CMPUT 201

Practical Programming Methodology

Instructor: Michael Buro

These are notes based on CMPUT 201 as it was taught at the University of Alberta in the Fall term of 2018. The presented material has been drawn in part from freely available sources, such as Wikipedia and C/C++ tutorial webpages, and notes I collected during years of C/C++ programming and teaching

These notes go beyond the slide shows that are usually presented in course lectures these days in an attempt to replace costly text books, except for exercise material which is not included. The notes might be restructured and certainly will be improved this term

I appreciate suggestions for improvements

– Michael Buro, Edmonton, August 2018

Course Contents

Part 1: Introduction, UNIX

Contents

[DOCUMENT NOT FINALIZED YET]

General Course Information

Week 1

- Section home page:
  - skatgame.net/mburo/courses/201
  - news, schedule, lecture notes, and additional material
- Except for the course newsgroup and assigment submissions we will not be using eClass
- My email address: mburo@ualