Associates
`(type)`	casts	unary	14	right-to-left
`* / %`	multiplicative	binary	13	left-to-right
`+ -`	additive	binary	12	left-to-right
`<< >>`	left, right shift	binary	11	left-to-right
`< <= > >=`	relational	binary	10	left-to-right
`== !=`	equality/ineq.	binary	9	left-to-right
`&`	bitwise and	binary	8	left-to-right
`^`	bitwise xor	binary	7	left-to-right
`|`	bitwise or	binary	6	left-to-right
`&&`	logical and	binary	5	left-to-right
`||`	logical or	binary	4	left-to-right
`?:`	conditional	ternary	3	right-to-left
`= += -= *= /= %= &= ^= |= <<= >>=`	assignment	binary	2	right-to-left
`,`	sequential eval.	binary	1	left-to-right

Structs and Unions

- structures
  - `struct MyPoint {int x; int y;};`
  - `typedef struct MyPoint MyPoint_t;`
  - `MyPoint_t point, *ptr;`
  - `point.x = 0;point.y = 10;`
  - `ptr = &point; ptr->x = 12; ptr->y = 40;`
- unions
  - `union MyUnion {int x; MyPoint_t pt; struct {int a; char c[4];} S;};`
  - `union MyUnion x;`
    - Can only use one of the elements. Memory will be allocated for the largest element

Conditional Statements (if/else)

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- If you have compound statements then use brackets (blocks)
  - ```
    if (a < 4 && b > 10) {
      c = a * b; b = 0;
      printf("a = %d, a\'s address = 0x%08x\n", a, (uint32_t)&a);
    } else {
      c = a + b; b = a;
    }
    ```
- These two statements are equivalent:
  - ```
    if (a) x = 3; else if (b) x = 2; else x = 0;
    ```
  - ```
    if (a) x = 3; else {if (b) x = 2; else x = 0;}
    ```
- Is this correct?
  - ```
    if (a) x = 3; else if (b) x = 2;
    else (z) x = 0; else x = -2;
    ```

Conditional Statements (switch)

```
int c = 0;
switch (c) {
  case 0:
    printf("c is 0... sad panda\n");
    break;
  case 1:
    printf("c is 1... cheer up\n");
  default:
    printf("Don't know what to do.\n");
    break;
}
```

- What if we leave the break statement out?
- Do we need the final break statement on the default case?

Loops

```
for (i = 0; i < MAXVALUE; i++) {
     dowork();
   }
while (c != 12) {
     dowork();
   }
do {
     dowork();
   } while (c < 12);
```

- flow control
  - **break** – exit innermost loop
  - **continue** – perform next iteration of loop
- Note, all these forms permit one statement to be executed. By enclosing in brackets we create a block of statements.

make and Makefiles, Overview

- Why use make?
  - convenience of only entering compile directives once
  - make is smart enough (with your help) to only compile and link modules that have changed or which depend on files that have changed
  - allows you to hide platform dependencies
  - promotes uniformity
  - simplifies my (and hopefully your) life when testing and verifying your code
- A makefile contains a set of rules for building a program

```
target ... : prerequisites ...
        command
        ...
```

- Static pattern rules.
  - each target is matched against target-pattern to derive stem which is used to determine prereqs (see example)

```
targets ... : target-pattern : prereq-patterns ...
   command
   ...
```

DLXOS Labs Makefile

```
CC = gcc-dlx
AS = dlxasm
CFLAGS = -mtraps -O3

#INCS = $(wildcard *.h)
#SRCS = $(wildcard *.c)
#OBJS = $(addsuffix .o, $(basename $
(wildcard *.c))) \
#       $(addsuffix .o, $(basename $
(wildcard *.s)))
INCS = dlxos.h traps.h filesys.h
memory.h misc.h process.h queue.h \
       synch.h syscall.h
SRCS = filesys.c memory.c misc.c
process.c queue.c synch.c traps.c
sysproc.c
OBJS = $(addsuffix .o, $(basename $
(SRCS)))

os.dlx.obj: os.dlx
     $(AS) -i _osinit -l os.lst
os.dlx
     mv os.dlx.obj ../execs

os.dlx: $(OBJS) dlxos.o trap_random.o
osend.o
     $(CC) -mtraps -O3 dlxos.o
trap_random.o $(OBJS) osend.o -o
os.dlx

osend.o: osend.s
     $(CC) -c osend.s

trap_random.o: trap_random.s
     $(CC) -c trap_random.s

dlxos.o: dlxos.s

$(CC) -c dlxos.s


usertraps.o: usertraps.s

$(CC) -c usertraps.s


userprog : userprog.o usertraps.o

$(CC) -mtraps -O3 userprog.o usertraps.o -o
userprog.dlx

$(AS) -l userprog.lst userprog.dlx

mv userprog.dlx.obj ../execs


Makefile.depend: depend


depend: $(SRCS) $(INCS)

$(CC) -MM $(SRCS) > Makefile.depend


clean:

/bin/rm -f *.o *.dlx *.lst *.obj Makefile.depend vm


include Makefile.depend
```

This is all worthless...

- … unless you go play
- Go play

```
[icrk@os ~]$ exit
logout
Connection to os.cs.siue.edu closed.
```