

# CSC322

## C Programming and UNIX

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

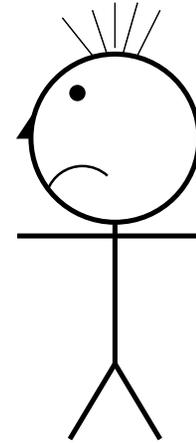
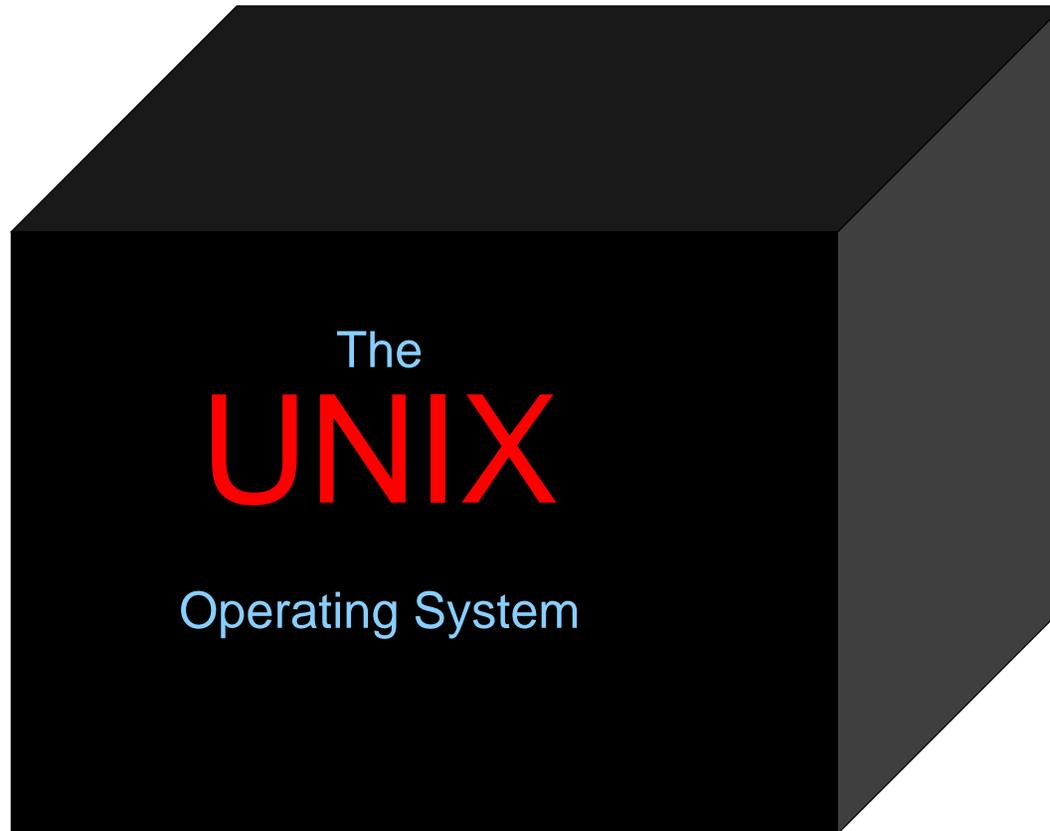
Prerequisites: CSC220 or EEN218

# Hackers!

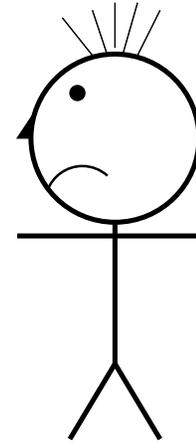
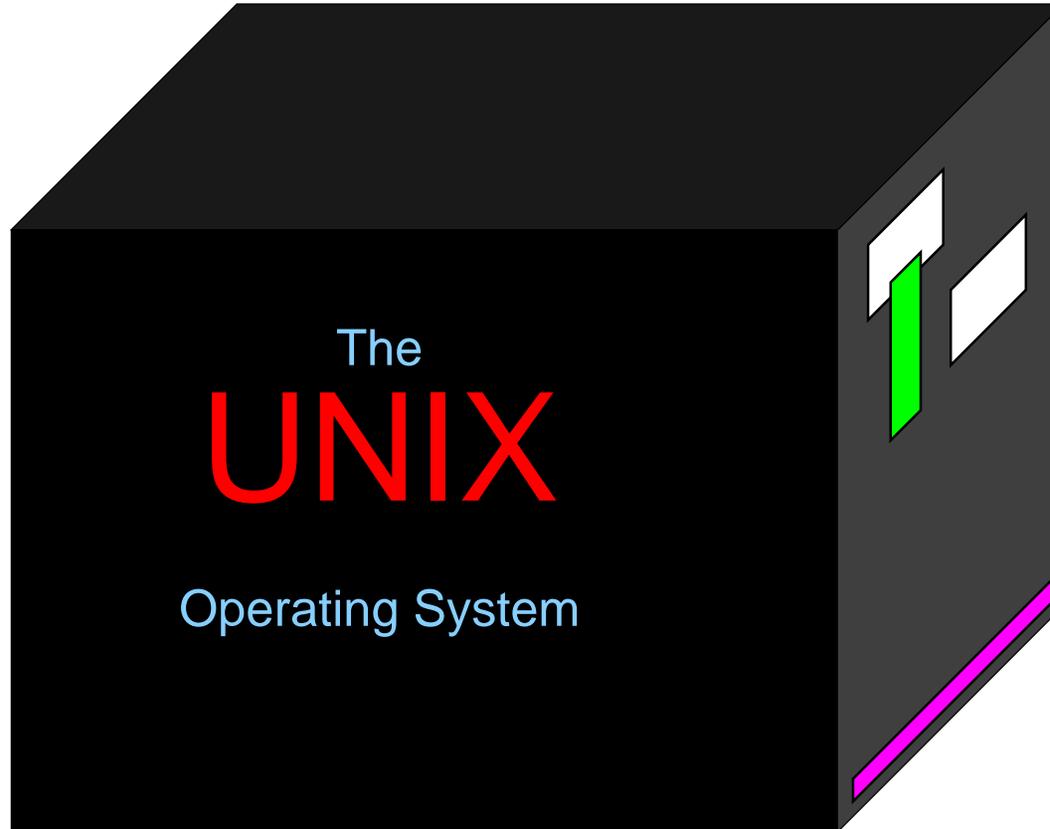
**Hacker** [originally, someone who makes furniture with an axe] 1. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary. 2. One who programs enthusiastically (even obsessively) or who enjoys programming rather than just theorizing about programming. 3. A person capable of appreciating hack value. 4. A person who is good at programming quickly. 5. An expert at a particular program, or one who frequently does work using it or on it; as in 'a Unix hacker'. (Definitions 1 through 5 are correlated, and people who fit them congregate.) 6. An expert or enthusiast of any kind. One might be an astronomy hacker, for example. 7. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations. 8. [deprecated] A malicious meddler who tries to discover sensitive information by poking around. Hence 'password hacker', 'network hacker'. The correct term for this sense is cracker.

The New Hacker's Dictionary (aka the Jargon File)

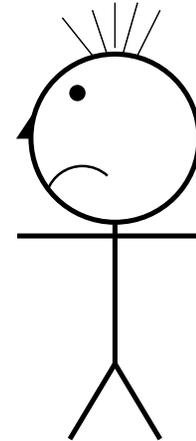
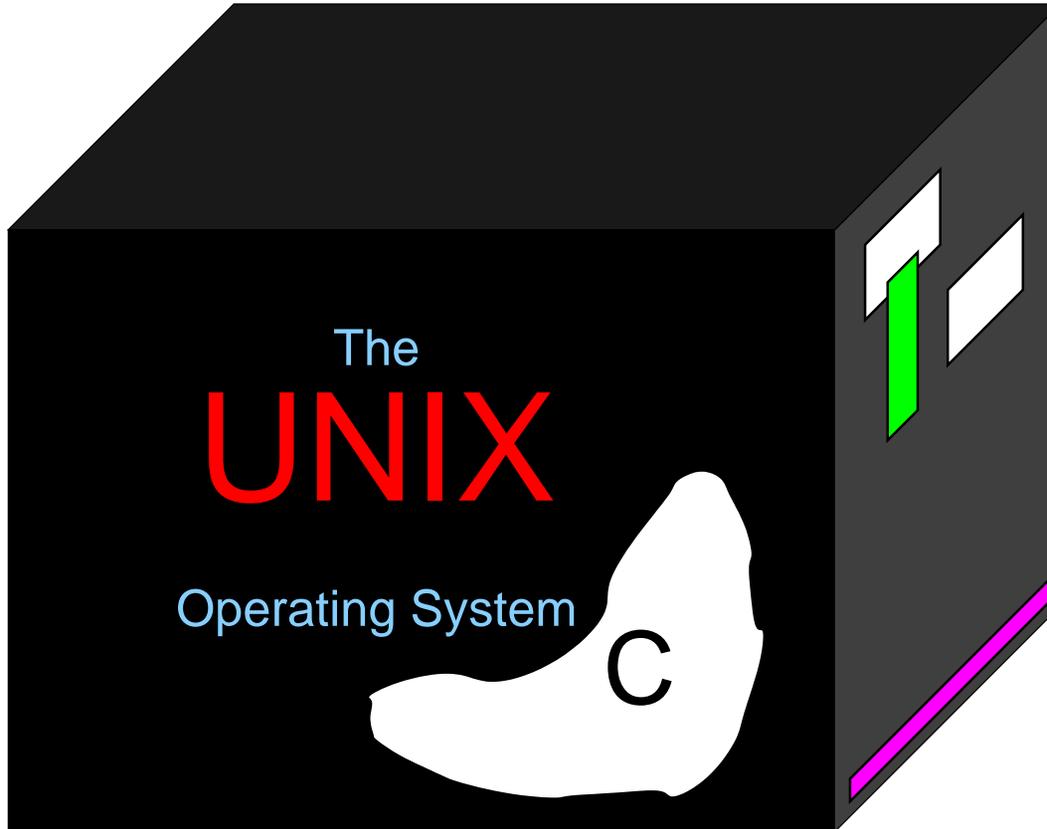
# UNIX and You



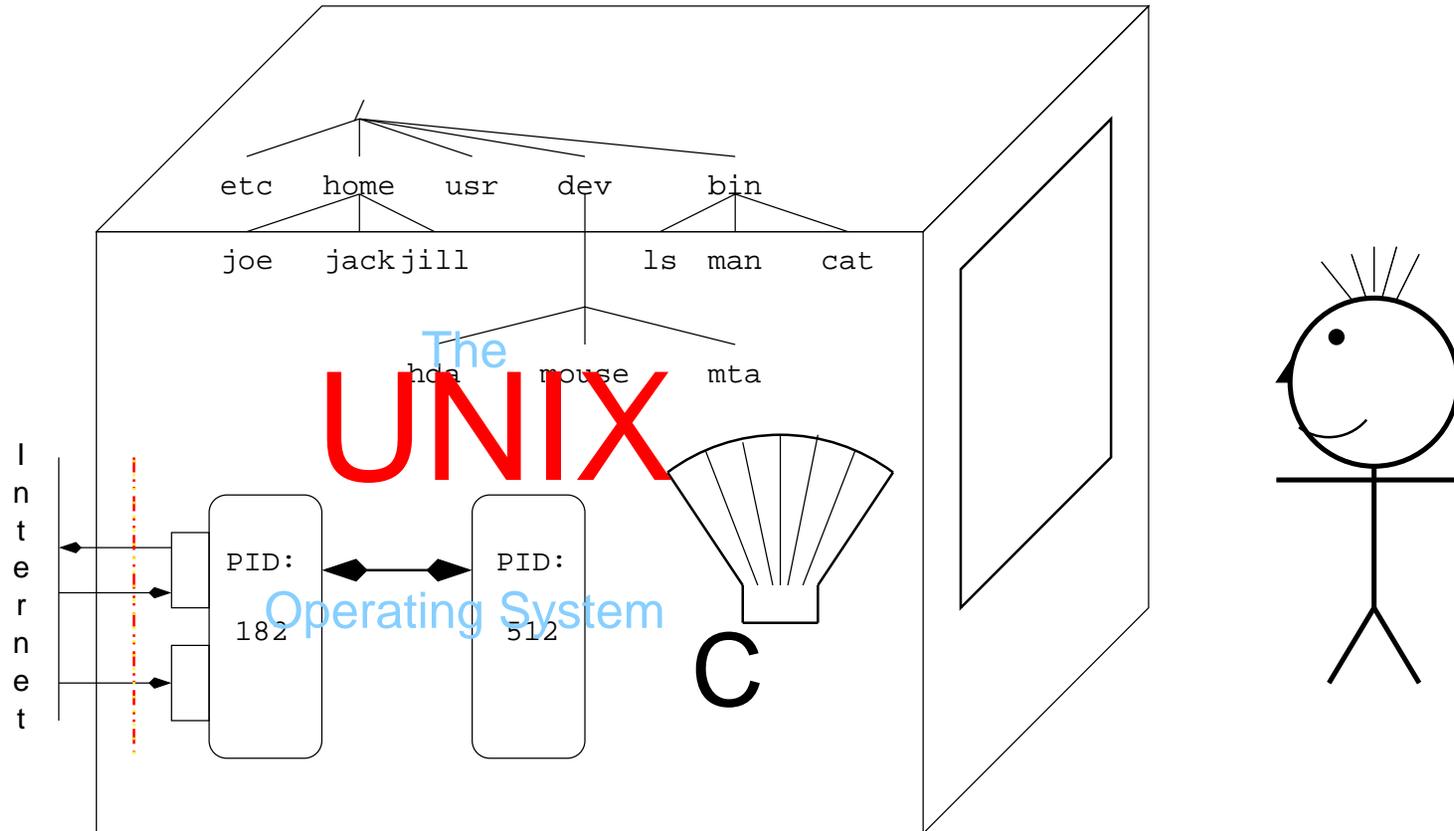
# UNIX and You



# UNIX and You



# Our AIM



# The Myth

UNIX is a big-iron operating system

UNIX is complicated

UNIX is hard to use

UNIX has been created by SUN, IBM, HP, and other large companies

UNIX is monolithic

# Counterpoint

UNIX was developed on small machines and became popular on the “killer micros”. UNIX dialects now run on everything from a PDA to CRAY supercomputers

UNIX is based on simple and elegant principles (but has added a some cruft over the years)

UNIX is not particularly hard to use (compared to the power it gives to the user), but has a reasonably steep learning curve. It’s not a “show-me” operating system, but a “tell me” operating system,

UNIX has been created in a research environment, and much of it has been developed in informal settings by hackers. Much of the impetus for UNIX comes from free versions (Linux, Net-, Open-, FreeBSD), although many companies contribute to it’s development

Many UNIX kernels are monolithic, but the UNIX system is extremely modular.

# UNIX

First **portable** operating system (NetBSD: 18 processor architectures,  $\approx$  50 computer architectures)

Written in a “high-level” language (C)

Small (for what it does):

- Recent LINUX kernel: 2.4 million LOC (1.4 million for driver, 0.4 million architecture-dependent stuff (16 ports))
- Windows 2000: Estimates range from 29 million to 65 million LOC, supports just 1.5 architectures

Modular (though often on a monolithic kernel)

- Separate windowing system (X) and window managers
- Various Desktop-Solutions (CDE, KDE, Gnome)
- **Toolbox-philosophy**: Combine (lot's of) simple tools
- Underneath: Strong and simple abstraction (“**Everything is a file**”)

# C

“Pragmatic” high level language:

- Handles characters, numbers, addresses as implemented by most computers
- Small core language, much functionality provided by libraries (mostly in C!)
- Compilers are easy to write
- Compilers are easy to port
- Even naive compilers produce reasonably efficient code

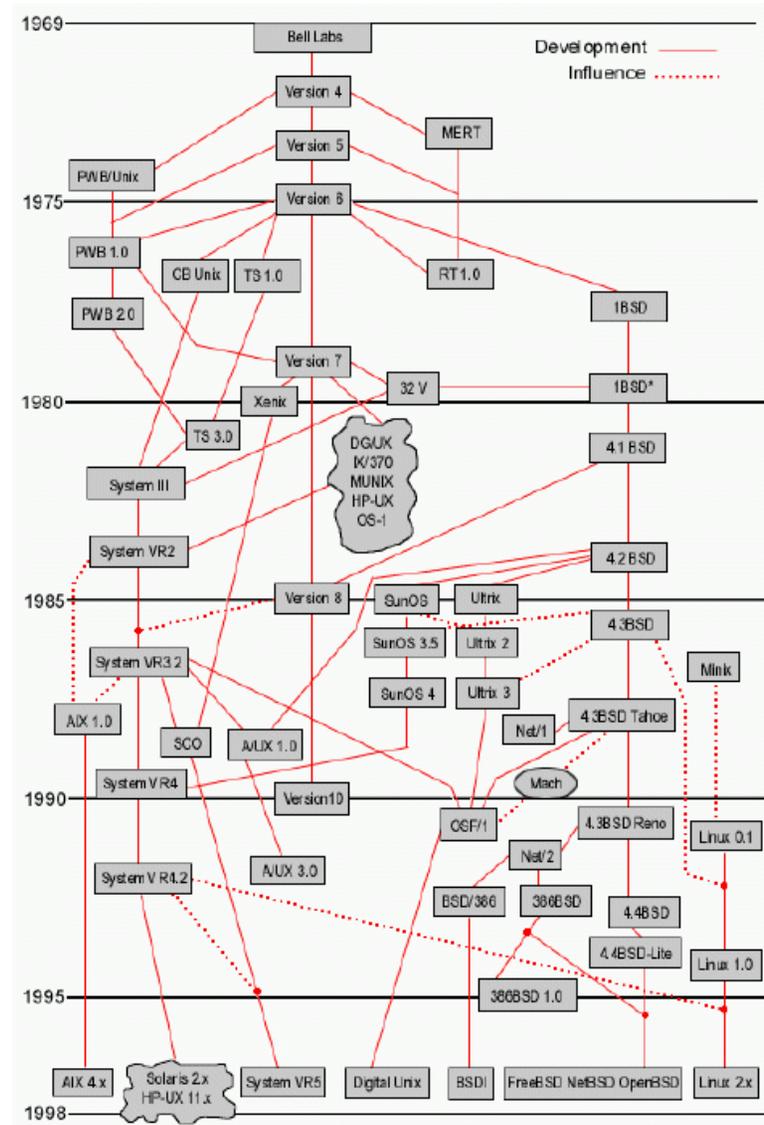
Hacker-friendly

- Straightforward compilation (nothing is hidden)
- Compact source code (fewer keystrokes, fast to read)
- Typed, but no bondage-and-discipline language

Adequate support for building abstractions

- Structures (composing objects), unions, enumerations
- Arrays and pointer
- Support for defining new types

# UNIX history tree (simplified)



For a fuller tree see <http://www.levenez.com/unix/>

# A Short History of UNIX and C

- 1969** Ken Thompson wrote the first UNIX (in assembler) on a PDP7 at AT&T Bell Labs, allegedly to play Space Travel
- 1970** Brian Kernighan coins the name UNIX. The UNIX project gets a PDP11 and a task: Writing a text processing system
- 1971-72** Creation of C (Dennis Ritchie), UNIX rewritten in C
- 1972** Pipes arrive, UNIX installed on 10 (!) systems
- 1975** AT&T UNIX “Version 6” distributed with sources under academic licenses
- 1976** Ken Thompson in Berkely, leading to BSD UNIX
- 1977** 1BSD release
- 1978** UNIX “Version 7”, leading to System V (AT&T)

# A Short History of UNIX and C

**1978** 3BSD, adding virtual memory

**1980** Microsoft XENIX brand of UNIX

**1982** 4.2BSD, adding TCP/IP

**1982** SGI IRIX

**1983** Bjarne Stroustrup creates C++ (at AT&T Bell labs)

**1983** GNU Project announced (Aim: Free UNIX-like system)

**1983-1984** Berkeley Internet Name Demon (BIND) created

**1984** SUN introduces NFS (Network File System)

**1985** Free Software Foundation (Stallman), GNU manifesto, GNU Emacs

# A Short History of UNIX and C

**1986** HP-UX, SunOS3.2 (from BSD Unix), “attack of the killer micros”

**1986** MIT Project Athena creates X11 (Network window system)

**1986** POSIX.1 (Portable operating system interface standard)

**1988** GNU GPL

**1988** System VR4 “One UNIX to rule them all” (AT&T+SUN)

**1988** NeXTCUBE with NeXTSTEP operating system

**1989** ANSI-C Standard “C89” (adds prototypes, standard library)

**1889** SunOS 4.0x

**1990** Net/1 Release (free BSD UNIX)

**1990** IBM AIX

# A Short History of UNIX and C

**1991** Linux 0.01, “attack of the killer PCs” (continuing till this day)

**1991** World Wide Web born

**1991–1992** Lawsuits around BSD UNIX Net/1 and Net/2 releases

**1992** SunOS 5 aka Solaris-2 (from System VR4)

**1993** FreeBSD 1.0

**1994** Linux 1.0

**1994** NetBSD 1.0, 4.4BSD Lite (unencumbered by AT&T copyrights, becomes new base for all non-commercial BSD flavours)

**1995** “UNIX wars” are over

**1996** Tux the Penguin becomes Linux mascot

# A Short History of UNIX and C

**1998** UNIX-98 branding (Single UNIX specification)

**2000** New ANSI “C99”

**2001** IBM runs prime time TV ads for Linux

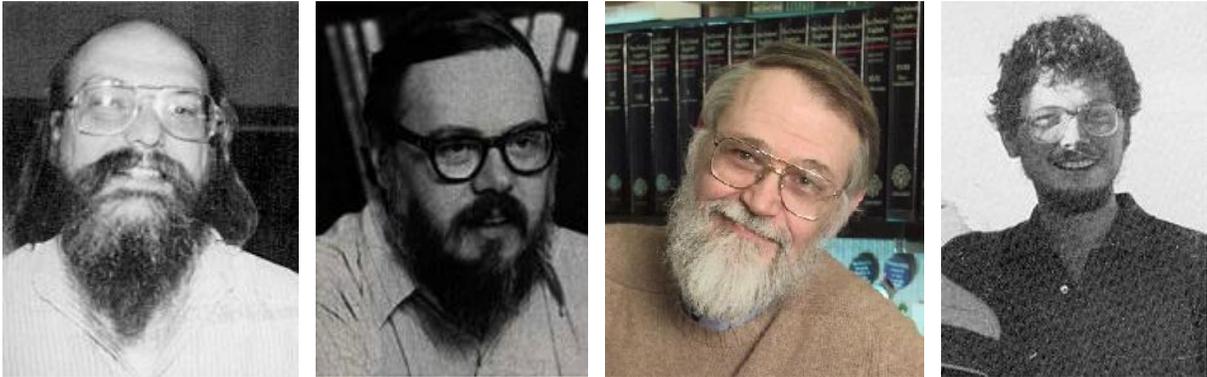
**2001** UNIX-based MacOS X

**2002** Linux is at version 2.4, Emacs is version 21.2, SunOS is at 5.9 (aka Solaris 9), BIND is version 9.2.1

# Another Opinion

UNIX is **not** an operating system. . .  
. . . but is the collected folklore of the  
hacker community!

# Spot the Even Ones



# Upshot

You don't **have** to grow a beard  
to become a world-class UNIX hacker. . .

. . . but it **does** seem to help!

**CSC322**  
**C Programming and UNIX**  
**UNIX from a User's Perspective**

Stephan Schulz

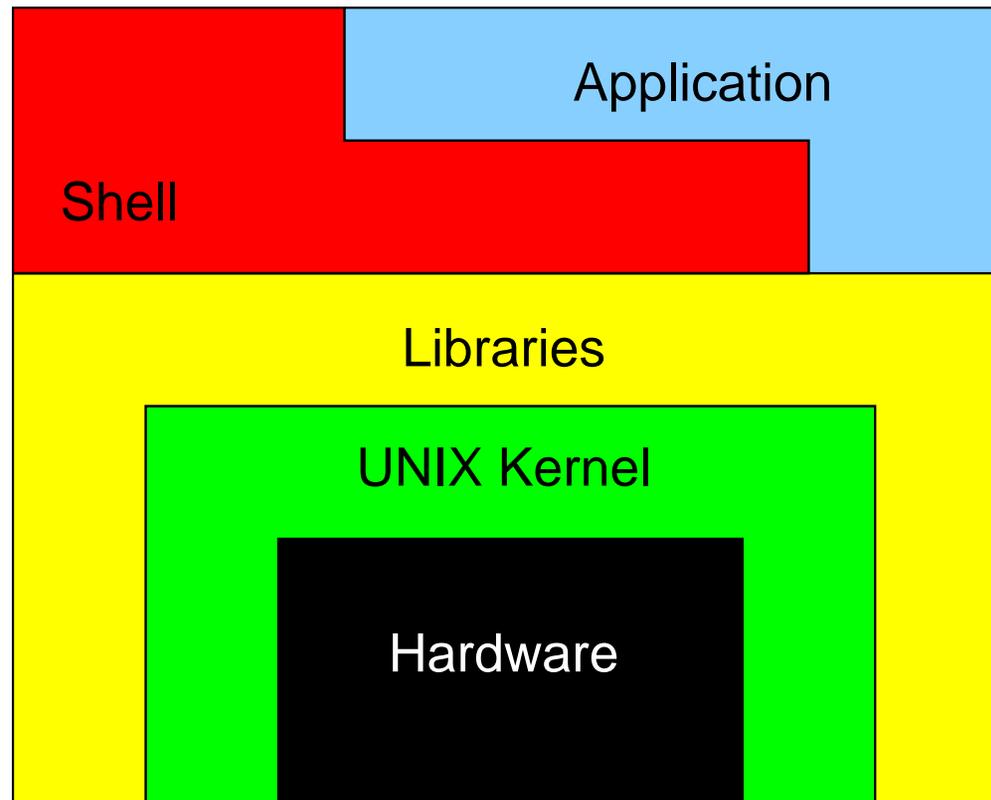
Department of Computer Science

University of Miami

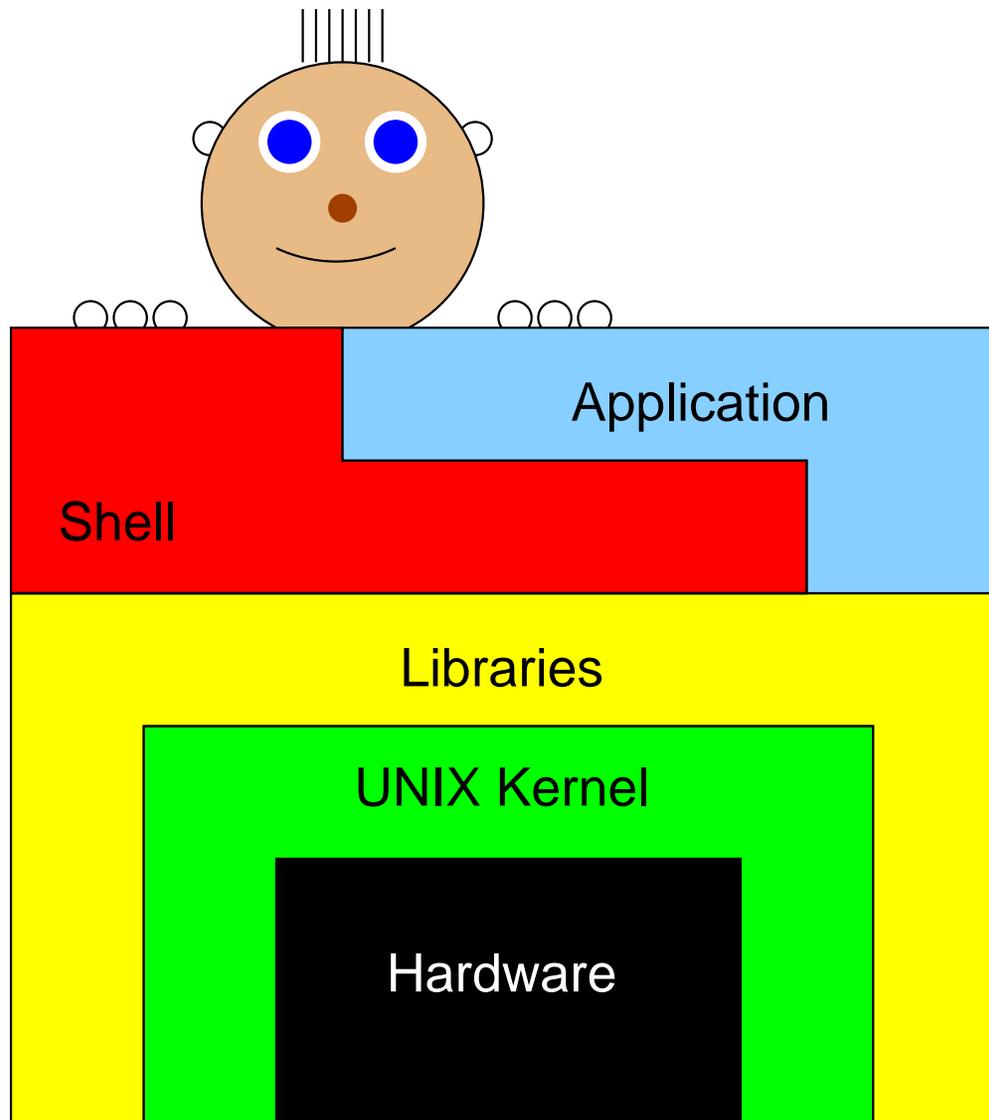
`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# UNIX Architecture



# UNIX Architecture



# Some Concepts

UNIX is a multi-user system. Each user has:

- User name (mine is schulz on most machines)
- Numerical user id (e.g. 500)
- Home directory: A place where (most of) his or her files are stored

UNIX is a multi-tasking system, i.e. it can run multiple programs at once. A running program (with its data) is called a **process**. Each process has:

- Owner (a user)
- Working directory (a place in the file system)
- Various resources

A **shell** is a command interpreter, i.e. a **process** accepting and executing commands from a user.

- A shell is typically owned by the user using it
- The initial working directory of a shell is typically the users home directory (but can be changed by commands)

# More on Users

There are two kinds of users:

- Normal users
- Super users (“root”)

Super-users:

- Have unlimited access to all files and resources
- Always have numerical user id **0**
- Normally have user name “root” (but there can be more than one user name associated with UID 0)
- **Can seriously damage the system!**

Normal users

- Can only access files if they have the appropriate permissions
- Can belong to one or more **groups**. Users within a group can share files
- **Usually cannot damage the system or other users files!**

# The User Interface

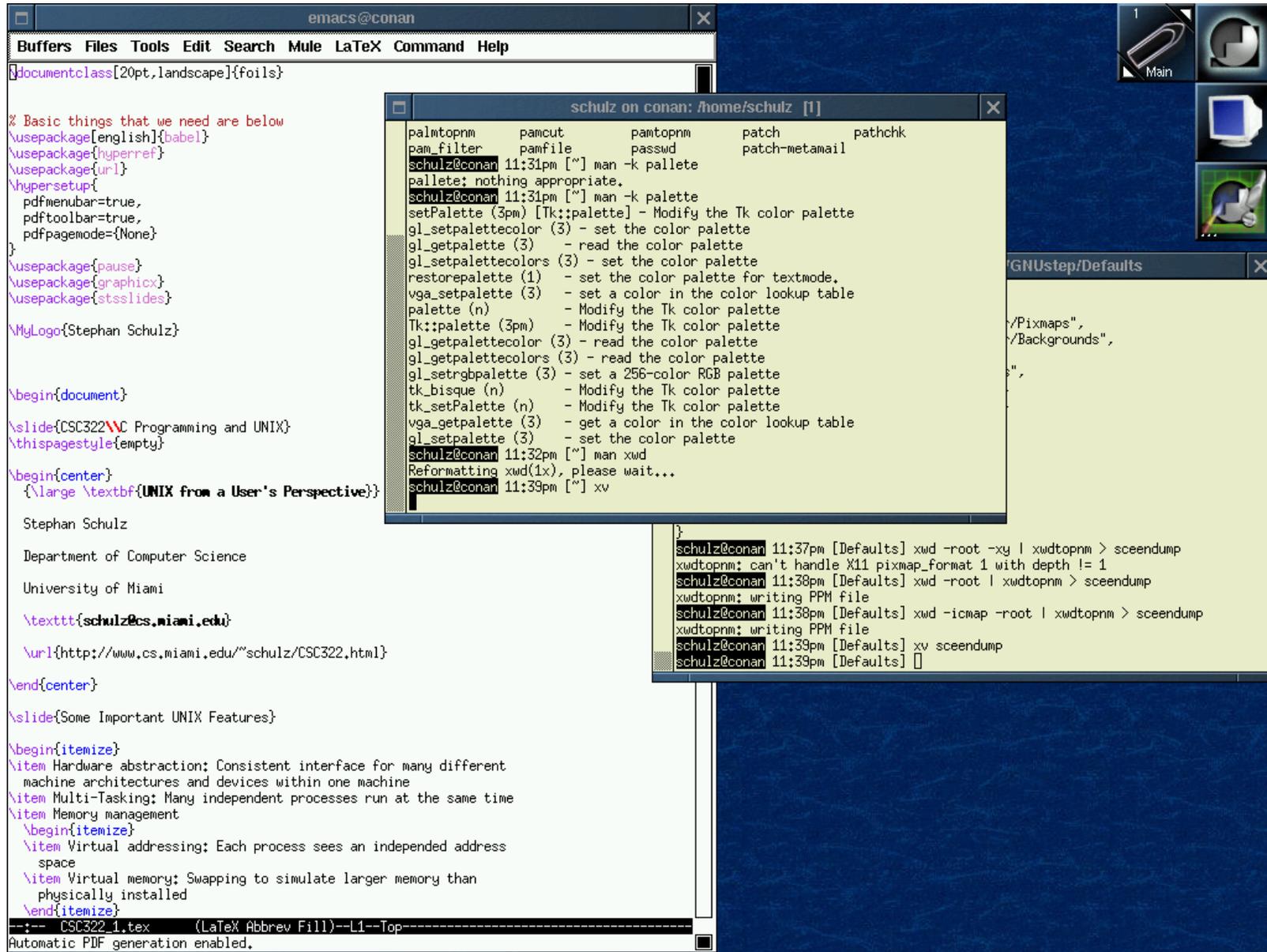
## UNIX: Provide Tools, Not Policy

- Most tools operate on all (ASCII) file formats
- Extremely configurable environment – different users have different user experiences
- No restrictions  $\Leftrightarrow$  Little consistency
- We will assume the default environment on the lab machines for examples

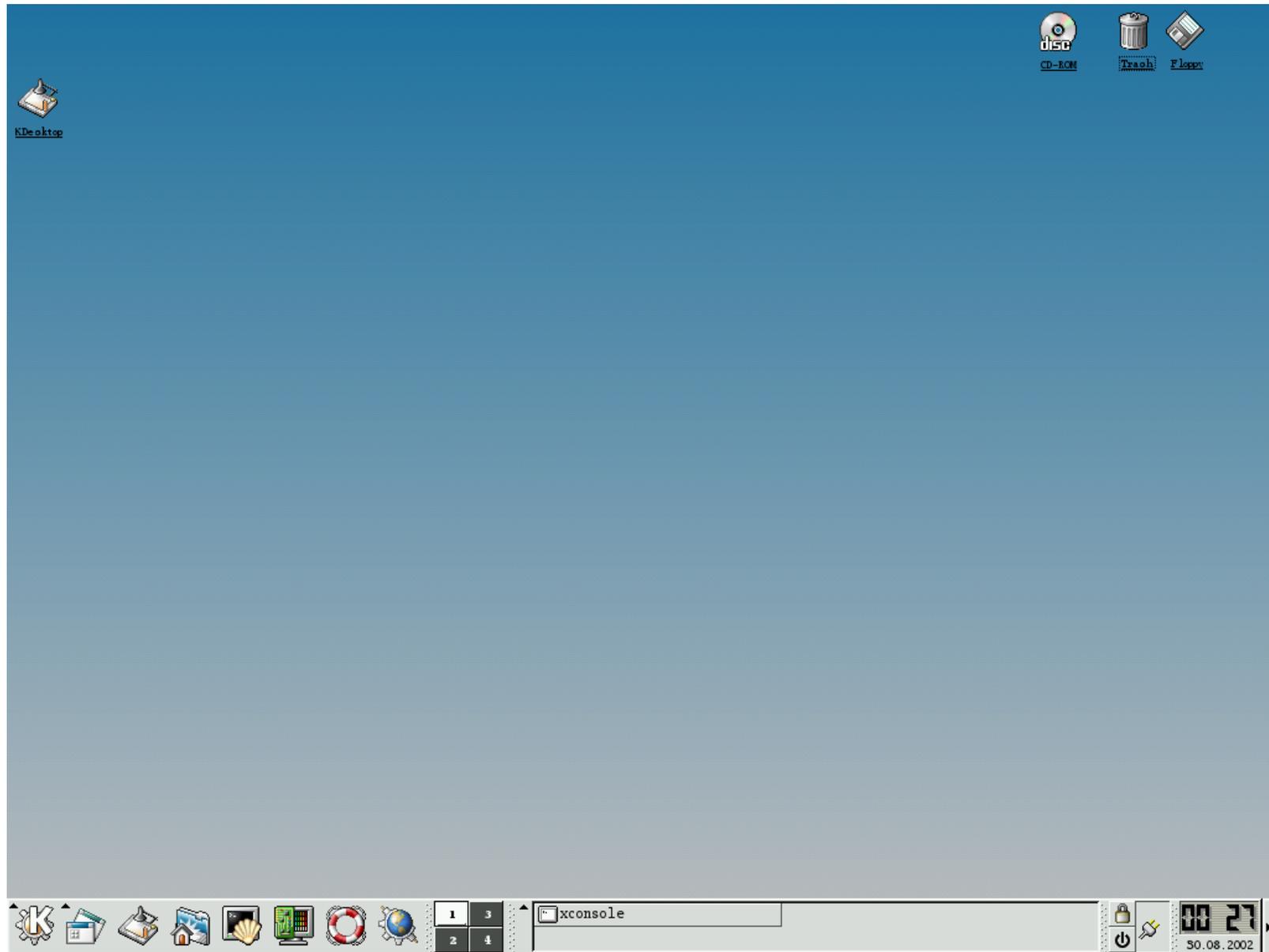
## X Window System: Provide Mechanisms, Not Policy

- Windowing system offers (networked) drawing primitives
- Different GUIs built on top of this
- GUI conventions may even differ from one application to the other!
- Modern desktop environments (GNOME/**KDE**) try to change this, but you are bound to use many legacy applications anyways!

# My Graphical Desktop



# Default KDE Desktop (SuSE Linux)



# Desktop Discussion

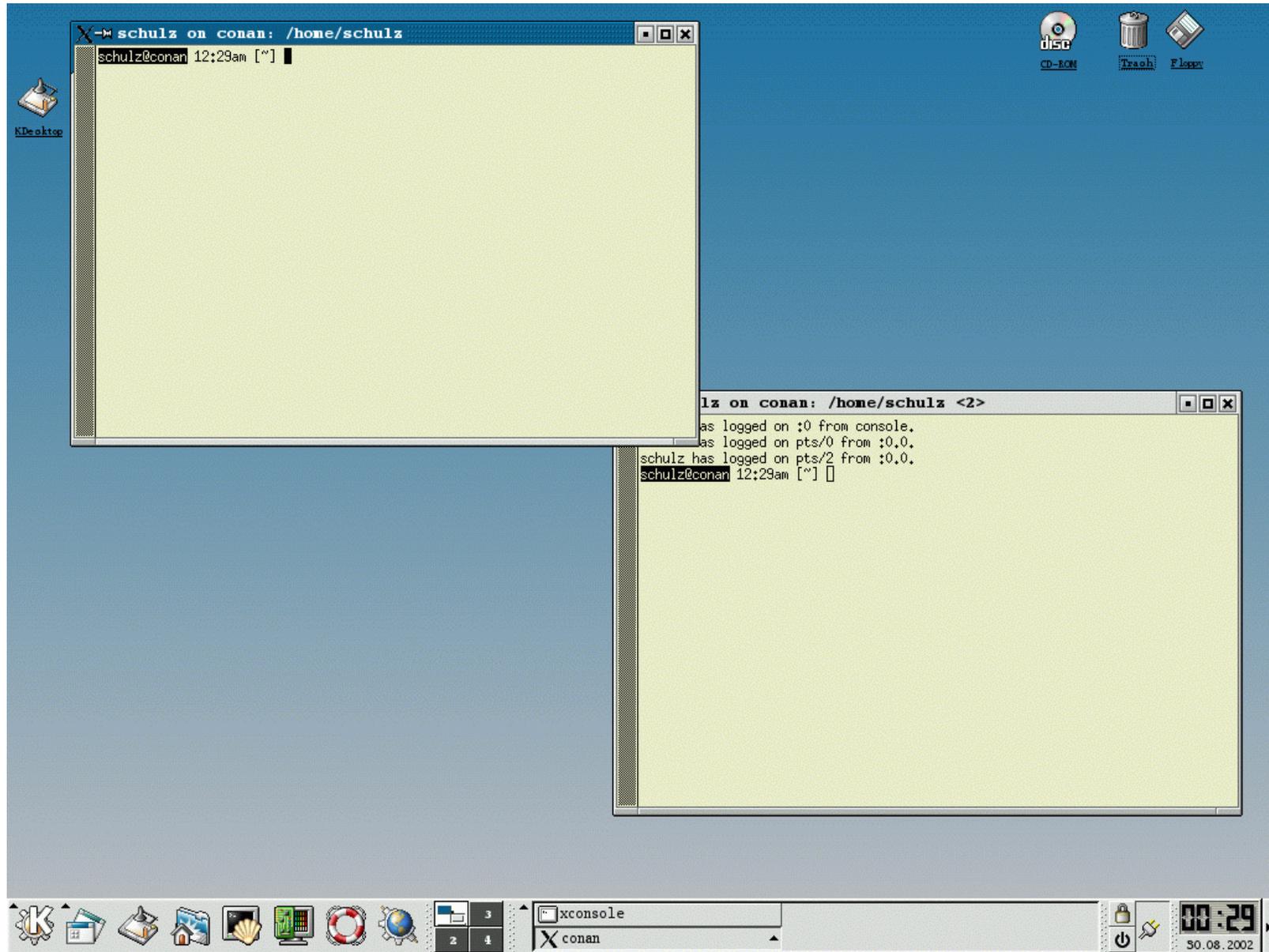
## My Desktop

- Uses windowing mostly to provide a better **text**-based interface (compared to pure text terminals)
- Text editor and shell (command line) windows
- (Can also run graphical applications)

## KDE Desktop

- Graphical, mouse-based user experience
- Mostly a launcher for **GUI**-based programs
  - \* Office programs
  - \* Graphics programs
  - \* Web browser
- **Can also run shell windows!**

# KDE Desktop with Terminal Application



## Exploring the Text Interface

Convention: System output is shown in typewriter font, user input is written in **bold face**, and comments (not to be entered) are written in *italics*.

**whoami** will print the user name of the current user (more exactly: It will print the first user name associated with the **effective user id**)

```
[schulz@gettysburg ~]$ whoami  
schulz
```

**pwd** prints the current **working directory** (more later):

```
[schulz@gettysburg ~]$ pwd  
/lee/home/graph/schulz
```

*Non-standard setup!*

**ls** lists the files in the current working directory:

```
[schulz@gettysburg ~]$ ls  
core Desktop
```

*Not much there at the moment*

## Text Interface Example (contd.)

Most UNIX programs accept **options** to modify their behavior. One-letter (“short”) options start with a single dash, followed by a letter:

```
[schulz@gettysburg ~]$ ls -a (Show all files, even hidden ones)
```

```
.                .gnome
..               .ICEauthority
.bash_logout     .kde
.bash_profile    .mcp
.bashrc          .MCOP-random-seed
core             .mcp.rc
.DCOPserver_hopewell.cs.miami.edu .screenrc
.DCOPserver_potomac.cs.miami.edu  .ssh
.DCOPserver_richmond.cs.miami.edu .tcshrc
Desktop          .xauth
.emacs           .Xauthority
.first_start_kde .xsession-errors
```

As you can see, hidden files start with a dot.

# The UNIX File System

In UNIX, all files are organized in a single **directory tree**, regardless of where they are stored physically

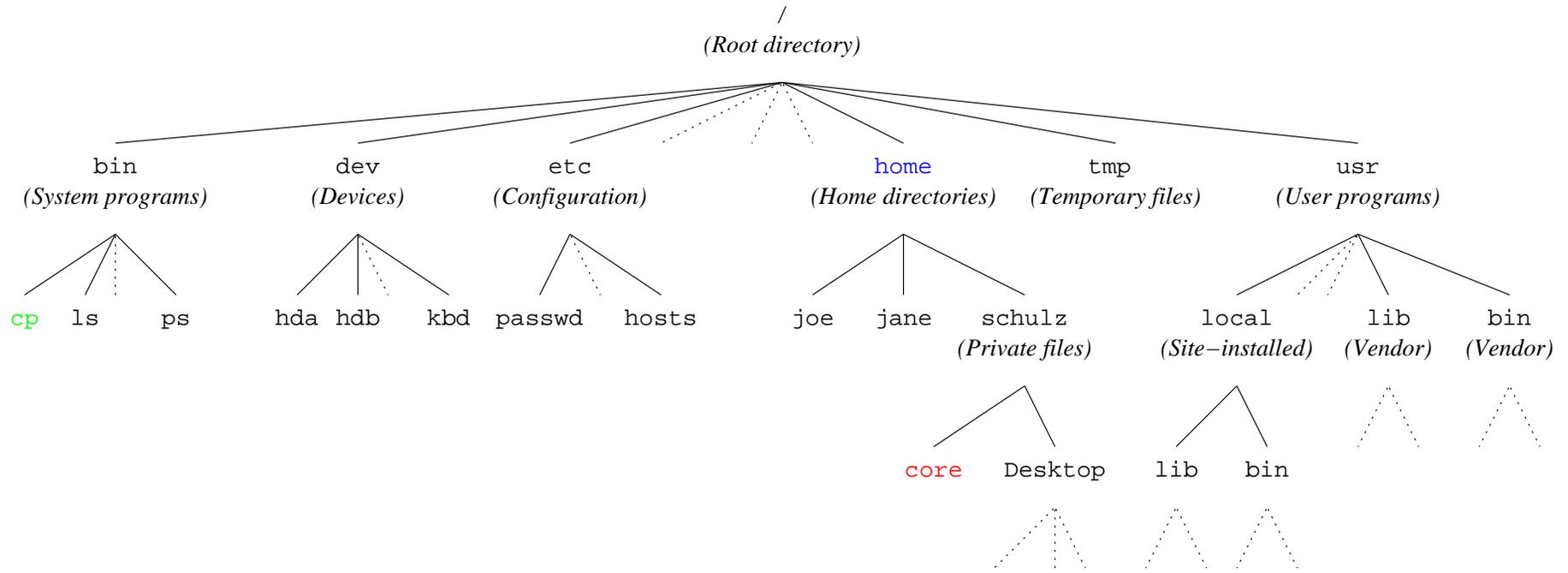
There are two main types of files:

- Plain files (containing data)
- Directories (“folders”), containing both plain files (optionally) and other directories

Each file in a directory is identified by its name and has a number of attributes:

- Name
- Type
- Owner
- Group (each file belongs to one group, even if the owner belongs to multiple groups)
- Access rights
- Access dates

# Typical File System Layout



Files in the directory trees are described by **pathnames**

- Pathnames consist of file names, separated by slashes (/)
- Absolute pathnames start with a /. /bin/cp denotes **cp**
- Relative pathnames are interpreted relative to the current working directory. If **/home** is the current working directory, then schulz/core denotes **core**

# Moving Through the File System

We can use the command **cd** to change our working directory:

```
[schulz@gettysburg ~]$ pwd
/lee/home/graph/schulz
cd /
[schulz@gettysburg /]$ pwd
/
[schulz@gettysburg /]$ cd bin
[schulz@gettysburg /bin]$ pwd
/bin
[schulz@gettysburg /bin]$ cd /lee/home/graph/schulz
[schulz@gettysburg ~]$ pwd
/lee/home/graph/schulz
```

Each directory contains two special entries: `.` and `..`

- `.` represents the directory itself. **cd .** is a NOP
- `..` normally represents the parent directory. **cd ..** moves the working directory up one level. In `/`, `..` points to `/` itself

## More about files

We can use the `-l` (“long format”) option to `ls` to show us all all attributes

```
[schulz@gettysburg ~]$ ls -l
-rw----- 1 schulz users 1531904 Aug 29 10:55 core
drwxr-xr-x 3 schulz users 4096 Aug 29 10:55 Desktop
```

The long format of `ls` shows us more about the files:

- The first letter tells us the **file type**. `d` is a directory, `-` means a plain file
- The next nine letters describe **access rights**, i.e. who is allowed to read, write, and execute the file. More on those later!
- The next number is the number of (hard) **links** to a file. More on that much later!
- Next is the **user** that owns the file
- After that, the **group** that owns the file
- Next comes the **file size** in bytes
- Then the **date** the file was changed for the last time
- Finally, the **name** of the file

# UNIX Online Documentation 1

The UNIX Programmer's Manual ("man pages")

- Traditionally available on every UNIX system, quite terse
- Usage: **man [section]** <command>
- Sections (may differ by UNIX flavour):
  1. **User commands**
  2. System calls
  3. **C library routines**
  4. Device drivers and network interfaces
  5. File formats
  6. Games and demos
  7. Misc. (ASCII, macro packages, tables, etc)
  8. Commands for system administration
  9. Locally installed manual pages. (i.e. X11)
- **man -k** <term> gives you a list of pages relevant to <term>
- To leave the man program (or rather the pager it uses), hit **q**

# UNIX Online Documentation 2

## GNU info files

- Available with most Linux systems and most GNU packages
- Usage: **info** <command>, then browse interactively
- You can also use the info reader build into GNU Emacs
  - \* Enter emacs, then type C-h i, then select topic
  - \* If you do not use Emacs, you should ;-)
  - \* . . . but we will introduce it later on

# Exercises

Move through the file system using **cd**. You can inspect most files using **more <file>** if they are ASCII text. Try e.g. `/etc/passwd` and `/etc/hosts`.

Try **man man** and **info info**

Read the man and info documentation for

- **ls**
- **whoami**
- **cd**
- **pwd**

Don't worry if you don't understand everything!  
(Do worry if you understand nothing!)

**CSC322**

**C Programming and UNIX**

**UNIX from a User's Perspective II**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Command Format

Normal UNIX command format: **<command>** **<arg<sub>1</sub>>**. . . **<arg<sub>n</sub>>**

- The first word is interpreted as a command
- The remaining words (separated by spaces or blanks) are **arguments**
- The implementation of a command is free in how it treats the arguments
- Convention: Arguments starting with a dash – are **options**

Many characters have special meaning in most shells, including \$, (, ), [, ], \*, &, |, ;, \, , ' , " , ' ' (blank, the argument separator)

- Arguments may be enclosed in single quotes ( ' ' ) or in double quotes ( " " ) to suppress most special meanings
  - \* Single quotes suppress (nearly) all special meanings
  - \* Double quotes suppress most special meanings
  - \* In particular, both suppress the meaning of blank: A string 'a a' will appear as a single argument to a command
  - \* Quotes are **not** passed on to the command!
- The backslash \ can be used to suppress the special meaning of individual characters. \" represents a double quote, \\ a backslash character

# Command Types

There are different types of commands a shell can execute:

Shell built-in commands are executed directly by the shell

- Examples: **cd**, **pwd**, **echo**, **alias**

Shell functions are user-defined shell extensions

- Particularly useful in **scripting**, rare in interactive use

Executable programs (the normal case) are loaded from the disk and executed

- Examples: **ls**, **whoami**, **man**
- If a **pathname** is given, that file is executed (if possible)
- If just a **filename** is given, bash searches in all directories specified in the variable `$PATH`
- Note that neither `.` nor `~` are necessarily in `$PATH`!

# UNIX User Commands: echo and touch

**echo** <arg<sub>1</sub>>. . . prints its arguments to the screen

– **echo** is often a shell built-in command. To guarantee the behavior described in the man-page, use **/bin/echo**

– Example:

```
[schulz@gettysburg ~]$ echo "Hello World"
```

```
Hello World      (simplest "Hello World" program in UNIX)
```

```
[schulz@gettysburg ~]$ echo '$PATH = ' $PATH
```

```
$PATH = ./usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr  
r/java/jdk1.3.1_01/bin:/home/graph/schulz/bin:/usr/X11R6/bin
```

**touch** <file<sub>1</sub>> . . . sets the access and modification time of the given files to the current time

– If one of the files does not exist, **touch** will create an empty file of that name

– Important option:

\* **-c**: Do **not** create non-existing files (long form **--no-create** is supported by modern implementations (GNU))

\* Other options: **man touch**

# UNIX User Commands: rm, mkdir, rmdir

**rm** <file<sub>1</sub>>, . . . will delete the named files

– Important options:

- \* **-f**: **F**orce removal, never ask the user (even if the user has withdrawn write permission for that file)
- \* **-i**: **I**nteractively ask the user about each file to be deleted
- \* **-r**: If some of the files are directories, **r**ecursively delete their contents first, then delete them

**mkdir** <file<sub>1</sub>>. . . will create the directories named (if the user has the permission to do so)

**rmdir** <file<sub>1</sub>>. . . will delete the directories named (if the user has the permission to do so and if they are empty)

# Effective Shell Use: History

Modern shells like the `bash` or the `tcsh` keep a **history** of your previous commands

- Type **history** to see these commands
- Type **!`<number>`** re-execute the command with the given number
- Type **!`<command>`** to re-execute the most recent command starting with the (partial) word `<command>`

Example:

```
[schulz@gettysburg ~]$ history
```

```
(...many entries omitted)
```

```
194 more CSC322.tex
```

```
195 gv CSC322_1.pdf
```

```
196 ls
```

```
197 ll CSC322_1.pdf
```

```
198 history
```

- **!`197`** will execute `ll CSC322_1.pdf`
- **!`g`** will execute `gv CSC322_1.pdf`

# Effective Shell Use: Editing/Completion

While typing commands, `bash` offers you many ways to ease your task:

- **[Backspace]** will delete the character preceding the cursor
- **[C-d]** (hold down **[CTRL]**, then press **[d]**) will delete the character under the cursor (if there is such a character)
- **[C-k]** will delete all characters under and right of the cursor
- Left arrow and right arrow move the cursor in the command line (alternatively, try **[C-b]** and **[C-f]**)
- **[C-a]** and **[C-e]** move to the begin and end of the line, respectively
- Up arrow and down arrow will move you through the history (as will **[C-p]** and **[C-n]**)!
- In general, default `bash` key bindings are inspired by `emacs` editing commands

One of the more intriguing features: **Name completion**

- At any time, hit **[TAB]**, and `bash` will complete the current word as far as possible. Hitting **[C-d]** at the end of a non-empty line will list possible completions
- It is quite smart (configurably smart, in fact) about this

# Effective Shell Use: Globbing

Idea: Use simple patterns to describe **sets** of filenames

A string is a **wildcard pattern** if it contains one of **?**, **\*** or **[**

A wildcard pattern expands into all file names matching it

- A normal letter in a pattern matches itself
- A **?** in a pattern matches any one letter
- A **\*** in a pattern matches any string
- A pattern **[l<sub>1</sub>. . . l<sub>n</sub>]** matches any one of the enclosed letters (exception: **!** as the first letter)
- A pattern **[!l<sub>1</sub>. . . l<sub>n</sub>]** matches any one of the characters **not** in the set
- A leading **.** in a filename is never matched by anything except an explicit leading dot
- For more: **man 7 glob**

Important: Globbing is performed **by the shell!**

## Example: File Handling and Globbing

```
$ mkdir TEST_DIR
$ cd TEST_DIR
$ touch a ba bba bbba bbbba bbbba LongFilename .LongHiddenFile
$ ls -a
.  ..  a ba bba bbba bbbba bbbba LongFilename .LongHiddenFile
$ echo *a* (Everything with an a anywhere)
a ba bba bbba bbbba bbbba LongFilename
$ echo *Long*
LongFilename (Note: Does not match .LongHiddenFile)
$ echo .* (all hidden files)
.  ..  .LongHiddenFile
$ echo [ab]*
a ba bba bbba bbbba bbbba
$ echo *[ae] (everything that ends in a or e)
$ echo ?*[ae] (everything that ends in a or e and has at least one more letter)
ba bba bbba bbbba bbbba LongFilename
```

## Example: File Handling and Globbing (Contd.)

```
$ cd ..  
$ rmdir TEST_DIR  
rmdir: 'TEST_DIR': Directory not empty  
$ rm TEST_DIR/*  
rmdir: 'TEST_DIR': Directory not empty  
$ rmdir TEST_DIR  
$ rm TEST_DIR/.L*  
$ rmdir TEST_DIR
```

Alternative:

```
$ mkdir TEST_DIR  
$ touch TEST_DIR/.HiddenFile  
$ rmdir TEST_DIR  
rmdir: 'TEST_DIR': Directory not empty  
$ rm -r TEST_DIR
```

# UNIX User Commands: cat/more/less

**cat** <file<sub>1</sub>> . . . will concatenate the named files and print them to **standard output** (by default, your terminal)

- It's usually just used to display short files ;-)

**more** and **less** are **paggers**

- Each will show you a text (e.g. the contents of a file given on the command line) by pages, stopping after each page and waiting for a key press (normally [space])
- Major differences:
  - \* **more** will automatically exit at the end of the data, **less** requires explicit termination with **[q]**
  - \* **less** allows you to scroll backwards (using **[p]**), **more** only allows scrolling forward
- For more (or less): **man more**, **man less**

# Text Editing under UNIX

There are 3 ways to edit text under UNIX:

1. The **vi** way
2. The **emacs** way
3. The wrong way

**vi** (the **v**isual editor) is the text editor written by Bill Joy for BSD UNIX (published about 1978)

- Screen-oriented WYSIWYG editor (for plain text)
- Available on just about any UNIX system
- About 35% of all serious UNIX hackers still prefer **vi** (or a derivative)!
- Current version on Lab machines: **vim 5.8.7** (**V**i **I**mproved)

**emacs** (**e**ditting **mac**ros) started in 1976 as a set of TECO macros on ITS

- Currently popular **emacs** versions (GNU Emacs and XEmacs) go back to 1985 GNU Emacs by Stallman. Both basically are a LISP system with a large text editing library and an editor-like user interface
- About 35% of all serious UNIX hackers use Emacs. Also widespread use on other operating systems
- **emacs** on the lab machines is **GNU Emacs 20.7.1**

# vi flyby

Getting into it: **vi <file>**

**Modal** interface: Normally letters denote editing commands, only in **insert mode** can actual letters be typed into the file

The editor starts in **command mode** (see next slide)

Insert mode (shows {-- INSERT --} in bottom line):

Key	Effect
<b>[ESC]</b>	Go back to command mode
Any normal key	Insert corresponding letter
<b>[Backspace]</b>	Delete last typed letter

Tutorials e.g. at [http://www.cfm.brown.edu/Unixhelp/vi\\_.html](http://www.cfm.brown.edu/Unixhelp/vi_.html).

## vi flyby II

Command mode (commands marked (\*)) change into insert mode):

Key(s)	Effect
Cursor keys	Move around
<b>:r &lt;file&gt;</b>	Insert file content at cursor position
<b>:w</b>	Write file
<b>:q</b>	Leave vi
<b>:wq</b>	Write file and leave
<b>:q!</b>	Leave vi even if unsafed changes
<b>:h</b>	Help!
<b>i</b>	Insert text at the cursor position (*)
<b>a</b>	Insert text after the cursor position (*)
<b>A</b>	Insert text at the end of the current line (*)
<b>o</b>	Open a new line and insert text (*)
<b>j</b>	Join two lines into one
<b>x</b>	Delete character under cursor
<b>dd</b>	Delete current line
<b>.</b>	Repeat last command
<b>:&lt;no&gt;</b>	Goto line number <no>

# Emacs for Everyone

Getting into it: **emacs** <file> or just **emacs** & (remark: Normally, **emacs** is only started once, and you **visit** different files from within the editor. Emacs can work on many files at once)

Emacs is extremely configurable and extendable:

- Special modes support nearly all programming languages
  - \* Indentation
  - \* Compilation/Error correcting
  - \* Debugging
- You can read email and USENET news in emacs
- Emacs can be used as a web browser

An Emacs window normally has different sub-regions:

- Menu bar (operate with a mouse, many frequently used commands)
- One or more text **windows**, each displaying a **buffer** (a text editing area)
- One **mode line** for each text window, displaying various pieces of information
- Finally, the **mini-buffer** for typing complex commands and dialogs

# Emacs for Everyone II

```
emacs@gettysburg.cs.miami.edu
Buffers Files Tools Edit Search Mule Emacs-Lisp Help
Copyright (c) 1985 Free Software Foundation, Inc; See end for conditions.
You are looking at the Emacs tutorial.

Emacs commands generally involve the CONTROL key (sometimes labeled
CTRL or CTL) or the META key (sometimes labeled EDIT or ALT). Rather than
write that in full each time, we'll use the following abbreviations:

C-<chr> means hold the CONTROL key while typing the character <chr>
Thus, C-f would be: hold the CONTROL key and type f.
M-<chr> means hold the META or EDIT or ALT key down while typing <chr>.
If there is no META, EDIT or ALT key, instead press and release the
ESC key and then type <chr>. We write <ESC> for the ESC key.

Important note: to end the Emacs session, type C-x C-c. (Two characters.)
The characters ">>" at the left margin indicate directions for you to
try using a command. For instance:

:-- TUTORIAL (Fundamental)--L18--Top
:; Red Hat Linux default .emacs initialization file
:; Are we running XEmacs or Emacs?
(defvar running-xemacs (string-match "XEmacs\\|Lucid" emacs-version))
:; Set up the keyboard so the delete key on both the regular keyboard
:; and the keypad delete the character under the cursor and to the right
:; under X, instead of the default, backspace behavior.
(global-set-key [delete] 'delete-char)
(global-set-key [kp-delete] 'delete-char)
:; Turn on font-lock mode for Emacs
(cond ((not running-xemacs)
      (global-font-lock-mode t)
      ))
:; Always end a file with a newline
(setq require-final-newline t)
:; Stop at the end of the file, not just add lines
(setq next-line-add-newlines nil)
:; Enable wheelmouse support by default
(if (not running-xemacs)
    (require 'mwheel) ; Emacs
    )
:-- .emacs (Emacs-Lisp)--L1--Top
Fontifying .emacs... (regexps.....)
```

# Emacs for Everyone III

Emacs is non-modal, normal keys always insert the corresponding letter

Commands are typed by using **[CTRL]** or **[ALT]** in combination with normal keys. We write e.g. **[C-a]** or **[M-a]** to denote **[a]** pressed with **[CTRL]** or **[ALT]** (**M** for **meta**). **[C-h t]** is **[C-h]** followed by plain **[t]**.

Key(s)	What it does
<b>[C-h t]</b>	Enter the emacs tutorial
<b>[C-x C-c]</b>	Leave emacs
<b>Cursor keys</b>	Move around
<b>[C-x C-f]</b>	Open a new file (*)
<b>[C-x C-s]</b>	Save current file
<b>[C-x s]</b>	Save all changed files (*)
<b>[M-x]</b>	Call arbitrary LISP function by name (*)
<b>[C-s]</b>	Incremental search (try it!) (*)

Entries marked with (\*) will ask for additional information in the mini-buffer

## Exercises

Experiment with bash command line editing and history

Create some files and play with globbing

Write a short text in both **vi** and **emacs**

Read the **vi** and **emacs** tutorials

Note: You are strongly encouraged to learn basics of both editors, and to become proficient in at least one of them. I'll not examine you about either, but don't complain if you have trouble with any other editor

# ed is the standard text editor

When I log into my Xenix system with my 110 baud teletype, both vi  
\*and\* Emacs are just too damn slow. They print useless messages like,  
'C-h for help' and '"foo" File is read only'. So I use the editor  
that doesn't waste my VALUABLE time.

Ed, man! !man ed

ED(1)

UNIX Programmer's Manual

ED(1)

## NAME

ed - text editor

## SYNOPSIS

ed [ - ] [ -x ] [ name ]

## DESCRIPTION

Ed is the standard text editor.

--

Computer Scientists love ed, not just because it comes first  
alphabetically, but because it's the standard. Everyone else loves ed  
because it's ED!

"Ed is the standard text editor."

And ed doesn't waste space on my Timex Sinclair. Just look:

```
- -rwxr-xr-x  1 root          24 Oct 29  1929 /bin/ed
- -rwxr-xr-t  4 root      1310720 Jan  1  1970 /usr/ucb/vi
- -rwxr-xr-x  1 root  5.89824e37 Oct 22  1990 /usr/bin/emacs
```

Of course, on the system \*I\* administrate, vi is symlinked to ed. Emacs has been replaced by a shell script which 1) Generates a syslog message at level LOG\_EMERG; 2) reduces the user's disk quota by 100K; and 3) RUNS ED!!!!!!

"Ed is the standard text editor."

Let's look at a typical novice's session with the mighty ed:

```
golem> ed
```

```
?
```

```
help
```

```
?
```

?  
?  
quit  
?  
exit  
?  
bye  
?  
hello?  
?  
eat flaming death  
?^C  
?  
^C  
?  
^D  
?

- ---

Note the consistent user interface and error reportage. Ed is generous enough to flag errors, yet prudent enough not to overwhelm the novice with verbosity.

"Ed is the standard text editor."

Ed, the greatest WYGIWYG editor of all.

ED IS THE TRUE PATH TO NIRVANA! ED HAS BEEN THE CHOICE OF EDUCATED AND IGNORANT ALIKE FOR CENTURIES! ED WILL NOT CORRUPT YOUR PRECIOUS BODILY FLUIDS!! ED IS THE STANDARD TEXT EDITOR! ED MAKES THE SUN SHINE AND THE BIRDS SING AND THE GRASS GREEN!!

When I use an editor, I don't want eight extra KILOBYTES of worthless help screens and cursor positioning code! I just want an EDitor!! Not a "viitor". Not a "emacsitor". Those aren't even WORDS!!!! ED! ED! ED IS THE STANDARD!!!

TEXT EDITOR.

When IBM, in its ever-present omnipotence, needed to base their "edlin" on a UNIX standard, did they mimic vi? No. Emacs? Surely you jest. They chose the most karmic editor of all. The standard.

Ed is for those who can \*remember\* what they are working on. If you are an idiot, you should use Emacs. If you are an Emacs, you should not be vi. If you use ED, you are on THE PATH TO REDEMPTION. THE

SO-CALLED "VISUAL" EDITORS HAVE BEEN PLACED HERE BY ED TO TEMPT THE FAITHLESS. DO NOT GIVE IN!!! THE MIGHTY ED HAS SPOKEN!!!

?

**CSC322**

**C Programming and UNIX**

**UNIX from a User's Perspective  
The Goodies**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# UNIX User Commands: grep

Usage: **grep** <regexp> <file<sub>1</sub>> . . .

- **grep** will scan the input file(s) and print all lines containing a string that matches the **regular expression** <regexp>
- Important options:
  - \* **-i**: Ignore upper and lower case in the regular expression
  - \* **-v**: Print all lines **not** matching the regular expression
- The name comes from an old editor command sequence standing for **g**lobally search for **r**egular **e**xpression, **p**rint matches
- It is one of the most useful UNIX tools!

Regular expressions (much more by **man grep**):

- A normal character matches itself
- A **.** matches any normal character
- A **\*** after a pattern matches any number of repetitions
- A range **[...]** works as for **globbing** (but use **^** instead of **!** for negation)
- Remember that many character are special for the shell – use quotes!
- Example: **grep "Ste.\*ulz" <file>** will find many versions of my full name in <file>

# Input and Output

Each UNIX process normally is created with 3 input/output streams:

- Standard Input or `stdin` (file descriptor 0) is used for normal input. Many UNIX programs will read from `stdin`, if no file names are given
- Standard Output or `stdout` (file descriptor 1) is used for all normal output
- Standard Error or `stderr` (file descriptor 2) is used for **out of band** output like warnings or error messages

By default, all three are connected to your terminal

It is possible to redirect the output streams into files

It is possible to make `stdin` read from a file

It is possible to connect one processes `stdout` to another ones `stdin`

# Simple Output Redirection

You can redirect the normal output of a command by appending `> <file>` to the command.

- Example 1:  
**\$ man stdin > stdin\_man\_page**  
**\$ more stdin\_man\_page**

```
STDIN(3)                               System Library Functions Manual                               STDIN(3)
```

```
NAME
```

```
    stdin, stdout, stderr - standard I/O streams
```

```
...
```

- Example 2: On the lab machines, the global password file is served over the NIS (or Yellow Pages) protocol, and is shown by the command **ypcat passwd**. **ypcat passwd > my\_passwd** gives you a private copy for password “quality checking”
- Example 3: **cat > myfile.c** can replace a text editor (hit **[C-d]** on a line of its own to indicate the end of input)

# Output Redirection II

By default, `stderr` is not redirected, so you can still see error messages on the terminal (and discard the normal output if it is useless)

To redirect `stderr`, use `2>` (redirect file descriptor 2):

- `$ man bla` will print `No manual entry for bla`
- `$ man bla 2> error` will save that error message in the file **error**

Special case: If you are not interested in any output, you can redirect it to `/dev/null` (a UNIX device file that just accepts data and ignores it):

- `$ man bla 2> /dev/null` will make sure that you do not see the error message
- Alternatively, `$ man if bla > /dev/null` will give you just the error message (even though `man` also prints the man page for the shell-built-in if)

# Input Redirection

You can also redirect the `stdin` file descriptor to read **from** a file

- Append `< <file>` to the command
- This is e.g. useful if you use an interactive program always for the same task (i.e. you always type the same data into the file)
- Some UNIX commands only accept input on `stdin` (e.g. the **tr** utility)

Example: **cat** `< file` is equivalent to **cat file**! (**Why?**)

# Shell Pipes

Pipes are a general tool for **inter-process communication** (IPC)

The shell allows us to easily set up pipes connecting stdout of one process to **stdin** of another

Example: **man bash | cat** will print the bash man page **without** using the pager

- This can be chained: **man bash | grep -i redir | grep -i input** will print just the lines containing information about input redirection
- **ypcat passwd | grep schulz** will give you just my entry in the password file

# Basic Process Control

You can start processes in the **foreground** or in the **background**

- Foreground processes are started by just typing a normal command
- Background processes are started by appending an ampersand (&) to the command. This is particularly useful for graphical applications: **emacs &**
- While a foreground process is running, the shell is blocked because the process is using the terminal as its `stdin` (i.e. you can have at most one non-suspended foreground process)
- (Most) foreground processes can be **terminated** by hitting **[C-c]** (often written as `^C`).
- (Most) foreground processes can be **suspended** by hitting **[C-z]**
- A suspended process can be continued by typing **fg** (to continue it as a foreground process) or **bg** (to let it run in the background)
- A background process will be suspended automatically, if it needs to read data from `stdin`
- **jobs** gives a numbered list of all processes started by the shell
- You can use **fg %<no>** to take a particular process into the foreground (**bg %<no>** works on the same principle)
- You can use **kill %<no>** to terminate the named job

# UNIX User Commands: Yes

Usage: **yes** [**arg**]

If no argument is given, **yes** will print an infinite sequence of lines containing just the character **y**

If an argument is given, **yes** will print an infinite sequence of lines containing that argument

Very little more is available by printing **man yes**

## Job Control Example

```
$ emacs &  
$ yes Hello
```

*(Start emacs in the background – it opens its own window)*  
*(Start yes in the foreground)*

```
Hello
```

```
Hello
```

```
Hello
```

```
...
```

```
^C
```

*(Enough of that)*

```
$ jobs
```

```
[1]      Running
```

```
emacs
```

*(Just my emacs)*

```
$ yes Hi
```

*(I can never get enough)*

```
Hi
```

```
Hi
```

```
...
```

```
^Z
```

*(Suspend it)*

```
Suspended
```

*(Indeed!)*

```
$ jobs
```

```
[1]      Running
```

```
emacs
```

```
[2]  + Suspended
```

```
yes
```

```
$ kill %1
```

*(Ooops! Emacs window closes)*

## **Notice: Lab Hours**

At the moment, a TA for CSC322 is in the lab Friday 4-6pm and Sunday 2-6pm.

**CSC322**  
**C Programming and UNIX**  
**Programming in C - Basics**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# A Bird's Eye View of C

A C program is a collection of

- Declarations
- Definitions

for

- Functions
- Variables
- Datatypes

A program may be spread over multiples files

A program file may contain preprocessor directives that

- Include other files
- Introduce and expand macro definitions
- Conditionally select certain parts of the source code for compilation

# A First C Program

Consider the following C program

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

int main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

Assume that it is stored in a file called `hello.c` in the current working directory.  
Then:

```
$ gcc -o hello hello.c
```

*(Note: Compiles without warning or error)*

```
$ ./hello
```

```
Hello World!
```

# A Closer Look (1)

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

int main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

We are including two **header files** from the **standard library**

- `stdio.h` contains declarations for buffered, stream-based input and output (we include it for the declaration of `printf`)
- `stdlib.h` contains declarations for many odds and ends from the standard library (it gives us `EXIT_SUCCESS`)
- In general, preprocessor directives start with a hash `#`

## A Closer Look (2)

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

int main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

The program consist of one function named `main()`

- `main()` returns a `int` (integer value) to its calling environment
- In this case, it takes no arguments (its argument list is `void`)
- In general, any C program is started by a call to its `main()` function, and terminates if `main()` returns

## A Closer Look (3)

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

int main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

The **function body** contains two statements:

- A call to the **standard library function** `printf()` with the argument "Hello World!\n" (a string ending with a newline character)
- A return statement, returning the value of the symbol `EXIT_SUCCESS` to the caller of `main()`

## A Closer Look (4)

**gcc** is the GNU C compiler, the standard compiler on most free UNIX system (and often the preferred compiler on many other systems)

- On traditional systems, the compiler is normally called **cc**

**gcc** takes care of all stages of compiling:

- Preprocessing
- Compiling
- Linking

It automagically recognizes what to do (by looking at the file name suffix)

Important options:

- **-o <name>**: Give the name of the output file
- **-ansi**: Compile strict ANSI-89 C only
- **-Wall**: Warn about all dubious lines
- **-c**: Don't perform linking, just generate a (linkable) object file
- **-O** – **-O6**: Use increasing levels of optimization to generate faster executables

## A More Advanced Example

```
/* A program that prints a Fahrenheit -> Celsius conversion table */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fahrenheit, celsius;

    printf("Fahrenheit -> Celsius\n\n");

    fahrenheit = 0;
    while(fahrenheit <= 300)
    {
        celsius = (fahrenheit-32)*5/9;

        printf("%3d   %3d\n", fahrenheit, celsius);
        fahrenheit = fahrenheit + 10;
    }
    return EXIT_SUCCESS;
}
```

# The Fahrenheit-Celsius Example

Compilation:

```
$ gcc -ansi -Wall -W -o celsius_fahrenheit celsius_fahrenheit.c
```

Running it:

```
$ ./celsius_fahrenheit | more
```

```
Fahrenheit -> Celsius
```

```
  0   -17
 10   -12
 20    -6
 30    -1
 40     4
 50    10
 60    15
 70    21
 80    26
 90    32
100    37
--More--
```

# Comments

Comments in C are enclosed in `/*` and `*/`

Comments can contain any sequence of characters except for `*/` (although your compiler may complain if it hits a second occurrence of `/*` in a comment)

Comments can span multiple lines

In assignments (and in live) use comments wisely

- Do explain important ideas, like i.e. what a function or program does
- Do explain clever tricks (if needed)
- Do **not** repeat things obvious from the program code anyways

Bad commenting will affect grading!

# Variables

“`int fahrenheit, celsius;`” declares two **variables** of type `int` that can store a signed integer value from a finite range

- By intention, `int` is the fastest datatype available on any given C implementation
- On most modern UNIX systems, `int` is a 32 bit type and interpreted in 2s complement, giving a range from -2 147 483 648 — 2 147 483 647. This is **not** part of the C language definition, though!

In general, a **variable** in a program corresponds to a memory location and can store a value of a specific type

- All variables must be declared, before they can be used
- Variables can be **local** to a function (like the variables we have used so far), local to a single source file, or global to the hole program

A variables value is changed by an **assignment**, an expression of the form  
“`var = expression;`”

## (Arithmetic) Expressions

- C supports various arithmetic operators, including the usual `+`, `-`, `*`, `/`
- Subexpressions can be grouped using parentheses
  - Normal arithmetic operations can be used on both integer and floating point values, with the type of the arguments determining the type of the result
  - Example: `(fahrenheit-32)*5/9` is an arithmetic expression in C, implementing the well-known formula  $C = \frac{5}{9}(F - 32)$  for converting Fahrenheit to Celsius
    - \* Since all arguments are integer, all intermediate results are also integer (as well as the final result)
    - \* Therefore we have to multiply with 5 first, then divide by nine (multiplying with `(5/9)` would effectively multiply with 0)

Bit-wise, logical and operator comparison operators also normally also return numeric values

Possible operands include variables, numerical (and other) constants, and function calls

Note: In C, **any** normal statement is an expression and has a value, including the assignment!

# Simple Loops

A **while-loop** has the form

```
while(<expr>)  
  <body>
```

where <body> either can be a single statement, terminated by a semicolon ';', or a statement block, included in curly braces '{}'

It operates as follows:

- At the beginning of the loop, the **controlling expression** is evaluated
- If it evaluates to a non-zero value, the loop body is executed once, and control returns to the `while`
- If it evaluates to 0, the body is skipped, and the program continues on the next statement after the loop

Note: The body can also be empty (but this is usually a programming bug)

# Formatted Output

`printf()` is a function for **formatted output**

It has at least one argument (the **format string**), but may have an arbitrary number of arguments

- The control string may contain various placeholders, beginning with the character `%`, followed by (optional) formatting instructions, and a letter (or letter combination) indicating the desired output format
- Each placeholder corresponds to exactly one of the additional arguments (modern compilers will complain, if the arguments and the control string do not match)
- In particular, `%3d` requests the output of a normal `int` in decimal representation, and with a width of at least 3 characters

Note: `printf()` is not part of the C language proper, but of the (standardized) C library

# UNIX User Commands: cp and mv

**cp** will either **copy** one file to another, or it will copy any number of files into a directory

- Usage: **cp** <file<sub>1</sub>> <file<sub>2</sub>>  
Copy <file<sub>1</sub>> to <file<sub>2</sub>>
- Usage: **cp** <file<sub>1</sub>>. . . <file<sub>n</sub>> <dest>  
Copy the named files into <dest>

**mv** will likewise **move** files

- Usage: **mv** <file<sub>1</sub>> <file<sub>2</sub>>  
Move <file<sub>1</sub>> to <file<sub>2</sub>>
- Usage: **mv** <file<sub>1</sub>>. . . <file<sub>n</sub>> <dest>  
Move the named files into <dest>

Warning: Unless used with option **-i**, both commands will happily overwrite existing files!

Again, a more complete description is available by man!

## Assignment(also see Website)

Write the following two C programs:

- `celsius2fahrenheit` should print a conversion table from Celsius to Fahrenheit, from -50 to +150 degrees Celsius, in steps of 5 degrees
- `imp_metric` should print two tables side by side (equivalent to a 4-column) table, one for the conversion from yards into meters, the other one for the conversion from km into miles. The output should use `int` values, but you can use the floating point conversion factors of 0.9144 (from yards to meters) and 1.609344 from mile to km. Try to make the program round correctly!

Sample Output:

Yards	Meters	Km	Miles
0	0	1	1
10	9	2	1
20	18	3	2
30	27	4	2
40	37	5	3
...			
100	91	11	7

# CSC322

## C Programming and UNIX

### Programming in C Extended Introduction

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Statements, Blocks, and Expressions

C programs are mainly composed of **statements**

In C, a statement is either:

- An expression, followed by a semicolon ';' (as a special case, an empty expression is also allowed, i.e. a single semicolon represents the empty statement)
- A flow-control statement (if, while, goto, break. . .)
- A **block** of statements (or **compound statement**), enclosed in curly braces '{}'.  
A compound statement can also include new variable declarations.

Note: The following is actually legal C (although a good compiler will warn you that some of your statements have no effect):

```
{  
    int a;  
  
    10+20;  
    10*(a=printf("Hello\n"));  
}
```

## Flow-Control: if

The primary means for conditional execution in C is the `if` statement:

```
if(<expr>
    <statement>
```

- If the expression evaluates to a non-zero (“true”) value, then the statement will be executed
- `<statement>` can also be a block of statements – in fact, it quite often is good style to always use a block, even if it contains only a single statement

An `if` statement can also have a branch that is taken if the expression is zero (“false”):

```
if(<expr>
    <statement>
else
    <statement>
```

## Flow-Control: while

C supports different **structured** loop constructs, including a standard `while`-loop (see also example from last lesson):

```
while(<expr>)  
    <statement>
```

When control reaches the `while` at the top of the loop, the expression is tested

- If it evaluates to true (non-zero), the body of the loop is executed and control returns to the `while`
- If it evaluates to false (i.e. zero), control directly goes to the statement following the body of the loop

Note: An empty loop body is possible (and sometimes useful)

Again: In many cases it is recommended to use a block even if it contains only one statement (or even no statements)

## Flow-Control: for

The `for`-loop in C is a construct that combines initialization, test, and update of loop variables in one place:

```
for(<expr1>; <expr2>; <expr3>)  
    <statement>
```

- Before the loop is entered, `<expr1>` is evaluated
- Before each loop iteration, `<expr2>` is evaluated
  - \* If it is true, the body is executed, then `<expr3>` is evaluated and control returns to the top of the loop
  - \* If it is false, control goes to the first statement after the body
  - \* In the typical case, both `<expr1>` and `<expr3>` are assignments to the same variable, while `<expr2>` tests some property depending on that variable

## Example

Here is the Fahrenheit/Celsius conversation using for:

```
/* A program that prints a Fahrenheit -> Celsius conversion table */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fahrenheit, celsius;

    printf("Fahrenheit -> Celsius\n\n");
    for(fahrenheit=0; fahrenheit<=300; fahrenheit=fahrenheit+10)
    {
        celsius = (fahrenheit-32)*5/9;
        printf("%3d   %3d\n", fahrenheit, celsius);
    }
    return EXIT_SUCCESS;
}
```

## for vs. while

Note that `for` is more general than `while`:

```
while(<expr>           and      for(;<expr>;)
    <statement>       <statement>
```

are equivalent.

Alternatively, you can achieve the functionality of `for` using just `while` (how?)

The preference for one or the other often is a matter of personal choice

- If there are clear initialization and update steps, `for` is often more convenient
- In other cases, `while` is more natural

# Variable Declarations

Variable names:

- A valid variable name starts with a letter or underscore (`_`), and may contain any sequence of letters, underscores, and digits
- Capitalization is significant – `a_variable` is different from `A_Variable`
- In addition to the language **keywords**, certain other names are reserved (by the standard library or by the implementation). In particular, avoid using names that start with an underscore!

Variable declarations:

- A (simple) variable declaration has the form `<type> <varlist>;`, where `<type>` is a type identifier (e.g. `int`), and `<varlist>` is a comma-separated list of variable names
- In ANSI-89 C, variables can only be declared outside any blocks or directly after an open curly brace. The new standard relaxes this requirement
- A variable declared in a block is (normally) visible just inside that block

# Types: Integers and Characters

C has a large number of **integer** data types:

- It offers `char`, `short`, `int`, `long` and (since the last language revision) `long long` types, all of which may represent integers from different ranges
- Note that in particular `char` is an integer data type, i.e. characters are represented by their numerical encoding in the character set (normally ASCII)
- Any of those can be `signed` or `unsigned`, i.e. capable of representing positive numbers only or both negative and positive numbers
- All types can be freely mixed in expressions
- **Unsigned** types always follow the rules of arithmetic modulo  $2^n$ , where  $n$  is the width (in bits) of their representation (i.e. values greater than  $2^n - 1$  are reduced by subtracting  $2^n$  until the result is in the range  $0 - 2^n - 1$ )

Integer **constants** are normally type `int` if they can be represented by that data type

- `123` is `int`
- `316L` is `long`
- `922U` is `unsigned int`

## Integer Type Table

Type	Alias	Signed/Unsigned?	Width(*)
char		Implementation	8 bit
signed char		Signed	8 bit
unsigned char		Unsigned	8 bit
short int	short	Signed	16 bit
signed short int	short	Signed	16 bit
unsigned short int	unsigned short	Unsigned	16 bit
int		Signed	32 bit
signed int	int	Signed	32 bit
unsigned int	unsigned	Unsigned	32 bit
long int	long	Signed	32 bit
signed long int	long	Signed	32 bit
unsigned long int	unsigned long	Unsigned	32 bit
long long int	long long	Signed	64 bit
signed long long int	long long	Signed	64 bit
unsigned long long int	unsigned long long	Unsigned	64 bit

**Note (\*)**: Width is **not** defined by the language standard but reflects currently common implementation choices!

# Simple Character I/O

The C library defines the three I/O streams `stdin`, `stdout`, and `stderr`, and guarantees that they are open for reading or writing, respectively

Reading characters from `stdin`: `int getchar(void)`

- `getchar()` returns the numerical (ASCII) value of the next character in the `stdin` input stream
- If there are no more characters available, `getchar()` returns the special value `EOF` that is guaranteed different from any normal character (that is why it returns `int` rather than `char`)

Printing characters to `stdout`: `int putchar(int)`

- `putchar(c)` prints the character `c` on the `stdout` stream
- (It returns that character, or `EOF` on failure)

`getchar()`, `putchar()`, and `EOF` are declared in `<stdio.h>`

## Example: File Copying

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

int main(void)
{
    int c;

    c=getchar();
    while(c!=EOF)
    {
        putchar(c);
        c=getchar();
    }
    return EXIT_SUCCESS;
}
```

Copies stdin to stdout – to make a a file copy, use  
**cat file | ourcopy > newfile**

Introduces != (“not equal”) relational operator

## Example: File Copying (idiomatic)

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

int main(void)
{
    int c;

    while((c=getchar())!=EOF)
    {
        putchar(c);
    }
    return EXIT_SUCCESS;
}
```

Remember: Variable assignments have a value!

Improvement: No duplication of call to `getchar()`

## Example: Character Counting

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

int main(void)
{
    int c;
    long count = 0;

    while((c=getchar())!=EOF)
    {
        count++;
    }
    printf("Number of characters: %ld\n", count);
    return EXIT_SUCCESS;
}
```

New idiom: ++ operator (increases value of variable by 1)

Test: **\$ man cat | charcount**  
1091

## Exercises

Write a program that continually increases the value of a `short` and a unsigned `short` variable and prints both (you can use `printf("%6d %6u", var1, var2);` to print them). What happens if you run the program for some time? You can pipe the output into **less** and search for interesting things (**man less** to learn how!). Remember that **[C-c]** will terminate most programs under UNIX!

Write a program that counts lines in the input. Hint: The standard library makes sure that any line in the input ends with a newline (`'\n'`)

Write a program that computes the factorial of a number (given as a constant in the program). Test it for values of 3, 18, and 55. Any observations?

# CSC322

## C Programming and UNIX

### Programming in C More on Expressions

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## Nomination: Most Useless Use of `cat` Award

If **ourcopy** is a program that just copies `stdin` to `stdout`, then

- **`cat file | ourcopy > newfile`** will indeed copy **file** to **newfile**
- So will **`ourcopy < file > newfile`**
- (So will **`cat < file > newfile`**)

# UNIX User Commands: `wc`

Usage: `wc <file1>...`

`wc` counts the bytes, words and lines in each file specified (or in `stdin` if none is given) and print the results, including the total for all input files.

Important options:

- `-c`: Print just the byte count
- `-w`: Print just the word count
- `-l`: Print just the line count

Example:

```
$ wc < wordcount.c
```

```
24      53      369
```

Notice: The program does not print unnecessary headers or footers. The output can easily be interpreted by other programs!

More: `man wc`

## Example: Word Counting

```
/* Count words */
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int c, in_word=0;
    long words = 0;
    while((c=getchar())!=EOF)
    {
        if(c == ' ' || c == '\n' || c == '\t')
        {
            in_word = 0;
        }
        else if(!in_word)
        {
            in_word = 1;
            words++;
        }
    }
    printf("%ld words counted\n", words);
    return EXIT_SUCCESS;
}
```

# Character Constants

In C, characters are just small integers

We can write character constants symbolically, by enclosing them in single quotes:

- `'a'` is the numerical value of a lower case **a** in the character encoding (97 for ASCII)
- `'A'` is upper case **A** (65 for ASCII)
- These values can be assigned to any integer variable!

You can use **escape sequences** (starting with `\`) for non-printable characters:

- `\t` is the **tabulator** character (HT), ASCII 9
- `\n` is the **newline** character (LF), ASCII 10 (and used by C to mark the end of the current line)
- `\a` is the **BEL** character, printing it will normally make the terminal beep
- `\0` is the NUL character, ASCII 0, and used by C to mark the end of a string
- `\\` is the backslash itself, ASCII 92

You can get all C escape sequences (and more) via **man ASCII**

## Another View at Expressions

Expressions are build from **operators** and **operands**, with parentheses for grouping

- Most operators are **unary** (taking one operand) or **binary** (taking two)
- Operands can be
  - \* (Sub-)Expressions
  - \* Constants
  - \* Function calls
- In C, binary operators are written in **infix**, i.e. between the operands:  $10+15$
- Unary operators are written either in prefix or postfix (some can even be written either way, with slightly different effects)

All operators have a **precedence**, defining how to interpret operations with multiple operators

- In the absence of parentheses, operators with a higher precedence bind tighter than those with a lower precedence:  $10+3*4 == 22$  is true,  $10+4*3==42$  is false
- In doubt, we can always fully parenthesize expressions:  $10+3*4 == 10+(3*4)$ , but  $(10+4)*3==42$

# Expressions: Associativity of Binary Operators

Binary operators have an additional property: **Associativity**

– Example:  $25+12+11$  can be interpreted as  $(25+12)+11$  or as  $25+(12+11)$

# Expressions: Associativity of Binary Operators

Binary operators have an additional property: **Associativity**

- Example:  $25+12+11$  can be interpreted as  $(25+12)+11$  or as  $25+(12+11)$
- **Worse:** What about  $25-12-11$ ?

By convention, standard arithmetic operators are **left-associative**, i.e. they bind to the left

- Thus:  $25-12-11 == (25-12)-11$  has the value 2

We will note associativity for many operators specifically, but unless otherwise noted, it's probably left-associative!

# Expressions: Relational Operators

Relational operators take two arguments and return a truth value (0 or 1)

We already have seen the equational operators. They apply to all basic data types and pointers:

- $a == b$  (**equal**) evaluates to 1 if the two arguments have the same value, otherwise it evaluates to 0
- $a != b$  evaluates to 1 if the two arguments have different values
- Note:  $a == b == c$  is evaluated as  $(a == b) == c$ , i.e. it compares  $c$  to either 0 or 1!

We can also compare the same types using the greater/lesser relations:

- $>$  evaluates to 1, if the first argument is greater than the second one
- $<$  evaluates to 1, if the second argument is greater than the first one
- $a >= b$  evaluates to 1, if either  $a > b == 1$  or  $(a == b) == 1$
- $a <= b$  evaluates to 1, if either  $a < b == 1$  or  $(a == b) == 1$

Precedence rule: The relational operators have lower precedence than the arithmetic ones (  $a+1 < 2*b$  makes sense)

# Expressions: Logical Operators

Logical operators operate on truth values, i.e. all non-zero values are treated the same way (representing **true**)

The binary logical operators are `||` and `&&`

- `a||b` evaluates to 1, if at least one of `a` or `b` is non-zero (otherwise it evaluates to 0)
- `a&&b` evaluates to 1, if both `a` and `b` are non-zero
- Both `||` and `&&` are evaluated left-to-right, and the evaluation stops as soon as we can be sure of the result (short-circuit evaluation)
  - \* Example: If `a!=b`, then `(a==b)&&c` will not evaluate `c`
  - \* Similarly: `(a==0 || 10/a >= 1)` will never divide by zero!

`!` is the (unary) logical negation operator, `!a` evaluates to 1, if `a` has the value 0, it evaluates to 0 in all other cases

Precedence rules:

- The binary logical operators have lower precedence than the relational ones
- `||` has lower precedence than `&&`
- `!` has a higher precedence than even arithmetic operators

# Expressions: Assignments

The **assignment operator** is = (a single equal sign)

- a = b is an expression with the value of b
- As a **side effect**, it will change the value of a to that same value

The expression on the left hand side of an assignment (a) has to be an **lvalue**, i.e. something we can assign to. Legal lvalues are

- Variables
- Dereferenced pointers (“memory locations”)
- Elements in a struct, union, or array

The assignment operator is **right-associative** (so you can write `a = b = c = d = 0;` to set all four variables to zero)

The assignment operator has extremely low precedence (lower than all other operators we have covered up to now)

# Floating Point Numbers

C supports three types of **floating point** numbers, `float`, `double`, and `long double`

- **float** is the most memory-efficient representation (typically 32 bits), but has limited range and precision
- **double** is the most commonly used floating point type. In particular, most numerical library functions accept and return **double** arguments. Doubles normally take up 64 bits
- **long double** offers extended range and precision (sometimes using 128 bits) and is a recent addition

Floating point constants are written using a decimal point, or exponential notation (or both):

- `1.0` is a floating point constant
- `1` is an integer constant. . .
- . . . but `1e0` and `1.0E0` are both floating point constants

If we mix integer and floating point numbers in an expression, a value of a “smaller” type is converted to that of the bigger one transparently:

- `9/2 == 4`, but `9/2.0 == 4.5` and `9.0/2 == 4.5`

## Fahrenheit to Celsius – More Exactly

```
/* A program that prints a Fahrenheit -> Celsius conversion table */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fahrenheit;
    double celsius;

    printf("Fahrenheit -> Celsius\n\n");
    for(fahrenheit=0; fahrenheit<=300; fahrenheit=fahrenheit+10)
    {
        celsius = (fahrenheit-32.0)*5.0/9.0;
        printf("%3d   %7.3f\n", fahrenheit, celsius);
    }
    return EXIT_SUCCESS;
}
```

Remark: The `%7.3f` conversion specification prints a float or double with a total width of 7 characters and 3 fractional digits

## Administrative Notes

Please **ssh** to `lee.cs.miami.edu` to use the lab machines over the net.

To change your password on the lab machines, use **yppasswd**. Also check <http://www.cs.miami.edu/~irina/password.html> for the password policy

To submit programming assignments, create a subdirectory with the name `ASSIGNMENT_<no>` (where `<no>` is the number of the current assignment) and copy the relevant files to it

Example: To submit the current assignment, do e.g.

```
$ cd ~ (go home)
```

```
$ mkdir ASSIGNMENT_2
```

```
$ cp mystuff/celsius2fahrenheit* ASSIGNMENT_2
```

```
$ cp mystuff/imp_metric* ASSIGNMENT_2
```

## Excercises

Expand the word count program to count characters, words, and lines (of `stdin`) as `wc` does

Write a program that prints useful imperial to metric (and back) conversion tables to a reasobale precision

# CSC322

## C Programming and UNIX

### Programming in C

### Simple Arrays and Functions

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Arrays

A **array** is a data structure that holds elements of **one type** so that each element can be (efficiently) accessed using an **index**

In C, arrays are always indexed by **integer** values

Indices always run from 0 to some fixed, predetermined value

`<type> <var> [<elements>];` defines a variable of an array type:

- `<type>` can be any valid C type, including user-defined types
- `<var>` is the name of the variable defined
- `<elements>` is the number of elements in the array (Note: Indices run from 0 to `<elements>-1`)

Example: `char x[10];` defines the variable `x` to hold 10 elements of type `char`, `x[5]` accesses the 5th element of that array

## Example: Counting Character Frequencies

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(void)
{
    int freq_count[128];
    int i, c;

    for(i=0; i<128; i++)
    {
        freq_count[i] = 0;
    }
    while((c=getchar())!=EOF)
    {
        if(c <= 127) /* Ignore non-ASCII */
        {
            freq_count[c]++;
        }
    }
}
```

## Example: Counting Character Frequencies (Contd.)

```
for(i=0; i<128; i++)
{
    if(isprint(i))
    {
        printf("%c: %d\n", i, freq_count[i]);
    }
}
return EXIT_SUCCESS;
}
```

Remark: `isprint(i)` is true, if `i` is a printable character. It is defined in `ctype.h`

# Initializing Arrays

In the example, we used an explicit loop to initialize the array

For short arrays we can also list the initial values in the definition of the array:

- `int days_per_month[12] = {31,28,31,30,31,30,31,31,30,31,30,31};`
- The number of values has to be smaller than or equal to the number of elements in the array
- Unspecified elements are initialized to **all bits zero**, (i.e. 0 for all basic data types)

If we give an explicit initializer, we can omit the size of the array:

- `int days_per_month[] = {31,28,31,30,31,30,31,31,30,31,30,31};`
- The compiler will automatically allocate an array of sufficient size to hold all the values in the initializer

# Array Layout

C arrays are implemented as a sequence of consecutive memory locations of the right size to hold the element

Example:

Address	Array Element	Content
0		
...		
112		Other data
120	days_per_month[0]	31
124	days_per_month[1]	28
128	days_per_month[2]	31
132	days_per_month[3]	30
136	days_per_month[4]	31
140	days_per_month[5]	30
144	days_per_month[6]	31
148	days_per_month[7]	31
152	days_per_month[8]	30
156	days_per_month[9]	31
160	days_per_month[10]	30
164	days_per_month[11]	31
168		Other data
...		

# No Safety Belts and No Air Bag!

C does not check if the index is in the valid range!

- If you access `days_per_month[13]` you might change some critical other data
- The **operating system** may catch some of these wrong accesses, but do not rely on it!)

This is source of many of the **buffer-overflow** errors exploited by crackers and viruses to hack into systems!

# Character Arrays

Character arrays are the most frequent kind of arrays used in C

- They are used for I/O operations
- They are used for implementing **string operations** in C

To make the use of character arrays easier, we can use string constants to initialize them. The following definitions are equivalent:

- `char hello[] = {'H', 'e', 'l', 'l', 'o', '\0'};`
- `char hello[] = "Hello";`
- `char hello[6] = "Hello";`

Notice that the string constant is automatically terminated by a NUL character!

# Functions

Functions are the primary means of structuring programs in C

A function is a named subroutine

- It accepts a number of arguments, processes them, and (optionally) returns a result
- Functions also may have **side effects**, like I/O or changes to global data structures
- In C, any subroutine is called a **function**, whether it actually returns a result or is only called for its side effect

Note: A function hides its **implementation**

- To use a function, we only need to know its interface, i.e. its name, parameters, and return type
- We can improve the implementation of a function without affecting the rest of the program

Function can be reused in the same program or even different programs, allowing people to build on existing code

# Function Definitions

A function definition consists of the following elements:

- Return type (or `void`) if the function does not return a value
- Name of the function
- Parameter list
- Function body

The name follows the same rules as variable names

The **parameter list** is a list of coma-separated pairs of the form `<type> <name>`

The body is a sequence of statements included in curly braces

Example:

```
int timesX(int number, int x)
{
    return x*number;
}
```

# Function Calling

A function is **called** from another part of the program by writing its name, followed by a parenthesized list of **arguments** (where each argument has to have a type matching that of the corresponding parameter of the function)

If a function is called, control passes from the call of the function to the function itself

- The parameters are treated as local variables with the values of the arguments to the call
- The function is executed normally
- If control reaches the end of the function body, or a `return` statement is executed, control returns to the caller
- A `return` statement may have a single argument of the same type as the return type of the function. If the statement is executed, the argument of `return` becomes the value returned to the caller

We can only call functions that have already been declared or defined at that point in the program!

## Example: Printing Character Frequencies

```
int print_freq(char c, int freq)
{
    int i;

    printf("%c :", c);
    if(freq < 75)
    {
        for(i=0; i<freq; i++)
        {
            putchar('#');
        }
    }
    else
    {
        printf("#....(%d)...#",freq);
        freq = -1;
    };
    printf("\n", c);
    return freq;
}
```

## Example: Printing Character Frequencies (contd.)

Assume that the previous function definition is inserted into the frequency counting program just in front of the `int main(void)` line

We can then modify `main` as follows:

```
...
    for(i=0; i<128; i++)
    {
        if(isprint(i))
        {
            print_freq(i, freq_count[i]);
        }
    }
    return EXIT_SUCCESS;
}
```

The program will then print frequency histograms instead of just numbers

## Exercises

Rewrite the Fahrenheit→Celsius Program to use a function for the actual conversion

# Assignment

A prime number is a (positive integer) number that is evenly divisible only by 1 and itself

1. Write a function `isprime()` that determines if an integer number is prime. You can use the `%` modulus operator (division rest on integers) or work with plain division. Use your function to implement a program `primes_simple` that prints all primes between 0 and 10000.
2. The **Sieve of Erathostenes** is a more efficient (and ancient) algorithm for finding all primes up to a given number. It starts with a list of all numbers from 2 to the desired limit. It traverses this list, starting at two. Whenever it encounteres a new number, it strikes all multiples of it from the list. What remains at the end is a list of prime numbers.

Example:

```
Initial list:           2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Striking multiples of 2: 2 3   5   7   9   11   13   15
Striking multiples of 3: 2 3   5   7           11   13
```

(There are no multiples of any remainig number, so we skip the  
Use the Sieve algorithm in a second program, `primes_sieve`, that prints all primes between 0 and 10000. Hint: Use an array!

# CSC322

## C Programming and UNIX

### Programming in C More on Functions

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Review of Function Properties

A **function** is a named subroutine

It accepts a number of arguments of a predetermined type and returns a value of a given type

It can have its own local variables

A function can be **called** from other places in the program, including other functions

Functions have to be known (either defined or declared) before they can be called

## Example: Reading Integers

We want to write a function that reads a positive integer number from `stdin`, using only `getchar()`

A number is defined as a sequence of decimal digits (characters from the range '0' to '9')

- We can use the function `isdigit(c)` from `ctype.h` to test if a character is a (decimal) digit
- The C standard guarantees that '0' to '9' have consecutive numerical values. We can thus get the value of a single character `c` that represents a digit by the expression `c-'0'`

Idea: We read the most significant digits first. So whenever we read a new digit, the value of what we have read so far increases 10-fold:

Read	Value
1	1
13	$10*1+3 = 13$
137	$10*13+7 = 137$
1375	$10*137+5 = 1375$

## Example: int read\_int10(void)

```
/* We assume that stdio and ctype have been included */

/* A function that reads a positive integer number in base 10 from
 * stdin. Return number or -1 on failure. Will read one character
 * ahead! */

int read_int10(void)
{
    int res = 0, c, count=0;

    while(isdigit(c=getchar()))
    {
        res = (res*10)+c-'0';
        count++;
    }
    if(count > 0) /* We read something */
    {
        return res;
    }
    return -1;
}
```

## Improving the Function

`read_int10(void)` works fine, but can only read number in **decimal notation**

We want to have a function that can read numbers in any base between 2 and 10 now

Examples:

- 142 in base 8 has the value  $1 * 8^2 + 4 * 8^1 + 2 * 8^0 = 1 * 64 + 4 * 8 + 2 = 98$
- 101010 in base two has the value  $1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 32 + 8 + 2 = 42$
- 1873 is not a valid number in base 6! All digits have to be smaller than the base

The principle is the same, we just use a parameter base instead of the hardwired value 10!

## Do we have a Valid Digit?

```
/* Is a character a valid digit in base b? */

int is_base_digit(int c, int base)
{
    if(c - '0' < 0)
    {
        return 0;
    }
    if(c - '0' >= base)
    {
        return 0;
    }
    return 1;
}
```

## Reading a Number in any Base ( $\leq 10$ )

```
/* A function that reads a positive integer number in any base from stdin */

int read_int_b(int base)
{
    int res = 0, c, count = 0;

    res = 0;
    while(is_base_digit((c=getchar()),base))
    {
        res = (res*base)+c-'0';
        count++;
    }
    if(count > 0) /* We read something */
    {
        return res;
    }
    return -1;
}
```

# Build General Functions!

Good programs are build by breaking the task into many functions that are:

- Small – at most one screen page (in your favourite editor)
- Simple – they only do one thing, and they do that well
- General – so that they can be reused at other parts in the program

Going from general to specific is (generally) easy:

```
/* Alternative to read_int10 */  
  
int read_int10b(void)  
{  
    return read_int_b(10);  
}
```

# Recursive Functions

As we stated above, functions can call other functions. They can also call themselves recursively

A recursive function always has to handle at least two cases:

- The **base case** handles a simple situation without further calls to the same function
- The **recursive cases** may do some work, and in between make recursive calls to the function for smaller (in some sense) subtasks

Recursion is one of the most important programming principles!

## Example: Printing Integers

We now want to print positive integer numbers to stdout, using only `putchar()`

Consider a number in base 10:  $421 = 42 * 10 + 1$

We can split the task into two subtasks:

- Print everything but the last digit (recursively)
- Print the last digit

Base case: There are no digits to print any more

Basic operations:

- To get the last digit, we use the modulus operator `%`
- To get rid of the last digit, we divide the number by the desired base (remember, integer division truncates)

## Example: Decimal Representation of 421

Let's do an example: We want to print the number 421 in base 10

- Step 1:  $421 \% 10 = 1$  and  $421 / 10 = 42$ . Hence the last number to print is 1 and the rest we still have to print is 42
- Step 2:  $42 \% 10 = 2$  and  $42 / 10 = 4$ . The second last digit is 2, the rest is 4
- Step 3:  $4 \% 10 = 4$  and  $4 / 10 = 0$ . The next digit is 4
- Step 4: Our rest is 0, hence there is nothing to do but printing the digits in the right order

The same principle applies for other bases (just replace 10 by your base)

## Writing a Number in any Base ( $\leq 10$ )

```
/* Write non-zero positive integer in any base to stdout */  
  
void write_int_b_rekursive(int value, int base)  
{  
    int digit;  
  
    digit = value % base;  
    value = value/base;  
  
    if(value!=0)  
    {  
        write_int_b_rekursive(value, base);  
    }  
    putchar(digit + '0');  
}
```

Problem: What happens if the input is 0?

(Answer: It works fine, but by accident, not by design!)

## Writing Integers (Contd.)

We can **wrap** the simple recursive function to handle the abnormal case (but, as we saw on the last slide, we don't need to):

```
/* Write positive integer in any base to stdout */  
  
void write_int_b(int value, int base)  
{  
    if(value == 0)  
    {  
        putchar('0');  
    }  
    write_int_b_rekursive(value, base);  
}
```

## Putting Things Together: A Base Converter

We now use the defined function to write a program that reads pairs **number base** and prints them back in the new **base**:

- **number** is considered to be a decimal number
- **base** should be a decimal number between 2 and 10 (inclusive)
- Numbers and pairs are separated by a single, arbitrary character (including space and newline)
- The program terminates, if one of the numbers is invalid

# The Base Converter

```
int main(void)
{
    int num, base;
    while(1)
    {
        printf("Input decimal value and desired base!\n");
        num = read_int10();
        if(num == -1)
        {
            return EXIT_SUCCESS;
        }
        base = read_int10();
        if(base == -1 || base < 2 || base > 10)
        {
            printf("Error: No valid base!\n");return EXIT_FAILURE;
        }
        write_int_b(num, base);
        putchar('\n');
    }
}
```

## Usage Example

```
$ ./base_converter
```

```
Input decimal value and desired base!
```

```
123123 3
```

```
20020220010
```

```
Input decimal value and desired base!
```

```
42 10
```

```
42
```

```
Input decimal value and desired base!
```

```
42 Hallo!
```

```
Error: No valid base!
```

```
$
```

## Exercise

Extend the base converter to work with base 16, using 0-9 and A-F as digits (allow both upper and lower case!)

Extend the base converter to accept triplets **input-base,value,outputbase**, where **value** is interpreted in **input-base** (and input-base is a single hexadecimal digit  $\geq 2$ ). Add reasonably robust error handling!

The complete base converter code from the lecture is available from the CSC322 web page or directly at [http://www.cs.miami.edu/~schulz/CSC322/base\\_converter.c](http://www.cs.miami.edu/~schulz/CSC322/base_converter.c)

# CSC322

## C Programming and UNIX

### Programming in C

### Program Structure and the C Preprocessor

Stephan Schulz

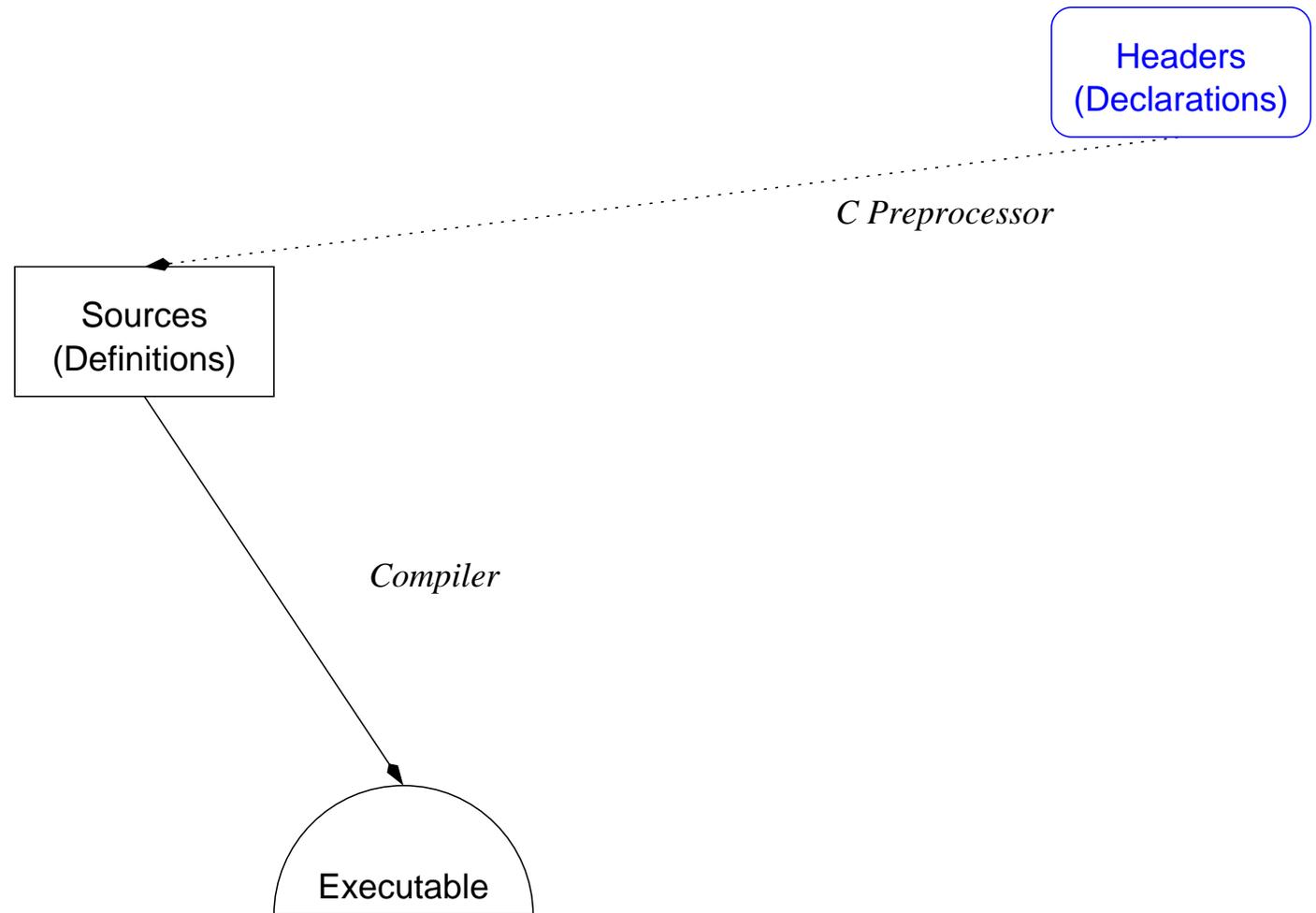
Department of Computer Science

University of Miami

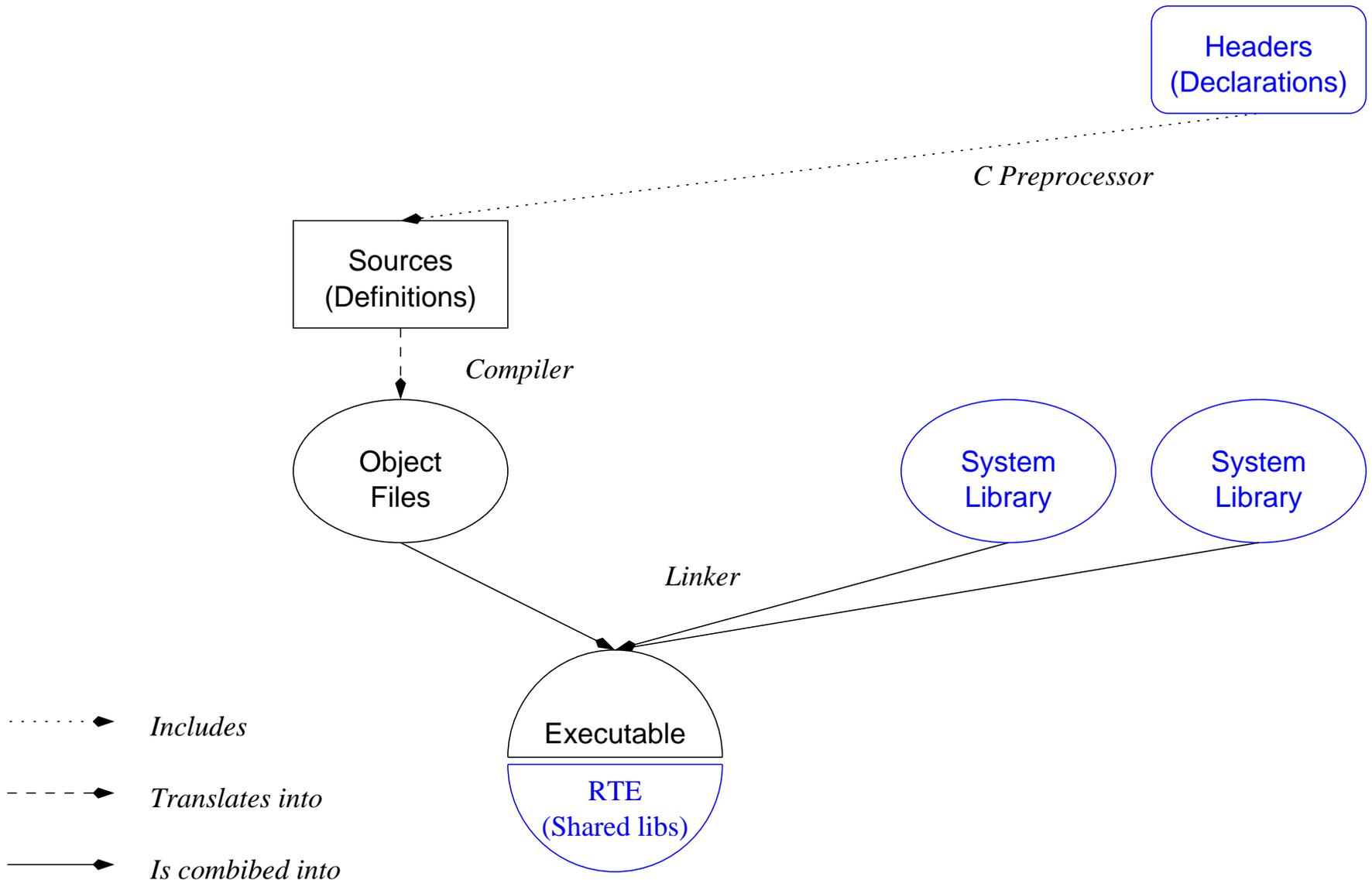
`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

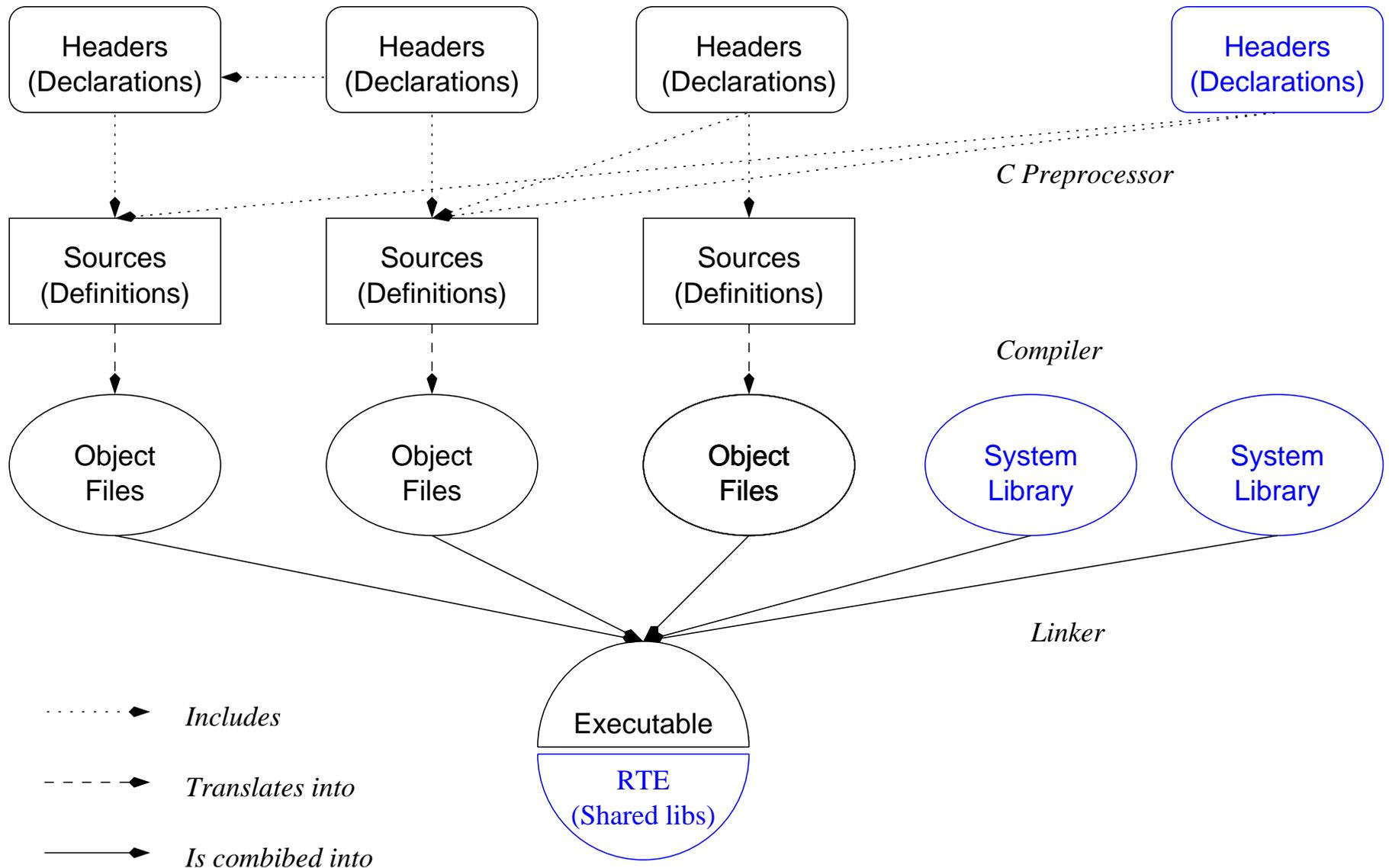
# Simple Program Structure



# Program Structure In Detail



# Program Structure for Multi-File Programs



# The C Preprocessor

The C preprocessor performs a **textual rewriting** of the program text before it is ever seen by the compiler proper

- It includes the contents of other files
- It expands macro definitions
- It conditionally processes or removes segments of the program text

Preprocessor directives start with a hash # and traditionally are written starting in the very first column of the program text

After preprocessing, the program text is free of all preprocessor directives

Normally, **gcc** will transparently run the preprocessor. Run **gcc -E <file>** if you want to see the preprocessor output

## Including Other Files: #include

The `#include` directive is used to include other files (the contents of the named file replaces the `#include` directive)

Form 1: `#include "file"`

- The preprocessor will search for `file` in the current directory
- What happens if `file` is not found in the current directory, is **implementation-defined**
  - \* UNIX compilers will typically treat `file` as a pathname (that may be either absolute or relative)
  - \* If the file is not found, the compiler prints an error message and aborts

Form 2: `#include <file>`

- `file` will be searched for in an implementation-defined way
- UNIX compilers will typically treat `file` as a file name relative to the **system include directory**, `/usr/include` on the lab machines
- You can add to the list of directories that will be searched using **`gcc -I<includedir>`**

## Example: Include

myfile.c:	mary:
A Poem  #include "mary"	Mary had a little lamb, Its fleece was white as snow; And everywhere that Mary went The lamb was sure to go.

```
$ gcc -E myfile.c
```

```
# 1 "myfile.c"
```

```
A Poem
```

```
# 1 "mary" 1
```

```
Mary had a little lamb,
```

```
Its fleece was white as snow;
```

```
And everywhere that Mary went
```

```
The lamb was sure to go.
```

```
# 4 "myfile.c" 2
```

# Include Discussion

Include directives are typically used for **sharing common declarations** between different program parts

**Libraries** (including the standard library) come with header files that define their **interface** by

- Defining data types and constants
- Declaring functions (and defining macros)
- Declaring variables

Note that included files can contain further `#include` statements (that will be automatically expanded by the preprocessor)

- This is frequent in system files, where the standard-prescribed include files often include system-specific files actually describing the features

## Simple Macro Definitions: #define

The #define directive is used to define **macros**

Simple Form: #define <name> <replacement text>

- This will define a macro for <name>, which has to follow the common rules for C identifiers (alphanumeric characters and underscore, should not start with a digit)
- Any normal occurrence of <name> after the definition will be replaced by <replacement text>
- Replacement will not take place in strings!
- The macro definition normally ends at the end of the line, however, it can be extended to the next line by appending \ as the very last character of the line

Note that macro expansion even takes place within further macro definitions!

Most common use: Symbolic constants (e.g. EOF)

## Simple #define Example

reality.c:

```
#define true 1
#define false 0
void reality_check(void)
{
    if(true == false)
    {
        printf("Reality is broken!\n");
    }
}
```

**\$ gcc -E reality.c**

```
# 4 "reality.c"
void reality_check(void)
{
    if(1 == 0)
    {
        printf("Reality is broken!\n");
    }
}
```

# Macros with Arguments

Macro definitions can also contain **formal arguments**

```
#define <name>(arg1,...,arg1) <replacement text>
```

If a macro with arguments is expanded, any occurrence of a formal argument **in the replacement text** is replaced with the actual value of the arguments in the macro call

This allows a more efficient way of implementing small “functions”

- But: Macros cannot do recursion
- Macro calls have slightly different semantics from function calls
- Therefore macros are usually only used for very simple tasks

By convention, preprocessor defined constants and many macros are written in ALL\_CAPS (using underscores for structure)

## #define Examples

macrotest.c:

```
#define XOR(x,y) ((!(x)&&(y))||((x)&&! (y))) /* Exclusive or */
#define EQUIV(x,y) (!XOR(x,y))

void test_macro(void)
{
    printf("XOR(1,1)   : %d\n", XOR(1,0));
    printf("EQUIV(1,0): %d\n", EQUIV(1,0));
}
```

\$ gcc -E reality.c

```
# 4 "macrotest.c"
```

```
void test_macro(void)
```

```
{
    printf("XOR(1,1)   : %d\n", ((!(1)&&(0))||((1)&&! (0))));
    printf("EQUIV(1,0): %d\n", (!(!(1)&&(0))||((1)&&! (0))));
}
```

## #define Caveats

Since macros work by textual replacement, there are some unexpected effects:

- Consider `#define FUN(x,y) x*y + 2*x`
  - \* Looks innocent enough, but: `FUN(2+3,4)` expands into `2+3*4+2*2+3` (not `(2+3)*4+2*(2+3)`)
  - \* To avoid this, **always** enclose each formal parameter in parentheses (unless you know what you are doing)
- Now consider `FUN(var++,1)`
  - \* It expands into `x+++*1 + 2*x++`
  - \* Macro arguments may be evaluated more than once!
  - \* Thus, avoid using macros with expressions that have side effects

Other frequent problems:

- Semicolons at the end of a macro definition (**wrong!**)
- “Invisible” syntax errors (run `gcc -E` and check the output if you cannot locate an error)

## Conditional Compilation: #if/#else/#endif

We can use preprocessor directives to conditionally include or exclude parts of the program:

- Program parts may be enclosed in #if <expr>/#endif pairs
- <expr> has to be a **constant integer expression**
- If it evaluates to 0, the text in the #if <expr>/#endif bracket is ignored, otherwise it is included
- There also is an optional #else “branch”

Most frequent use: Test for the definition of macros

- defined(<macro>) evaluates to 1 if <macro> is defined (even as the empty string), 0 otherwise
- Short form: #if defined(<macro>) is equivalent to #ifdef <macro>, #if !defined(<macro>) is equivalent to #ifndef <macro>,  
– E.g.: #ifndef EOF  
    #define EOF -1  
    #endif

## Example: #ifdef

cond\_preproc.c:

```
#define hallo
#define fred barney
#define test 2+2
#if defined(hallo)
"Hallo"
#else
#ifdef fred
"Fred"
#endif
#endif
#if test
"test"
#endif
```

```
$ gcc -E cond_preproc.c
```

```
# 5 "cond_preproc.c"
```

```
"Hallo"
```

```
"test"
```

# Exercises

Search the `/usr/include` directory (use **grep** for faster progress) and find out where the following functions/macros are defined, and, for the macros, what their value is

- `LONG_MAX`
- `ULONG_MAX`
- `getchar()`
- `getc()`
- `EOF`
- `EXIT_FAILURE`
- `EXIT_SUCCESS`
- `NULL`

# CSC322

## C Programming and UNIX

### Programming in C

### C Preprocessor/Declarations and Scoping

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## Conditional Compilation: #if/#else/#endif

We can use preprocessor directives to conditionally include or exclude parts of the program:

- Program parts may be enclosed in #if <expr>/#endif pairs
- <expr> has to be a **constant integer expression**
- If it evaluates to 0, the text in the #if <expr>/#endif bracket is ignored, otherwise it is included
- There also is an optional #else “branch”

Most frequent use: Test for the definition of macros

- defined(<macro>) evaluates to 1 if <macro> is defined (even as the empty string), 0 otherwise
- Short form: #if defined(<macro>) is equivalent to #ifdef <macro>, #if !defined(<macro>) is equivalent to #ifndef <macro>,  
– E.g.: #ifndef EOF  
    #define EOF -1  
    #endif

## Example: #ifdef

cond\_preproc.c:

```
#define hallo
#define fred barney
#define test 2+2
#if defined(hallo)
"Hallo"
#else
#ifdef fred
"Fred"
#endif
#endif
#if test
"test"
#endif
```

```
$ gcc -E cond_preproc.c
```

```
# 5 "cond_preproc.c"
```

```
"Hallo"
```

```
"test"
```

## More on Preprocessor Definitions

You can use `#undef <name>` to get rid of a definition

– This is most often used to start from a clean slate:

```
#undef true
#undef false
#define true 1
#define false 0
```

– It is, however, forbidden to undefine implementation-defined names

You can use the `-D` option to **gcc** to cause certain names to be defined throughout the process

– This is often used to select one of many alternatives for compilation

- \* With or without internal consistency checks
- \* With or without certain features (e.g. Demo version vs. commercial version)
- \* . . .

Certain names may be predefined by the implementation (most starting with two underscores: `__FILE__`, `__STDC__` . . . )

## Combinations of #ifdef and #include

#ifdef/endif also can be used to conditionally include or exclude files

Usage: Compile for different operating systems:

```
#ifdef __LINUX__
#include "linux.h"
#elif defined(__BSD__)
#include "bsd.h"
#else
#include "default.h"
#endif
```

Usage: Guarding against multiple inclusions

```
#ifndef THIS_HEADER
#define THIS_HEADER
<lots of stuff>
#endif
```

# Separate Compilation

C supports the separate compilation of multiple source files

- Each source file is translated into an **object file**
- A **linker** combines different object files into the final executable

**gcc** by default tries to create an executable program by performing operations as follows:

1. Preprocessing
2. Compilation (and assembly)
3. Linking

For multi-file programs, we have to perform separate compilation:

- **gcc -c file.c -o file.o** will compile **file.c** into **file.o** without linking
- **gcc -o progname file1.o file2.o file3.o** will link the three precompiled object files into an executable

# Definitions and Declarations

**Definitions** cause the defined objects to be created

- Variable definitions allocate an appropriate amount of memory (and associate it with the variable name)
- Function definitions cause code to be generated

**Declarations** only state information about an object

- For variables, they state the type
- For functions, they state return type and argument types

There can be any number of compatible **declarations** for an object

There can be only one **definition** for the object

A function or variable can only be used inside the **scope** of a matching declaration

Any definition also implicitly declares an object

## Explicit Declarations

Variables can be declared by adding the `extern` keyword to the syntax of a definition:

- `extern int counter;`
- `extern char filename[MAXPATHLEN];`

Function declarations just consist of the function header, terminated by a semi-colon:

- `int isdigit(int c);`
- `int putchar(int c);`
- `bool TermComputeRWSequence(PStack_p stack, Term_p from, Term_p to);`

Alternatively, the `names` of the formal parameters can be omitted

- `int isdigit(int);`
- `int putchar(int);`
- `bool TermComputeRWSequence(PStack_p, Term_p, Term_p);`
- However, the first form is often preferred because the parameter names may document the purpose of the parameter

# Scoping Rules

There are two kinds of declarations in C

- Declarations written inside a **block** are called **local** declarations
- Declarations outside any block are **global** declarations

The scope of a local declaration begins at the declaration and ends at the end of the innermost enclosing block

The scope of a global declaration begins at the declaration and continues to the end of the source file

- Note that this refers to files after preprocessing, i.e. a declaration in a header file also is visible in the including file (from the point of the `#include` statement)



# Limiting Potential Scope

By default, all declared variables and functions are accessible from any source file in the program

- Of course, they may have to be declared to be visible

Problems: We have no control over the use of these objects in other source files

- Reuse of libraries may fail because of **namespace pollution**
- Unintentional or malicious misuse of internal functions may lead to program misbehaviour

The `static` keyword, applied to a **global** definition (or declaration), limits the accessibility of the declared object to the source file it is defined in

- `static int internal_help_fun(int a1, int a2);`

In general, it is a good idea to declare everything not expected to be used by other program part `static`

# Lifetime and Initialization of Variables

Global variables have unlimited lifetime

- They are created and initialized when the program starts
- The expression used in the initialization has to be **constant**, i.e. it has to be fully evaluable at compile time
- If not explicitly initialized, they are guaranteed to be initialized to 0
- They keep their values until the program terminates (unless explicitly changed, of course)

Most local variables (and function parameters) only have limited lifetime

- They are also called **automatic** variables and are typically allocated on the **stack**
- They are created when the variable comes into scope and are destroyed when the variable goes out of scope – in particular, each recursive call gets a fresh copy of the variable
- The initializing expression can use all variables and functions currently in scope
- They are reinitialized every time they come into scope, if not initialized explicitly, they contain undefined values (“junk”)

## Persistent Local Variables: `static` again

`static` local variables have unlimited lifetime

- They are initialized the very first time they come into scope
- They are shared between different calls to the same function
- They keep their values in between calls
- However, they can only be accessed from inside their corresponding block

## Example: Static and Automatic Variables

```
#include <stdio.h>
#include <stdlib.h>
static int global_count = 0;
void counter_fun(void)
{
    static int static_count = 0;
    int auto_count = 0;
    int pseudo_count = global_count;

    global_count++; auto_count++; static_count++; pseudo_count++;
    printf("Global: %3d Auto: %3d Static: %d Pseudo: %d\n",
           global_count, auto_count, static_count, pseudo_count);
}
int main(void)
{
    counter_fun();
    counter_fun();
    global_count = 0;
    counter_fun();
    counter_fun();
    return EXIT_SUCCESS;
}
```

## Example: Static and Automatic Variables(Contd.)

```
$ gcc -o vartest vartest.c
```

```
$ ./vartest
```

```
Global:  1 Auto:  1 Static:  1 Pseudo:  1
```

```
Global:  2 Auto:  1 Static:  2 Pseudo:  2
```

```
Global:  1 Auto:  1 Static:  3 Pseudo:  1
```

```
Global:  2 Auto:  1 Static:  4 Pseudo:  2
```

```
$
```

# Assignment

Write a **data safe** library offering the following functionality:

- Calling `data_safe(ds_register, 0, 0)` will return a **unique** random key (a positive integer). Use `rand()` to generate random numbers (and **man rand** to find out how).
- Calling `data_safe(ds_store, key, value)` will store the value (a positive integer) in the data safe (under the key). It should return the value if everything worked, -1 otherwise (e.g. if there is no space left)
- Calling `data_safe(ds_retrieve, key, n)` will retrieve the `n`th value stored under the key, or -1 if less than `n` values have been stored under the key
- Calling `data_safe(ds_delete, key, 0)` will delete all entries stored under key (you may then reuse key for future register calls, as long as you still generate a random key)
- Make sure that at least 100 keys can be in use in parallel, and that at least 10000 data items can be stored in total

Make sure that the data is not accessible in any other way (using legal C)

Implement the library in its own source file, with a header file `data_safe.h` that contains **all** necessary declarations

Write a main program `ds_test.c` that uses the library, storing 10 values under 3 different keys, retrieving them and delete them. Use a reasonably varied sequence of storage, retrieval, and registration

Hints:

- Use static local variables to store the necessary data in the `data_safe()` function
- Use preprocessor `#define` statements to define the symbolic constants `ds_register`, `ds_store`,...
- Be careful to avoid handing a key already in use out on registration. Carefully design your data structures first, the operations will be simple to implement

# CSC322

## C Programming and UNIX

### Programming in C

### rpn\_calc: An Extended Example

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Project: An RPN Calculator

Aim: A calculator program that can do simple arithmetic

- Conversion between different bases
- Addition, subtraction, multiplication...

We'll use **reverse polish notation**

- Operator is written **after** arguments:  $7\ 5\ + = 7+5$
- More complicated:  $12\ 2\ 5\ 2\ * + - = (12-(2+(5*2)))$

Advantages of RPN

- Easy to understand
- Easy to implement
- No hassle with recursive parsing of parentheses and precedences
- Can easily and consistently handle operators of any arity (number of arguments)

## Some Suggested Operators

Arithmetic operators (others may be added):

- + Pop two numbers , add them
- Pop two numbers, subtract first from second
- \* Pop two numbers , multiply them
- / Pop two numbers, divide second by first
- % Pop two numbers, divide second by first, giving the division rest

Non-Arithmetic operators (non-exclusive):

- p Print the topmost number on the stack
- o Pop topmost number on the stack, use it as new output base
- i Pop topmost number on the stack, use it as new input base
- S Print the whole stack (mainly for debugging)
- P Print input and output bases (in decimal)

# Usage Example

```
$ ./rpn_calc
```

```
10 8 +
```

```
S
```

```
18
```

```
3 / p
```

```
6
```

```
3 / p
```

```
2
```

```
o
```

```
p
```

```
Stack underflow error
```

```
P
```

```
Input base (decimal): 10 Output Base (decimal): 2
```

```
255
```

```
p
```

```
11111111
```

```
16 p
```

```
10000
```

```
10 o
```

```
S
```

```
255 16
```

# Implementation

Basic idea:

- Input is a sequence of **numbers** and **operators**
- If a number is read, it is pushed onto a **stack**
- If an operator is read, the necessary number or arguments is popped of the stack, the operation is performed, and the result is placed in the stack

Input and output can happen in any representation from base 2-16

- There is a strong convention for representing these numbers:
  - \* Digits are 0-9 with nominal value, A-F (or a-f) with values 10-15
- Input and output use independent bases (base conversion made easy)

Recognizing numbers and operators

- Any string of valid digits in the current input base is a number
- Any string starting with `-` and directly followed by valid digits in the current input base is a number
- Everything else is treated as an operator

# Subtasks

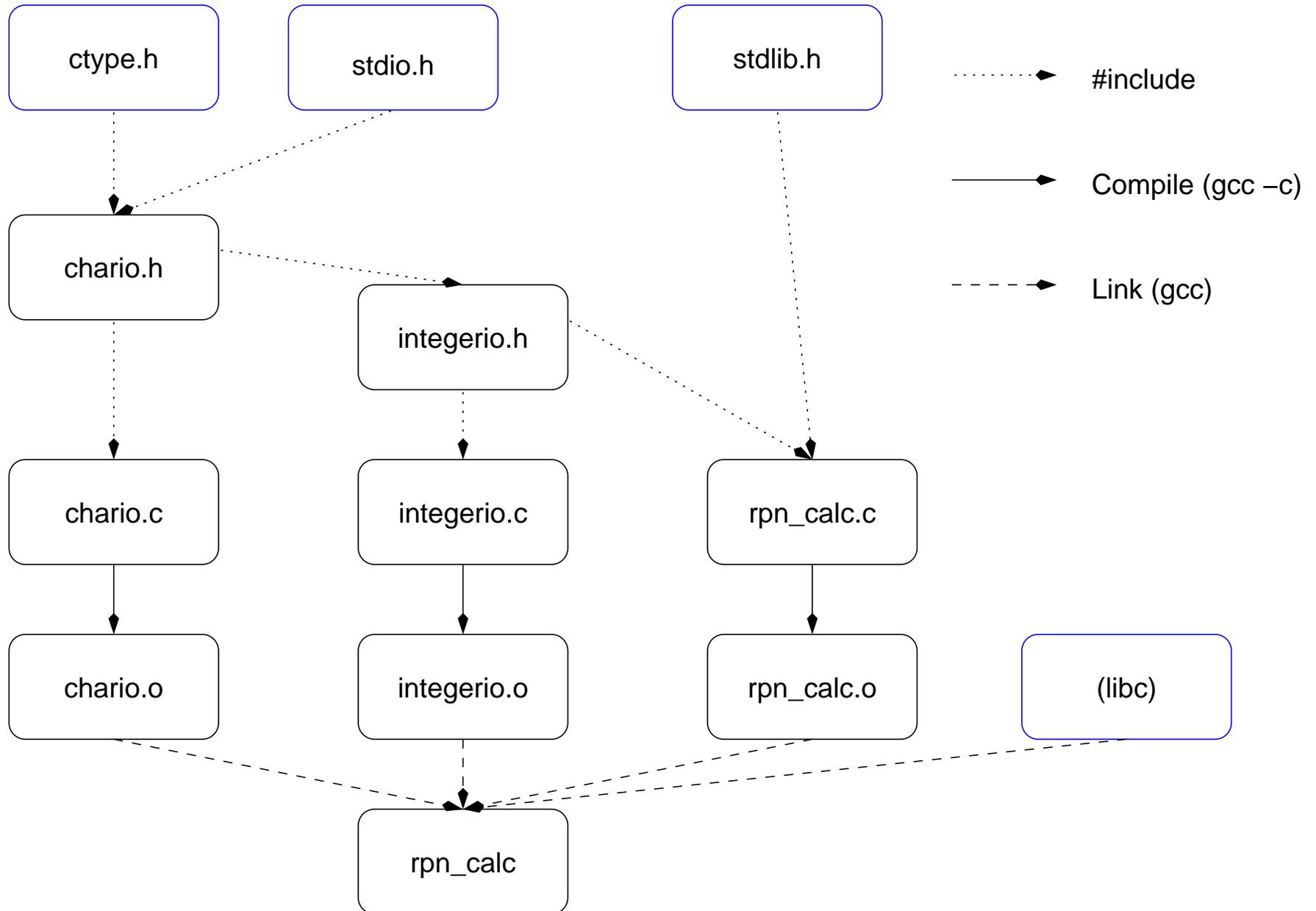
From the above, we can identify a number of subtasks:

- Reading numbers and operators
- Printing numbers
- Handling the stack
- Executing the actual operations

Input handling is the hardest task!

- We need to read up to 2 characters to decide if we read a number or an operator ('-+' represents two operators, '-1' a number)
- Rather than handling explicit lookahead variables throughout the program, we can build a general character I/O-library that allows us to read ahead, but to maintain (or restore) the status of the input queue

# Program Organization



# The Character I/O Library: Ideas

Main interface similar to `getchar()`

Read character can be “pushed back” into the input queue

Implementation:

- Internal **buffer** of character
- **Pushed** characters go into the buffer
- Reading first tries the buffer, and only reads `stdio` if the buffer is empty

Additional help-functions

- Look at a character, but don't read it
- Skip while space

## The Character I/O Library: chario.h

```
#ifndef UNGETCHAR
#define UNGETCHAR
#include <stdio.h>
#include <ctype.h>
/* Maximal number of characters the can be pushed back */
#define MAX_BUFFERED_CHARS 1024

/* As getchar(), but with unget cabability (provided by PushChar() */
int GetChar(void);

/* Push back a character into the read queue. Return c or EOF if the
   queue is full. */
int PushChar(int c);

/* Return the next character, but do not read it */
int LookChar(void);

/* Skip over white space characters. Return the first non-white
   character (but it is not removed from the queue), or EOF if the
   pushback queue is full. */
int SkipSpace(void);
#endif
```

# The Character I/O Library: Global Variables and Includes

```
#include "chario.h"
```

```
static int char_buff[MAX_BUFFERED_CHARS];
```

```
static int buff_pos = 0;
```

# The Character I/O Library: Reading and Unreading

```
int GetChar(void)
{
    if(buff_pos)
    {
        buff_pos--;
        return char_buff[buff_pos];
    }
    return getchar();
}

int PushChar(int c)
{
    if(buff_pos < MAX_BUFFERED_CHARS)
    {
        char_buff[buff_pos] = c;
        buff_pos++;
        return c;
    }
    return EOF;
}
```

# The Character I/O Library: Help Functions

```
int LookChar(void)
{
    int c = GetChar();

    PushChar(c);
    return c;
}
```

```
int SkipSpace(void)
{
    int c;

    while(isspace((c=GetChar())))
    { /* Empty body */ }
    return PushChar(c);
}
```

# The Integer I/O Library: Ideas

We use the same algorithms as discussed before

However, because we allow bases up to 16, we add some additional helper functions for

- Recognizing valid digits
- Converting numerical values to character representation of digits
- Giving the numerical value of digits

Second difference: We allow negative numbers

- We cannot use  $-1$  to signal failure
- Instead: We write a separate function that predicts the presence (or absence) of a number in the input stream
- The calling functions have to make sure that the integer reading function is only called if there is valid input (i.e. success is guaranteed)

# The Integer I/O Library: integerio.h

```
#include "chario.h"
```

```
/* Read an integer in base base. */
```

```
int read_int_base(int base);
```

```
/* Check if there is a integer to be read, i.e. a digit or '-'  
   directly followed by a digit */
```

```
int int_available(int base);
```

```
/* Write integer in any base to stdout */
```

```
void write_int_base(int value, int base);
```

# The Integer I/O Library: Includes

```
#include "integerio.h"
```

# The Integer I/O Library: Helper functions 1

/\* Consider c as a hexadecimal digit (0..9, a..f, A..F) and return its numerical value. If not a valid digit, return -1 \*/

```
static int hex_digit_value(int c)
```

```
{
    if(c >= '0' && c <= '9')
    {
        return c - '0';
    }
    if(c >= 'a' && c <= 'f')
    {
        return c - 'a' + 10;
    }
    if(c >= 'A' && c <= 'F')
    {
        return c - 'A' +10;
    }
    return -1;
}
```

## The Integer I/O Library: Helper functions 2

```
/* Check if a character c is a valid digit in base. */
static int is_base_digit(int c, int base)
{
    int value = hex_digit_value(c);

    if(value < 0 || value >= base)
    {
        return 0;
    }
    return 1;
}
```

## The Integer I/O Library: Helper functions 3

```
/* Given an int 0<= value < 16, return the Hexadecimal digit with that
   value */
static int int_to_hexdigit(int value)
{
    if(value<=9)
    {
        return value + '0';
    }
    else
    {
        return value - 10 + 'A';
    }
}
```

# The Integer I/O Library: Reading Integers

```
/* Read an integer in base base. */
int read_int_base(int base)
{
    int res = 0, c, sign = 1;

    if((c=GetChar())=='-')
    {
        sign = -1;
    }
    else
    {
        PushChar(c); /* Unread Character */
    }
    while(is_base_digit((c=GetChar()),base))
    {
        res = (res*base)+hex_digit_value(c);
    }
    PushChar(c);
    return res*sign;
}
```

# The Integer I/O Library: Checking for Integer Presence

```
/* Check if there is a integer to be read, i.e. a digit or '-'
   directly followed by a digit */
int int_available(int base)
{
    int save_char , res = 0;

    if(is_base_digit(LookChar(), base))
    {
        res = 1;
    }
    else if(LookChar() == '-')
    {
        save_char = GetChar();
        if(is_base_digit(LookChar(), base))
        {
            res = 1;
        }
        PushChar(save_char);
    }
    return res;
}
```

# The Integer I/O Library: Writing Integers

```
/* Write integer in any base (2<= base <=16) to stdout */
void write_int_base(int value, int base)
{
    int digit;

    if(value < 0)
    {
        putchar('-');
        value = -1*value;
    }
    digit = value % base;
    value = value/base;

    if(value!=0)
    {
        write_int_base(value, base);
    }
    putchar(int_to_hexdigit(digit));
}
```

## Exercises

Download the program from <http://www.cs.miami.edu/~schulz/CSC322.html>, compile it, and read the source code. You may want to add more operators (e.g. `t` to duplicate the top of the stack, `s` to switch the two topmost numbers, . . .

# CSC322

## C Programming and UNIX

### Programming in C

### rpn\_calc: An Extended Example (Part 2)

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## Recapitulation: Some of our Library Functions

The `integerio` library offers functions for reading and printing integers. All functions have a parameter `base` for selecting the number system (2–16, or binary to hexadecimal)

```
int read_int_base(int base);
```

- Reads an integer from the standard input (using our `GetChar()/PushChar()` interface), returning its value
- If no valid integer can be found, behavior is undefined!

```
int int_available(int base);
```

- Returns 1 (true), if a valid integer can be read from standard input, 0 otherwise
- Does not consume any characters from the input stream!

```
void write_int_base(int value, int base);
```

- Prints an integer number to `stdout`, using the number system selected by `base`

Additional function from `chario.c`: `int SkipSpace(void)`

# The Main Calculator Program

Aim: RPN (Postfix) calculator program

- Input: Operators and Numbers (operands)
- Numbers are pushed on a stack
- Operators pop operands and push the result of the operation

```
while(there is input)
{
    if(input is a number)
    {
        num = read_number();
        push(num);
    }
    else if(input is a valid operator)
    {
        pop operands, apply operator, push result;
    }
    else
    {
        print error mesage;
    }
}
```

# Case Distinctions

Note: The operator determines which actions we have to perform

- This is a **case distinction**: Based on a single (integer) value, we have to select one alternative
- Possible implementation:

```
if(value == val1)
{
    action1;
}
else if((value == val2)
{
    action2;
}
...
else
{
    default_action;
}
```

## C Alternative: switch

```
switch(E)
{
case val1: action1;
           break; /* Otherwise we fall through! */
case val2: action2;
  ...     break;
default:  default_action;
           break;
}
```

E has to be an integer-valued expression

val1, val2, . . . have to be **constant** integer expressions

E is evaluated and the result is compared to each of the constants after the **case** labels. Execution starts with the first statement after the matching case. If no case matches, execution starts with the (optional) default case.

Note: Execution does **not** stop at the next case label! Use `break;` to break out of the `switch`

# The Stack Abstract Datatype

A **stack** is a **last-in first-out** (LIFO) data structure

- It can store values of a given type
- Values can be **pushed** onto a stack
- The topmost element can be retrieved by **popping** it off the stack
- Typically, only the top element is accessed (enforced either by convention or by design)
- Stacks can have a predetermined size (maximal number of elements) or grow as needed

Stack implementation in C:

- Values are stored in an array of the correct type
- A **stack pointer** contains the index of the next unused cell

## Stack Implementation in `rpn_calc.c`

We use a fixed maximal stack size:

```
#define STACKSIZE 1024
```

- Using a symbolic constant avoids mistyping and misreading, and allows us to easily change the stack size later!

Our stack data structure is realized by two variables:

- `int stack[STACKSIZE]`; stores the values
- `int sp = 0`; is the **stack pointer**, and initially points to the first element of `stack`

Stack operations are implemented as specialized macros

## Pushing things onto the stack: PUSH()

```
/* If stack is full, print an error message,  
   otherwise push the value onto the stack */  
#define PUSH(value) \  
if(sp < STACKSIZE) \  
{ \  
    stack[sp] = (value);\  
    sp++;\  
}  
else\  
{\  
    printf("Stack overflow error\n");\  
}
```

## Popping values: POP\_OR\_FAIL()

```
/* If stack is empty, print an error message and "break;",  
   otherwise pop the top value into varname */  
#define POP_OR_FAIL(varname) \  
if(sp > 0) \  
{ \  
    sp--; \  
    (varname) = stack[sp]; \  
} \  
else \  
{ \  
    printf("Stack underflow error\n"); \  
    break; \  
}
```

Note that the macro contains a `break;` statement in the error case

- Limits general usability but. . .
- . . . exits the case it is used in early!

# The Main Program: Preliminaries and Declarations

```
int main(void)
{
    int num, arg1, arg2, i;
    int stack[STACKSIZE];
    int sp = 0, in_base = 10, out_base = 10;

    SkipSpace();
```

The number systems to be used for input and output is determined by `in_base` and `out_base`

- Both are initialized to 10 (decimal)

Note that the next character to be read is meaningful (not white space) now

- This will be a **loop invariant** of the main loop)

# The Main Loop: Overall Structure

```
while(LookChar()!=EOF)
{
    if(int_available(in_base))
    {
        num = read_int_base(in_base);
        PUSH(num);
    }
    else
    { /* Operator! */
        switch(GetChar())
        {
            case 'o':
                ... /* Handle different cases */
            default:
                printf("Unknown operand\n");
                break;
        }
    }
    SkipSpace();
}
return EXIT_SUCCESS;
}
```

# The Main Loop: Arithmetic operators

```
switch(GetChar())
{
  ...
  case '+':
    POP_OR_FAIL(arg2);
    POP_OR_FAIL(arg1);
    num = arg1+arg2;
    PUSH(num);
    break;
  case '-':
    POP_OR_FAIL(arg2);
    POP_OR_FAIL(arg1);
    num = arg1-arg2;
    PUSH(num);
    break;
  case '*':
    POP_OR_FAIL(arg2);
    POP_OR_FAIL(arg1);
    num = arg1*arg2;
    PUSH(num);
    break;
```

```
case '/' :
    POP_OR_FAIL(arg2);
    POP_OR_FAIL(arg1);
    num = arg1/arg2;
    PUSH(num);
    break;
case '%' :
    POP_OR_FAIL(arg2);
    POP_OR_FAIL(arg1);
    num = arg1%arg2;
    PUSH(num);
    break;
...
}
```

# The Main Loop: I/O operators

```
switch(GetChar())
{
    ...
    case 'p':
        POP_OR_FAIL(num);
        write_int_base(num,out_base);
        putchar('\n');
        PUSH(num);
        break;
    case 'o':
        POP_OR_FAIL(num);
        if(num < 2 || num >16)
        {
            printf("Only bases 2-16 (decimal) supported\n");
        }
        else
        {
            out_base = num;
        }
        break;
}
```

```
case 'i':
    POP_OR_FAIL(num);
    if(num < 2 || num >16)
    {
        printf("Only bases 2-16 (decimal) supported\n");
    }
    else
    {
        in_base = num;
    }
    break;
```

```
case 'S':
    for(i=0; i<sp; i++)
    {
        write_int_base(stack[i],out_base);
        putchar(' ');
    }
    putchar('\n');
    break;
case 'P':
    printf("Input base (decimal): %d  Output Base (decimal): %d\n",
           in_base,out_base);
    break;
...
}
```

# Manual Compilation

First, we compile all of the source files individually:

```
$ gcc -ansi -Wall -c -o chario.o chario.c
```

```
$ gcc -ansi -Wall -c -o integerio.o integerio.c
```

```
$ gcc -ansi -Wall -c -o rpn_calc.o rpn_calc.c
```

Then we perform the linking step:

```
$ gcc -ansi -Wall -o rpn_calc chario.o integerio.o rpn_calc.o
```

Now the program is ready to run:

```
$ ./rpn_calc
```

```
2 o 10 p
```

```
1010
```

## UNIX User Commands: `dc`

**dc** is an arbitrary precision RPN calculator

- It handles floating point numbers (to any preselected precision)
- It handles **bignums**, i.e. integers that do not fit into any standard data type
- It has a lot of build-in functionality and can be extended by user-defined macros

Usage is quite similar to our **rpn\_calc**

For more: **man dc** or (particularly) **info dc** (or read info in emacs: **[C-h i]**)

# Exercises

Read the man and info pages for **dc**

Play with the program

Enjoy the weekend and be merry

Note: I've updated the `rpn_calc` sources on the web page to the latest version (changes only comments and style)

# CSC322

## C Programming and UNIX

### Programming in C

### More on Operators and Expressions

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Increment and Decrement Operators

C supports the unary operators ++ and -- for incrementing and decrementing variables

- ++ increments a variable by 1
- -- decrements a variable by 1

Both can be used as **prefix** and **postfix** operators: x++ or ++x

- In both cases, x is incremented by 1
- The difference is in the **value** of the expression:
  - \* The expression x++ has the value of x **before** incrementing
  - \* ++x has the value of x after incrementing, i.e. it is equivalent to the assignment x=x+1

Both forms are used, but the postfix form is more common

# Example

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int x,y;

    x=5;    y=5;
    printf("x    = %d    y    = %d\n", x, y);
    printf("x++ = %d    ++y = %d\n", x++, ++y);
    printf("x    = %d    y    = %d\n", x, y);
    printf("x-- = %d    --y = %d\n", x--, --y);
    printf("x    = %d    y    = %d\n", x, y);
    return EXIT_SUCCESS;
}
```

## Output:

```
x    = 5    y    = 5
x++ = 5    ++y = 6
x    = 6    y    = 6
x-- = 6    --y = 5
x    = 5    y    = 5
```

# Binary Number Representation

C guarantees a base 2 representation for all unsigned integer types:

- Example: 16 bit representation (short on many implementations) of 42

0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
$2^{15}$	$2^{14}$	$2^{13}$	$2^{12}$	$2^{11}$	$2^{10}$	$2^9$	$2^8$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

$$42 = 2^5 + 2^3 + 2^1 = 32 + 8 + 2$$

- If a result of an arithmetic operation results in a value not representable by the result type, it is reduced modulo  $2^n$ , where  $n$  is the width of the result type

An unsigned number of a narrower type is converted to a wider type by adding an appropriate number of leading zeroes:

- The 8 bit representation (char on many implementations) of 42 is:

0	0	1	0	1	0	1	0
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

The exact representation for signed integers is not fixed, however, positive signed integers are guaranteed to have the same representation in signed and unsigned types

# Bitwise Operators

Bitwise operators operate on the **binary representation** of numbers

The binary bitwise operators include

– Bitwise **and** (&) sets a bit in the result, if it is set in both operands:

$$6 \& 3 == 2$$

– | is the bitwise **or**, i.e. the result bit is set, if at least one of the corresponding bits in the input is set:

$$6 | 3 == 7$$

– ^ is the bitwise exclusive or (or **xor**) (the result bit is set if and only if the two operands differ at that position):

$$6 \wedge 3 == 5$$

The bitwise **not** (or **one's complement**) toggles all bits

– The result value depends on the number format

– For 16 bit unsigned short,  $\sim 42 == 65493$

# Bitwise Shifting

C also supports the **shifting** of binary numbers

The binary operator `<<` shifts an integer value left, filling up vacant spaces with zero: `1 << 3 == 8`

- Left-shifting by  $n$  bits is equivalent to multiplication with  $2^n$  (but may be faster on ancient compilers)

The binary operator `>>` shifts an integer value right

- For unsigned value, the new bits become zero
- For signed values, either zeroes are shifted in (**logical shift**), or the first (sign) bit is replicated (arithmetic shift, equivalent to division by  $2^n$ )

Note: The shift operators are used seldomly

- C++ has even recycled them for I/O operations
- Binary and, or, and not, on the other hand, are used frequently to manipulate binary flags packed into a single integer value

## Example

These macros can be used to set and query properties in a variable, where each property is encoded in a single bit

```
#define SetProp(var, prop) ((var) = (var) | (prop))
#define DelProp(var, prop) ((var) = (var) & ~(prop))
#define FlipProp(var, prop) ((var) = (var) ^ (prop))

/* Absolutely assign properties masked by sel */
#define AssignProp(var, sel, prop) DelProp((var),(sel));\
                                   SetProp((var),(sel)&(prop))

/* Are _all_ properties in prop set in var? */
#define QueryProp(var, prop) (((var) & (prop)) == (prop))

/* Are any properties in prop set in var? */
#define IsAnyPropSet(var, prop) ((var) & (prop))
```

# Assignment Operators

Very frequently, programming tasks require the updating of a variable, based on its old value

- Frequent example: `i=i+1;`

In addition to the general assignment operator, C offers operators combining update and assignment

- If `<op>` is a binary operator, then `<op>=` is the corresponding **assignment operator**
- `x <op>= <expr>` is equivalent to `x = x <op> <expr>`
- This is supported for `<op> ∈ { +, -, *, /, %, <<, >>, &, ^, | }`

Most frequently used

- `+=` (as in `fahrenheit += 10`)
- `-=` (e.g. in the update part of a for loop)

# Conditional Expressions

Similarly to conditional statements (if/else), C has conditional expressions:

- If `<test>`, `<e1>`, `<e2>` are expressions, then `<test> ? <e1> : <e2>` is a **conditional expression**
  - \* If `<test>` evaluates to true (non-zero), then `<e1>` is evaluated and its value returned
  - \* Otherwise, `<e2>` is evaluated and returned

Example 1:

```
#define MAX(a,b) ((a>b)?a:b)
```

Example 2:

```
printf("There %s %d item%s left\n",  
      (count==1)?"is":"are",  
      count,  
      (count==1)?"":"s");
```

# Expression Sequences

The coma operator separates two expressions: `<expr1>, <expr2>`

- Expressions are evaluated left to right
- The value of a coma-separated sequence is the value of the last expression in it
- Don't confuse it with the coma separating function call arguments!

Nearly only legitimate use: Initialize and update in for loops:

```
for(cels=0, fahr=-32; cels <= 100; cels+=5,fahr+=9)
{
    printf("%3d    %3d\n", cels, fahr);
}
```

## Type Conversion (Casting)

As already stated, C performs type conversion in many situations automatically

- If different numeric types are used in an expression, all values are promoted to the “largest” type
- If a value of an unsigned integer type is assigned to a “smaller” variable of smaller type, excess bits are dropped
- For signed types, conversion is only partially specified

In addition, values can be coerced to a different type

- A **cast expression** has the syntax (<type>) <value>

Example:

```
printf("Int: %d   Float: %d\n",  
      (1/2)*2,  
      (int) (((float)1/2)*2));
```

```
Int: 0   Float: 1
```

## Exercises

Write a function that counts the number of bits that are one in an unsigned long number (Footnote: Allegedly the NSA sponsors the inclusion of hardware to make this operation fast in many chips because they need it for speeding up the cracking of encrypted documents)

Rewrite `imp_metric` to use comma-separated expressions to build the tables

# CSC322

## C Programming and UNIX

### Programming in C

### Expressions and the Type System

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Getting the Size of Objects and Types

A final operator is `sizeof`

- `sizeof` can be applied to an expression or to a parenthesized type name
- Applying it to an expression is equivalent to applying it to the type of the expression

`sizeof` returns the number of character-sized memory units necessary to store an object of the type

- By definition, `sizeof (char) == 1`

Example:

```
printf("sizeof 1: %d  sizeof (short)1: %d\n", sizeof 1, sizeof ((short)1));
```

```
sizeof 1: 4  sizeof (short)1: 2
```

Note: `sizeof` will be useful for dynamic memory handling

# Order Of Execution

In general, the order of execution of subexpressions is **not** defined!

Exceptions:

- `&&`, `||`, `?:`, and `,`

If you need a particular order of execution, you must force it

- Since statements are executed sequentially, compute subexpression in separate statements (assigning them to different variables)
- Other **sequence points** are set by the operators listed above

The example on the next page may print `One Two One Two` or `Two One One Two`

# Example

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

int one(void)
{
    printf("One ");
    return 1;
}
int two(void)
{
    printf("Two ");
    return 1;
}
int main(void)
{
    one()+two();
    one()&&two();
    printf("\n");
    return EXIT_SUCCESS;
}
```

# Types in C

C offers a set of **basic types** built into the language

We can define new, quasi-basic types as **enumerations**

We can construct new types using **type construction**:

- Arrays over a base type
- Structures, combining different base types in one object
- Unions (can store different type values alternatively)
- **Pointer** to a base type

This generates a recursive type hierarchy!

- We can use new types to build further on them
- E.g. Arrays of Pointers, Structures combining unions and enumerations, . . .

# Basic Types

Basic types in C:

- char (typically used to represent characters)
- short
- int
- long
- long long
- float
- double

All integer types come in and **unsigned** variety

# Defining New Types with typedef

The typedef keyword is used to define **new names** for types in C

General syntax: If we add typedef to a variable definition, it turns into a type definition

Examples:

```
unsigned long ulong; /* Define variable */
typedef long ulong_t; /* Define a new type ulong_t */
ulong_t ulong1; /* Define variable of new type */

char string[80]; /* Defining an array variable */
typedef char string_t[80]; /* Define a string type */
string_t string1; /* Define a variable of that type -- we can use
                  string1[32] now */
```

# Symbolic Names in the Data Safe Assignment

The data safe assignment calls for a function `data_safe()` with three arguments

- The first argument is a symbolic method: `ds_register`, `ds_store`, `ds_retrieve`, `ds_delete`
- We can implement this using a `int` argument and `#define`:

```
#define ds_register 1
#define ds_store    2
#define ds_retrieve 3
#define ds_delete   4
```

```
int data_safe(int method, int key, int value_or_index);
```

Problems:

- Nothing in the declaration of `data_safe()` tells us that the `int` is anything but a number
- The `#define` statements are independent

Wouldn't it be nice to create a new type to reflect the intended use?

# Enumerations in C

Enumeration data types can represent values from a **finite domain** using **symbolic** names

- The possible values are explicitly listed in the definition of the data type
- Typically, each value can be used in only one enumeration

In C, enumerations are created using the `enum` keyword

In C, enumeration types are **integer types**

- A definition of an enumeration type just assigns numerical values to the symbolic name
- Unless explicitly chosen otherwise, the symbolic names are numbered starting at 0, and increasing by one for each name
- However, **any** `int` value can be assigned to a variable of an enumeration type
- Likewise, we can assign any enumeration constant to any integer type variable

C enumerations have only mnemonic value, they do not enable the compiler to catch bugs resulting from mixing up different types

# Enumeration Syntax

An enumeration type is defined by the `enum` keyword, followed by a list of identifiers (**enumeration constants**) in curly brackets

The following code describes an enumeration data type for the data safe methods:

```
enum{ds_register, ds_store, ds_retrieve, ds_delete}
```

It can be used like any other type specifier:

```
int data_safe(enum{ds_register, ds_store, ds_retrieve, ds_delete}method,  
             int key, int value_or_index);
```

```
...
```

```
key = data_safe(ds_register, 0, 0);
```

## enum and typedef

Typically, enumeration data type are used to define new types

- The `enum` keyword describes the new type
- The `typedef` keyword assigns a name to the type
- The new type can then be used consistently throughout the program

Example:

```
typedef enum{ds_register, ds_store, ds_retrieve, ds_delete}DS_operation;  
  
int data_safe(DS_operation method, int key, int value_or_index);  
  
...  
key = data_safe(ds_register, 0, 0);
```

Typically, enumerations (and other new data types) are declared in **header files** (.h files), and form part of the interface of a module

## More on Enumerations

Since enumerations are actually integer types, we can assign specific values to the constants

- We can even assign the same value to different constants!

Example (also note preferred form of formatting for enums):

```
typedef enum
{
    ds_register = 1,
    ds_store = 2,
    ds_retrieve = 3,
    ds_delete = 4,
    ds_forget = 4
}DS_operation;
```

# Aggregating Data Types

Let's again look at the data safe assignment

- We somehow have to associate a **key** and a **value** (or multiple values)
- Simple approach: Use two arrays, one for keys, one for values
- If `keys[i] = key`, then `values[i]` holds a value associated with key

However, the association between those two elements is not reflected by this construction

- The two arrays are independent
- They can be manipulated independently
- There is not even a guaranty that both arrays have the same size!
- If we pass key and value to a function, we have to pass them as individual elements (what if we have 132 different elements?)

Solution: Creating **structures** that combine different elements into one

## struct

A **structure** is a datatype that may have any number of **members**

- Members can have different types
- Members can have any other type (including arrays or other structures)
- Members are referred to by their name in the structure

Java analogy: A structure type is a class, but:

- No member functions
- All members are public

Structures are defined using the `struct` keyword, followed by an optional name and a list of member definitions in curly braces

- Each member definition is a normal variable definition, giving type and name of the member

# Structure Example

Consider the following definition:

```
struct key_assoc {int key; int value;} key_pair;
```

- It creates a variable `key_pair` with two members
- They can be referred to by name:

```
key_pair.value = 10;  
...  
if(key_pair.key == user_key)  
{  
    count++;  
}
```

## struct and typedef

As with enumerations, structures are usually used with typedef:

```
typedef struct key_assoc
{
    int key;
    int value;
} key_pair_t;

static key_pair_t key_value_array[10000];
```

- The first definition defines a new type, `key_pair_t`
- The second one creates an array of 10000 of these pairs

Using the name (`struct key_assoc`), we can refer to the array even before we have seen the full definition

- Important for self-referential data types using **pointers**

## Exercises

Create a function that has two primary colours (red, blue, yellow) as input, and returns the colour that results from mixing them

- Use an enumeration type for the colours
- Use an struct to hold triples (colour1, colour2, mix) and an array to store all associations
- You can use **linear search** to find matching patterns for your input

# CSC322

## C Programming and UNIX

### Programming in C

### Data Structures and Pointers

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Representing Related Objects

Assume the following problem:

- In a drawing program, we need to represent geometrical shapes (circles, squares, rectangles, triangles...)
- There is some common information for all shapes:
  - \* Border colour
  - \* Line width
  - \* Fill colour (if any)
- However, the coordinates are different for each shape:
  - \* For a circle, we need center point and radius
  - \* For a square or rectangle we need two corners
  - \* For a triangle we need three corners

Object-oriented languages allow a base class shape, and derived classes for the different shapes

- In C, we have to program this explicitly, using **unions!**

# Unions

**Unions** of base types allow the new type to store one value of any of its base types (but only one at a time)

The syntax is analogous to that of structures:

- The keyword `union` is followed by a list of **member definitions** in curly braces

Example

- `union {int i; float f; char *str;} numval`
- `numval` can store either an integer or a floating point number, or a pointer to a character (normally a string)
- Access is as for structures: `numval.i` is the integer value

Note: Unions weaken the type system:

- `numval.f=1.0; printf("%d\n",numval.i);`
- Situations like that are, in general, impossible to detect at compile time

## Shape Example Continued (1)

```
typedef enum
{
    circle,
    square,
    rectangle,
    triangle
}ShapeType;
```

```
typedef enum
{
    red,
    green,
    blue,
    black, white
}ColourType
```

```
typedef struct
{
    int center_x;
    int center_y;
    int radius;
}CircleCoord;
```

## Shape Example Continued (2)

```
typedef struct
{
    int lower_left_x;
    int lower_left_y;
    int upper_right_x;
    int upper_right_y;
}RectangleCoord;
```

```
typedef RectangleCoord SquareCoord;
```

```
typedef struct
{
    int point1_x;
    int point1_y;
    int point2_x;
    int point2_y;
    int point3_x;
    int point4_y;
}TriangleCoord;
```

## Shape Example Continued (3)

```
typedef union
{
    CircleCoord    circle_coord;
    RectangleCoord rect_coord;
    SquareCoord    square_coord;
    TriangleCoord  tria_coord;
}ShapeCoord;
```

```
typedef struct
{
    ShapeType  type;
    int        border_width;
    ColourType border_colour;
    ColourType fill_colour;
    ShapeCoord coords;
}Shape;
```

## Shape Example Continued (4)

```
void draw_shape(Shape draw_obj)
{
    switch(draw_obj.type)
    {
        case circle:
            draw_circle(draw_obj.coords.circle_coord.center_x,
                        draw_obj.coords.circle_coord.center_y,
                        draw_obj.coords.circle_coord.radius,
                        draw_obj.border_width,
                        draw_obj.border_colour,
                        draw_obj.fill_colour);

            break;
        case square:
            draw_square(draw_obj.coords.square_coord.lower_left_x,
                       draw_obj.coords.square_coord.lower_left_y,
                       draw_obj.coords.square_coord.upper_right_x,
                       draw_obj.coords.square_coord.upper_right_y,
                       draw_obj.border_width,
                       draw_obj.border_colour,
                       draw_obj.fill_colour);

            break;
        ...
    }
}
```

# Pointers

**Pointers** are derived types of a base type

- A **pointer** is the memory address of an object of the base type
- Given a pointer, we can manipulate the object pointed to

Notice that there are **two** parts to a pointer:

- The actual memory address (a **dynamic** feature in the running program)
- The **type** of the pointer (pointer to `int`, pointer to `Shape`. . . ) telling us how to interpret the data at that address (a **static** feature that can be determined at compile time)

C uses the unary `*` to define variables of pointer types:

- `int *count;` defines the variable `count` as a pointer to `int`
- Notice that this pointer does not contain a valid address - there is no object of type `int` created along with the pointer!
- Pointers can be defined for any valid type in C: `struct{double real;double imag;} *complex` defines `complex` as a pointer to the struct

# Basic Pointer Operations in C

The most basic operations on pointers are:

- Given an object, return a pointer to it
- Given a pointer, give the object it points to (**dereference** the pointer)

C uses the unary `*` operator for both pointer definition and pointer dereferencing, and `&` for getting the address of an existing object

- `int var; int *p;` defines `var` to be a variable of type `int` and `p` to be a variable of type **pointer to int**
- `p = &var` makes `p` point to `var` (i.e. `p` now stores the address of `var`)
- `*p = 17;` assigns 17 to the `int` object that `p` points to (in our example, it would set `var` to 17)
- Note that `&(*p) == p` always is true for a pointer variable pointing to a valid object, as is `*(&var) == var` for an arbitrary variable!

# Pointers - A simple Example

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

void swap(int *x, int *y)
{
    int z;

    z  =*x;
    *x =*y;
    *y = z;
}

int main(void)
{
    int var1=7, var2=42;

    printf("var1: %d  var2: %d\n", var1, var2);
    swap(&var1, &var2);
    printf("var1: %d  var2: %d\n", var1, var2);
    return EXIT_SUCCESS;
}
```

## Example Continued

Output of the program:

```
var1: 7  var2: 42  
var1: 42 var2: 7
```

Note that this technique is an example of a frequent way to simulate **call by reference** in C

- Instead of passing an object, we pass a reference to it
- Allows changes to the object inside the function
- Often cheaper (especially for big objects)

# Why Pointers?

There are two main reasons for using pointers:

- Efficiency
- Dynamically growing data structures

## Efficiency Aspects

- Pointers are typically represented by one machine word
- Storing pointers instead of copies of large objects saves memory
- Passing pointers instead of large objects is much more efficient

## Dynamically growing data structures

- Each data type has a fixed size and **memory layout**
- Pointers allow us to build dynamically growing data structures by adding and removing fixed size cells

# Pointing at Nothing and Pointing Nowhere

Pointers of type `void*` are a special case:

- A `void*` pointer is a **generic pointer**, without associated base type
- `void*` pointers can be assigned to variables of any other pointer type (and vice versa)
- Such pointers are used primarily for **dynamic memory handling**

C has a special, reserved **NULL pointer** of type `void*`

- The NULL pointer is guaranteed to be different from all pointers pointing to legitimate objects
- It can be written as plain 0 (in a pointer context)
- `stdlib.h` defines a symbolic name, `NULL`, for the NULL pointer
- Dereferencing `NULL` is illegal!
- Notice that `NULL` is considered to be **false** if used in logical expressions
- Note: For most current machines, the NULL pointer actually is address 0. However, this is not guaranteed (and is false for some machines with strange memory models)

## Exercises

Write a program that prints the sizes of various build-in and self-defined data types (e.g. the `Shape` type and its subtypes). Do you see a relation between them?

Write a program that uses `swap()` to sort an array of integers and print it. If you feel adventurous, use `read_int_base()` from the `rpn_calc` example (or a similar function) to read integers to fill the array

## Notes

Please email the TA, Raghu, at his UMiami address, `raghu@lee.cs.miami.edu` from now on

Your grades for the assignments will be placed into your home directories

Solutions to the prime number assignment will be available shortly after noon

# CSC322

## C Programming and UNIX

### Programming in C Dynamic Data Structures

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Refresher: Pointers

A **pointer type** is a derived type of a **base type**

- A pointer is the address of an object of the base type
- Given a pointer `p`, `*p` gives us the object it points to
- Given an object `o`, `&o` gives us a pointer to that object in memory

An object of type `void*` is a generic pointer (i.e. a plain address without associated base type)

- A pointer of type `void*` can be assigned to a variable of any other pointer type
- Similarly, a value of any pointer type can be assigned to a `void*` variable

The special value `NULL` is a pointer of type `void*`

- It is guaranteed different from all pointers to valid object
- Its logical value is **false**, while that of all other pointers is **true**

# Dynamic Memory Handling

The C library offers functions for **dynamic memory handling**

- We can request a block of memory of a certain size
- If such a block is available, we will get a `void*` pointer to it
- This block can be used to store any object that fits into it
- If we do not need that object anymore, we can return it to the library

Such blocks can be used to build arbitrary sized data structures

- . . . e.g. by allocating bigger and bigger arrays if the need arises
- . . . or by using **pointers** within a structure to point to additional structures (which may contain further pointers)

## The malloc() function

We request a block of memory using `malloc()` (declared in `<stdlib.h>`)

- It's declared as `void *malloc(size_t size);`, i.e. it returns a generic pointer
- `size_t` is a new data type from the standard library. It's guaranteed to be an unsigned integer data type (often unsigned int)
- `malloc()` allocates a region big enough to hold the requested number of bytes on the **heap** (a reserved memory region) and returns the address of the first byte (a pointer to that region)
- The `sizeof` operator is used to get the necessary size for the object datatype
  - `p = malloc(sizeof(int));` allocates a memory region big enough to store an integer and makes `p` point to it
  - The `void*` pointer is silently converted to a pointer to `int`
- If no memory is available on the heap, `malloc()` will return the `NULL` pointer (also written as plain `0`)

# Freeing Allocated Memory

The counterpart to `malloc()` is `free()`

- It is declared in `<stdlib.h>` as  
`void free(void* ptr);`
- `free()` takes a pointer allocated with `malloc()` and returns the memory to the heap

Note that it is a bug to call `free()` with a pointer **not** obtained by calling `malloc()` (i.e. a pointer generated by applying `&` to a variable)

It also is a bug to call `free()` with the same pointer more than once

## More on Dynamic Memory Allocation

Good programming practice **always** checks if `malloc()` succeeded (i.e. returns not `NULL`)

- In multi-tasking systems, even small allocations may fail, because other processes consume resources
- The OS may limit memory usage to small values
- Failing to implement that check can lead to erratic and non-reproducible failure!

Similarly, each call to `malloc()` should (eventually) be followed by a call to `free()` for the pointer obtained

- If you do not know if you still need a piece of memory, or if a pointer still points somewhere, you are in **deep trouble**, anyways!
- By consequently freeing all allocated memory, you can easily check if you return the same number of block you allocate!

# Pointers are a Mixed Blessing!

## Dangling pointers

- A **dangling pointer** is a pointer not pointing to a valid object
- A call to `free()` leaves the pointer dangling (the pointer variable still holds the address of a block of memory, but we are no longer allowed to use it)
- Copying a pointer may also lead to additional dangling pointer if we call `free()` on one of the copies
- Trying to access a dangling pointer typically causes hard to find errors, including crashes

## Memory leaks

- A **memory leak** is a situation where we lose the reference to an allocated piece of memory:

```
p = malloc(100000 * sizeof(int));  
p = NULL; /* We just lost a huge gob of memory! */
```
- Memory leaks can cause programs to eventually run out of memory
- Periodically occurring leaks are catastrophic for server programs!

## Example: SecureMalloc()

Note: In my programs, there is typically at most a single call to `malloc()`:

```
void* SecureMalloc(size_t size)
{
    void* res = malloc(size);

    if(!res)
    {
        printf("malloc() failure -- out of memory?");
        exit(EXIT_FAILURE);
    }
    return res;
}
```

# Pointers and Structures/Unions

Most interesting data structures use pointers to structures

- Examples: Linear lists (see below), binary trees, terms, . . .

Most frequent operation: Given a pointer, access one of the elements of the structure (or union) pointed to

- `(*list).value = 0;`
- Note that that requires parentheses in C

More intuitive alternative:

- The `->` operator combines dereferencing and selection
- `list->value = 0;`
- This is the preferred form (and seen nearly exclusively in many programs)

## Example: Linear Lists (of Integers)

A list over  $a$  can be recursively defined as follows:

- The empty list is a list
- If  $l$  is a list and  $e$  is an element of the base type, then  $e . l$  is a list

We can represent that in C as follows:

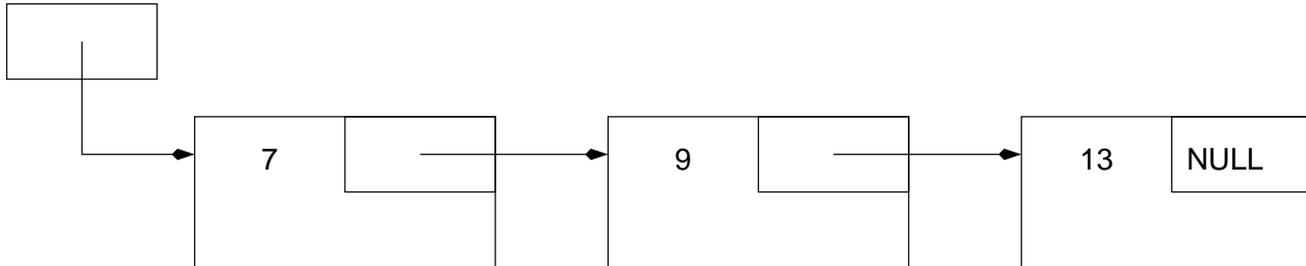
- The empty list is represented by the NULL pointer
- A non-empty list is represented by a pointer to a struct containing the element and a pointer to the rest of a list

Some list operations:

- Insert an element as the first element
- Insert an element as the last element
- Print the list elements in order
- Free the memory taken up by a list

## Example Continued

Graphical representation of the list structure for (7,9,13):



Notice the **anchor** of the list

## Example – Declarations

```
#ifndef INT_LISTS
#define INT_LISTS
#include <stdlib.h>
#include <stdio.h>

typedef struct int_list_cell
{
    int          value;
    struct int_list_cell *next;
}IntListCell;

typedef IntListCell *IntList_p;

void* SecureMalloc(size_t size);

void IntListInsertFirst(IntList_p *list, int new_val);
void IntListInsertLast(IntList_p *list, int new_val);
void IntListFree(IntList_p list);
void IntListPrint(IntList_p list);

#endif
```

## Example – Inserting At the Front

```
/* Insert a new integer as the first element of an integer list */  
  
void IntListInsertFirst(IntList_p *list, int new_val)  
{  
    IntList_p handle;  
  
    handle = SecureMalloc(sizeof (IntListCell));  
    handle->value = new_val;  
    handle->next = *list;  
    *list = handle;  
}
```

## Example – Inserting At the End

```
/* Insert a new integer as the last element of an integer list */

void IntListInsertLast(IntList_p *list, int new_val)
{
    IntList_p handle, last;

    handle = SecureMalloc(sizeof (IntListCell));
    handle->value = new_val;
    handle->next = NULL;

    if(!*list)
    {
        *list = handle;
    }
    else
    {
        last = find_last_element(*list);
        last->next = handle;
    }
}
```

## Example – Helper Function

```
/** Helper function: Given a non-empty list, return last element */  
  
IntList_p find_last_element(IntList_p list)  
{  
    if(list->next)  
    {  
        return find_last_element(list->next);  
    }  
    return list;  
}
```

## Example – Freeing Lists

```
/* Free the memory taken up by a list */  
  
void IntListFree(IntList_p list)  
{  
    if(list)  
    {  
        IntListFree(list->next); /* Free rest */  
        free(list); /* Free this cell */  
    }  
}
```

## Example – Printing Lists

```
/* Print a list as a sequence of numbers */

void IntListPrint(IntList_p list)
{
    IntList_p handle;

    for(handle = list; handle; handle = handle->next)
    {
        printf("%d ", handle->value);
    }
    putchar('\n');
}
```

## Example – Main Function

```
int main(void)
{
    int value;
    IntList_p list1 = NULL, list2 = NULL;

    SkipSpace();
    while(int_available(10))
    {
        value = read_int_base(10);
        IntListInsertFirst(&list1, value);
        IntListInsertLast(&list2, value);
        SkipSpace();
    }
    printf("List1: ");
    IntListPrint(list1);
    printf("List2: ");
    IntListPrint(list2);

    IntListFree(list1);
    IntListFree(list2);
    return EXIT_SUCCESS;
}
```

# Assignment

A **binary search tree** is either empty, or it consists of a node storing a **key** (the **root** of the tree), and a left and right subtree, such that all keys in the left subtree are smaller than the key in the node, and all keys in the right subtree are bigger

- To print a tree in (left-to-right) **preorder**, you first print the root, then the left subtree, then the right subtree
- To print a tree in (left-to-right) **postorder**, you first print the left subtree, then the right subtree, then the root
- To print a tree in **natural order**, you first print the left tree, then the root, then the right tree

Design a data structure for binary search trees with `int` keys, using dynamic memory handling

Implement functions to:

- Insert keys into the tree (ignoring keys already in the tree)
- Print a tree in preorder, natural order, and postorder
- Free the memory taken up by the tree

Use this datatype and the functions from `integerio` to write a program that reads a list of integers from `stdin` into a tree, and prints that tree in the three different orders

You can use the code from the linear list example as a base. The complete code will be available from the course homepage

# CSC322

## C Programming and UNIX

### Programming in C Pointers and Arrays

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Midterm Examn

Monday, Oct. 14th, 11:00 – 11:50

Topics: Everything we did so far

- UNIX file system layout
- Simple UNIX utilities
- Job Control
- Basic C
- Compilation and the preprocessor
- C flow control and functions
- Data structures in C
- Pointers

Friday we will refresh some of that stuff (but **do** reread the lecture notes yourself, and check the example solutions on the web)

# Refresher: Pointers

A **pointer type** is a derived type of a **base type**

- A pointer is the address of an object of the base type
- Given a pointer `p`, `*p` gives us the object it points to
- Given an object `o`, `&o` gives us a pointer to that object in memory

An object of type `void*` is a generic pointer (i.e. a plain address without associated base type)

- A pointer of type `void*` can be assigned to a variable of any other pointer type
- Similarly, a value of any pointer type can be assigned to a `void*` variable

The special value `NULL` is a pointer of type `void*`

- It is guaranteed different from all pointers to valid object
- Its logical value is **false**, while that of all other pointers is **true**

# Refresher: Dynamic Memory Handling

`void* malloc(size_t size);` is a function from `<stdlib.h>`

- It will return a pointer to an otherwise unused block of memory with at least `size` bytes (or `NULL` if no memory is available)
- Typical use: `int *p = malloc(sizeof(int));`

`void free(void* ptr);` is the counterpart to `malloc()`

- It takes a pointer to a block allocated with `malloc()` and returns the block to the heap
- It is a **(usually fatal)** bug to call `free()` more than once for the same block, or with a pointer not obtained from `malloc()`

Very frequent case: Allocation of memory for structs

- Accessing elements in a struct: `(*list).value = 0;`
- More readable alternative: `list->value = 0;`

# Pointers and Arrays in C

In C, arrays and pointers are strongly related:

- Everywhere except in a definition and the left hand side of an assignment, an array is equivalent to a pointer to its first element
- In particular, arrays are passed to functions by passing their address!
- More exactly: An array **degenerates** to a pointer if passed or used in pointer contexts

Not only can we treat arrays as pointers, we can also apply array operations to pointers:

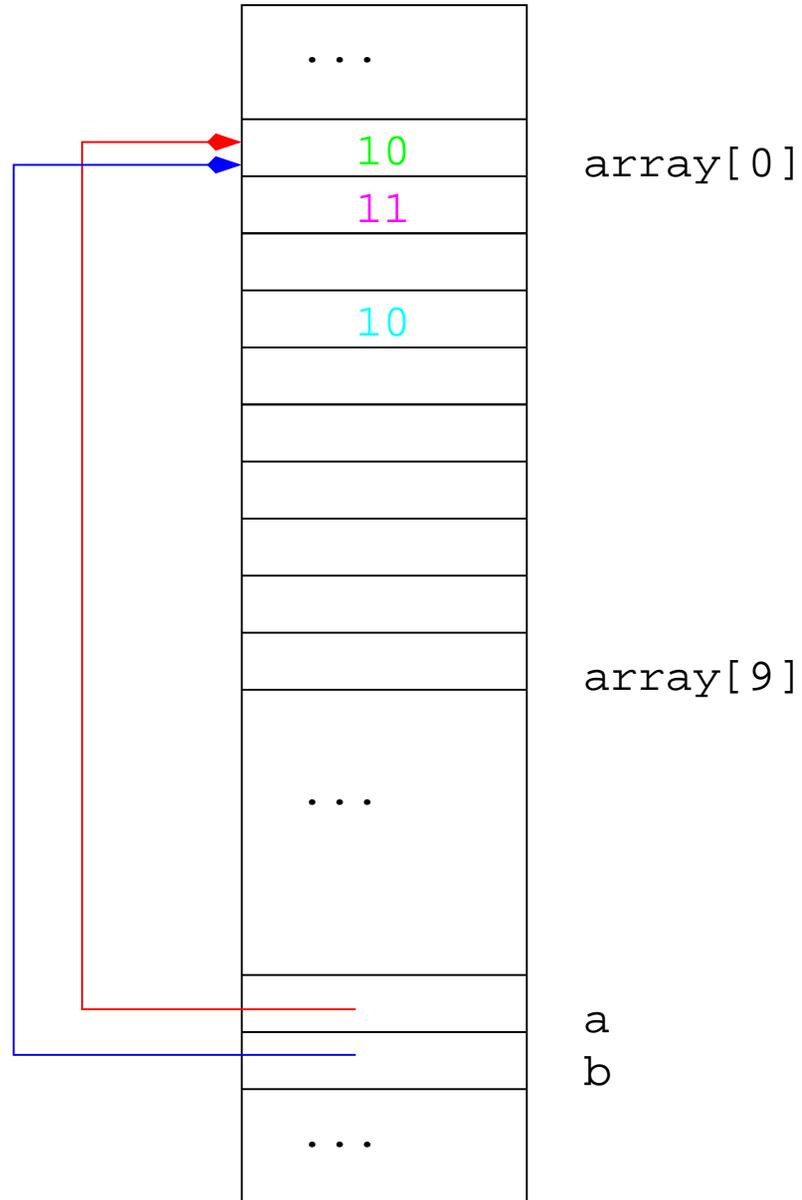
- If  $p$  is a pointer to the first element of an array, we can use  $p[3]$  to access the third element of that array
- In general, if  $p$  points to some memory address corresponding to an array element  $a[j]$ ,  $p[i]$  points to  $a[j+i]$

# Graphic Example

```
int array[10];
int *a, *b;

a = array;
b = &(array[0]);

array[0] = 10;
a[1] = 11;
b[3] = *a;
```



# Example

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

int main(void)
{
    char a[] = "CSC322\n";
    char *b;
    int i;

    b=a;

    printf(b);
    for(i=0;b[i];i++)
    {
        printf("Character %d: %c\n", i, b[i]);
    }
    return EXIT_SUCCESS;
}
```

## Example Output

Compiling: `gcc -o csc322 csc322.c`

Running:

CSC322

Character 0: C

Character 1: S

Character 2: C

Character 3: 3

Character 4: 2

Character 5: 2

Character 6:

# Parameter Passing in C

In C, parameters to functions are always **passed by value**

- The formal parameter (in the function) is a local variable
- It is initialized to the **value** of the actual parameter (the expression we used in the function call)
- Changing the local variable in the function does not change the formal parameter

Arrays degenerate into pointers to the first element, however!

- That pointer is still passed by value, however, in effect the **array** is passed **by reference**
- We can thus change the array elements from inside the function!

This is frequently used for efficient array manipulation!

- Sorting arrays
- Reading elements into an array from `stdin`
- Applying a transformation to all elements

## Example

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void upcase(char *string)
{
    int i;

    for(i=0; string[i]; i++)
    {
        string[i] = toupper(string[i]);
    }
}
int main(void)
{
    char str[] = "A test string.";

    printf("%s\n", str);
    upcase(str);
    printf("%s\n", str);
    return EXIT_SUCCESS;
}
```

## Example Output

A test string.

A TEST STRING.

**CSC322**  
**C Programming and UNIX**  
**Midterm Review**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# UNIX Concepts

UNIX is a multi-user system

- Users have a user name, a numerical user id (e.g. 500), and a home directory
- The privileged user root with UID 0 has (essentially) unlimited access

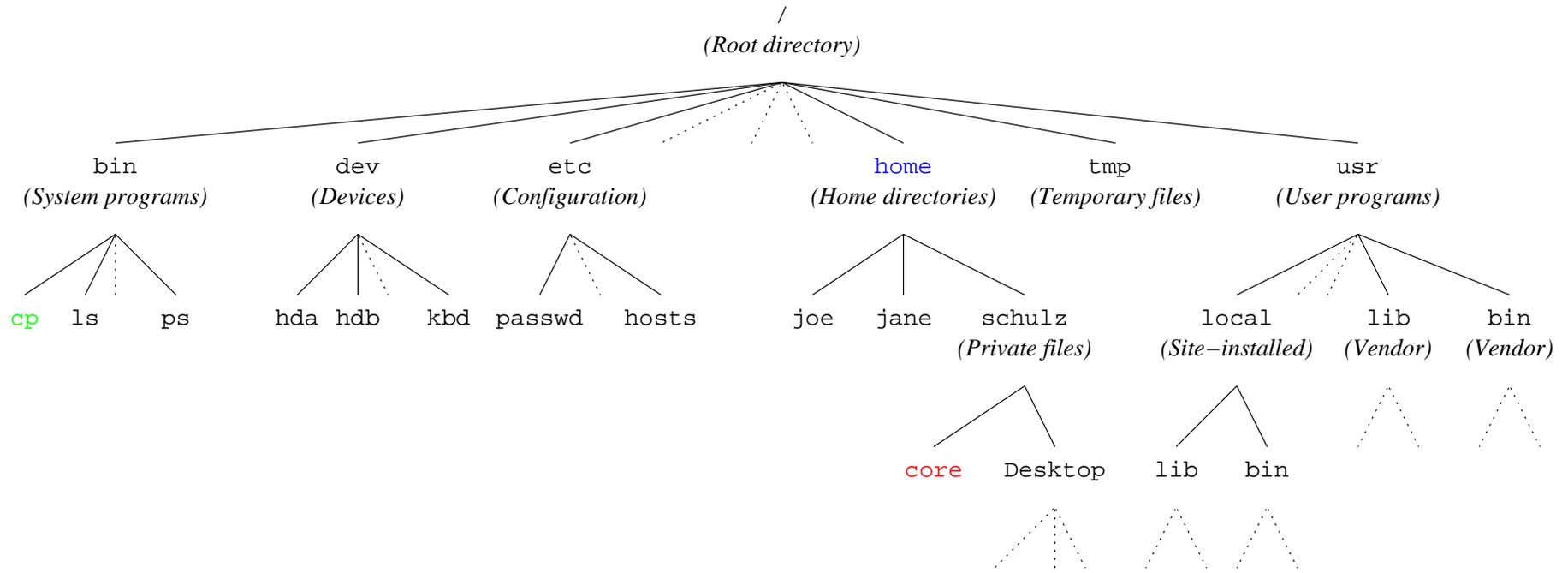
UNIX is a multi-tasking system, i.e. it can run multiple programs at once. A running program (with its data) is called a **process**. Each process has:

- Owner (a user)
- Working directory (a place in the file system)
- Various resources

A **shell** is a command interpreter, i.e. a **process** accepting and executing commands from a user.

- A shell is typically owned by the user using it
- The initial working directory of a shell is typically the user's home directory (but can be changed by commands)

# The File System



In UNIX, all files are organized in a single **directory tree**

There are two main types of files:

- Plain files (containing data)
- Directories, containing both plain files (optionally) and other directories

# Globbering

Glob patterns describe **sets of file names**

A string is a **wildcard pattern** if it contains one of **?**, **\*** or **[**

A wildcard pattern expands into all file names matching it

- A normal letter in a pattern matches itself
- A **?** in a pattern matches any one letter
- A **\*** in a pattern matches any string
- A pattern **[l<sub>1</sub>. . . l<sub>n</sub>]** matches any one of the enclosed letters (exception: **!** as the first letter)
- A pattern **[!l<sub>1</sub>. . . l<sub>n</sub>]** matches any one of the characters **not** in the set
- A leading **.** in a filename is never matched by anything except an explicit leading dot

Important: Globbing is performed **by the shell**, not an application program!

# Some Important UNIX Commands (1)

Orientation and moving around

- **whoami**
- **pwd** – print working directory
- **cd** – change directory
- **ls** – list files (Important options: **-a**, **-l**)

Operating on files

- **cat** – concatenate and print files
- **less** and **more** – print files page by page
- **touch** – change access dates (or create empty files)
- **mv** – move files
- **cp** – copy files
- **rm** – remove files
- **wc** – count words (and lines and characters)

## Some Important UNIX Commands (2)

Working on Directories:

- **mkdir** – make a new directory
- **rmdir** – remove an empty directory

Miscellaneous

- **man** – read the manual (-k: Search for keywords in the manual)
- **info** – read info format documentation (also available through **emacs**)
- **echo** – Print arguments
- **grep** – Search lines matching a **regular expression**

# Input and Output Redirection, Piping

The three standard UNIX IO channels are

- `stdin` (Standard Input)
- `stdout` (Standard Output)
- `stderr` (Errors)

Normal output redirection redirects `stdout` into a file:

Input redirection makes `stdin` read from a file

Piping connects one processes `stdout` to the `stdin` of another process

```
cat > newfile           # Read stdin, write to newfile
cat < newfile           # Read newfile, write to terminal
cat > newfile < oldfile # Poor man's copy
cat newfile | wc        # Count words in newfile
```

# Process Control

Processes started from the shell can be

- Running or Suspended
- In the foreground (accepting keyboard input) or in the background

Simple process control:

- Running a command followed by `&` starts it in the background (normally commands are executed in the foreground)
- `^Z` (Control-Z) will suspend a foreground process
- `^C` (Control-C) will terminate it
- **fg** wakes a suspended process and puts it into the foreground
- **bg** puts it into the background
- **kill** can be used to terminate it
- **jobs** prints a list of active processes started from a shell

# C Compiling with gcc

Programs consisting of a **single .c file** can be compiled in one step

- **gcc -o file file.c** will compile **file.c** into an executable program **file**

Multiple C files must be compiled and linked separately!

- **gcc -c -o file1.o file1.c** compiles the file into an object (**.o**) file
- **gcc -o file file1.o file2.o...** links the different object files together to form an executable

Important **gcc** options:

- **-o <name>**: Give the name of the output file
- **-ansi**: Compile strict ANSI-89 C only
- **-Wall**: Warn about all dubious lines
- **-c**: Don't perform linking, just generate a (linkable) object file
- **-O** – **-O6**: Use increasing levels of optimization to generate faster executables

# C Datatypes

The language offers a set of **basic types** built into the language

- char, short, int, long, long long
- float, double
- Integer data types come in signed and unsigned variety!

We can define new, quasi-basic types as enumerations (enum)

We can derive new types as follows:

- Arrays over a base type ([])
- Structures combining base types (struct)
- Unions (able to store alternative types) (union)
- Pointer to a base type (\*)

typedef is used to define named new types

# Flow Control

`if...else`

- Conditional execution

`switch`

- Select between many alternatives, based on a single integer type variable
- Remember **fall through** property and `break;!`

`while`

- Loop as long as a condition is true

`for`

- As while, but included initialization and update in a single statement

# Functions

Any C program is a collection of **functions**

- There has to be exactly **one** function called `main()` in the program
- Execution starts by a call to `main()` (executed by the OS)
- A **function** definition consists of a header and a body

The header consists of:

- The **return type** of the function
- The **name** of the function
- A parenthesized list of **formal arguments**

The **body** of the function is a sequence of declarations and statements

- Execution of the function ends when a `return` statement is encountered or the end of the body is reached
- The argument of the `return` statement is the value returned from the function call

# C Preprocessor

The `#include` directive is used to include other files (the contents of the named file replaces the `#include` directive)

The `#define` directive is used to define **macros**

- Macros can simply define a textual constant
- Macros can have formal arguments, which will be instantiated in the replacement text

`#if/#else/#endif` is used for conditional compilation

- The controlling expression of the `#if` has to be a **constant integer expression**
- Special case: `#ifdef` tests if a macro is defined
- Special case: `#ifndef` tests if a macro is **not** defined

# Exercises

Reread the lecture notes

Download the C examples from the Web

- Read the code
- Compile them by hand
- Run them

# CSC322

## C Programming and UNIX

### Programming in C

### Dynamic Arrays and Pointer Arithmetic

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Dynamically Allocated Arrays

Since pointers and arrays can be used interchangeably in many contexts, we can use `malloc()` to allocate arrays of whatever size we need!

- The size of an array of `n` elements of type `t` is just `n*sizeof(t)`

Applications:

- We can allocate arrays in a function and return pointers to them (remember that local variables are destroyed when control leaves a function)
- We can determine array size at run time
- We can dynamically increase array size by:
  - \* Allocating a bigger array
  - \* Copying the old array into the initial part of the new array
  - \* Freeing the old array

## Example

```
#include <stdio.h>
#include <stdlib.h>
#define BUF_SIZE 1024
int main(void)
{
    int c, count=0;
    char* buffer;

    buffer = malloc(sizeof(char)*BUF_SIZE);/* Missing check! */
    while((c=getchar())!=EOF)
    {
        if(count == BUF_SIZE-1)
        {
            printf("Buffer full\n"); exit(EXIT_FAILURE);
        }
        buffer[count++] = c;
    }
    buffer[count] = '\0';
    printf("%s\n", buffer);
    free(buffer);
    return EXIT_SUCCESS;
}
```

## Changing Allocated Block Size: `realloc()`

`void* realloc(void* ptr, size_t size);` is defined in `<stdlib.h>`

- It's first argument is a pointer to a block of memory on the heap (obtained with `malloc()`, `realloc()`, or an equivalent function)
- The second argument is a desired new size of the block
- `realloc()` returns a pointer to a new block of memory, of the desired size (if available, otherwise `NULL`)
- If `realloc()` is successful, the initial part of the new block (up to the smaller of the two sizes) will be identical to the old block

Special cases:

- if `ptr` is `NULL`, `realloc()` is equivalent to `malloc()`
- If `size` is `NULL`, `realloc()` is equivalent to `free`
- As with `malloc()`, we always have to check the return value!

Most common use: Increase the size of some array

## Example: Growing the Buffer as Needed

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int c, count=0, size = 2;
    char* buffer;

    buffer = malloc(sizeof(char)*size); /* Missing check! */
    while((c=getchar())!=EOF)
    {
        if(count == size - 1)
        {
            size = size * 2;
            buffer = realloc(buffer, size); /* Missing check! */
        }
        buffer[count++] = c;
    }
    buffer[count] = '\0';
    printf("%s\n", buffer);
    free(buffer);
    return EXIT_SUCCESS;
}
```

# Additional Pointer Properties

Pointers of the same type can be compared using `<`, `>`, `<=`, `>=`

- The result is only defined, when both pointers point at elements in the same array or struct, or if both pointers point to addresses within the same `malloc()`ed block
- Pointers to elements with a smaller index are smaller than pointers to elements with a larger index

**Pointer arithmetic** allows addition of integers to (non-void) pointers

- If `p` points to element `n` in an array, `p+k` points to element `n+k`
- As a special case, `p[n]` and `*(p+n)` can again be used interchangeably (and often are in practice)
- Most frequent case: Use `p++` to advance a pointer to the next element in an array
- Note that pointer arithmetic only works on non-void pointers

# Pointer Arithmetic

```
char *cp, *cq;
```

```
int *ip, *iq;
```

	char arr1[28]
cp	a
cp+1	b
cp+2	c
	d
	e
	f
	g
	h
	i
	j
	k
	l
cq=cp+12	m
	n
	o
	p
	q
	r
	s
	t
	u
	v
	w
	x
	y
	z
	0
	\0

	int arr2[7]
ip	17
p+1	42
&ip[2]	-13
iq=ip+3	2
	2147483647
iq+2	1024
	-1

# Pointer Arithmetic Example

```
#include <stdlib.h>
#include <stdio.h>
int print_str(char *string)
{
    int i = 0;
    while(*string)
    {
        putchar(*string);
        string++;
        i++;
    }
    return i;
}
int main(int argc, char* argv[])
{
    char message[] = "Hello World!\n";
    int count;
    count = print_str(message);
    printf("Printed %d characters!\n", count);
    return EXIT_SUCCESS;
}
```

## Reading the Command Line: `argc` and `argv`

The C standard defines a standardized way for a program to access its (command line) arguments: `main()` can be defined with two additional arguments

- `int argc` gives the **number** of arguments (including the program name)
- `char *argv[]` is an array of pointers to character strings each corresponding to a command line argument

Since the name under which the program was called is included among its arguments, `argc` is always at least one

- `argv[0]` is the program name
- `argv[argc-1]` is the last argument
- `argv[argc]` is guaranteed to be `NULL`

## Example: Echoing Arguments

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

int main(int argc, char* argv[])
{
    int i;

    for(i=1; i<argc; i++)
    {
        printf("%s ", argv[i]);
    }
    putchar('\n');

    return EXIT_SUCCESS;
}
```

## Example: Echoing Arguments – Idiomatic

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

int main(int argc, char* argv[])
{
    char **p;

    for(p=argv+1; *p; p++)
    {
        printf("%s ", *p);
    }
    putchar('\n');

    return EXIT_SUCCESS;
}
```

## Exercises

Write a function that reads a line (terminated by `'\n'`) into an array, and a program that reads files line by line and prints it back. You can assume a reasonable fixed length (e.g. 1024 characters) per line

Write a library that implements a dynamic array type for `char` arrays.

- Implement functions that can assign and retrieve values from arbitrary positions, e.g. `void darrayassign(darray_p array, int index, char newval)` and `char darrayvalue(darray_p array, int index)`
- Write a function `darrayalloc()` that returns a pointer to a freshly allocated dynamic array
- Write a function `darrayfree()` that frees such an array
- Hint: Use a struct that contains at least a pointer to the dynamically allocated proper array and the currently allocated array size. If an index greater than the size occurs, use `realloc()` to increase the size

Put the two together: Write a function that can read a line of any length and returns (a pointer to) it

**CSC322**  
**C Programming and UNIX**  
**Making Programming Easier**

Stephan Schulz

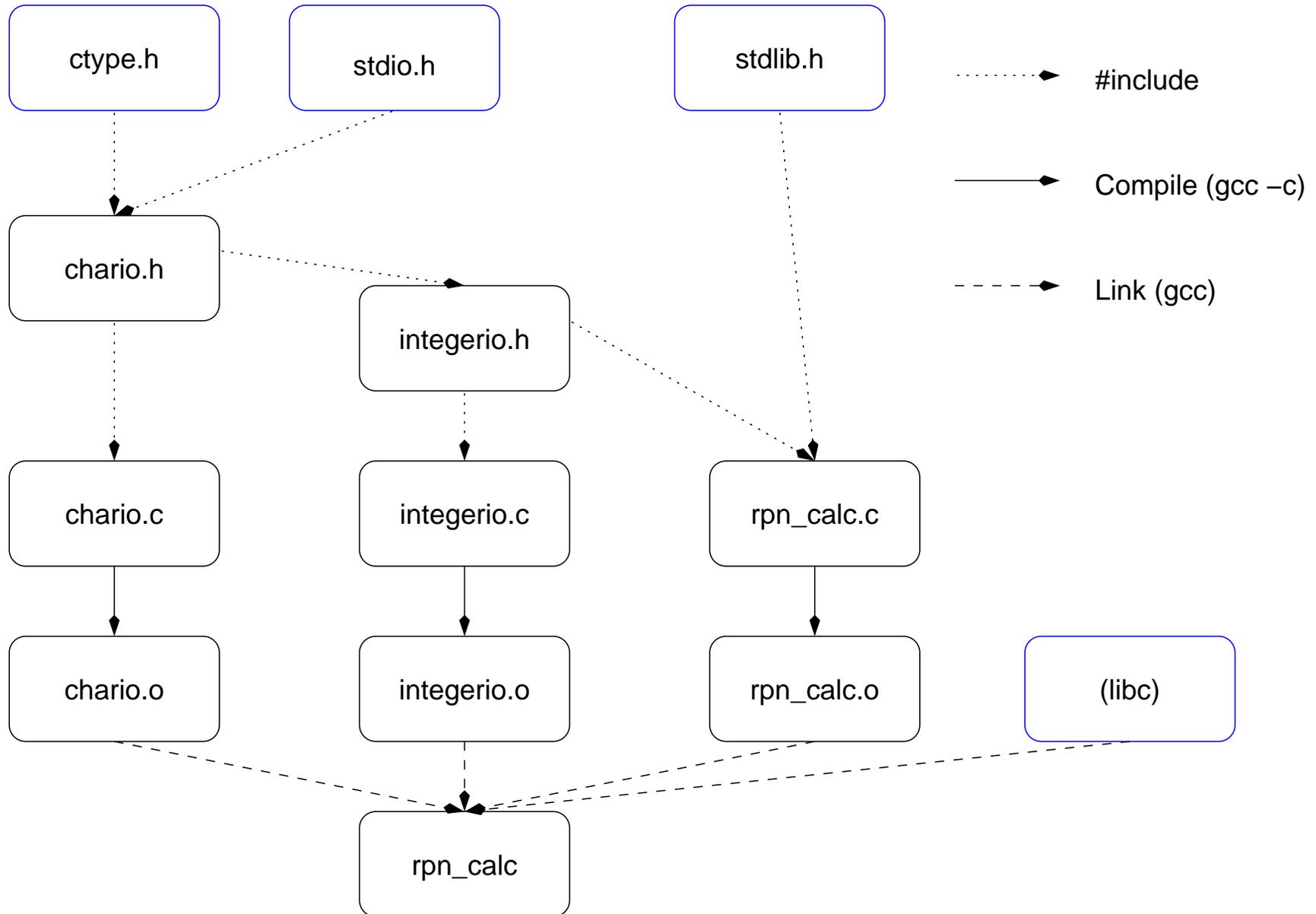
Department of Computer Science

University of Miami

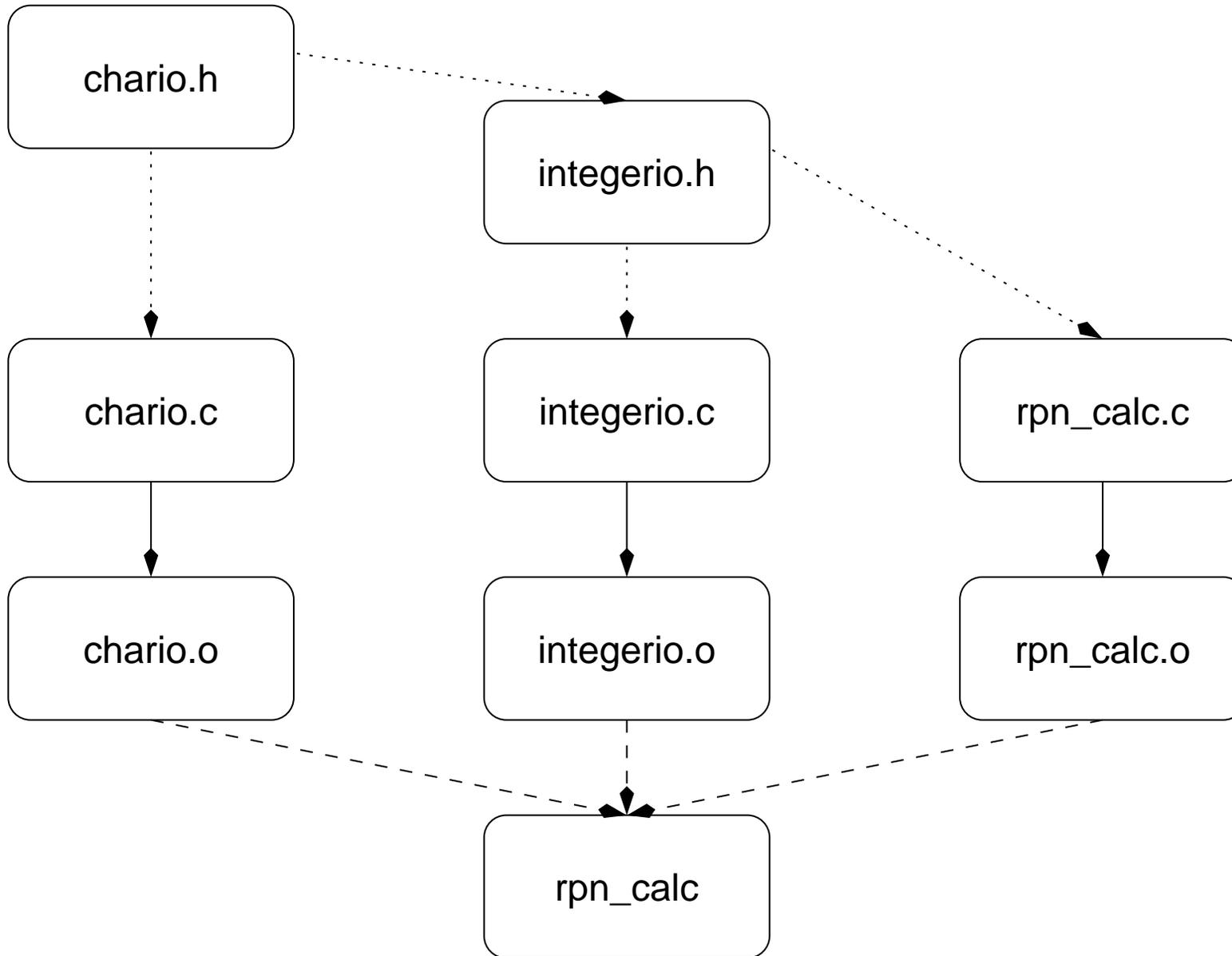
`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# The rpn\_calc Example



## The rpn\_calc Example (Simplified)



# Program Dependencies

In the example, changing one file may make many steps necessary to propagate the change

- If any `.h` file has been changed, all `.c` files that include it may have to be recompiled
- If any `.c` file has changed, it has to be recompiled
- If any `.o` file has changed, we need to relink the program
- In more complex programs, even more such situations exist!

Recompiling **all** files and relinking (in the right order) solves the problem. . .

- Very expensive for large programs
  - \* Mozilla, Windows NT: Many hours
  - \* Linux kernel (on modern machine): Many minutes
  - \* E theorem prover: 1-2 minutes
- We still need to know the right order!

Recompiling by hand is error-prone (and inconvenient)

# UNIX User Utilities: make

make is a UNIX utility that can automatically update large projects with complex dependencies

- Dependencies and build instructions are described in a file called Makefile (preferred form) or makefile

A makefile contains a number of **rules** for rebuilding the project

A rule consist of a **target**, a list of **prerequisites**, and commands for rebuilding

- The **target** normally is a file that needs rebuilding
- The **prerequisites** are all files that are needed to rebuild the target
- Finally, the commands describe how to rebuild the target

Semantics:

- Execution begins with the first target (or a target given on the command line)
- First, rules for all prerequisites are activated (if any)
- Then, if the target does not exist, or if any of the prerequisites is younger than the target, the commands are executed

## Example: rpn\_calc makefile

```
# Relink rpn_calc if one of the object files changed
rpn_calc: chario.o integerio.o rpn_calc.o
    gcc -ansi -Wall -o rpn_calc chario.o integerio.o rpn_calc.o

# Recompile chario if either the .h or the .c changed
chario.o: chario.h chario.c
    gcc -ansi -Wall -c -o chario.o chario.c

#...
integerio.o: chario.h integerio.h integerio.c
    gcc -ansi -Wall -c -o integerio.o integerio.c

#...
rpn_calc.o: integerio.h chario.h rpn_calc.c
    gcc -Wall -ansi -c -o rpn_calc.o rpn_calc.c

# General format:
#
# TARGET: PREREQUISITES
# [TAB] command1
# [TAB] command2 ...
```

# Built-In Rules and Makefile Variables

make knows how to remake many types of files!

- In particular, make knows how to run the C compiler to build object (.o) files from .c files

We could have omitted the compiler command e.g. from the rule for chario.o:

```
chario.o: chario.h chario.c
```

make allows the use of **variables**, both for customization and for more compact makefiles

- Variables are set using the assignment operator:  
RPN=chario.o integerio.o rpn\_calc.o
- Variables are referenced using a \$: \$(RPN)

Important predefined variables:

- CC: Name of the C compiler
- CFLAGS: Additional flags for the C compiler

## Example: rpn\_calc makefile revisited

```
CC=gcc
CFLAGS=-Wall -ansi -O6

RPN=chario.o integerio.o rpn_calc.o
# Relink rpn_calc if one of the object files changed
rpn_calc: chario.o integerio.o rpn_calc.o
    gcc -ansi -Wall -o rpn_calc $(RPN)
chario.o: chario.h chario.c
integerio.o: chario.h integerio.h integerio.c
rpn_calc.o: integerio.h chario.h rpn_calc.c
```

Rebuilding from scratch:

```
$ rm *.o
$ make
gcc -Wall -ansi -O6 -c -o chario.o chario.c
gcc -Wall -ansi -O6 -c -o integerio.o integerio.c
gcc -Wall -ansi -O6 -c -o rpn_calc.o rpn_calc.c
gcc -ansi -Wall -o rpn_calc chario.o integerio.o rpn_calc.o
```

# Phony Targets

Not all targets need to correspond to files

- Targets not corresponding to a file are called **phony**
- Since no corresponding file exists, commands in rules with phony targets are always executed

Frequent use: Cleanup commands

```
clean:  
    rm *.o  
    rm rpn_calc
```

# Assignment

Write a program `sort_csc322` that reads an arbitrary length file line by line (allowing for arbitrary line length), sort the lines in ASCIIbetical order, and prints it back

- Order: A letter that has a smaller numerical value is smaller than a letter that has a bigger numerical value. To compare strings, find the first character that differs (including the terminating `'\0'`)
- Hints:
  - \* If you are lazy, reuse the binary tree code for sorting!
  - \* Define a data type for the lines, using `struct` and `char*`

Include a `Makefile` for building your final program from the sources!

- More hint: If you are lazy, read **man makedepend**

# CSC322

## C Programming and UNIX

### Odds And Ends

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Errors, Bugs, and Other Unpleasant Animals

Most hard-to-handle errors are **not** syntax errors

- Most syntax errors go away with experience
- Even if not, they are usually easy to find and fix!

Most serious problems are runtime errors, resulting from faulty program logic

- Finding logic errors is hard
- Not finding them is worse!

Examples:

- Spacecraft may crash (Mars Climate Orbiter) or explode (Ariane-5)
- Medical devices may actually kill patients (Therac-25 cancer treatment device)
- The IRS may decide you are a tax evader, and have you arrested!

Ways to (more) correct software:

- Formal methods and a controlled development process
- Testing
- **Internal consistency checks**

# Assertions

Internal consistency checks are used to verify that assumptions about the state of the program are true

- Very frequent use: Check if parameters to functions have valid values
- Check **loop invariants**
- Check array boundaries

## Problems

- Checks are inconvenient to program
- The checks may cause unacceptable slowdowns (E theorem prover: Factor of 2–3, depending on input data)

C solution: The `<assert.h>` header file and macros

- Convenient way to add simple consistency checks
- Checks can be disabled at compile time (now slow-down for final product)

## <assert.h> and assert()

The `assert()` macro is defined in `assert.h`

It is used with a single argument

If that argument has the truth value “true”, nothing happens

Otherwise, `assert()` prints an error message and aborts the program

- Error message contains text of the assertion, name of source file, line in file

If the preprocessor macro `NDEBUG` is defined, `assert()` is ignored (defined as the empty macro)

Careful use of `assert()` while testing makes your programs much more robust and helps you weed out errors earlier!

## Example

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int gcd(int a, int b)
{
    assert(a>0);assert(b>0);
    if(a==b)
    {
        return a;
    }
    if(a > b)
    {
        return gcd(a-b,b);
    }
    return gcd(b-a,a);
}
int main(void)
{
    printf("Result: %d\n", gcd(15,3));
    printf("Result: %d\n", gcd(0,2));
    return EXIT_SUCCESS;
}
```

## Example (Continued)

```
$ gcc -ansi -Wall -o gcd_assert gcd_assert.c
```

```
$ ./gcd_assert
```

```
Result: 3
```

```
gcd_assert: gcd_assert.c:7: gcd: Assertion 'a>0' failed.
```

```
Abort
```

```
$ gcc -ansi -Wall -o gcd_assert gcd_assert.c -DNDEBUG
```

```
$ ./gcd_assert
```

```
Result: 3
```

```
Segmentation fault
```

# Search in Loops

A frequent use of loops is to search for something in a sequence (list or array) of elements

First attempt: Search for an element with property P in array

```
for(i=0; (i< array_size) && !P(array[i]); i=i+1)
{ /* Empty Body */ }
if(i!=array_size)
{
    do_something(array[i]);
}
```

- Combines property test and loop traversal test (unrelated tests!) in one expression
- Property test is negated
- We still have to check if we found something at the end (in a not very intuitive test)

Is there a better way?

## Early Exit: break

C offers a way to handle early loop exits

The `break;` statement will always exit the innermost (structured) loop (or `switch`) statement

Example revisited:

```
for(i=0; i< array_size; i=i+1)
{
    if(P(array[i])
    {
        do_something(array[i]);
        break;
    }
}
```

- I find this easier to read
- Note that the loop is still single entry/single exit, although control flow in the loop is more complex

## Selective Operations and Special Cases

Assume we have a sequence of elements, and have to handle them differently, depending on properties:

```
for(i=0; i< array_size; i=i+1)
{
    if(P1(array[i])
    {
        /* Nothing to do */
    }
    else if(P2(array[i]))
    {
        do_something(array[i]);
    }
    else
    {
        do_something_really_complex(array[i]);
    }
}
```

Because of the special cases, all the main stuff is hidden away in an else

Wouldn't it be nice to just goto the top of the loop?

## Early Continuation: `continue`

A `continue` statement will immediately start a new iteration of the current loop

– For C for loops, the update expression will be evaluated!

Example with `continue`:

```
for(i=0; i< array_size; i=i+1)
{
    if(P1(array[i])
    {
        continue;
    }
    if(P2(array[i]))
    {
        do_something2(array[i]);
        continue;
    }
    do_something_really_complex(array[i]);
}
```

## do/while Loops

Both `while` and `for` loops in C are controlled at the top

- If the controlling expression is false, the loop is not entered at all

Occasionally, we can express some algorithms more conveniently, if we have a controlling expression at the end of the loop

- Loop body is always executed at least once!

C language construct: `do/while()` loop

```
do
{
    loop body
}while(E);
```

- If E evaluates to true at the end of the loop, control is transferred back to the `do`

## Example

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

    do
    {
        printf("Please choose 1 for half of a bad joke or 2 for a cool number!\n");
        c=getchar();
    }while(!(c=='1' || c=='2'));
    if(c=='1')
    {
        printf("Why did the chicken cross the road? ...\n");
    }
    else
    {
        printf("42\n");
    }
    return EXIT_SUCCESS;
}
```

# Some Loop Statistics

E theorem prover

- State of the art **automated theorem prover**
- About 100000 lines of C code (20000 statements, the rest is comments, white space, definitions....)
- Total of 942 structured loop statements in code base

521 for() loops

- Most iterate over integer values (for i=0; i<limit; i++)

421 while loops

- Many iterate over linked structures:

```
while(handle is not NULL)
{
    do something with *handle;
    make handle point to "next" element;
}
```

0 do loops, but plenty of recursion

# Exercises

Go back over your exercises and assignments, and think about good places to insert `assert()` statements

Write a non-recursive function that searches for a value in a binary search tree. Use `break` to leave the loop if you found it!

Think about uses for `do/while` ;-)

# CSC322

## C Programming and UNIX

### Function Pointers C Standard Library (1)

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Functions as Arguments

Occasionally, you want to be able to pass around functions just like data

Example:

- Configure an event handler (“call this function if the UPS signals power-down”)
- Simulate some object-oriented techniques (virtual functions), e.g. to implement **destructors**
- Most importantly: Parameterize algorithms

Functional languages have functions as **first class** objects

C is less flexible, but gives us **function pointers** to pass as arguments and store in variables

- Idea: Pointers are addresses in memory
- Functions are pieces of code in memory

# Function Pointers

We can use the address of a function to call it!

- As with normal pointers, we need know the type of the function (in this case, the return type and the type of the arguments it takes)

Syntax: Same principle as for other type!

- To declare a function pointer, use a function declaration, but add parentheses and add a \* to denote that it is a pointer:

```
int (*add)(int x1, int x2);
```

- This declares add to be a pointer to a function accepting two integer arguments and returning a third integer

To use a function pointer: Just dereference the pointer

```
a = (*add)(10,20);
```

To assign a value to the pointer, just get the address of a function:

```
add = &some_function_name;
```

## Function Pointers (2)

To confuse students (and for convenience), it is possible to omit both the dereferencing in calling and the ampersand in assigning:

```
add = somefunction
a = add(10,20);
```

- Since there is nothing else you can do with functions in C, these simplifications do not create an ambiguity
- They tend to make code **easier** to read, though, especially with functions that return pointers

Note: Since declarations quickly become hard to read, it is wise to always use `typedef` to define a suitable function pointer type!

## Example

```
#include <stdlib.h>
#include <stdio.h>
int add(int x1, int x2)
{ return x1+x2; }
int subtract(int x1, int x2)
{ return x1-x2; }
void use_fun(int limit, int (*fun)(int x1, int x2))
{
    int i;
    for(i=0; i<limit; i++)
    {
        printf("Result: %d\n",fun(20,i));
    }
}
int main(int argc, char* argv[])
{
    use_fun(5, &add); /* Can drop & here */
    printf("-----\n");
    use_fun(5, subtract); /* Can add & here! */
    return EXIT_SUCCESS;
}
```

## Example Output

Result: 20

Result: 21

Result: 22

Result: 23

Result: 24

-----

Result: 20

Result: 19

Result: 18

Result: 17

Result: 16

## C Library Functions: `qsort()`

`qsort()` is a very useful C library function (declared in `<stdlib.h>` that is able to sort any kind of array (and normally does so very efficiently)!

`qsort` is defined as follows:

```
void qsort(void *base, size_t nmemb, size_t size,  
           int(*compar)(const void *, const void *));
```

- The first argument points to the array to be sorted (i.e. to its first element)
- The second argument is the **number** of elements in the array
- The third argument gives the size of a single element
- Finally, the last element is a function pointer of a function taking two pointer arguments, and returning an integer value

## C Library Functions: qsort() (2)

qsort definition (repeated):

```
void qsort(void *base, size_t nmemb, size_t size,  
           int(*compar)(const void *, const void *));
```

Purpose of compar: Let the caller define an order on elements

- (\*compar)() is called by qsort() to compare two arguments
- It gets pointers to two array elements as arguments
- It should compare these elements and return
  - \* 0, if the two elements are equal (under the order)
  - \* A negative integer, if the first element is smaller
  - \* A positive integer, if the first element is greater

## Example

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

typedef int (*CompareFun)(const void* arg1, const void* arg2);

int compare_ints(int *arg1, int* arg2)
{
    if(*arg1 < *arg2)
    {
        return -1;
    }
    if(*arg1 > *arg2)
    {
        return 1;
    }
    return 0;
}
```

## Example (continued)

```
int main(int argc, char* argv[])
{
    int array[10], i;

    for(i=0; i<10; i++)
    {
        array[i] = rand()%128;
    }
    printf("Unsorted: \n");
    for(i=0; i<10; i++)
    {
        printf("%d\n", array[i]);
    }
    qsort(array, 10, sizeof(int), (CompareFun)compare_ints);
    printf("Sorted: \n");
    for(i=0; i<10; i++)
    {
        printf("%d\n", array[i]);
    }
    return EXIT_SUCCESS;
}
```

## Example (Output)

Unsorted:

103

70

105

115

81

127

74

108

41

77

Sorted:

41

70

74

77

81

103

105

108

115

127

# The C Standard Library

The C Standard Library contains a large number of functions, some data types and system dependent constants

- Covers many things that other languages handle in the main language
- Also contains primitives for extending some parts of the language
- Notably missing: Any functionality for graphics (only stream-based I/O)

Most parts of the library are automatically linked with the C programs (exception: Floating point math functions)

The standard library is part of the C standard, and has to be supported on any standards-compliant full C implementation

- Code written using only the standard library should be highly portable

The library has 15 parts with corresponding header files

- Some declarations are repeated in different headers

# C Standard Library Organisation

- `assert.h`: Assertions (\*)
- `ctype.h`: Character classes (+)
- `errno.h`: Error reporting for library functions
- `float.h`: Implementation limits for floating point numbers
- `limits.h`: Limits for other things
- `locale.h`: Localization support
- `math.h`: Mathematical functions
- `setjmp.h`: Non-local function exits
- `signal.h`: Signal handling
- `stdarg.h`: Support for functions with a variable number of arguments (as e.g. `printf()`)
- `stddef.h`: Standard macros and typedefs
- `stdio.h`: Input and output (+)
- `stdlib.h`: Miscellaneous library functions (+)
- `string.h`: String (character array) handling
- `time.h`: Functions about time and date

## Error Handling: `errno.h`

Library functions typically signal an error by returning an **out of range** value, i.e. a value that cannot possibly be correct

- For many functions that is `-1` or `NULL`

They communicate the cause of the error by setting the global `int` variable `errno` to a specific value

- At the program start, `errno` is guaranteed to have the value `0`
- No library function will ever set `errno` to `0`, but failed library functions will set it to an implementation-defined value encoding the cause of the error

Error codes have symbolic names (with `#define`):

- `EDOM`: (Required by the standard) Domain error for some math functions
- `ERANGE`: (Required by the standard) Range error for some math functions
- `EAGAIN`: (UNIX) Temporary problem, try again
- `ENOMEM`: (UNIX) Out of memory
- `EBUSY`: (UNIX) Some necessary resource is already in use
- `EINVAL`: (UNIX) Invalid argument to some function

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc, char* argv[])
{
    char *res;

    printf("errno: %d\n", errno);
    res = strdup("Hallo"); /* Allocate space, copy the string to it */
    if(!res)
    {
        printf("Could not copy string, errno: %d = %d\n", errno, ENOMEM);
    }
    else
    {
        printf("All is fine, errno: %d\n", errno);
        free(res);
    }
    return EXIT_SUCCESS;
}
```

## Exercises

Write a program that sorts an arbitrary sized array of double values

Think about a program that sorts **pointers** to char, based on the characters (or character arrays) the pointers point to (yes, this is a hint for your assignment)

Check out `/usr/include/errno.h` and `/usr/include/asm/errno.h`

# CSC322

## C Programming and UNIX

### C Standard Library Characters and Strings

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## Character Classes and `<ctype.h>`

The C standard defines several character classes in a **portable** way

- We can use these functions regardless of the underlying character set of the implementation
- Most of these functions can be (and are) implemented in a very efficient manner for ASCII characters

C characters are integer values, typically 8 bits wide

- On most implementations, `char` is an 8 bit extension to ASCII (in recent time, `isolatein-1` or variants have become popular)
- There is limited support for bigger character sets using `wchar_t`

Character handling functions are defined in `<ctype.h>`

## Some C Character Classes

All character class functions accept and return `int` values

- Behaviour is only defined if the input is from the range of `unsigned char` or `EOF`
- Each function returns true (non-0) if the character is in the range, 0 otherwise

Character class test functions

- `isdigit(c)`: Digits, i.e. `{0-9}`
- `isalpha(c)`: Upper and lower case characters (`{a-z,A-Z}`, in some locales additional characters, e.g. umlauts like `ä, ö, . . .`)
- `isalnum(c)`: Equivalent to `(isdigit(c) || isalpha(c))`
- `iscntrl(c)`: Control characters, i.e. non-printable characters (in ASCII, those are characters with codes 0 to 31 and 127)
- `isxdigit(c)`: Hexadecimal digits, `{0-9,a-z,A-Z}`
- `islower(c)`: Lower case letters
- `isupper(c)`: Upper case letters
- `ispunct(c)`: Printing characters that are neither letters, digits, nor space
- `isprint(c)`: Normal, printable characters

# Character Class Conversion Functions

There are two functions for converting characters from one class to another:

- `tolower(c)` converts upper case characters to lower case characters
- `toupper(c)` converts lower case characters to upper case characters

Both functions return the character unchanged, if it is not a upper or lower case character, respectively

# Example

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(void)
{
    int c;

    while((c=getchar())!=EOF)
    {
        if(iscntrl(c))
        {
            printf("<control-character %d\n", c);
        }
        else
        {
            putchar(toupper(c));
        }
    }
    return EXIT_SUCCESS;
}
```

# Example Output

```
$ man man | ctypedemo
```

```
MAN(1)                MANUAL PAGER UTILS                MAN(1)<control-charac
ter 10>
<control-character 10>
<control-character 10>
<control-character 10>
N<control-character 8>
NA<control-character 8>
AM<control-character 8>
ME<control-character 8>
E<control-character 10>
    MAN - AN INTERFACE TO THE ON-LINE REFERENCE MANUALS<control-character 10>
<control-character 10>
S<control-character 8>
SY<control-character 8>
YN<control-character 8>
NO<control-character 8>
OP<control-character 8>
PS<control-character 8>
SI<control-character 8>
```

# Strings

Strings are not part of the C language proper

- String literals are supported
- Limited support by functions the C standard library

String-handling functions are operating on `char*` (pointer to char) values

- It is the responsibility of the program to make sure that there is sufficient space for the operations available!

Convention for strings:

- Strings are `\0` terminated arrays of character
- Important: Size of the array is **not** taken into account!

```
char excess[10000] = "a"; /* String length 1, takes up two
                           characters, a and \0 */
```

```
char tooshort[2];
tooshort[0] = 'a';
tooshort[1] = 'b'; /* tooshort is not a valid string, if treated
                    as one, behaviour is undefined */
```

## String Functions from <string.h> (1)

`char *strcpy(char* s, const char *ct)`

- Copy a `'\0'`-terminated string from `ct` to `s`
- Returns `s`
- `s` **must** point to a sufficiently large area of memory!
- Note: For all string functions that copy strings, source and target areas may not overlap (otherwise, behaviour is undefined)

`char *strncpy(char* s, const char *ct, size_t n)`

- As `strcpy()`, but copies at most `n` characters
- Note: If `ct` is longer than `n`, `s` will not be `'\0'`-terminated
- If `ct` is shorter than `n`, then the result will be padded with additional `'\0'` characters (i.e. `s` must always have space for `n` characters, even if `ct` is shorter than `n` characters)

`size_t strlen(const char *cs)`

- Return the length of the string at `cs`
- Does not count the trailing `'\0'`

## Example: Duplicating Strings

Several UNIX standards define a function `strdup()` that allocates enough memory for a string, and then copies it, returning the pointer to the newly allocated memory

Our version also makes sure that there is memory available:

```
char* SecureStrdup(char* str)
{
    char *newstr = SecureMalloc(strlen(str)+1);

    return strcpy(newstr,str);
}
```

## String Functions from <string.h> (2)

```
char *strcat(char *s, const char *ct)
```

- Concatenates ct at the end of s
- Returns s
- Result is always '\0' terminated

```
char *strncat(char *s, const char *ct, size_t n)
```

- As strcat(), but copies at most n characters from ct
- Result is always '\0' (even if ct is longer than n

Examples:

```
char *t="World";  
char s[10] = "Hello";  
strncat(s,t,3); /* Ok, t now points to "HelloWor" */  
strcat(s,t); /* Error: "HelloWorld" requires 11 character ('\0'!) */
```

## String Functions from <string.h> (3)

```
int strcmp(const char* cs, const char* ct)
```

- Compare two strings in the lexical extension of the natural order on characters
- First differing character decides which string is bigger (including terminating '\0', i.e. a substring is always smaller than a superstring)
- Return value: Integer <0, if cs is smaller, >0, if ct is smaller, or 0 if both are equal

```
int strncmp(const char* cs, const char* ct, size_t n)
```

- As strcmp(), but compare at most n characters

```
char *strchr(const char *s, int c)
```

- Return pointer to the first occurrence of c in cs (or NULL, if c is not present in cs)

```
char *strrchr(const char *s, int c)
```

- Return pointer to the **last** occurrence of c in cs (or NULL)

## String Functions from <string.h> (4)

`char *strpbrk(const char *cs, const char *ct)`

– Returns pointer to first character from `ct` in `cs` (or `NULL`), i.e. `ct` is treated as a **set** of characters

– Example:

```
strpbrk("Hello", "eul"); /* Returns pointer to the "e" in "Hello"
```

`char* strstr(const char *cs, const char *ct)`

– Return pointer to first occurrence of `ct` in `cs`, or `NULL` if `ct` is not a substring of `cs`

`char *strerror(int n)`

– Return a pointer to a string description of the library error with error code `n` (as defined in <errno.h>)

– If `n` is not a known error code, a pointer to a generic “unknown error code” message is returned

# Generic Memory Access Functions

The original C standard used `char*` as a generic pointer, hence generic memory handling functions are lumped in with strings

- Character is just another name for Byte in C, anyways
- However, ANSI C has the generic `void*` pointer type

The following functions are generally very similar to the string functions, but do not use a delimiter like `'\0'`

- All operations specify a length parameter `n`, and handle exactly `n` characters

These functions basically treat the virtual memory as one big character array!

- Used to implement many basic operations
- Typically implemented very efficiently (often by processor specific assembler subroutines)

## Memory Functions from `<string.h>` (1)

```
void *memcpy(void *s, const void *ct, size_t n)
```

- Copy a sequence of `n` bytes from `ct` to `s`
- The regions may not overlap!

```
void *memmove(void *s, const void *ct, size_t n)
```

- Copy a sequence of `n` bytes from `ct` to `s`
- There are no additional constraints (i.e. `memmove()` has to handle cases where the regions overlap)

```
int memcmp(const void *cs, const void *ct, size_t n)
```

- Compare the first `n` characters found at `cs` and `ct`
- Return value: As `strcmp()` (`<`, `>`, `= 0`)

## Memory Functions from <string.h> (2)

```
void *memchr(const char *s, int c, size_t n)
```

- Search for character `c` in the first `n` bytes at `cs`, return pointer to it (or `NULL`)

```
void *memset(void *s, int c, size_t n)
```

- Place character `c` into the first `n` characters at `s`, returning `s`

## Exercises

Write a simple version of `grep` (looking for plain strings in `stdin` only)

Write a version of `memmove()` (the hard part is handling overlapping arrays – remember that you can compare pointers with `<`, `>` and `==`)!

# CSC322

## C Programming and UNIX

### C Standard Library Memory Handling and IO

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Generic Memory Access Functions

The original C standard used `char*` as a generic pointer, hence generic memory handling functions are lumped in with strings

- Character is just another name for Byte in C, anyways
- However, ANSI C has the generic `void*` pointer type

The following functions are generally very similar to the string functions, but do not use a delimiter like `'\0'`

- All operations specify a length parameter `n`, and handle exactly `n` characters

These functions basically treat the virtual memory as one big character array!

- Used to implement many basic operations
- Typically implemented very efficiently (often by processor specific assembler subroutines)

## Memory Functions from <string.h> (1)

```
void *memcpy(void *s, const void *ct, size_t n)
```

- Copy a sequence of n bytes from ct to s
- The regions may not overlap!

```
void *memmove(void *s, const void *ct, size_t n)
```

- Copy a sequence of n bytes from ct to s
- There are no additional constraints (i.e. memmove() has to handle cases where the regions overlap)

```
int memcmp(const void *cs, const void *ct, size_t n)
```

- Compare the first n characters found at cs and ct
- Return value: As strcmp() (<, >, = 0)

## Memory Functions from <string.h> (2)

```
void *memchr(const char *s, int c, size_t n)
```

- Search for character `c` in the first `n` bytes at `cs`, return pointer to it (or `NULL`)

```
void *memset(void *s, int c, size_t n)
```

- Place character `c` into the first `n` characters at `s`, returning `s`

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[])
{
    char carray[10];
    int iarray[10], i;
    memset(&carray[0], 'a', 10*sizeof(char));
    memset(&iarray[0], 'a', 10*sizeof(int));
    for(i=0; i<10; i++)
    {
        printf("%c : %d\n", carray[i], iarray[i]);
    }
    memset(&carray[0], 'b', 10*sizeof(char));
    memmove(&iarray[0], &carray[0], 10*sizeof(char));
    for(i=0; i<10; i++)
    {
        printf("%c : %d\n", carray[i], iarray[i]);
    }
    return EXIT_SUCCESS;
}
```

# Example Output

a : 1633771873  
b : 1650614882  
b : 1650614882  
b : 1633772130  
b : 1633771873  
b : 1633771873

# Input and Output in the Standard Library

Input and output in C is based on the concept of **streams** of bytes

- Binary streams are raw, unprocessed bytes (only guarantee: If you write data to a binary stream, and then read it back, it is unchanged)
- Text streams are composed of (possibly empty) lines, separated by a single newline (`'\n'`) character (the library has to make sure other text representations are converted properly)
- In UNIX, text and binary streams are identical
- In Windows, the library has to convert the newline/linefeed sequence used to separate lines to a single newline for text streams (but, of course, may not mangle binary streams)

Streams are represented by `FILE*` objects in C (“file pointers”)

- The `FILE` type is defined in `<stdio.h>`
- A stream normally has to be explicitly opened (connected to an input and output device) and should be closed (made available for reuse)

# Standard Streams

By default, each program has three text streams open on startup:

- `stdin` is the standard input (normally reading from keyboard)
- `stdout` is the standard output (normally connected to the terminal)
- `stderr` is the standard error channel (also connected to the terminal)

The I/O-functions we have used so far implicitly use the default streams:

- `printf()` and `putchar()` write to `stdout`
- `getchar()` reads from `stdin`

# Opening File Streams

In addition to the standard streams, we can create additional streams, normally associated with a **file**. The most general function is:

```
FILE* fopen(const char *filename, const char *mode)
```

- The first argument has to be a valid filename
- The second argument describes the mode in which the file should be opened

The **mode** is a string of characters

- "r" opens a file for reading in text mode
- "w" opens a file for writing in text mode (will create new file, overwriting an existing file)
- "a" opens a file for writing in text mode (but will append new output to the end of an existing file)
- Adding a "b" will open the file as a binary file (e.g. "rb": Read binary)

`fopen()` returns a valid file pointer, if successful, or NULL if it fails

- In the case of failure, it sets `errno` to an appropriate value!

# Closing and Reopening File Streams

Once we are done with a certain file, we have to **close** it

- The number of simultaneously open files is limited for most operating systems. Closing a stream makes it available for other purposes
- Streams may be **buffered**. Closing a stream flushes the buffer (i.e. prints all remaining characters)

`int fclose(FILE *stream)` closes the file associated with stream

- It returns 0 if no errors occurred, EOF otherwise

`FILE* freopen(const char *filenam, const char *mode, FILE *stream)`

- This function closes `stream` and reopens it with a new associated file
- It is useful to e.g. redirect `stdin` into a file (from within the program)

## Simple Stream Based I/O Functions (Characters)

```
int fgetc(FILE *stream)
```

- Return the next character from the named stream (or EOF if no character is available or an error occurs)
- Note: `getchar()` is equivalent to `fgetc(stdin)`

```
int fputc(int c, FILE *stream)
```

- Print the character `c` to the stream, returning `c` or EOF in case of error
- `putchar(c)` is equivalent to `fputc(c, stdout)`

`int getc(FILE *stream)` is equivalent to `fgetc()`, except that it may be implemented as a macro (and may hence evaluate `stream` more than once)

Similarly, `int putc(int c, FILE *stream)` is equivalent to `fputc`, but may be a macro

## Simple Stream Based I/O Functions (Strings)

```
int fputs(const char *s, FILE *stream)
```

- Writes the string pointed to by the first argument to the denoted stream
- Returns EOF on failure, a non-negative value otherwise

```
char *fgets(char *s, int n, FILE *stream)
```

- Read at most  $n-1$  characters into the array pointed to by  $s$ , stops early if a newline is encountered
- $*s$  is always `'\0'` terminated
- Returns  $s$ , or `NULL` on error

Note: There also is a function `char *gets(char *s)` that attempts to read a line of input from `stdin`

- **Never use `gets()`!**
- Since there is no way to specify a maximal number of characters to read, we cannot ensure that `gets()` will not result in a buffer overflow error!

## Example: Simple cat Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <assert.h>

void print_file(FILE *stream)
{
    int c;

    while((c=fgetc(stream))!=EOF)
    {
        fputc(c, stdout);
    }
}
```

## Example Continued

```
int main(int argc, char *argv[])
{
    int i;
    FILE *file;

    if(argc == 1)
    {
        print_file(stdin);
    }
    else
    {
```

## Example Continued

```
for(i=1; i<argc; i++)
{
    file = fopen(argv[i], "rb");
    if(!file)
    {
        int errno_safe = errno;

        assert(errno);
        fputs(argv[0],stderr); /* Print program name */
        fputs(": ", stderr);
        fputs(strerror(errno_safe), stderr);
        fputc('\n', stderr);
        return EXIT_FAILURE;
    }
    print_file(file);
    fclose(file); /* Assuming it works... */
}
return EXIT_SUCCESS;
}
```

# Example Output

```
$ man man | ./mycat1
```

```
man(1)
```

```
man(1)
```

```
NAME
```

```
man - format and display the on-line manual pages
```

```
manpath - determine user's search path for man pages
```

```
...
```

```
$ ./mycat1 does_not_exist
```

```
./mycat1: No such file or directory
```

```
$ ./mycat1 mycat1.c
```

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

```
#include <stdlib.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```
#include <assert.h>
```

```
void print_file(FILE *stream)
```

## Exercises

Write a version of `memmove()` using pointer assignment (the hard part is handling overlapping arrays – remember that you can compare pointers with `<`, `>` and `==`)! You may need to cast `void*` to `char*` to access individual bytes.

Write a version of `wc` that more closely mimics the behaviour of the UNIX version, i.e. that gives separate accounts and a total if called with more than one argument (if called with a single arguments, it just gives an account for that file, if called with none, it reads from `stdin`)

# CSC322

## C Programming and UNIX

### C Standard Library Input and Output

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## Remark about `fgets()`

```
char *fgets(char *s, int n, FILE *stream)
```

- Read at most `n-1` characters into the array pointed to by `s`, stops early if a newline is encountered
- `*s` is always `'\0'` terminated
- Returns `s`, or `NULL` on error

### Note:

- It is the responsibility of the **caller** (i.e. your program) to provide enough memory!
- `s` already has to point to an array (or `malloc()`ed area of sufficient size

This holds for most standard library functions!

- . . . including `gets()` (never use `gets()`!)

## Example: Simple cat Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <assert.h>

void print_file(FILE *stream)
{
    int c;

    while((c=fgetc(stream))!=EOF)
    {
        fputc(c, stdout);
    }
}
```

## Example Continued

```
int main(int argc, char *argv[])
{
    int i;
    FILE *file;

    if(argc == 1)
    {
        print_file(stdin);
    }
    else
    {
```

## Example Continued

```
for(i=1; i<argc; i++)
{
    file = fopen(argv[i], "rb");
    if(!file)
    {
        int errno_safe = errno;

        assert(errno);
        fputs(argv[0],stderr); /* Print program name */
        fputs(": ", stderr);
        fputs(strerror(errno_safe), stderr);
        fputc('\n', stderr);
        return EXIT_FAILURE;
    }
    print_file(file);
    fclose(file); /* Assuming it works... */
}
return EXIT_SUCCESS;
}
```

# Example Output

```
$ man man | ./mycat1
```

```
man(1)
```

```
man(1)
```

```
NAME
```

```
man - format and display the on-line manual pages
```

```
manpath - determine user's search path for man pages
```

```
...
```

```
$ ./mycat1 does_not_exist
```

```
./mycat1: No such file or directory
```

```
$ ./mycat1 mycat1.c
```

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

```
#include <stdlib.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```
#include <assert.h>
```

```
void print_file(FILE *stream)
```

## Reminder: Using `stdin`

You can redirect files into `stdin`:

- `mycat1 < mycat.c`

You can type into `stdin` from your terminal

- Type `[C-d]` (`^d`), "Control-D" to indicate **end of input**
- Depending on your version of UNIX and your terminal, you may have to type `[C-d]` on a line of it's own

# Buffering and Flushing

Both input and output streams can be buffered

- Unbuffered streams will pass on each individual character as soon as possible
- Fully buffered streams will wait until the (arbitrary sized) buffer is full until they pass on the collected data as one chunk
- Text streams can also be **line buffered**. A line buffered stream will collect at most one line of data

`int fflush(FILE* stream)` will **flush** all buffers associated with an output stream

- Causes data to be actually written (if the writing process dies, the data is safe), although the OS may still have another layer of buffers
- Return value: 0 on success, EOF on failure
- Calling `fflush(NULL)` flushes **all** open streams
- Calling `fflush(NULL)` on an input stream invokes undefined behaviour

# Buffering

By default, the standard streams are buffered as follows:

- `stdin` is line buffered
- `stdout` is line buffered
- `stderr` is unbuffered

You can change the buffering state with the function

```
int setvbuf(FILE *stream, char* buff, int mode, size_t size)
```

- `buff` points to a buffer of at least `size` byte (or it is `NULL`, in which case a buffer will be `malloc()`ed)
- Mode can be one of three predefined values:
  - \* `_IOFBF` for full buffering
  - \* `_IONBF` to disable buffering
  - \* `_IOLBF` to enable line buffering

`void setbuf(FILE *stream, char *buff)` is a simpler interface:

- If `buff` is zero, buffering is switched off
- Otherwise, full buffering with a buffer size `BUFSIZ` is enabled (and `buff` has to point to a large enough buffer!)

## Example

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

int main(int argc, char* argv[])
{
    char name[80];
    char buffer[BUFSIZ];

    setbuf(stdout, buffer);
    printf("Please enter name: ");
    fgets(name,80,stdin);
    printf("Your name is: %s\n", name);

    setbuf(stdout, NULL);
    printf("Please enter name: ");
    fgets(name,80,stdin);
    printf("Your name is: %s\n", name);

    return EXIT_SUCCESS;
}
```

## Example Behaviour

```
$ ./bufftest
```

```
Stephan
```

```
Please enter name: Your name is: Stephan
```

```
Please enter name: Schulz
```

```
Your name is: Schulz
```

## More Operations on Files

```
int remove(const char *filename)
```

- Removes a file (as in `rm`)
- Return 0 on success, something else on failure

```
int rename(const char *oldname, const char *newname)
```

- Rename a file (as in `mv`)
- Return 0 on success, something else on failure

```
FILE *tmpfile(void)
```

- Creates a temporary file with mode `wb+` (reading and writing in binary)
- The file will vanish if the program terminates normally
- On failure, `NULL` will be returned

## Even More File Operations

```
char *tmpnam(char *s)
```

- Creates a file name that is different from any existing name
- If called with argument `NULL`, will return a pointer to a static buffer containing the name
- Otherwise, `s` has to point to an array of at least `L_tmpnam` bytes
- Note: Using `tmpnam()` in security-critical applications is discouraged, as it creates a **race** condition (what if another process creates a file with the name in between the call to `tmpnam()` and `fopen()`?)

# Error Functions

Each FILE data structure stores two pieces of information:

- If end-of-file has been reached during reading
- If an error occurred

`int feof(FILE *stream)` returns true if the end-of-file indicator has been set

`int ferror(FILE *stream)` returns true if the error indicator is set

`void clearerr(FILE *stream)` clears both indicators

`void perror(const char *s)` prints an error message to `stderr` as follows:

- First, the supplied string is printed, followed by a colon
- Then the error message for the current value of `errno` is printed, followed by a newline

# Exercises

Write a simple database that keeps given name, family name, and date of birth for a person. Subtasks:

- Create a dialog where people can enter data
- Create an interface for searching for data, based on any criterium
- Create an interface where you can print lists of people, possibly sorted by any of the data fields

You need to think about the data base structure (a flat text file should work, see e.g. `/etc/passwd` for ideas)

You need an architecture for your overall program

- The conventional way is to use one monolithic program with a a menue structure (use text menus...)
- The UNIX way would be to write one program for each task

# CSC322

## C Programming and UNIX

### C Standard Library Formatted Output

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Formatted Output

The formatted output functions offer a very convenient way of printing data in a controlled manner

- They are able to print all basic C datatypes (and strings)
- They can print any number of arguments with one command
- For most datatypes, there are multiple useful formats
- Argument output and descriptive strings can be interspersed easily

Output format is determined by a **format** string argument

- The format string contains ordinary text that is copied directly to the output
- It also contains **conversion specifiers** that describe how to format additional arguments

Formatted output functions are **variadic**, i.e. they take a variable number of arguments

- Number of arguments is determined by the number of conversion specifiers
- Modern compilers check this property if the format string is constant

## A first Example

```
printf("%d divided by %d = %f\n",22,7,22/7.0);
```

- The first argument to printf is the **format string**
- It contains 3 conversion specifiers:
  - \* The first %d specifies an **int** argument that should be printed in decimal notation and corresponds to the first extra argument, 22
  - \* The second %d corresponds to the third argument, 7
  - \* Finally, the %f specifies a **double** (floating point) argument that should be printed in pure decimal notation (with fractional part after the decimal dot)

The format string also contains additional text

- Text is printed
- Note that normal conventions hold, i.e. `\n` in a string literal is the newline character

Output printed:

```
22 divided by 7 = 3.142857
```

# The printf() Family of Functions

All functions are declared in `<stdio.h>`

```
int printf(char *format, ...);
```

- Print the additional arguments under control of the argument string to `stdout`
- Returns number of characters printed, or any negative number on error

```
int fprintf(FILE *stream, char *format, ...);
```

- As `printf()`, but print to the designated output stream

```
int sprintf(char *s, char *format, ...);
```

- Instead of actually printing anything, `sprintf()` will store the output characters in the character array `s` points to
- The string will be `\0` terminated
- It is the responsibility of the programmer to make sure `*s` is big enough
- The returned count of characters does not include the terminating `null` character (i.e. it is the same value that `printf()` would return)

# Format Specifiers

Format specifiers always start with a % character, and end in a **conversion letter**

- The conversion letter describes the basic output format
- It normally also describes which kind of argument has to follow

Optional parts of a format specifier include (in order)

- Flags (affect how the result will be printed)
- Minimum field width (if fewer characters are necessary, **padding** will be used)
- Precision (number of significant digits/characters)
- Size modifier (e.g. require `short` or `long` instead of `int`)

## Some Conversion Letters (1)

d: Convert an `int` argument and print it in decimal representation

i: Alias for d

u: Convert an `unsigned int` argument and print it in decimal representation

o: Convert an `unsigned int` argument and print it in **octal** representation

x: Convert an `unsigned int` argument and print it in **hexadecimal** representation, using {a, b, c, d, e, f} for the extra hexadecimal digits

X: As x, but use upper case hex digits ({A, B, C, D, E, F})

p: Convert a `void*` pointer and print it in an implementation-defined manner (for our system, and for many other systems, the argument is printed as a hexadecimal number representing the address)

## Some Conversion Letters (2)

`f`: Print a `double` argument (`float` is converted automatically) as a sequence of digits with a decimal point

– Unless otherwise specified via the `precision` modifier, 6 digits are printed after the decimal point

`e`: Print a `double` argument in normalized exponential form, with 1 digit before the decimal dot (and by default 6 digits after the dot). Example: `3.141593e+01` ( $= 3.141593 * 10^1$ )

`E`: As `e`, but print upper case `E` before exponent

`g`: “Human-friendly floating point output”. Print a `double` number either as with `e` (for very small numbers) or with `f` letters, cutting off unnecessary trailing zeros

`G`: As `g`, but use `E` instead of `e`

## Some Conversion Letters (3)

`c`: Print a single `int` argument by converting it to `char` and printing the corresponding **character** (use `%i` to print the numeric value of a character)

`s`: Print a C style string, converting a `char*` argument pointing to a `\0`-terminated string

`%`: Convert no arguments, just print a single `%` character (i.e. `%%` in the format string generates a single `%` in the output)

Remarks:

- We have not covered some of the more esoteric conversions
- The 1995 addendum to C89 and the C99 standard add additional conversion characters
- For more details, check `man 3 printf`

## Example

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

int main(void)
{
    char *p = "This is a string";

    printf("12 with... %%d: %d, %%u: %u, %%o: %o, %%x: %x, %%X: %X\n",
           12, 12, 12, 12, 12);

    printf("12.5 with... %%f: %f, %%e: %e, %%E: %E, \n%%g: %g, %%G: %G\n",
           12.5,12.5,12.5,12.5,12.5);

    printf("Printing a character with %%c: %c and %%d: %d\n",
           'a', 'a');

    printf("This is a string \"%s\" and its address: %p\n", p,p);

    return EXIT_SUCCESS;
}
```

## Example Output

```
12 with... %d: 12, %u: 12, %o: 14, %x: c, %X: C
```

```
12.5 with... %f: 12.500000, %e: 1.250000e+01, %E: 1.250000E+01,  
%g: 12.5, %G: 12.5
```

```
Printing a character with %c: a and %d: 97
```

```
This is a string "This is a string" and its address: 0x80485a0
```

# Size Modifiers

Size modifiers are used to change the default argument size:

- The `l` modifier changes integer arguments to their long variants
- It changes `h` modifier indicates that the argument is of type `short` or `unsigned short` instead of the default `int`

The C99 standard introduces additional size modifiers:

- `z` indicates argument of type `size_t` (for integer arguments)
- `ll` indicates long long versions of the integers
- `hh` indicates `char` arguments instead of `int` types

For us, the `%ld` version (long integer) is probably the most important one

## Specifying Minimum Field Width

The minimum field width is an integer literal between the % and the conversion letter (with optional size modifier)

- It may be preceded by any flags
- The precision, if any, follows it

By default, any value is printed **right-justified** in its field

- Padding is done with spaces:

```
printf("|%7d|\n", 12);  
|    12|
```

If the natural value representation is bigger than the minimum field width, the specification has no effect

```
printf("|%7s|\n", "A long string");  
|A long string|
```

# The Flags

- : Left-justify output (only useful in connection with a width specification)
- 0: Use 0 for padding to requested field width (by default, ' ' (space) is used)
- +: For numerical values: Always print a sign, either + or -
- ' ' (space): Always print a character for the sign, - for negative numbers, ' ' for positive ones
- #: Use a variant of the conversion operation
  - For %o, print a leading 0
  - For %x, print a leading 0x
  - For %X, print a leading 0X
  - For floating point numbers, trailing digits and decimal dot are always printed with the # flag

# The Precision

The precision field is used for a number of different things:

- For any integer conversion character, it gives a minimum number of digits to print (by adding leading zeros)
- For %e, %E and %f, it gives the number of digits in the fractional part
- For %g and %G, it is the number of significant digits to be printed
- Finally, for strings (%s), it gives the maximal number of characters to be printed from the string

# Example

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

int main(void)
{
    printf("Floating point example: |%+8.2f|\n", 3.0/7.0);
    printf("Floating point example: |% 8.2f|\n", 3.0/7.0);
    printf("String: %-7.7s\n", "Longish String");
    printf("String: %-7.7s\n", "short");
    printf("String: %7.7s\n", "short");

    return EXIT_SUCCESS;
}
```

Output:

```
Floating point example: |  +0.43|
Floating point example: |   0.43|
String: Longish
String: short
String:  short
```

# Assignment

Write an `archiver` program `arch322`. Your program should accept any number of arguments (to be treated as filenames). It should write (to `stdout`) an archive, i.e. a file that contains enough information to recreate the original files with their names. For simplicity, allow only files in the current directory to be archived (check, if the arguments contain a `/` and print an error message if yes). Also print useful error messages if one of the named files does not exist, etc.

Write a `dearchiver` program `dearch322` that accepts an archive file (in your format) on `stdin` and recreates the original files in the current directory. Print an error message if the file is not a valid archive.

You are free to design your own archive format, but you may get some ideas from reading the documentation (`man/info`) on `tar/gtar`. Please document your format in one or two paragraphs. You may assume UNIX I/O, i.e. no difference between text and binary I/O.

Example:

```
$ arch322 Makefile sort_csc322.c utilities.c > myarch.arch
$ mkdir NEW
$ cd NEW
$ dearch322 < ../myarch.arch
Recreating Makefile
Recreating sort_csc322.c
Recreating utilities.c
$ ls
$
Makefile sort_csc322.c utilities.c
```

**CSC322**  
**C Programming and UNIX**  
**Asynchronous Events and Signals**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Processes

A UNIX **process** is an instance of a program in execution. It can be described by

- The executable code (stored in the **text segment** of the virtual memory image of the process)
- The program data (stored in the data segment)
- The state, including stack pointer and stack, program counter, etc. (usually collected in a process control block, or PCB)

A process uses certain resources:

- Processor time on a CPU
- Memory, both virtual or real
- File descriptors
- . . . .

Some of its important properties are

- Owner
- Process id (pid), a unique non-negative integer
- Parent (exception: `init`)

# UNIX User Commands: ps

Usage: **ps** <complicated options>

- **ps** shows information about currently executing processes
- It is one of the least standardized UNIX tools

Our Linux **ps** can assume many different **personalities**

- Different personalities show different behaviour
- . . . and accept different options.

Default behaviour (**ps** without options):

- Show information about all existing processes of the current user controlled by the same terminal **ps** was run on
- For each process, list:
  - \* Process Id (PID)
  - \* Controlling terminal (TTY)
  - \* CPU time used by the process
  - \* Name of the executable program file

# Vanilla ps Example

\$ ps

PID	TTY	TIME	CMD
1125	pts/3	00:00:01	tcsh
7157	pts/3	00:00:00	xevil
7189	pts/3	00:00:00	gv
7193	pts/3	00:00:00	gs
7194	pts/3	00:00:00	ps

## Some ps Options

Some simple BSD style options for the default personality (note: BSD style options for **ps** are **not** preceded by a dash!)

- **a**: Print information about **all** processes that are connected to any terminal
- **x**: Print information about processes **not** connected to a terminal
- **U <username>**: Print information about processes owned by the named user
- **u**: User oriented output with more interesting information:
  - \* Owner of a process (USER)
  - \* Process Id (PID)
  - \* Percentage of available CPU used by the process (%CPU)
  - \* Percentage of memory used (%MEM) (note that this measures virtual memory usage, real memory usage may be lower because of shared pages)
  - \* Virtual memory size of the process in KByte (VSZ)
  - \* Size of the **resident set**, i.e. the recently referenced pages not swapped out (RSS)
  - \* Controlling terminal (TTY)
  - \* Time or date when the process was started (START)
  - \* Seconds of CPU time used (TIME)
  - \* Full command used to start the process (COMMAND)

## Interesting ps Example

\$ ps aux

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	1368	432	?	S	Oct30	0:04	init
root	2	0.0	0.0	0	0	?	SW	Oct30	0:03	[keventd]
root	3	0.0	0.0	0	0	?	SW	Oct30	0:00	[kapmd]
...										
root	486	0.0	0.1	1372	408	?	S	Oct30	0:00	/sbin/dhcpd -n -h wo
root	551	0.0	0.2	1644	668	?	S	Oct30	0:00	syslogd -m 0
...										
schulz	1095	0.0	4.8	16112	12268	?	S	Oct30	4:40	emacs -geometry 96x77
schulz	1096	0.0	0.8	4944	2216	?	S	Oct30	0:05	xterm -geometry 80x40
schulz	1997	0.0	0.5	3072	1476	?	S	Oct31	0:12	ssh sherman emacs
root	4073	0.0	1.0	7480	2768	pts/3	S	Oct31	0:03	/usr/local/lib/xmcd/b
schulz	22637	0.0	0.5	2940	1444	pts/5	S	Nov05	0:03	ssh -X sunbroy2.infor
schulz	22645	4.0	18.7	82248	47832	?	S	Nov05	31:04	/usr/local/mozilla/mc
schulz	6722	0.0	0.0	0	0	?	Z	Nov05	0:00	[plugger <defunct>]
schulz	7189	0.0	0.8	3948	2220	pts/3	S	00:15	0:00	gv CSC322_1.pdf
schulz	7235	0.4	2.2	10060	5668	pts/3	S	00:41	0:00	gs -dNOPLATFONTS -sDE
schulz	7236	76.8	38.0	98072	96896	pts/0	R	00:43	0:33	eprover /home/schulz/
news	7237	0.5	1.0	3704	2796	?	S	00:43	0:00	leafnode
schulz	7258	0.0	0.2	2624	708	pts/3	R	00:43	0:00	ps aux

**CSC322**  
**C Programming and UNIX**  
**Signals and Signal Handlers**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Signals

**Signals** are a way to signal unusual events to a process

- Run time errors
- User requests
- Pending communication

In general, signals can arrive asynchronously, i.e. at any time

Signals can have many different values, depending on the value, the process can

- Ignore a signal
- Perform a **default action** (defined by the implementation)
- Invoke an explicit signal handler

# Standard C Signals

Standard C defines a small number of signals, UNIX defines many more

Signal	Meaning	Default Action (UNIX)
SIGABRT	Abort the process	Terminate
SIGFPE	Floating point exception	Terminate with core
SIGILL	Illegal instruction	Terminate with core
SIGINT	Interactive interrupt	Terminate
SIGSEGV	Illegal memory access	Terminate with core
SIGTERM	Termination request	Terminate

Note: SIGINT is generated when you press [CTRL-C]!

- The signal is delivered to the process
- The default action is to terminate the process

## Some UNIX Signals

UNIX defines about 60 different signals, including all Standard C signals

Some important UNIX signals:

Signal	Meaning	Default Action (UNIX)
SIGHUP	Terminal connection lost (or controlling process dies)	Terminate
SIGKILL	Kill process, cannot be caught or ignored	Terminate
SIGBUS	Bus error	Terminate with core
SIGSTOP	Stop a process (does not terminate, cannot be caught or ignored)	Suspends process
SIGCONT	Continue suspended process	Ignored (*)
SIGURG	Out of band data arrived on a socket	Ignore
SIGXCPU	CPU time limit reached	Terminate with core

(\*) OS will still wake process up

[CTRL-Z] generates SIGSTOP!

## UNIX User Command: `kill`

Note: `kill` is often implemented as a shell built-in

- Syntax may differ slightly from the `kill` program
- Allows use of `kill` in job control

Usage for our `kill`: `kill [-<SIG>] <pid> ...`

- If no signal is specified, `SIGTERM` is sent
- Signals can be specified symbolically (for a list of names run `kill -l`) or numerically (`man 7 signal` gives a list of signals and their numeric values)

`kill` accepts a list of `<pid>` arguments

- Most common case: `<pid>` is a normal process id (a positive integer). The signal is sent to the corresponding process
- If `<pid>` is `-1`, the signal is sent to all processes of the user (`kill -KILL -1` is a surefire way to log yourself out)
- Finally, if `<pid>` is any other negative number, the signal is sent to the corresponding process group

## UNIX User Commands: top

top is an interactive version of ps

- It shows various information about the top processes currently running
- Also shows general system information
- All information is periodically updated
- top seems to be more consistent between different UNIX dialects, and is often preferred for interactive use (or even for scripting)

top also can be used to send signals to processes

- Press [k] and then specify process and signal

Non-interactive use of top (“better ps”):

- `top -b -n1` will print a single page in a **ps**-like manner

For more information: `man top` or run top and hit [h] for help

## top Example

11:09pm up 8 days, 1:15, 7 users, load average: 0.59, 0.21, 0.07  
78 processes: 71 sleeping, 4 running, 3 zombie, 0 stopped  
CPU states: 95.2% user, 4.7% system, 0.0% nice, 0.0% idle  
Mem: 254576K av, 249892K used, 4684K free, 0K shrd, 7428K buff  
Swap: 522072K av, 30888K used, 491184K free 68440K cached

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	COMMAND
12692	schulz	25	0	25548	24M	664	R	89.3	10.0	0:08	eprover
1040	root	15	0	89416	15M	5424	S	5.5	6.4	919:35	X
1097	schulz	15	0	2324	2124	1676	S	3.7	0.8	0:15	xterm
12693	schulz	16	0	924	924	728	R	1.1	0.3	0:00	top
1096	schulz	15	0	2512	2252	1708	R	0.1	0.8	0:07	xterm
1	root	15	0	472	432	416	S	0.0	0.1	0:04	init
2	root	15	0	0	0	0	SW	0.0	0.0	0:04	keventd
3	root	15	0	0	0	0	SW	0.0	0.0	0:00	kapmd
4	root	34	19	0	0	0	SWN	0.0	0.0	0:00	ksoftirqd_CPU0
5	root	15	0	0	0	0	SW	0.0	0.0	0:09	kswapd
6	root	15	0	0	0	0	SW	0.0	0.0	0:00	bdflush
7	root	15	0	0	0	0	SW	0.0	0.0	0:00	kupdated
8	root	25	0	0	0	0	SW	0.0	0.0	0:00	mdrecoveryd
12	root	15	0	0	0	0	SW	0.0	0.0	0:01	kjournald

...

# Catching Signals

User programs can set up a **signal handler** to catch signals

- A signal handler is a normal function
- It has to be explicitly set up for each signal type
- It will be called asynchronously when a signal of the correct type has been caught
- When the signal handler returns, the program will resume execution at the old spot

UNIX implements several different ways of handling signals, we will concentrate on the ANSI C signal handling

- All use the same signal: Signals are small integers
- However, for all existing signals, we use the `#defined` name showed above (`SIGHUP...`)

Signal handling stuff is defined in `<signal.h>`

## ANSI C Signal Handling with `signal.h`

`signal.h` defines the `signal()` function for establishing signal handlers as follows:

```
void (*signal(int sig, void (*handler)(int)))(int)
```

**Huh?**

## ANSI C Signal Handling with `signal.h`

`signal.h` defines the `signal()` function for establishing signal handlers as follows:

```
void (*signal(int ig, void (*handler)(int)))(int)
```

We can break this definition up as follows:

```
typedef void (*SigHandler)(int);
```

```
SigHandler signal(int sig, SigHandler handler);
```

- The first argument to `signal()` is the signal to be caught
- The second argument is a pointer to the new **signal handler**
- Return value is a pointer to the old signal handler for that signal (or `SIG_ERR` if no signal handler could be established)

Predefined (pseudo) signal handlers (possible arguments to `signal()`):

- `SIG_DFL`: Revert to the default behaviour for that signal
- `SIG_IGN`: Ignore the signal from now on

## ANSI C Signal Handling (Continued)

Additional definitions in `signal.h`:

`sig_atomic_t` is an integer type

- We are guaranteed that an assignment to a variable of this type is atomic, i.e. will not be interrupted by e.g. another signal
- That means that its value will always be well-defined

`int raise(int sig)` raises a signal to the program

- Return value: 0 on success, something else otherwise

# ANSI C Signal Handlers

A signal handler is a function that returns nothing and gets the signal that was caught as an argument

There are several limitations on signal handler:

- Since signals can arrive asynchronously, the state of the program is not well-defined!
- Signals may be handled even within a single C statement
- Therefore a signal handler cannot make many assumptions about the state of the program
- For maximum portability, a signal handler should only
  - \* Reestablish itself by calling `signal()`
  - \* Assigning a value to a variable of type `volatile sig_atomic_t`
  - \* Return or terminate the program (e.g. calling `exit()`)

Once a signal has been caught, the signal handler for that signal is reset to default behaviour

- If you want to catch multiple signals, the signal handler has to reestablish itself

## Common UNIX functions: sleep()

Often, a program only has to perform task only occasionally, or it has to wait for a certain event to happen. ANSI C has no way of delaying a program

- Old-style home computer programmers use **busy delay loop**
- However, those are unacceptable on multi-user systems
- Moreover, they can usually be optimized away by a good compiler

All UNIX versions address this problem with the `sleep()` function (normally defined in `<unistd.h>`):

```
unsigned int sleep(unsigned int seconds);
```

`sleep()` makes the current process sleep (do nothing ;- ) until either

- (At least) `seconds` seconds have elapsed or
- A non-ignored signal arrives

Return value:

- 0 if sleep terminated because of elapsed time
- Number of seconds left when the process was awakened by a signal

## Example: Counting Signals (Fluff)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <assert.h>

typedef void (*SigHandler)(int);
volatile sig_atomic_t sig_int_flag = 0;
volatile sig_atomic_t sig_term_flag = 0;

void EstablishSignal(int sig, SigHandler handler)
{
    SigHandler res;

    res = signal(sig, handler);
    if(res == SIG_ERR)
    {
        perror("Could not establish signal handler");
        exit(EXIT_FAILURE);
    }
}
```

## Example: Counting Signals (The Signal Handlers)

```
void sig_int_handler(int sig)
{
    EstablishSignal(SIGINT, sig_int_handler);

    assert(sig == SIGINT);
    printf("Caught SIGINT!\n"); /* Risky */
    sig_int_flag = 1;
}

void sig_term_handler(int sig)
{
    EstablishSignal(SIGTERM, sig_term_handler);
    assert(sig == SIGTERM);
    printf("Caught SIGTERM!\n"); /* Risky! */
    sig_term_flag = 1;
}
```

## Example: Counting Signals (Main)

```
int main(int argc, char* argv[])
{
    int i;
    int int_counter = 0;

    EstablishSignal(SIGTERM, sig_term_handler);
    EstablishSignal(SIGINT, sig_int_handler);

    for(i=0; i<1000 && !sig_term_flag; i++)
    {
        printf("Going to sleep!\n");
        sleep(30);
        if(sig_int_flag)
        {
            sig_int_flag = 0;
            int_counter++;
        }
    }
    printf("SIGINTs: %d  Iterations:%d\n", int_counter, i);
    return EXIT_SUCCESS;
}
```

## Example Session: Live

(Change to Terminal!)

## Exercises

Start a long running process (e.g. `top`) in one xterm

Send it various signals and see how it behaves

Read `man 7 signal`, `man kill` and `man 2 kill`

Download the source to the signal handler example and play with it

- Send different signals to it
- Add your own signal handler
- Write a generic signal handler function that catches more than one signal (and works correctly for multiple signals)

**CSC322**  
**C Programming and UNIX**  
**The UNIX File System**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# UNIX File System

UNIX philosophy: **Everything is a file**

- Plain files
- Hardware devices (Keyboard, mouse, hard drives)
- Network connections

Consequently, UNIX specifies a lot more properties and has more ways of manipulating a file than ANSI C

- Low-level IO
- File access rights
- Different file types

Note: These are not ANSI C features

- We have to call **gcc** **without** the `-ansi` option to use most of these features (otherwise, most UNIX extensions are disabled)

# UNIX File Types (1)

## Regular files

- Boring old data file (most common type of file)
- UNIX does not care **what** is inside that file

## Directories

- Stores names and pointers to more information
- **Write** access is limited to kernel file system functions to assure the integrity of the file system

## Character special files

- Represent hardware devices that generate individual characters (/dev/kbd, /dev/mouse)

## Block special files

- Represent hardware where data is available in fixed-size blocks (e.g. hard drives, /dev/hda in Linux)

# UNIX File Types (2)

FIFOs (named pipes)

- Special files used for interprocess communication

Sockets

- Special files used for network communication (or local interprocess communication)
- Not available in all UNIX versions (some don't represent network connections as files in the file system)

Symbolic links

- A symbolic link is a file containing just a file name
- The kernel normally automatically redirects any access to the link to the named file

## The stat() Functions

The three functions in the stat family all allow us to extract information about a file

- Who owns it
- How big is it
- What kind of file is it
- ...

They are specified as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

## The stat() Functions (2)

All three functions perform the same basic function:

- Write information about a file into the structure `buf` points to (and which we have to provide)
- Return 0 if the operation was possible, -1 otherwise (in which case they also set `errno`)

Differences:

- `fstat()` accepts a low level **file descriptor** referring to an open file
- `lstat()` will not follow symbolic links, but give information about the link itself (`stat()` given information about the file pointed to)

How exactly `struct stat` is defined may differ

- It always contains certain standard members

## The stat() Functions (3)

```
struct stat {
    dev_t      st_dev;      /* device number*/
    dev_t      st_rdev;     /* device type (if inode device) */
    ino_t      st_ino;     /* inode number */
    mode_t     st_mode;     /* access rights and file type */
    nlink_t    st_nlink;   /* number of hard links */
    uid_t      st_uid;     /* user ID of owner */
    gid_t      st_gid;     /* group ID of owner */
    off_t      st_size;    /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;   /* time of last access */
    time_t     st_mtime;   /* time of last modification */
    time_t     st_ctime;   /* time of last change */
};
```

Interpretation of some fields is supported by predefine macros

– E.g. `st_mode`

## Example: Simple `ls -l` Version

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

void stat_file(char *fname)
{
    struct stat buff;
    char* type = "Unknown";

    if(lstat(fname, &buff) < 0)
    {
        err_sys("lstat");
    }
}
```

## Example Continued

```
if(S_ISREG(buff.st_mode))
{
    type = "Regular file";
}
else if(S_ISDIR(buff.st_mode))
{
    type = "Directory";
}
else if(S_ISCHR(buff.st_mode))
{
    type = "Character special file";
}
else if(S_ISBLK(buff.st_mode))
{
    type = "Block special file";
}
else if(S_ISFIFO(buff.st_mode))
{
    type = "Pipe or FIFO";
}
```

## Example Continued

```
else if(S_ISLNK(buff.st_mode))
{
    type = "Symbolic link";
}
else if(S_ISSOCK(buff.st_mode))
{
    type = "Socket";
}
printf("%-30s  %10ld Bytes  %s\n", fname, buff.st_size, type);
}

int main(int argc,  char *argv[])
{
    int i;

    for(i=1; i<argc; i++)
    {
        stat_file(argv[i]);
    }
    return EXIT_SUCCESS;
}
```

## Example Output

```
$ /SOURCES/CSC_322/myls *
```

BINTREE	533 Bytes	Directory
LIST_DEMO	549 Bytes	Directory
Makefile	1322 Bytes	Regular file
Makefile~	1277 Bytes	Regular file
RPN_CALC	630 Bytes	Directory
RPN_CALC.tgz	10197 Bytes	Regular file
SORT	373 Bytes	Directory
a.out	13756 Bytes	Regular file
base_converter	14634 Bytes	Regular file
base_converter.c	1918 Bytes	Regular file
base_converter.c~	430 Bytes	Regular file
celsius2fahrenheit	13633 Bytes	Regular file
celsius2fahrenheit.c	395 Bytes	Regular file
charcount	13639 Bytes	Regular file
charcount.c	216 Bytes	Regular file
charcount.c~	114 Bytes	Regular file
charuniq	13643 Bytes	Regular file
charuniq.c	571 Bytes	Regular file
...		

## Example Output (of device directory /dev/)

```
$ /SOURCES/CSC_322/myls *
```

```
...  
cdrom                8 Bytes  Symbolic link  
cdu535              0 Bytes  Block special file  
cfs0                0 Bytes  Character special file  
cm205cd            0 Bytes  Block special file  
cm206cd            0 Bytes  Block special file  
console            0 Bytes  Character special file  
core               11 Bytes  Symbolic link  
cpu               196 Bytes  Directory  
cua0               0 Bytes  Character special file  
cua1               0 Bytes  Character special file  
...  
ham                0 Bytes  Character special file  
hda                0 Bytes  Block special file  
hda1              0 Bytes  Block special file  
hda10             0 Bytes  Block special file  
hda11             0 Bytes  Block special file  
hda12             0 Bytes  Block special file  
...
```

# Links

Links form a connection between a file name and the actual file

There are two kinds of links:

- Hard links
- Symbolic (or soft) links

A hard link links a name and a file

- Each file can have multiple hard links
- All are equivalent (no concept of “original link”), access is equally efficient for all hard links
- `rm` actually only removes a link, if the number of links becomes 0, the file is finally removed)
- Typically, it is only possible to have hard links to a file on the same physical partition or medium

## Links (2)

Soft links create indirect aliases for a file

- They are just files that contain another file name
- Following a soft link incurs a small performance penalty
- Symbolic links point anywhere in the file system (no limitations as to physical medium, networked file system, . . . )
- Symbolic links do not influence the file pointed to at all!
- If the file does not exist any more, the link still exists, but is **broken**

Most user-created links are soft links nowadays

- Used to share files
- Used to hide file system reorganization

# UNIX User Commands: `ln`

`ln` is used to create both hard and symbolic links

Usage is similar to `mv` and `cp`:

- `ln <target>`: Create a link to `<target>` in the current directory (under the same file name)
- `ln <target> <name>`: Make `<name>` a link to `<target>`
- `ln <target1>...<targetn> <dir>`: Create links to all targets in the current directories

Important option: `-s` (create symbolic links)

More: `man ln`

## Exercises

Read `man stat` and extend the `ls` example to show more information (e.g. everything `ls -l` shows)

Explain the difference between `mv filea fileb`, `cp filea fileb` and `ln filea fileb`

# CSC322

## C Programming and UNIX

### The UNIX File System

#### File modes

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# File Ownership

All files have an owner (a user)

- `ls -l` displays the user name (if available) or the numerical user id (e.g. for files of a user that no longer exists)

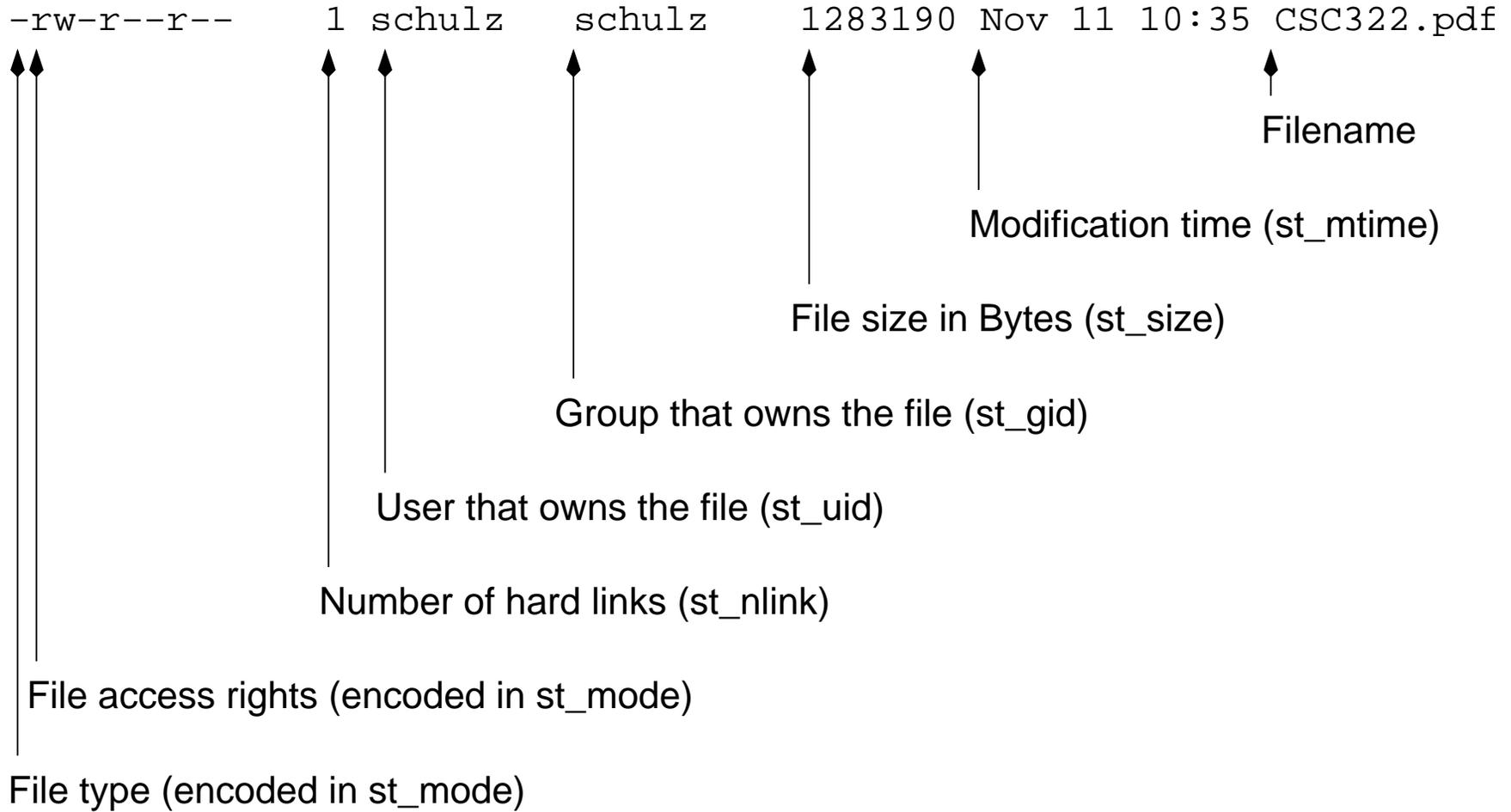
Similarly, each file has a **group** associated with it

- This will be similarly displayed by `ls -l`

Owner and group of a file determine who has what kind of access to that file. Access types are

- Read access (open a file for reading, reading data)
- Write access (change a file)
- Execute access (run a file as a program, or, for directories, access file names in that directory)

# ls -l Output Explained



Note: All information (except for the file name) are available by calling one of the `stat()` functions!

# User Groups

**Groups** are used in UNIX to give a group of users the ability to access a common resource

- Most obvious use: Share files on the disk
- In practice more important: Allow access to a hardware device (Note: A modem is a file, e.g. /dev/modem!)

Every user belongs to a **primary group**

- The primary group for a user is listed in the passwd file (as a numerical **group id** or gid):  
schulz:x:500:500:Stephan Schulz:/home/schulz:/bin/tcsh
  - \* For normal UNIX systems, /etc/passwd
  - \* For systems running NIS, see the file with ypcat passwd
- After logging in, the users primary group is active (the gid of the shell has the value for the primary group)
  - \* Processes started by another process (including the shell) inherit the gid

## Groups (Continued)

Additional group information is in `/etc/group`:

- For each group, a symbolic name (displayed by `ls -l`) and a list of users belonging to that group:

```
daemon:x:2:root,bin,daemon
```

```
schulz:x:500:
```

Secondary groups are additional groups which list the user as a member

- A user can explicitly change to such a group using the `newgrp` command (`man newgrp`)

## UNIX User Utilities: `chown` and `chgrp`

`chown` is used to change the owner of a file

- Usage: `chown <newuser> <file1> ...`
- On most systems, only root is allowed to use `chown` (there are security issues even with giving away files!)

`chgrp` changes the group of a file

- Usage: `chgrp <newuser> <file1> ...`
- On most systems, you can only change the group of a file to a group in which you are a member (see above)

Important option for both: `-R`

- Recursively apply the operation to subdirectories and files in them

# File Mode Bits

The status word of a file (the `st_mode` field in `struct stat` also contains 9 bits describing file access rights

- Note: These rights exist for all files, including special files and directories!

There are three different groups with potentially different access rights:

- The **u**ser who owns the file
- Members of the **g**roup associated with the file
- **O**ther users

There are also three different types of access:

- **R**ead access
- **W**rite access
- **E**xecute access

There are three more bits describing special properties

- The **setuid** bit: If true, the file will run under the **effective user id** of the program owner (not the one who started it)
- The **setgid** bit: Same thing for the group id
- The **sticky** bit with complex semantics and interesting history

# Symbolic Encoding

`ls -l` prints a string of 9 letters to represent the 9 12 file mode bits

Normal case: The `setuid`, `setgid` and sticky bit are all cleared (0):

- The mode has the form `uuugggooo` to encode user, group, and other access rights
- Each letter may be `-` to denote that that bit is clear
- Or it may have the mnemonic value of that right:
  - \* `r` for read (first letter)
  - \* `w` for write (second letter)
  - \* `x` for execute (third letter)

If one of the special bits is set, this is denoted by changing the last letter of each group (`x`) to another letter. Common cases (more: `info ls`):

- `s` in the user executable position: The file is user executable and the `setuid` bit is set
- `s` in the group executable position: The file is group executable, and the `setgid` bit is set

# Numerical Encoding

The 12 permission bits are normally represented by 4 octal digits (each digit represents 3 bits):

- 0001 represents execute access for others
- 0002 represents write access for others
- 0004 represents read access for others
- 0010 represents execute access for group
- 0020 represents write access for group
- 0040 represents read access for group
- 0100 represents execute access for user
- 0200 represents write access for user
- 0400 represents read access for user
- 1000 is the sticky bit
- 2000 is the setgid bit
- 4000 is the setuid bit

To generate a composite mode, just add up the individual modes

Leading zeroes (especially the first one) are often omitted

# Examples

`rw-r--r--` is the most common mode for a regular file on a conventional UNIX system:

- The user is allowed to read and write the file
- Everyone else is allowed to read the file (no secrets ;-)
- Corresponding numerical value:

0004 Other read

0040 Group read

0400 User read

0200 User write

-----

0644

Numeric mode 666 (the number of the beast) gives full read and write access for everyone (`rw-rw-rw-`)

- Some people claim that this is not coincidence. . .

# UNIX User Utilities: chmod

chmod is used to change the file access bits

Usage 1: `chmod <numeric-mode> files`

– Sets the file mode of the named files to the octal mode absolutely

Usage 2: `chmod <symbolic-mode-command> files`

– The symbolic mode command can add or remove privileges for the different groups

– Format: `<who><what><right>`

\* `<who>` can be any sequence of letters from ugo or a (equivalent to ugo)

\* `<what>` can be

· + to add rights

· - to remove rights

· = to absolutely assign rights

\* `<right>` can be any combination of letters from rwx

Important option: `-R`

– Recursively modify files and subdirectories

## chmod Examples

```
chmod ugo+rwx myfile      # Grant full access rights to everybody
chmod 777 myfile          # Grant full access rights to everybody
chmod -R go-rwx .        # Paranoid: Remove read, write, and execute
                           # rights for all other people on the current
                           # directory and all files and subdirectory
                           # Trying to fix things, but removed all
                           # execute rights from programs _and_
                           # directories (makes things hard to fix ;-)
```

# File Mode Creation Mask

Each process maintains a **file mode creation mask**

- This mask determines, which access rights are granted for newly created files and directories
- The colloquial name is **umask**
- The umask is inherited by new processes started (i.e. your files will be created with rights based on the umask of your shell)

The umask contains 9 bits, corresponding to the `rwXrwxrwx` access rights

- Bits set in the mask are always cleared
- All other rights are granted by default (with the `x` bits only set for executables and directories)

The `shell` maintains a umask that can be set with the `umask` command (which is normally in a user configuration file)

- Example: `umask 022`
- Removes write permissions for everybody but the owner

## Exercises

Read the man and info pages on `chmod`, `chown` and `chgrp`

The UNIX commands `chmod` and `chown` correspond to system calls of the same name. To find out how they work, read:

- `man 2 chmod`
- `man 2 chown`

Use this information to implement a rudimentary version of `chmod`

# CSC322

## C Programming and UNIX

### The UNIX File System File Descriptors

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# File Descriptors

Files are identified for the kernel as **file descriptors**

- A file descriptor is a small, non-negative integer
- It's used as an index into the file descriptor table of a process to obtain more information

For many purposes, file descriptors are quite similar to file pointers (FILE\*) from the C standard I/O library

However, file descriptor I/O is much more lowlevel

- No formatted I/O
- No buffering – each I/O operation directly causes a system call to actually perform the data transfer

Notes:

- UNIX's standard I/O library is implemented using file descriptors
- Network communication also works via file descriptors

## Opening Files: `open()`

The `open()` system call opens a named file and returns a file descriptor (or `-1` on failure)

It is defined as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int oflag, mode_t mode);
```

Arguments:

- `pathname` is a standard UNIX file name as for `fopen()`
- `oflag` contains the options. The value is created by bitwise **OR**ing of one of the following values with a number of option flags:
  - \* `O_RDONLY`: Open the file for reading
  - \* `O_WRONLY`: Open the file for writing
  - \* `O_RDWR`: Open for reading and writing
- The third argument is only interpreted if `open()` is used for file creation (and can be omitted otherwise)

## Option flags for `open()`

Note: All of the following flags have to be ORed (using the bitwise or operator `|`) with the main access mode (`O_RDONLY`, `O_WRONLY`, `O_RDWR`)

Options:

- `O_APPEND`: All output on this file descriptor is appended at the end of the file
- `O_CREAT`: If the file does not exist, create it
- `O_EXCL`: Only used with `O_CREAT` – give an error, if the named file already exists
- `O_TRUNC`: If the file exists and is opened for writing or r/w, truncate it to length 0
- `O_SYNC`: Only return from writes to that file when the physical output is complete

There are some more flags that we only discuss when necessary

Example: `fd = open("/tmp/testfile", O_WRONLY|O_APPEND|O_SYNC)`

## Using `open()` to create files

If the option `O_CREAT` is given, `open()` will create a file if no file with the given name exists

This also requires the third argument to `open()` (which otherwise is ignored or can be omitted)

- This argument describes the access rights set for the new file
- It is created by binary ORing of the following constants:

<code>S_IRUSR</code>	Read Permission for the user
<code>S_IWUSR</code>	Write permission for the user
<code>S_IXUSR</code>	Execute permission for the user
<code>S_IRGRP</code>	Read Permission for the group
<code>S_IWGRP</code>	Write permission for the group
<code>S_IXGRP</code>	Execute permission for the group
<code>S_IROTH</code>	Read Permission for the others
<code>S_IWOTH</code>	Write permission for the others
<code>S_IXOTH</code>	Execute permission for the others

- Note: These are the same values used by `st_mode` in `struct stat`

## Notes on `open()` and `close()`

The mode given to `open()` is modified by the `umask` of the process

The file mode is only set if the file is actually created (not even if it exists but is truncated with `O_TRUNC`)

A file is **closed** using the `close()` function:

```
#include <unistd.h>
int close(int fd);
```

– Return value: 0 on success, -1 on failure

There are three predefined file descriptors that are open by default, corresponding to the 3 standard I/O channels:

- `STDIN_FILENO` (traditionally 0)
- `STDOUT_FILENO` (traditionally 1)
- `STDERR_FILENO` (traditionally 2)

## File Descriptor I/O: read() and write()

The functions `read()` and `write()` perform unbuffered input and output:

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `ssize_t` is an integer type defined in `<unistd.h>`
- `fd` is the file descriptor for input or output
- `buf` is a pointer to an area of memory
  - \* `write()` reads the data to write from this buffer
  - \* `read()` stores the read data in the buffer
- `count` is the number of bytes to transfer (and should not be bigger than the size of `*buf!`)

Both functions return the number of bytes transmitted

- For `write()`, a smaller number than requested signals an error
- For `read()`:
  - \* 0 indicates end of file
  - \* -1 signals error
  - \* Everything else is normal (there may be fewer characters than requested currently available)

## Example: Simple cat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

## Example Continued

```
#define BUF_SIZE 1024

int main(int argc, char* argv[])
{
    int fd;
    char buf[BUF_SIZE];
    ssize_t count, check;

    if(argc!=2)
    {
        fprintf(stderr, "USAGE: mycat2 file");
        exit(EXIT_FAILURE);
    }
    fd = open(argv[1], O_RDONLY);
    if(fd == -1)
    {
        err_sys("open");
    }
}
```

## Example Continued

```
while((count = read(fd,&buf,BUF_SIZE)))
{
    if(count== -1)
    {
        err_sys("read");
    }
    check = write(STDOUT_FILENO, &buf, count);
    if(check!=count)
    {
        err_sys("write");
    }
}
if(close(fd) == -1)
{
    err_sys("close");
}
return EXIT_SUCCESS;
}
```

# The Standard I/O Library and File Descriptors

Remember that a file pointer is actually of type `FILE*`

It typically points to a structure in an array

- `stdin` points to element number 0
- `stdout` points to element number 1
- `stderr` points to element number 2
- More elements are filled in for each use of `fopen()`

Each of the structures contains:

- A buffer
- Some counters and positions to manage the buffer
- A file descriptor
- Flags for the access mode (read or write)

Consider the case of writing:

- All write commands just write into the buffer space
- If the buffer is full or a `fflush()` command is issued (or the stream is closed), all of the buffer is written using a single `write()` command

Reading similarly reads a large block and hands it out piecewise

## Cheating with fdopen()

Formatted, buffered output is very convenient and quite efficient for many small I/O operations (`getchar()`, `fprintf()`, ...)

- Normally much better than `read()` and `write()`
- But some I/O methods only give us file descriptors (dammit!)

Solution: The function `fdopen()` will generate an entry in the `FILE` array from a file descriptor and return the pointer to it

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

`fildes` has to be an open file descriptor

`mode` is a string as for `fopen()` ("`r`", "`w`"...) and must be compatible with the flags of the file descriptor

## Exercises

Write simple version of `cp` using `open()`, `read()` and `write()`. Use a default buffer size, but support an option `-b` that allows you to set the buffer size from the command line. Measure the speed of copying a large file for different sizes

Examples:

- `mycp file1 file2` copies `file1` to `file2`, using the default buffer size
- `mycp -b3 file2 file2` copies the file using a buffer of 3 bytes

Use the `fstat()` command on both files to get the native block size of the file systems for both files (the `st_blksize` field in `struct stat`). What do you notice? Can you write a better `cp` now?

**CSC322**  
**C Programming and UNIX**  
**More on File Descriptors**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

## More on the UNIX I/O System

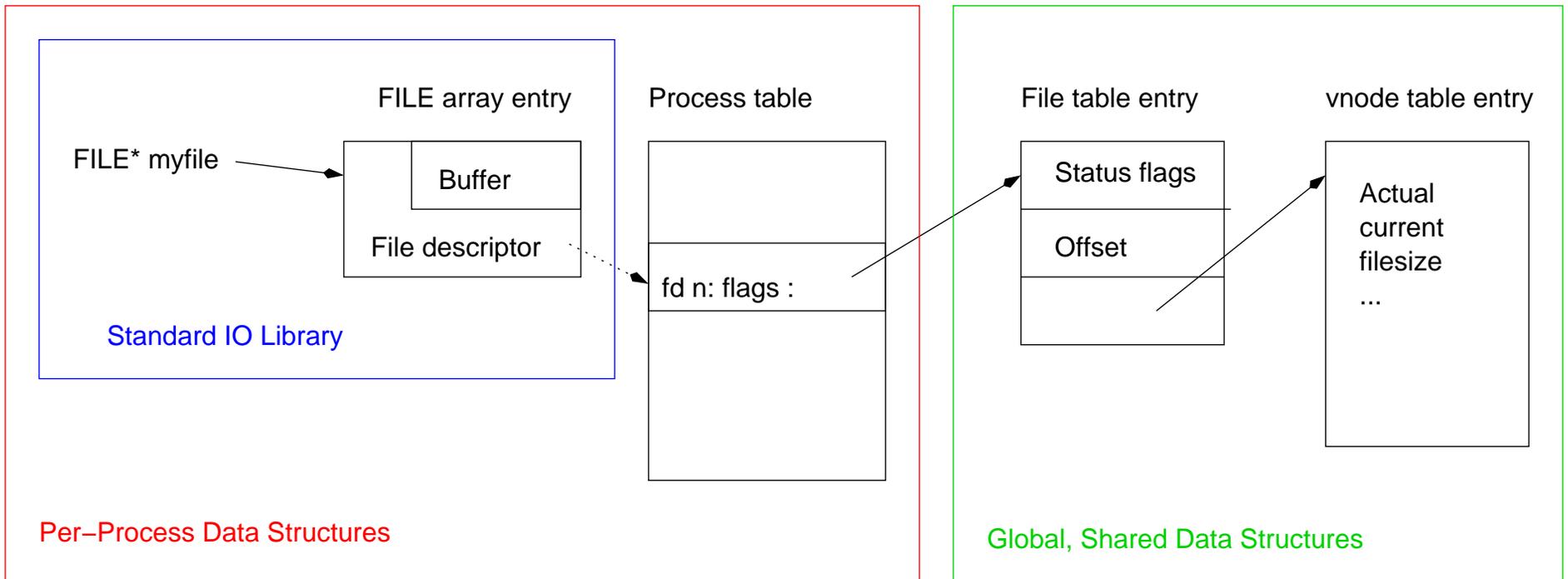
The file descriptor typically is an index into a table that contains information about all open files of the **process**

- That table contains just the flags (read/write) for that file descriptor and a pointer to the kernels global **file table**

The **file table** is **global** and shared by all processes. It has one entry per opened file , containing:

- File status flags (read, write, append, sync. . . , the things we passed to `open()`)
- Current offset into the file: The position where the next read or write will start
- A pointer to the `vnode` of the file
  - \* The `vnode` contains the file type and information about how to actually access the file, as well as the current real file size
  - \* It also gives us a way to access the `inode` that contains all the information we get with `stat()`
  - \* There is only one `vnode` per file, i.e. the `vnode` is the same for all file descriptors and all processes that access the same file

# The UNIX File I/O System



# Blocking vs. Nonblocking I/O

All I/O we have seen so far is blocking

- `read()` waits (**blocks**) until some input becomes available
- It then returns the read data
- Similarly, if `write()` temporarily cannot write the data, it blocks until it can

**Non-blocking** I/O always returns immediately from the I/O function

- If the I/O failed temporarily, the functions return -1
- `errno` is set to **EWOULDBLOCK**

**Question:** How do we achieve non-blocking I/O?

**Answer:** By manipulating the file descriptor

- Each file descriptor has a number of associated flags
- One of these selects blocking vs. non-blocking behaviour

## Manipulating File Descriptors: `fcntl()`

`fcntl()` is a catch-all function for manipulating file descriptors

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

We are only interested in the use of `fcntl()` for getting and changing the **file status flags**:

- `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- `O_APPEND`
- **`O_NONBLOCK`**
- `O_SYNC`
- ... (depending on UNIX version)

`fcntl()` may return various values, depending on `cmd`

- On error, it always returns `-1` and sets **`errno`**

## fcntl() Continued

Using `fcntl()` to get the file status flags:

```
flags = fcntl(fd, F_GETFL);
```

- To interpret the result, we need to logically **AND** it with the flag we are interested in (see example)
- To get the read/write status, **AND** the result with `O_ACCMODE`

To set the file status flags:

```
fcntl(fd, F_SETFL, newflags);
```

- If we only want to change a single flag, we have to get the old value and use binary operations to change just that flag!

– Example:

```
int flags = fcntl(STDIN_FILENO, F_GETFL);  
flags = flags | O_NONBLOCK;  
fcntl(STDIN_FILENO, F_SETFL, flags);
```

## Example: Printing Flags for a File Descriptor

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

## Example (2)

```
void print_fd_file_status(int fd)
{
    int flags = fcntl(fd, F_GETFL);
    if(flags == -1)
    {
        err_sys("fcntl");
    }
    printf("Flags for file descriptor %d\n", fd);
    switch(flags & O_ACCMODE)
    {
    case O_RDONLY:
        printf("Read only\n");
        break;
    case O_WRONLY:
        printf("Write only\n");
        break;
    case O_RDWR:
        printf("Read/Write\n");
        break;
    default:
        printf("Strange\n");
    }
}
```

## Example (3)

```
if(flags & O_APPEND)
{
    printf("Append is set\n");
}
if(flags & O_NONBLOCK)
{
    printf("Non-blocking\n");
}
if(flags & O_SYNC)
{
    printf("Synchronous writes\n");
}
}
int main(int argc, char* argv[])
{
    print_fd_file_status(STDIN_FILENO);
    print_fd_file_status(STDOUT_FILENO);
    print_fd_file_status(STDERR_FILENO);
    print_fd_file_status(42);
    return EXIT_SUCCESS;
}
```

## Example Output

```
$/fcntl_example
```

```
Flags for file descriptor 0
```

```
Read/Write
```

```
Flags for file descriptor 1
```

```
Read/Write
```

```
Flags for file descriptor 2
```

```
Read/Write
```

```
fcntl: Bad file descriptor
```

```
$/fcntl_example < signal_test.c
```

```
Flags for file descriptor 0
```

```
Read only
```

```
Flags for file descriptor 1
```

```
Read/Write
```

```
Flags for file descriptor 2
```

```
Read/Write
```

```
fcntl: Bad file descriptor
```

# Multiplexing I/O

Often, a program has to be able to read data from multiple sources

- Data from the user
- Data from the network
- Data from a file that is in the process of being written

Bad solution: Polling

- Switch all file descriptors to non-blocking
- Test them one after the other, until one of them has data
- **Uses too much system resources!**

Minimally better: Polling with a short waiting time between I/O attempts

- But: Lousy reaction time

Right solution: Use the right tool (`select()`)

## Multiplexing I/O: select()

select() is used to watch a set of file descriptors for one of three conditions:

- A file descriptor is ready for reading
- A file descriptor is ready for writing
- Is there an exceptional condition for a file descriptor?

We can tell the function to either

- Return immediately, telling us the current status
- Wait until at least one of the conditions becomes true
- Wait until at least one of the conditions becomes true, but at most a fixed amount of time

Specification:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
int select(int max_fd+1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *tvptr);
```

## select() Arguments

`fd_set` is defined in `sys/types.h`

- It is a data type that can store a set of file descriptors
- We only know how to manipulate it:
  - \* `FD_ZERO(fd_set *set)` removes all file descriptors from the set
  - \* `FD_SET(int fd, fd_set *set)` inserts `fd` into the set
  - \* `FD_CLR(int fd, fd_set *set)` removes `fd` from the set
  - \* `FD_ISSET(int fd, fd_set *set)` returns true, if `fd` is contained in `*set`

The three `fd_set*` arguments are used for input **and** output of `select()`

- The `fd_set` structures the arguments point to describe which file descriptors we are interested in
- If the pointer is `NULL`, we are not interested in any file descriptor for the corresponding property
- If `select()` returns, the set have been modified to contain just the descriptors for which the property is true

`int max_fd+1` has to be at least one bigger than the biggest file descriptor in any one of the three sets

- It is used to speed up things in the UNIX kernel

## `select()` Arguments and Return Value

The last argument to `select()` is a pointer to a struct `timeval`

This struct has two fields:

- `long tv_sec; /* Seconds */`
- `long tv_usec; /* Microseconds */`

There are two possible cases:

- `tvptr` is `NULL`: In this case, `select()` waits until one of the file descriptors is ready (or a signal is caught)
- `tvptr` points to a valid struct `timeval`: In this case, `select()` waits at most the specified time

Return value:

- `-1` on error or if `select()` returned because of a signal (`errno` will be set!)
- Otherwise, the number of file descriptors for which the specified condition is true is returned

## Example

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    fd_set readfds;
    fd_set writefds;
    int res;

    FD_ZERO(&readfds);
    FD_ZERO(&writefds);

    FD_SET(STDIN_FILENO, &readfds);
    FD_SET(STDOUT_FILENO, &writefds);
    FD_SET(STDERR_FILENO, &writefds);

    res = select(3, &readfds, &writefds, NULL, NULL);
    printf("%d file descriptors are ready\n", res);
}
```

## Example (2)

```
if(FD_ISSET(STDIN_FILENO, &readfds))
{
    printf("STDIN is ready for reading\n");
}
if(FD_ISSET(STDOUT_FILENO, &writefds))
{
    printf("STDOUT is ready for writing\n");
}
if(FD_ISSET(STDERR_FILENO, &writefds))
{
    printf("STDERR is ready for writing\n");
}

return EXIT_SUCCESS;
}
```

## Example Output

```
$ ./select_example
2 file descriptors are ready
STDOUT is ready for writing
STDERR is ready for writing
$ ./select_example < select_example.c
3 file descriptors are ready
STDIN is ready for reading
STDOUT is ready for writing
STDERR is ready for writing
```

# Internet Assignment (I)

On the assignment home page you will find links to two binary programs, a chat server and a chat client. In the end, you should turn in a program that has the same functionality as the client

Step 1:

- Download the programs and understand what they do
- To start the server, type `./chat_server <port>`, where `<port>` is an integer greater than 1024
- To connect to the sever, type `./chat_client <ip-addr> <port> <nick>`
  - \* `<ip-addr>` is the **IP-Address** of the server host (use 127.0.0.1 if the server runs on the same host, use `nslookup <name>` for other hosts)
  - \* `<port>` is the same number as used for the server
  - \* `<nick>` is the nickname under which you will chat

Caveats:

- Due to firewalling, do not expect to be able to reach a server from outside the lab
- The binaries only run under Linux

I will try to keep a server running on lee on port 6666 for all of you to share

# CSC322

## C Programming and UNIX

### Basic UNIX Network Programming Introduction

Stephan Schulz

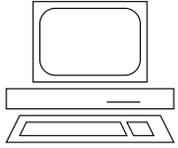
Department of Computer Science

University of Miami

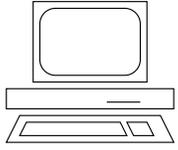
`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

# Networking



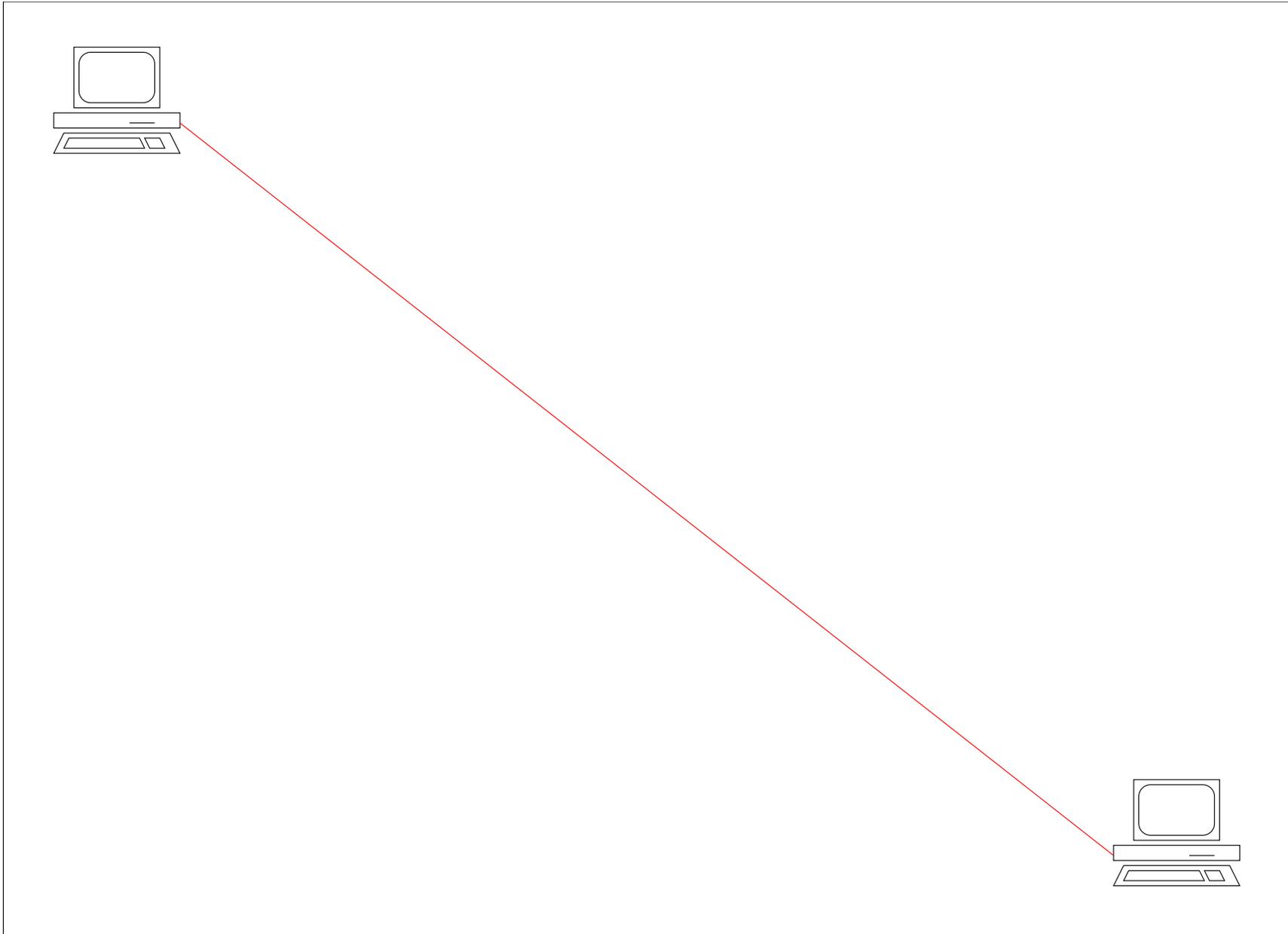
# Networking



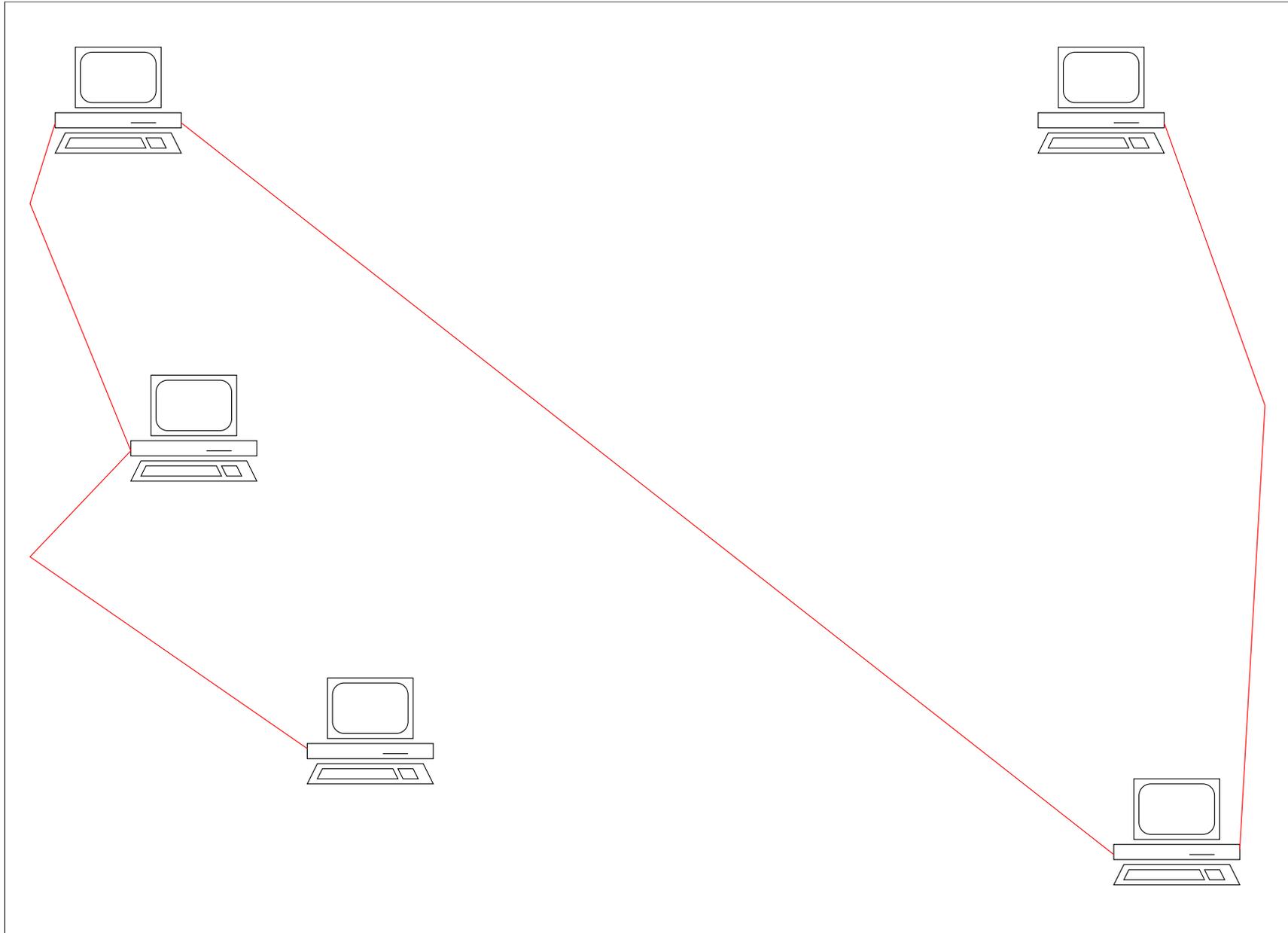
# Networking



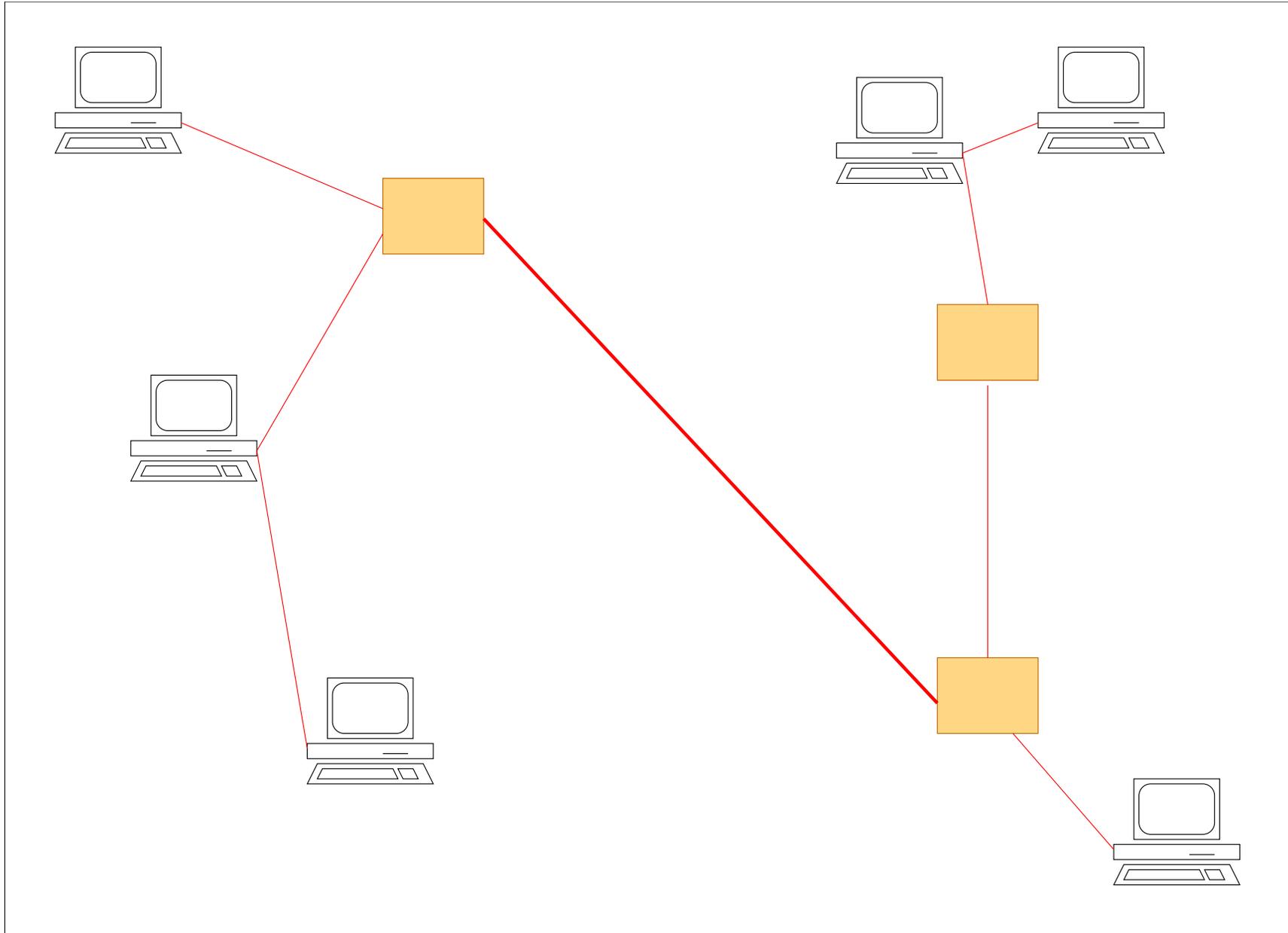
# Networking



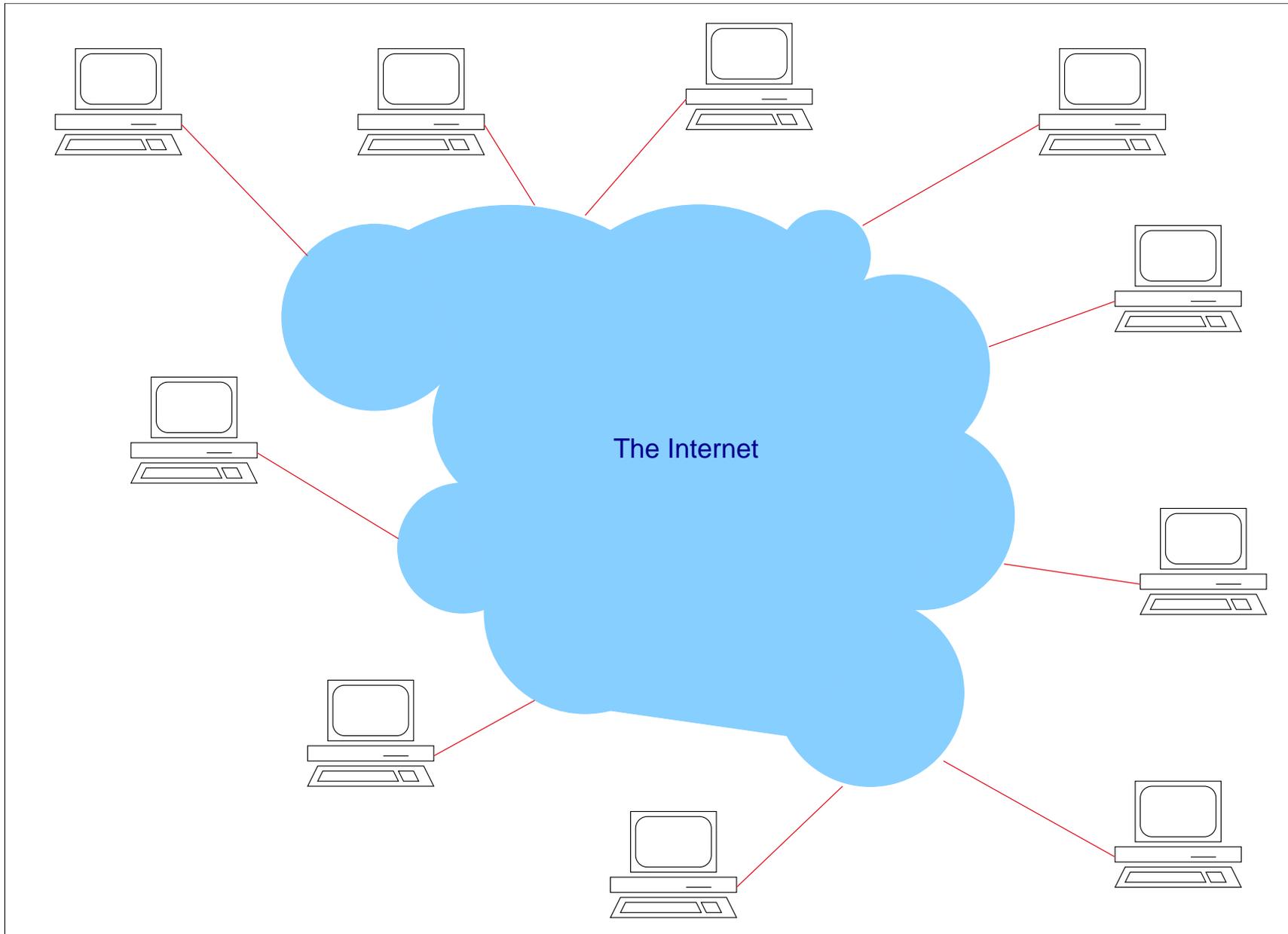
# Networking



# Networking



# Networking



# Networking Concepts

Is communication occurring between two partners, or is it a broadcast communication?

- How are partners identified (**addressed**)?

Is traffic **stream oriented** or **packet oriented**?

- Stream-oriented: Messages arrive as stream of bytes (similar to reading from a file)
- Packet oriented: Traffic arrives in the form of distinct packets of a fixed (or fixed maximal) size

Is the communication reliable or unreliable?

- Can messages disappear?
- Can the order of messages change?
- Can messages be duplicated?

# Network Layers

Level 0: Physical or Hardware layer

- Copper wires or optical fiber
- Radio waves or laser beams for wireless protocols

Level 1: Data Link Layer

- How is data transported?
- Examples: Ethernet, Token ring, ATM, ISDN

Level 2: Network layer

- How are individual hosts or networks assembled into a network?
- Examples: Internet protocol (IP)

Level 3: Transport layer

- Converts from standard user packets to network layer packets
- May include error checking and correcting
- Examples: TCP and UDP

Higher layers. . .

- Take care of data representation at various levels

# The Internet Protocol (IP)

Level 2 protocol (Hardware-Agnostic)

Prevalent protocol today: IPv4

- Unreliable (“best effort”)
- Packet-oriented (**IP-Datagrams**)
- Can be addressed to individual hosts or **broadcast addresses**
- Addresses are 32 bit numbers (“4 binary octets”), normally written as dotted decimal numbers: 127.0.0.1
- Addresses denote individual hosts!

Currently being deployed: IPv6

- Shares many properties
- But: 128 bit addresses (8 4-digit hex numbers, written in Hex and separated by colons: 21DA:00D3:0000:2A3B:02AA:00BF:FE28:9C5A)

# The User Datagram Protocol (UDP)

Based on IP

Still. . .

- paket-oriented
- unreliable

Adds: Service multiplexing

- The same host can have many different communications
- Each communication uses a different **port**

Supported by UNIX with sockets with socket type `SOCK_DGRAM`

Used for:

- DNS (Domain name service)
- NFS (Network file system)

# The Transmission Control Protocol (TCP)

Based on IP, but. . .

- Connection-based
- Stream-oriented
- Reliable
- Service multiplexing (with ports)

Supported by UNIX with sockets with socket type `SOCK_STREAM`

Addresses for TCP (and UDP) have two parts:

- The **IP number** for specifying the host
- The **port number** for specifying the port

Most services are associated with a fixed port number:

- HTTP (WWW): Port 80
- SMTP (Email transport): Port 25
- FTP (File Transfer): Port 21
- For a semi-complete list: `more /etc/services`
- Server port numbers up to 1024 are normally reserved for root

# UNIX Sockets

Sockets are special **file descriptors** used for many different kind of inter-process communication

- Local
- Networked

We can create sockets for different communication styles

- Stream oriented
- Datagram

Sockets are used on both sides of a communication

- The receiver creates a socket and associates it with a port
- The sender creates a socket and connects it to the receiver

# Client/Server Model

A **server** offers a certain service

– It is ready to **accept** connections on a certain port

A **client** initiates communication by trying to **connect** to that port

# CSC322

## C Programming and UNIX

### Basic UNIX Network Programming Simple Connections

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# The Client Side for TCP Connections

The client has to perform the following steps:

- Create a socket for stream-oriented communication over IP
- Create an address structure for the server address
- Connect the socket to the server port
- Use the connection

# Creating Sockets with `socket()` (1)

To create a socket, we call `socket()`

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

On success, `socket()` returns a valid file descriptor just like `open()`

– After enough black magic, we can use it with `read()` and `write()`

On failure, the function returns `-1` and sets `errno`

## Creating Sockets with `socket()` (2)

```
int socket(int domain, int type, int protocol);
```

The `domain` argument describes the protocol family that will be used. Interesting values:

- `PF_INET`: Internet with IPv4
- `PF_INET6`: Internet with IPv6
- `PF_LOCAL`: Local communication

The `type` describes the communication style:

- `SOCK_STREAM` for connection based streams
- `SOCK_DGRAM` for datagrams

The last argument specifies the protocol

- There normally is only a single protocol for each domain/type pair, use 0 to select this (the default)
- `PF_INET/SOCK_STREAM` gives us TCP/IPv4
- `PF_INET/SOCK_DGRAM` gives us UDP/IPv4

# Socket Adresses

This is a reasonably ugly topic!

Because sockets are used for so many things, there is no single data type for socket addresses

- Instead, each address family has its own format
- We have to pass this by address (casted to a bogus type struct sock\_addr\*)
- Additionally, we have to tell the system the size of our address format

Because different computer models use different data formats (**Big Endian** vs. **Little Endian**), we have to convert values to **network order** using:

```
#include <netinet/in.h>
uint32_t htonl(uint32_t hostlong); /* Host to Network conversion for long */
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort); /* Network to host conversion for short */
```

## Socket Adresses for IPv4

For IPv4 addresses, we use the data type struct `sock_addr_in`

It contains the following fields we have to fill:

```
u_char  sin_family;      /*----Internet address family */
u_short sin_port;       /*----Port number */
struct  in_addr sin_addr; /*----Holds the IP address */
```

For `sin_family`, we use a predefined constant `AF_INET`

For the port, we use the port number, converted with `htons()`

`sin_addr` is filled in by the function `inet_pton()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
```

## inet\_pton()

```
int inet_pton(int af, const char *src, void *dst);
```

`inet_pton()` converts an internet address in string form to a network address structure

First argument: Address family

- `AF_INET` for IPv4 addresses
- `AF_INET6` for IPv6 addresses

Second argument: Pointer to string containing address

- For IPv4: IP-Numbers (4 numbers with dots)
- IPv6: Hex representation (8 4-digit hex numbers separated by colons)

Third argument: Pointer to the destination

- Normally a pointer to the `sin_addr` field in a `struct sockaddr_in`

## Connecting to a Remote Port: `connect()`

After we have prepared an address in a struct `sock_addr_in`, we can **connect** an existing socket to a remote port:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

- `sockfd`: Socket you want to connect
- `serv_addr`: Pointer to the carefully prepared address you want to connect to, casted to `struct sockaddr*`
- `addrlen`: Size of **your** actual structure, i.e. `sizeof(struct sock_addr_in)`
  - \* Remember that by casting the second argument, we actually lie to the about the data structure we are pointing to
  - \* That's ok – the socket library knows that we are probably lying
  - \* Passing the length helps the library to straighten things out

Return value: 0 on success, -1 on failure

## Example: Getting an Insult

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    int sock;
    struct sockaddr_in server_addr;
    char buf[80];
    int msg_len, res;
```

## Example (2)

```
sock = socket(PF_INET, SOCK_STREAM, 0); /* Check against -1 omitted! */

memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(1695);
res = inet_pton(AF_INET, "128.138.196.16", &server_addr.sin_addr);
if(res < 0)
{
    err_sys("inet_pton (no valid address family)");
}
if(res == 0)
{
    fprintf(stderr, "Not a valid IP address");
    exit(EXIT_FAILURE);
}
res = connect(sock, (struct sockaddr *) &server_addr,
              sizeof(server_addr));
if(res == -1)
{
    err_sys("connect");
}
```

## Example (3)

```
while(1)
{
    msg_len = read(sock, buf, 80);
    if(msg_len == 0)
    {
        break;
    }
    write(STDOUT_FILENO, buf, msg_len);
}
close(sock);

return EXIT_SUCCESS;
}
```

# The Server Side

A **server** has a more complex task than a client

General steps:

- Create a socket (we now how to do this)
- Create an address (on its own machine)
- **Bind** the socket to the address
- **Listen** for incoming connections on that socket

For each client:

- **Accept** the connection (on a new socket)
- Use the connection
- **Close** the connection

# Server Side Addresses

We need to specify a local address for the listening port

– It contains the address family, IP address, and port

Instead of actually digging out the servers IP address (which may be complex), we use the special address 0.0.0.0 or INADDR\_ANY

Given this address, the server will accept connections on any IP address which refers to it

Example:

```
struct sockaddr_in sock_name;
int sock;
short port;
...Get socket, set port to some value...
memset(&sock_name, 0, sizeof(sock_name));      /* Clear address */
sock_name.sin_family = AF_INET;                /* Set address family */
sock_name.sin_port = htons(port);              /* Set port */
sock_name.sin_addr.s_addr = htonl(INADDR_ANY); /* Set address */
```

# Naming a Socket (Binding a Socket to an Address)

Once we have created a socket and a local address, we need to **bind** the socket to an address

- All future operations will make use of that address

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

- `sock_fd`: Socket we want to bind
- `my_addr`: Pointer to the address
- `addrlen`: Length of that address
- See remarks for `connect()`!

Return value:

- 0 on success
- -1 on failure

# Listening for Incoming Connections

We use the `listen()` function call to make a socket listen for incoming connections:

```
#include <sys/socket.h>
int listen(int sock, int backlog);
```

- `sock` is the file descriptor we want to set to listening state
- `backlog` is the number of pending connections allowed at any one time
  - \* If more unanswered connection request are received, they will be refused or ignores
  - \* If we accept a connection, that slot becomes available again
  - \* A good value is 5 ;-)

Return value: 0 on success, -1 on failure

# Accepting Connections

To finally establish a connection, we have to accept it:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sock, struct sockaddr *addr, socklen_t *addrlen);
```

- `sock`: The socket we expect connection on
- `addr`: A pointer to an address structure (or NULL)
- `addrlen`: A pointer to an integer variable of type `socklen_t` that initially has to contain the size of `*addr`

If `accept()` returns. . .

- The return value is the file descriptor of a **new** socket (or -1)
- If `addr` is not NULL, the address of the remote socket is written into it
- `*addrlen` is changed to the actual size of the new variable

By default, `accept()` **blocks** until a connection request is received

- If we set the socket to non-blocking (using `fcntl()`), it will return with -1 and set `errno` to `EWOULDBLOCK` if there are no pending requests

## Example: Greeting the World

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <ctype.h>
#include <unistd.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    int sock, con_sock;
    struct sockaddr_in sock_name;
```

## Example (2)

```
if(argc!=2)
{
    fprintf(stderr, "Usage: simple_server <port>\n");
    exit(EXIT_FAILURE);
}
sock = socket(PF_INET, SOCK_STREAM, 0);
if(sock == -1)
{
    err_sys("socket");
}
sock_name.sin_family = AF_INET;
sock_name.sin_port = htons(atoi(argv[1]));
sock_name.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sock, (struct sockaddr *) &sock_name, sizeof(sock_name)) < 0)
{
    err_sys("bind");
}
if(listen(sock, 1) == -1)
{
    err_sys("listen");
}
```

## Example (3)

```
while(1)
{
    con_sock = accept(sock, NULL, NULL);
    if(con_sock == -1)
    {
        err_sys("accept");
    }
    write(con_sock, "Hiho and welcome!\n", strlen("Hiho and welcome!\n"));

    if(close(con_sock) == -1)
    {
        err_sys("close(con_sock)");
    }
}
/* sock closed automatically when we exit via ^C */
}
```

## More Information

man pages:

- man socket
- man 2 bind
- man accept

The GNU C library documentation on sockets

- Available by doing `info libc`
- In emacs: **[C-h i]**
- On the internet, e.g. at:
  - \* [http://www.gnu.org/manual/glibc-2.2.3/html\\_chapter/libc\\_16.html](http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_16.html)
  - \* [http://www.gnuenterprise.org/doc/glibc-doc/html/chapters\\_16.html](http://www.gnuenterprise.org/doc/glibc-doc/html/chapters_16.html)

## Internet Assignment (II)

Step 2: Write a simple client that

- reads IP address, port and nickname from the command line
- Connects to the specified server
- Uses `select()` or non-blocking `read()` to read everything the server transmits
- Closes the connection

Step 3: Modify the client to keep on reading. Be sure to use `select()` now!

Step 4: Write a second client that connects, reads input from the terminal, and sends it to the server (prepended with the nickname and a colon).

- You should be able to see what you send if you simultaneously connect with the client from step 3.
- If the user types [C-D] to signal end of input, close the connection to the server and terminate

Step 5: Put everything together, using `select()` on the network connection and standard input.

- Send input from the terminal to the server
- Set input from the network to standard output

**CSC322**

**C Programming and UNIX**

**Process Creation and Termination (I)**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# Subprocesses

UNIX is a multi-process operating systems

- Many processes run at the same time
- Processes can be created and can terminated

Processes form a hierarchy

- All processes have a unique parent
- In the end, all (real) processes descent from the `init` process

Parent and child share a special relationship

- The parent has to retrieve the termination status of a process
- The child can get his parents process id
- If a parent dies, its special role is taken over by the `init` process

# Process Properties

For each process, we can get various identifiers:

- The process id
- The process id of the parent
- The **real** user id of the process (i.e. the user id of the owner)
- The **effective** user id of the process (i.e. the user id that is used to check access rights). It can differ e.g. for programs with the `setuid` bit set
- The real group id
- The effective group id

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getpid(void);      /* Get process id */
pid_t getppid(void);    /* Get parent process id */
uid_t getuid(void);     /* Get real user id */
uid_t geteuid(void);    /* Get effective user id */
gid_t getgid(void);     /* Get real group id id */
gid_t getgid(void);     /* Get effective group id */
```

# Standard Execution of a UNIX Program

Creation of the process

- Can only happen via the `fork()` process

Execution of a program

- Via the kernel system call `exec()`
- Comes in various handy library variants

Running

- Process runs in its own process space (virtual memory)

Termination

- Normal exit
- Call to `abort()`
- Catching a signal for which the default action is aborting

# Exiting

There are three normal ways of terminating a program

Calling `return st;` from `main()` (ANSI C)

- In that case the **exit status** of the program is `st`
- Interpretation of the exit status is implementation-defined for ANSI C (but defined for UNIX)

Calling `exit(st);` from anywhere in the program (ANSI C)

- Exit status is `st`
- In `main()`, `exit()` and `return` are equivalent
- In both cases, some cleanup actions are performed
  - \* **Exit handlers** are called
  - \* All open files are flushed and closed

Calling `_exit(st)` (UNIX) or `_Exit(st)` (new in ANSI-C 99, may not be widely supported)

- Program is immediately terminated
- Exit status is `st`

# Exit Formalities

```
#include <stdlib.h>
```

```
void exit(int status);
```

```
void _Exit(int status); /* New in C99 */
```

```
#include <unistd.h>
```

```
void _exit(int status);
```

ANSI C defines three different exit statuses:

- EXIT\_SUCCESS (in `stdlib.h`)
- EXIT\_FAILURE (in `stdlib.h`)
- 0 (equivalent to EXIT\_SUCCESS)

In practice, EXIT\_SUCCESS is nearly always just #defined as 0

## Cleaning up: atexit()

ANSI C allows us to register up to 32 functions that will be called whenever the program terminates normally:

```
#include <stdlib.h>
```

```
int atexit(void (*func)(void));
```

- Argument is a pointer to a function that neither takes an argument nor returns a value
- Return value for `atexit()` is 0 on success, -1 on error

Each call to `atexit()` results in a single call to the registered function

- Registered functions are called in reverse order of registration
- We can register the same function more than once

Note: Exit handlers should only access global variables

# Example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int handler_counter=0;

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

void handler1(void)
{
    printf("Handler1, counter = %d\n", handler_counter);
    handler_counter++;
}

void handler2(void)
{
    printf("Handler2, counter = %d\n", handler_counter);
    handler_counter++;
}
```

## Example (2)

```
int main(void)
{
    if(atexit(handler1) != 0)
    {
        err_sys("atexit");
    }
    if(atexit(handler2) != 0)
    {
        err_sys("atexit");
    }
    if(atexit(handler1) != 0)
    {
        err_sys("atexit");
    }
    if(atexit(handler1) != 0)
    {
        err_sys("atexit");
    }
    printf("My PID is %d and my parents PID is %d\n", getpid(), getppid());
    return EXIT_SUCCESS;
}
```

## Example Output

```
My PID is 2019 and my parents PID is 746  
Handler1, counter = 0  
Handler1, counter = 1  
Handler2, counter = 2  
Handler1, counter = 3
```

## Running other Programs: `system()`

The `system()` function is defined by ANSI C

```
#include <stdlib.h>
int system(const char *command);
```

`system()` hands the string pointed to by `command` to the systems **command processor** for execution

- `system()` returns, when the command returns
- Return value of `system()` in this case is implementation-defined

If `command` is `NULL`, `system()` checks if the implementation **has** a command processor

- It returns 0, if not
- Anything else, otherwise

## system() in UNIX

On UNIX, there **always** is a command processor

- The command is handed to the standard shell, `/bin/sh`
- It can make use of all shell facilities, including I/O redirection

The return value of the `system()` command normally is an encoding of the **exit status** of the executed command

- If for some reason no new process for the shell can be created, `-1` is returned (and `errno` is set to specify what went wrong)
- If the shell cannot be executed, it is treated as if the shell returned `127`
- Otherwise, the return value is an encoding of the **exit status** of the shell (which always returns the exit status of the command, if it could be executed)

# Termination Status Interpretation

Termination status can come from multiple sources

- `system()` (which nicely packs up all the work for us)
- Functions that retrieve the exit status of a child process: `wait()` and `waitpid()` (more later)

Interpretation depends on the cause of the termination of the child process.

Assume that `status` is the termination status

- If `WIFEXITED(status)` is true, the process terminated normally (i.e. via `exit()`, `_exit()` or `return` from `main`)
  - \* `WEXITSTATUS(status)` returns the (lower 8 bit of) the value that was passed to `exit()`
- If `WIFSIGNALED(status)` is true, the process was terminated because of an uncaught signal with default action `abort`
  - \* `WTERMSIG(status)` gives the number of the signal

If `WIFSTOPPED(status)` is true, the process is currently stopped (via `SIGSTOP` or `SIGSTP`)

- `WSTOPSIG(status)` returns the number of the stop signal

## Example: Executing Commands

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int handler_counter=0;

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

## Example: Executing Commands

```
int main(int argc, char* argv[])
{
    int i, status;
    for(i=1; i<argc; i++)
    {
        status = system(argv[i]);
        if(status == -1)
        {
            err_sys("system");
        }
        if(WIFEXITED(status))
        {
            printf("Exited normally, returning %d\n", WEXITSTATUS(status));
        }
        else
        {
            printf("Handle this in your assignment\n");
        }
    }
    return EXIT_SUCCESS;
}
```

## Example Output

```
$ ./system_example "date" "man does_not_exist" "whoami -q"  
Tue Nov 26 04:59:29 CET 2002  
Exited normally, returning 0  
No manual entry for does_not_exist  
Exited normally, returning 16  
whoami: invalid option -- q  
Try 'whoami --help' for more information.  
Exited normally, returning 1
```

## Exercises

Write a program that prints its parents PID and modify the last example to print its PID. Run the program via the example code. What do you notice? Why?

Extend the example to a shell that reads commands from the user and executes them

- Handle all cases of why a process can terminate, and print a useful message for all cases

**CSC322**

**C Programming and UNIX**

**Process Creation and Termination (II)**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# Creating new Processes: fork()

The **only** way of creating a new process under UNIX is via the `fork()` function

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork(void);
```

`fork()` creates a new child process that is in nearly all ways an exact **copy** of the parent

Execution continues in both parent **and** child

Only (major) differences:

- New PID and new parent PID
- Return value of `fork`

Return value of `fork()`

- On failure: `-1`, `errno` will be set
- On success:
  - \* In the child, `0` will be returned
  - \* In the parent, the PID of the child (a value `>0`) will be returned

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    pid_t pid, ppid, child_pid;
    int some_var = 42;

    pid = getpid();
    printf("Parent. My PID is %d and I am about to procreate\n", pid);
    child_pid = fork();
    if(child_pid<0)
    {
        err_sys("fork");
    }
}
```

## Example

```
if(child_pid == 0)
{
    pid = getpid();
    ppid = getppid();
    printf("Child. My PID is %d, my parent is %d\n", pid, ppid);
    printf("Child: some_var=%d - Changing it now!\n", some_var);
    some_var=7;
    printf("Child: some_var=%d\n", some_var);
}
else
{
    printf("Parent. My PID is %d, my child is %d\n", pid, child_pid);
    printf("Parent: some_var=%d\n", some_var);
    printf("Going to sleep now, waiting for my child to die...\n");
    sleep(5);
    printf("I'm awake again. some_var is still %d\n", some_var);
}
return EXIT_SUCCESS;
}
```

## Example Output

```
Parent. My PID is 12625 and I am about to procreate
Parent. My PID is 12625, my child is 12626
Parent: some_var=42
Going to sleep now, waiting for my child to die...
Child. My PID is 12626, my parent is 12625
Child: some_var=42 - Changing it now!
Child: some_var=7
I'm awake again. some_var is still 42
```

Notice that I took a snapshot of the processes with top:

PID	USER	PRI	...	SHARE	STAT	%CPU	%MEM	TIME	COMMAND
12625	schulz	16	...	280	S	0.0	0.1	0:00	fork_example
12626	schulz	16	...	0	Z	0.0	0.0	0:00	fork_example <defunct>

- As long as the parent lives, the child remains around as a **zombie**
- As the parent dies, the `init` process gets the termination status and is delivered from its undead state

## Comments on `fork()`

Order of execution for parent and child is unpredictable!

Forked processes **behave** as if an actual copy has been made

- All of the processes memory is accessible in both parent and child
- Changing them in one does not affect the other

On modern UNIX versions, `fork()` is implemented with **copy on write**

- Both processes actually **share** the same pages in memory
- Only when a process actually tries to change a value in memory is a private copy created
- Consequence: Forking is very cheap – it only has to copy basic process structures

UNIX programmers use forking a lot!

- Servers may fork one process for each connection!
- Shells fork for executing commands

# Don't Do This!

```
#include <unistd.h>

int main(int argc, char* argv[])
{
    while(1)
    {
        fork();
    }
}
```

# Don't Do This!

```
#include <unistd.h>

int main(int argc, char* argv[])
{
    while(1)
    {
        fork();
    }
}
```

It is the simplest version of a **fork bomb**

- Will create an exponentially growing number of processes
- Quickly consumes all system resources
- Makes system essentially unusable

# Forking and I/O

As the example showed, both parent and child were able to write to `stdout`

- In general, parent and child share file descriptors open at the time of `fork()`
- This can be problematic, as the order in which output is written is undefined
- Even worse for input or output to files or sockets (on the screen, we can usually figure things out)

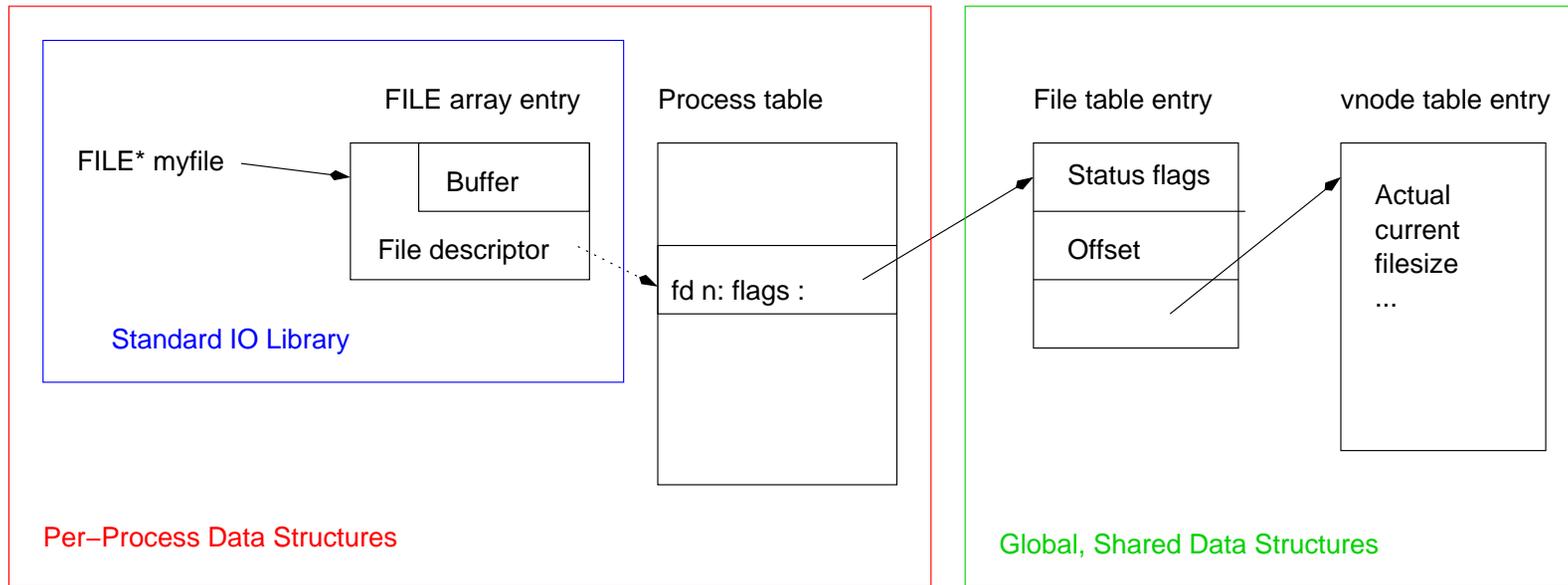
If responsibility for file descriptors is clear, parent can delegate communication to child

- Example: Parent just accepts() connections
- Child actually performs communication on the file descriptor
- Both parent and child need to close an open file descriptor!

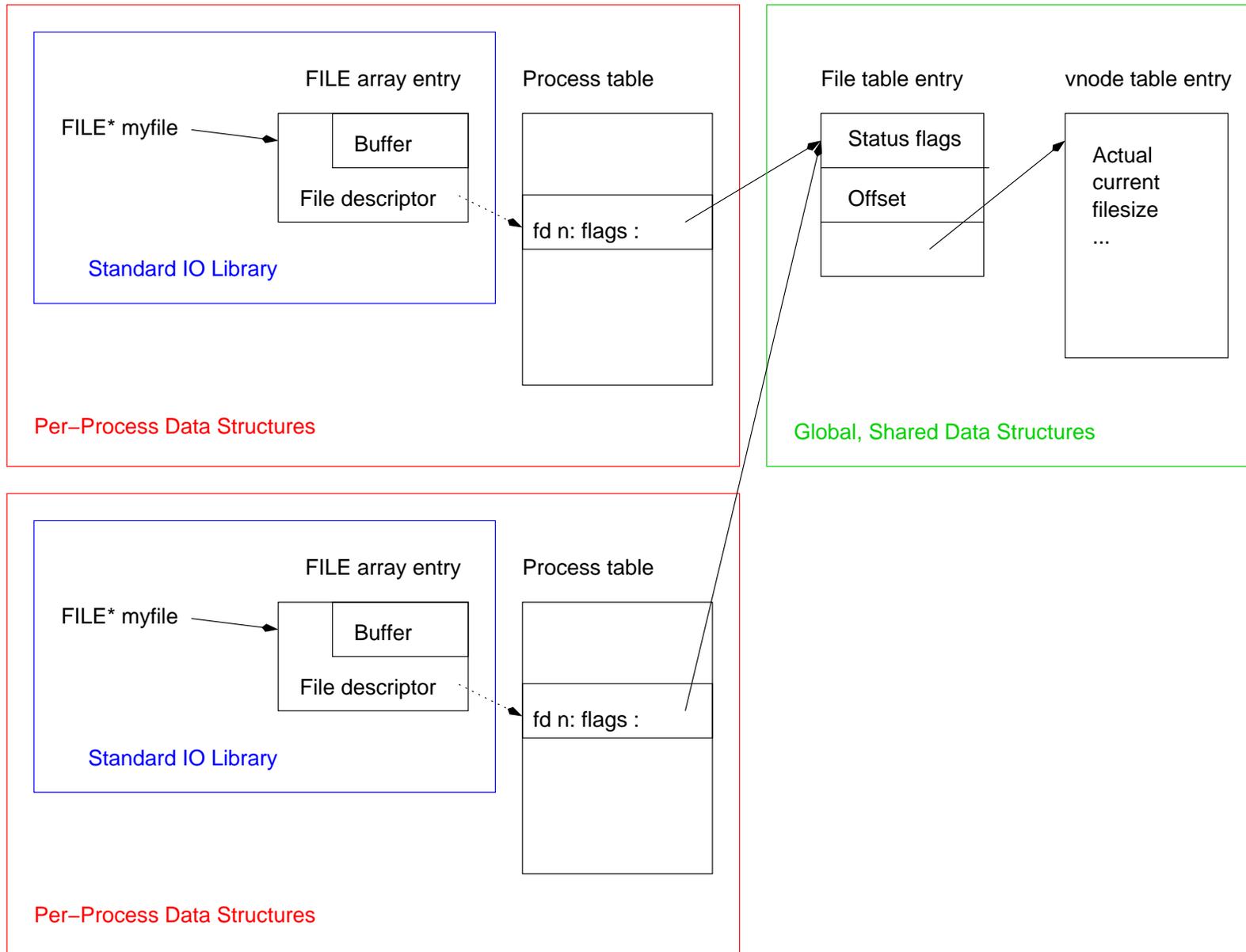
Parent and child share file descriptor, but not standard I/O library buffers

- Can have unexpected effects!

# I/O Setup before Forking



# I/O Setup after Forking



## Example: Bufferd I/O and Forking

```
/* Usual includes and stuff omitted */
int main(int argc, char* argv[])
{
    pid_t child_pid;

    printf("Hiho "); /* <--- Note: No Newline! */
    child_pid = fork();
    if(child_pid<0)
    {
        err_sys("fork");
    }
    if(child_pid == 0)
    {
        printf("from the child!\n");
    }
    else
    {
        printf("from the parent!\n");
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

## Example Output

```
$fork_example2
```

```
Hiho from the parent!
```

```
Hiho from the child!
```

`stdout` is line buffered

- Since we did not print a full line (and did not call `flush()`, the string was not printed
- Calling `fork()` duplicated the buffer contents
- Then, both parent and child caused a flush

# Waiting for Children to Die

As stated above, parents need to get the termination status of their children (otherwise those children become zombies)

They can do so by calling `wait()`

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

- `wait()` waits until a child terminates
- It returns the PID of the terminated child
- If `status` is not equal to `NULL`, it writes the termination status of the child into the variable it points to
- Note: If some children have already terminated, `wait()` picks one of those and returns its data
- If there are no children, `wait()` returns `-1` and sets `errno`

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    pid_t pid, ppid, child_pid;
    int i, status;

    pid = getpid();
    printf("Parent. My PID is %d and I am about to procreate\n", pid);
    fflush(stdout);
```

## Example (2)

```
for(i=0; i<3; i++)
{
    child_pid = fork();
    if(child_pid<0)
    {
        err_sys("fork");
    }
    if(child_pid == 0)
    {
        break; /* Only the parent forks! */
    }
}
if(child_pid == 0)
{
    pid = getpid();
    ppid = getppid();
    printf("Child. My PID is %d, my parent is %d\n", pid, ppid);
    sleep(1);
    exit(i);
}
```

## Example (3)

```
printf("Parent: Waiting for my children\n");
while((child_pid = wait(&status))!=-1)
{
    printf("Child %d terminated with termination status %d\n", child_pid, status);
    if(WIFEXITED(status))
    {
        printf("Termination normal, exit status %d\n", WEXITSTATUS(status));
    }
}
return EXIT_SUCCESS;
}
```

## Example Output

Output:

```
Parent. My PID is 13565 and I am about to procreate
Child. My PID is 13567, my parent is 13565
Child. My PID is 13568, my parent is 13565
Child. My PID is 13569, my parent is 13565
Parent: Waiting for my children
Child 13569 terminated with termination status 512
Termination normal, exit status 2
Child 13568 terminated with termination status 256
Termination normal, exit status 1
Child 13567 terminated with termination status 0
Termination normal, exit status 0
```

## Exercises

Here is a function that computes the **rollercoaster numbers**

```
long rollercoaster(long i)
{
    printf("%ld\n", i);
    if(i==1)
    {
        return 0;
    }
    if(i%2==0)
    {
        return 1+rollercoaster(i/2);
    }
    return 1+rollercoaster(3*i+1);
}
```

Write a program that forks of 10 processes, each of which computes the rollercoaster numbers for one of the numbers from 11 to 20 and prints it

Make the parent wait for all children and print the PID's and the exit status of each in the order in which the children terminate, then terminate the parent

**CSC322**  
**C Programming and UNIX**  
**Process Control (System Calls)**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# Process Groups

UNIX processes are organized in **process groups**

- A process group has a **group leader**
- All processes in the group have the same **process group id** (which is the **process id** of the group leader)

Some operations can be done not just for single processes, but for a whole group:

- Delivering signals with `kill`
- Waiting for process termination with `waitpid()` (later)

By default, a process inherits the process group id from its parent

- Processes can change their own process group id
  - \* . . . to become process group leaders in a new process group, or
  - \* . . . to join an existing process group
- Parents can change the process group id of their children (unless the children already called `exec()`)

Note: Don't confuse the **pgid** (process group) with the **gid** (user/owner group)

# Getting and Changing Process Groups

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
int  setpgid(pid_t pid, pid_t pgrp);
```

`getpgrp()` always returns the process group id of the current process

- No error condition!

`setpgid(pid_t pid, pid_t pgrp)` sets the process group id of the process with the PID `pid` to `pgrp`

- Return value: 0 on success, -1 on error (`errno` set)

- Special values:

- \* If `pid` is 0, the PID of the calling process is assumed

- \* If `pgrp` is 0, the process id denoted by the first argument is assumed (i.e. that process is made into a process group leader of a new process group)

- Note that this means that `setpgid(0,0)` makes the current process into a process group leader

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    pid_t pid, pgid, child_pid;
    int i, res;

    pid = getpid();
    pgid = getpgrp();
    printf("Parent. My PID is %d and my process group is %d\n",pid,pgid);
```

## Example (2)

```
res = setpgid(0,0);
if(res== -1)
{
    err_sys("setpgid");
}
printf("Parent. I'm now the process group leader.\n");

for(i=0; i<3; i++)
{
    child_pid = fork();
    if(child_pid<0)
    {
        err_sys("fork");
    }
    if(child_pid == 0)
    {
        break; /* Only the parent forks! */
    }
}
```

## Example (3)

```
if(child_pid == 0)
{
    pid = getpid();
    pgid = getpgrp();
    printf("Child %d. My PID is %d, my process group is %d.\n", i, pid, pgid);
    sleep(1);
    res = setpgid(0,0);
    if(res== -1)
    {
        err_sys("setpgid");
    }
    pid = getpid();
    pgid = getpgrp();
    printf("Child %d. I'm now independent, pid %d and pgid %d\n",i, pid,pgid);
    printf("Child %d exiting\n", i);
    exit(EXIT_SUCCESS);
}
printf("Parent, sleeping.\n");
sleep(3);
printf("Parent, exiting.\n");
return EXIT_SUCCESS;
}
```

## Example Output

```
$ ./pg_example
```

```
Parent. My PID is 1946 and my process group is 1946
```

```
Parent. I'm now the process group leader.
```

```
Parent, sleeping.
```

```
Child 0. My PID is 1947, my process group is 1946.
```

```
Child 1. My PID is 1948, my process group is 1946.
```

```
Child 2. My PID is 1949, my process group is 1946.
```

```
Child 0. I'm now independent, pid 1947 and pgid 1947
```

```
Child 0 exiting
```

```
Child 1. I'm now independent, pid 1948 and pgid 1948
```

```
Child 1 exiting
```

```
Child 2. I'm now independent, pid 1949 and pgid 1949
```

```
Child 2 exiting
```

```
Parent, exiting.
```

Note that the parent **starts out** as a process group leader!

- Most shells with build-in job control will always execute commands in their own process group

# UNIX System Call: kill

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

`kill()` sends the signal `sig` to the process or processes specified by `pid`

- `pid > 0`: Signal is sent to process with PID `pid`
- `pid == 0`: Signal is sent to all processes in the same process group (if process has permission to send it)
- `pid < 0`: Signal is sent to all processes with **process group id** `|pid|`
- Special case: `pid == -1`: Most UNIX versions send signal to all processes with the same user id (real or effective) as the caller

Possible signals: As for the `kill` command (defined in `<signal.h>`)

- Also see `man signal`

Note: `kill()` is the function used to implement the `kill` command

## Is this good for Something?

There are many possible situations where an application consists of a **set** of processes:

- Server may have one process that `accepts()` connections, multiple workers that serve individual connections
- Competitive theorem prover runs many search strategies in parallel

If we make the top level control program into a process group leader, termination becomes a lot easier

- We can kill whole process group with one command
- The leader can be made to automatically kill all processes

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
    pid_t pid, pgid, child_pid;
    int i, res;

    res = setpgid(0,0);
    if(res==-1)
    {
        err_sys("setpgid");
    }
}
```

## Example (2)

```
pid = getpid();
pgid = getpgrp();
printf("Queen bee:PID is %d process group is %d\n",pid,pgid);

for(i=0; i<3; i++)
{
    child_pid = fork();
    if(child_pid<0)
    {
        err_sys("fork");
    }
    if(child_pid == 0)
    {
        break; /* Only the parent forks! */
    }
}
```

## Example (3)

```
if(child_pid == 0)
{
    while(1)
    {
        printf("Worker bee %d gathering honey\n", i);
        sleep(1);
    }
}
for(i=0; i<3; i++)
{
    printf("Queen bee sleeping\n");
    sleep(1);
}
printf("Queen bee terminates\n");
kill(-getpgrp(), SIGTERM); /* Commented out for version 2 */
return EXIT_SUCCESS;
}
```

## Example Output with kill

Queen bee:PID is 2412 process group is 2412

Queen bee sleeping

Worker bee 0 gathering honey

Worker bee 1 gathering honey

Worker bee 2 gathering honey

Queen bee sleeping

Worker bee 0 gathering honey

Worker bee 1 gathering honey

Worker bee 2 gathering honey

Queen bee sleeping

Worker bee 0 gathering honey

Worker bee 1 gathering honey

Worker bee 2 gathering honey

Queen bee terminates

# Example Output **without** kill

```
schulz@leonardo 4:31am [CSC_322] ./pgkill_example
Queen bee:PID is 2460 process group is 2460
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee terminates
schulz@leonardo 4:32am [CSC_322] Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
...
```

## Waiting for Termination: `waitpid()`

The `wait()` function waits for termination of **any** child of a process

- It blocks until a child terminates
- It cannot check the status of a specific child

POSIX introduced `waitpid()` as a more general interface solving this problem:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t wpid, int *status, int options);
```

## waitpid() continued

Return value: PID of terminated child (or 0 if no child terminated, or -1 on error)

wpid: Process id describing processes we are waiting for

- wpid == -1: Wait for all processes
- wpid > 0: Wait for process with PID wpid
- wpid < -1: Wait for all processes in process group with PID |wpid|
- wpid == 0: Wait for all children with PGID of the caller

status: As for wait(), if !=NULL, termination status is written into it

options: (Can be combined with |)

- 0: Normal blocking wait
- WNOHANG: Return immediately with 0 if no child is available
- WUNTRACED: Used for job control and stopped processes

# Exercises

Write a program that keeps a network server alive (or reanimates it):

- The server accepts connections
- For each connection, it forks a child that reads input from the net and appends it to a log file
- All those processes should be in the same process group

The monitor program just starts the main server process, makes it the group leader, and waits for the server to terminate

- In that case, it kills all of the server processes and restarts the server

**CSC322**  
**C Programming and UNIX**  
**Program Execution**

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# Process Environment

Each UNIX process has an **environment**

- the Environment consists of a list of strings
- Normally, those strings have the form `name=value` (and most functions for manipulating the environment assume this form)
- The `name` is called an **environment variable**
- Since most environment variables are created and maintained by the shell, they are often also called **shell variables**

Children inherit the environment of their parents

- Note that children get a **copy** of the environment
- Each process can change its own environment, but not that of its parent

Environment variables are used for a large number of things

- Where to look for executable programs
- Which editor to use (in well-written applications)
- What is the users username?
- Some mandated by standards (POSIX, SUSv2), others just customary

# Environment and the Shell

You can print the environment using the `printenv` program

- Just `printenv` prints all environment variables (and their values)
- `printenv <name>` prints the value of the variable with name `<name>`

Since no process can modify its parents environment, you need to use a build-in command to change a shells environment

- `tcsh`: `setenv VAR VALUE` and `unsetenv VAR`
- `bash`: `export VAR=VALUE` and `unset VAR`

# Example: Part of my Environment

```
$ printenv
```

```
PWD=/home/schulz/SOURCES/CSC_322
```

```
VENDOR=intel
```

```
HOSTNAME=wombat
```

```
QTDIR=/usr/lib/qt3-gcc2.96
```

```
LESSOPEN=|/usr/bin/lesspipe.sh %s
```

```
USER=schulz
```

```
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=01;32
```

```
MACHTYPE=i386
```

```
XDM_MANAGED=/var/run/xdmctl/xdmctl-:0,maysd,mayfn,sched
```

```
XMODIFIERS=@im=none
```

```
EDITOR=emacsclient
```

```
LANG=C
```

```
HOST=wombat
```

```
DISPLAY=:0.0
```

```
FROM=Stephan Schulz <schulz@cs.miami.edu>
```

```
LOGNAME=schulz
```

```
SHLVL=3
```

```
GROUP=schulz
```

```
TEXINPUTS=~ /TEXT/TEXLIB/
```

```
SUPPORTED=en_US.iso885915:en_US:en:de_DE@euro:de_DE:de
```

```
SHELL=/bin/tcsh
```

```
HOSTTYPE=i386-linux
```

```
CVSROOT=stephan@gw.safelogic.se:/CVS
```

```
OSTYPE=linux
```

```
HOME=/home/schulz
```

```
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
```

```
PATH=/home/schulz/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:/usr/X11R6/bin:.
```

```
_=/usr/X11R6/bin/xterm
```

```
TERM=xterm
```

```
WINDOWID=18874382
```

## Some Important Environment Variables

PATH (POSIX) determines where the shell looks for executable programs

- List of directory names, separated by colon
- Can contain . to include working directory (**Dangerous** on multi-user systems)

EDITOR (traditional) is used by good UNIX program to determine which editor to run if you have to edit text

LOGNAME (POSIX) is your user name

TERM (POSIX) is your (text) terminal type

- If you have trouble with remote logins, set it to vt100

HOME (POSIX) is your home directory

DISPLAY (X11 Window System) is the name of your display

- UNIX can run programs on one host, and display them on another
- DISPLAY tells it where to show output for X programs

# Accessing the Environment from a Program

There are two ways to access the environment of a process:

- Via the `environ` variable
- Via `getenv()` and `putenv()`

If we want to go through all of the environment, we need to declare the `environ` variable:

```
extern char **environ;
```

- It points to a NULL-terminated array of pointers
- Each array element points to \0-terminated C string of the form `<name>=<value>`

## Example

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

extern char **environ;

int main(int argc, char* argv[])
{
    char **handle;

    for(handle=environ; *handle; handle++)
    {
        printf("%s\n", *handle);
    }
    return EXIT_SUCCESS;
}
```

# The POSIX Interface to the Environment

```
#include <stdlib.h>
char *getenv(const char *name);
int  putenv(char *string);
```

`getenv()` takes a pointer to an environment variable name and returns its value (or NULL if the variable does not exist)

- It's even part of ANSI C (but ANSI C says nothing about the environment)

`putenv()` takes a single string of the form `<name>=<value>`

- Adds the string (i.e. the `<name>=<value>` pair) to the environment
- If `<name>` exists, the old definition is changed
- Note that some versions of UNIX include just the pointer in the environment, while others create a copy of the string

Additional functions of interest:

- `clearenv()`: Clears environment (POSIX, but not traditional)
- `unsetenv()`: Remove a single variable (traditional)
- `setenv()`: More flexible version of `putenv()` (traditional)

# Executing New Programs

A process can cause the execution of a new program via one of the exec functions

- Causes this same process to replace its own program, data, and stack with new data
- Program code is loaded from disk
- Heap and stack are reinitialized
- New program starts running at its `main()` function

There are 6 different exec functions that differ in:

- How they look for the program to run (via `path` or via absolute filename)
- How they accept arguments for the new program (as additional arguments to the exec function or via an array of pointers)
- How they handle the environment (inheritance of completely new environment)

## The 6 exec Functions

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg , ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [], char *const envp[]);
```

All return -1 on error, and not at all on success

execlp() and execvp() take a filename and search the PATH directories for the program

execl(), execlp() and execl\_e() take arguments for the new program as additional arguments

- The list has to end with an additional NULL argument
- The others take a pre-created argv vector

Finally, execl\_e() and execve() take an explicit environment pointer

## The `execvp()` function

`execvp(const char *file, char *const argv[])` is reasonably easy to use:

- First argument is a file name (not containing any /)
- The program to be executed is found as by the shell, by looking through all the directories in `PATH`

Second argument is a pointer to an array of argument pointers

- Same format and conventions as `argv` in `main`
- First argument should be program name
- Array should be `NULL` terminated

Upon execution, the new program runs

- Keeps old `PID`, `GID`, `PGID`, working directory, . . .
- Normal file descriptors stay open (unless the the flag `FD_CLOEXEC` is set using `fcntl()`)

## Example: A mini-Shell

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

#define MAX_LINE 1024

void err_sys(char* message)
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

## Example (2)

```
void* secure_malloc(int size)
{
    void* res = malloc(size);

    if(!res)
    {
        fprintf(stderr, "malloc() failure -- out of memory?");
        exit(EXIT_FAILURE);
    }
    return res;
}

char* secure_strdup(char* source)
{
    void* res = secure_malloc(strlen(source)+1);

    strcpy(res, source);
    return res;
}
```

## Example (3)

```
int count_words(char* line)
{
    int words=0, in_word=0;
    while(*line)
    {
        if(isspace(*line))
        {
            in_word = 0;
        }
        else
        {
            if(in_word == 0)
            {
                words++;
                in_word = 1;
            }
        }
        line++;
    }
    return words;
}
```

## Example (4)

```
char **build_argv(char* line)
{
    int argc = count_words(line);
    int i;
    char *new;
    char **argv;

    if(argc == 0)
    {
        return NULL;
    }
    argv = secure_malloc(sizeof(char*)*(argc+1));
```

## Example (5)

```
for(i=0; i<argc; i++)
{
    while(isspace(*line))
    {
        line++;
    }
    for(new=line; *new && !isspace(*new); new++);
    /* Empty body */
    *new = '\0';
    argv[i] = secure_strdup(line);
    line = new+1;
}
argv[i] = NULL;
return argv;
}
```

## Example (6)

```
void print_argv(char **argv)
{
    int i;

    printf("Command: %s\n", argv[0]);
    printf("Arguments:\n");
    for(i=0; argv[i]; i++)
    {
        printf("%s\n", argv[i]);
    }
    printf("=====\n");
}
```

## Example (7)

```
int main(void)
{
    pid_t  child_pid;
    char   line[MAX_LINE];
    char   *line_res;
    char  **argv;

    while(1)
    {
        printf("# ");fflush(NULL);
        line_res = fgets(line, MAX_LINE, stdin);
        if(!line_res)
        {
            break;
        }
        argv = build_argv(line);
        if(!argv)
        {
            continue;
        }
        print_argv(argv);
    }
}
```

## Example (8)

```
child_pid = fork();
if(child_pid == -1)
{
    err_sys("fork");
}
if(child_pid == 0) /* Child! */
{
    setpgid(0,0);
    if(execvp(argv[0], argv) == -1)
    {
        err_sys("execvp");
    }
}
else
```

## Example (9)

```
{ /* Parent */
    setpgid(child_pid, child_pid);
    if(wait(NULL) == -1)
    {
        err_sys("wait");
    }
    free(argv);
}
return EXIT_SUCCESS;
}
```

# Example Usage

```
schulz@wombat 2:01am [CSC_322] ./shell_example
```

```
# echo Hallo
```

```
Command: echo
```

```
Arguments:
```

```
echo
```

```
Hallo
```

```
=====
```

```
Hallo
```

```
# ls -l macrotest.c wordcount env_example.c
```

```
Command: ls
```

```
Arguments:
```

```
ls
```

```
-l
```

```
macrotest.c
```

```
wordcount
```

```
env_example.c
```

```
=====
```

```
-rw-rw-r-- 1 schulz schulz 233 Dec 3 21:31 env_example.c
```

```
-rw-rw-r-- 1 schulz schulz 206 Nov 26 23:41 macrotest.c
```

```
-rwxrwxr-x 1 schulz schulz 13715 Nov 26 23:47 wordcount
```

## Example Usage (2)

```
# ls *
```

```
Command: ls
```

```
Arguments:
```

```
ls
```

```
*
```

```
=====
```

```
ls: *: No such file or directory
```

```
# hallo
```

```
Command: hallo
```

```
Arguments:
```

```
hallo
```

```
=====
```

```
execvp: No such file or directory
```

## Exercises

Extend the shell example (code is on the web page) to

- Have better error handling
- Do background processing (with `&`)
- Support job control
- Offer I/O redirection with `>` and `<`

Read the man pages on `popen()` and `pipe()` to see how we could achieve piping

If you are adventurous, implement:

- Piping
- Globbing (read man `glob`)

# CSC322

## C Programming and UNIX

### Final Review

Stephan Schulz

Department of Computer Science

University of Miami

`schulz@cs.miami.edu`

`http://www.cs.miami.edu/~schulz/CSC322.html`

Prerequisites: CSC220 or EEN218

# Final Examn

Place and Time:

- Room LC 192 (the normal room)
- Monday, Dec. 16th, 11:00 a.m. – 13:30 p.m.

Topics:

- Everything we covered in this class
- Emphasis will be on second half

You may bring:

- Lecture notes, your own notes, books, printouts of your (or my solutions)to the exercises. . .
- . . . but no computers, PDAs, mobile phones (switch them off and stow them away) or similar items

Note: I'll only review material from the second half of the semester today

- Check lecture notes, pages 299–312 for overview of first half

# Pointers and Dynamic Arrays

Arrays are passed as pointers to the first element

- Arrays and pointers (to an allocated memory region) can be used in the same way (i.e. we can index a pointer: `p[5]`)
- We can use `realloc()` to dynamically enlarge dynamically allocated arrays

Pointer arithmetic: We can add and subtract integers to pointers to step through an array

- `p[5]` is equivalent to `*(p+5)`

The following two program snippets are equivalent:

```
int a[SIZE], i;                                int a[SIZE], *handle;

/* Assume some initialization in both versions */

for(i=0; a[i]; i++)                             for(handle = a; *handle; handle++)
{
    printf("%d\n", a[i])                       {
                                                printf("%d\n", *handle)
                                                }
}
```

# Make

Make is a tool for automating multi-program builds

- Rule-based (rules are stored in Makefile)
- Performs just the necessary operations to update all program parts
- You specify **dependencies** and **actions**

Example:

```
PROGS=hello fahrenheit2celsius fahrenheit2celsius2 fahrenheit2celsius3 \  
      charcount ourcopy wordcount escape base_converter inc_example
```

```
all: $(PROGS)
```

```
clean:
```

```
    rm $(PROGS)
```

```
hello: hello.c
```

```
    gcc -ansi -Wall -o hello hello.c
```

```
fahrenheit2celsius: fahrenheit2celsius.c
```

```
    gcc -ansi -Wall -o fahrenheit2celsius fahrenheit2celsius.c
```

```
...
```

# New Flow Control Constructs

`break` is used to break out of loops (and `switch` statements)

- Immediately transfers control to the first statement after the loop

`continue` allows early continuation of a loop

- Transfers control back to the beginning of the loop
- In case of `for` loops, update expression will be evaluated

`do/while` loops test the condition at the end of the loop

- Loop body always gets executed once
- Otherwise similar to plain `while` loop

# Function Pointers and `qsort()`

We can use **pointers to functions** (of a specific type) to

- Implement generic functions and data types
- Emulate object-oriented constructs (virtual functions)
- Implement call back and signal handlers

Using function pointer:

- Just use the function name or use the address operator (`&fun`)
- Calling the function: Either use pointer as is, or dereference: `(*fun)(arg1)`

Example for function pointer usage: `qsort()` from `stdlib`

```
void qsort(void *base, size_t nmemb, size_t size,  
           int(*compar)(const void *, const void *));
```

# Standard Library: Characters and Strings

`ctype.h` contains character classification functions:

- `isspace(c)`
- `isprint(c)`
- `isdigit(c)`
- `isalpha(c)`
- `isalnum(c) ...`
- Also: `toupper(c)`, `tolower(c)`

String (`\0` terminated sequence of characters) functions are defined in `string.h`

- `strcpy(to,from)` copies a `\0`-terminated string **to existing memory**
- `strcat(to,from)` appends a string at the end of an existing string
- `strcmp(s1,s2)` compares two strings, returns value `<0`, `0`, `>0`
- `strncpy()`, `strncat()`, `strncmp()` limit operation to a given number of characters
- `strpbrk()` searches for characters in a string
- `strstr()` searches for a substring

# Standard Library: Memory Accesses

Memory access functions treat memory as a large array of characters

- Important difference to string functions: Not `\0`-terminated, you **always** have to give a length

Functions:

- `memcpy(to, from, n)` copies `n` bytes
- `memmove(to, from, n)` does the same even for overlapping regions of memory
- `memcmp(s1, s2, n)` compares two memory regions
- `memchr(s, c, n)` searches for character `c` in memory region starting at `s`
- `memset(s, c, n)` writes `n` copies of character `c` into memory (used e.g. to zero out socket address data structures)

# Standard Library: Buffered I/O

Standard library supports **buffered IO** via **streams**

- Stream creation: `fopen(filename, mode)`
- Stream destruction: `fclose(stream)`
- Predefined streams: `stdin`, `stdout`, `stderr`
- Text streams: Lines separated by `\n`
- Binary streams: Raw data (under UNIX, no difference)

Basic I/O functions:

- `fgetc(stream)` reads a single character and returns it as an `int` (and EOF on end of file)
- `fputc(c, stream)` writes a single character to a stream
- `fgets(s, n, stream)` reads a single line or `n` characters (whichever is less) into the **preallocated** memory at `s`
- `fputs(s, stream)` writes a `\0`-terminated string to the stream

Streams can be `fflush()`ed, and we can change buffering with `setvbuf()` and `setbuf()`

# Standard Library: Formatted Output

`printf(format,...)` and `fprintf(stream, format, ...)` write an arbitrary number of arguments under the control of a **format string**

- Format string contains plain characters and **conversion specifiers** starting with a %
- Each conversion specifier must have a matching argument
- Conversion specifiers specify in which form argument is printed

Conversion specifier format:

- %, followed by optional flags, field width, precision, size modifier
- Ends in a conversion letter

Example: `printf("%-5ld\n", i)`

- Prints integer, at least 5 characters, left-justified (fills up with spaces), followed by a newline

Important conversion letters: d (int), s (string), c (character), g (floating point number)

# Processes and Signals

Processes are running programs and have a number of properties

- Owner, PID, GID, PGID, Parent
- Each process has its own virtual memory and cannot (directly) access other processes data
- Multiple processes can run “at the same time”

We can use a number of tools to work with running processes:

- `ps` lists running processes
- `top` gives an interactive view of running processes

`kill <pid>` can be used to send signals to process `<pid>`

- By default sends `SIGTERM`
- You can also send other signals, e.g. `kill -HUP <pid>`

Signals can also be generated by other events, e.g.

- Floating point exception
- Illegal memory access

# Signal Handling

Each signal has a **default action** (either abort, abort with core dump, or ignore)

- Action can be changed!

The `signal(sig, handler)` function can be used to change the behaviour of a process to a signal

- `sig` is the signal to respond to
- `handler` is a pointer to a function that returns `void` and takes an `int` (the signal) as an argument
- Predefined pseudo-handlers: `SIG_DFL` (re-establish default behaviour), `SIG_IGN` (ignore signal)

Established signal handlers catch a single signal!

- Must reestablish handler from within the handler

Signals can occur at any time, state of the program may be undefined

- It's dangerous to do much beyond exiting, manipulating variables of type `volatile sig_atomic_t`, and calling `signal()` again

# UNIX File System (I)

UNIX: Everything is a file

File types:

- Regular file
- Directory
- Character special file
- Block special file
- Socket
- Symbolic link

`stat()` functions give us information about files

- Owner
- Mode
- Size
- Access and modification times

# UNIX File System (II)

Important concepts:

- File ownership and group ownership
- Access rights (read, write, execute for user/group/others)
- Links: Connect a name to a file
  - \* Hard links: Directory entries
  - \* Soft links: Files with names of another file as data

Important utilities:

- `ln`: Creates links (both symbolic and hard)
- `ls`: Shows files and file information
- `chmod`: Allows us to change the mode of a file
- `chgrp`: Changes group
- `chown`: Changes owner

## File Descriptors and `select()`

File descriptors are used by the UNIX kernel to represent open files

- File descriptors are small integers (indices into the process file table)
- Can be associated with a number of flags we can manipulate with `fcntl()` or set when we open the file: `O_NONBLOCK`, `O_APPEND`, . . .
- Predefined: `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`
- Opening files: `open()`
- Using files: `read(fd, buf, n)` and `write(fd, buf, n)`
- Closing: `close()`

`select(maxfd, readfds, writefds, exceptfds, time)` waits for certain things to become true for sets of file descriptors

- Any of the file descriptors in `readfds()` is ready for reading
- Any of the file descriptors in `writefds()` is ready for writing
- An exceptional circumstance happens for one of the file descriptors in `exceptfds()`
- Return value: Number of file descriptors for which condition is true
- Also removes all file descriptors from sets for which condition is **not** true

# Networking Concepts

Communication can be

- Broadcast vs. dedicated partners
- Stream-oriented vs. packet-oriented
- Reliable vs. unreliable

Communication partners need to be uniquely identified

- For IP: IP addresses (denote hosts) (4 8 bit numbers, e.g. 127.0.0.1)
- For TCP/IP: IP address and **port** (16-bit integer)

UNIX uses **sockets** (a special kind of file descriptors) for communication

- Bi-directional streams
- Use with `read()` and `write()`

# TCP/IP (v4) Connections

Reliable, stream-oriented, between two partners

Client:

- Create a socket: `socket(PF_INET, SOCK_STREAM, 0)`
- Fill in struct `sockaddr_in` address structure
  - \* `sin_family = AF_INET`
  - \* `sin_port = htons(port)`
  - \* `sin_addr` filled in with `inet_pton()`
- Connect socket to address: `connect(sock, addr, addr_len)`
- Use socket and `close()` it

Server:

- Create socket
- Create its own address (normally with `INADDR_ANY`)
- `bind()`ing the socket to the address
- `listen()`ing on the socket
- `accepting()` the connection (giving a **new** socket)
- Use and close the socket

# Creating and Ending Processes

`fork()` creates new process

- Both parent and child execute the same program

Parent has to `wait()` or `waitpid()` to pick up the child's termination status

- Otherwise child becomes zombie
- But orphans are inherited by `init`

Process termination

- `exit()`
- `return` from `main()`
- Abort (from a signal)

# Process Environment and Program Execution

Processes have access to environment variables

- Inherited from (or set up by) parent
- Can be modified

To start a new program:

- `fork()` to create a new process
- Call one of the exec functions with:
  - \* Executable name (filename or path name)
  - \* Arguments (individual or as array)
  - \* For some functions, environment pointer

# Exercises

Learn hard ;-)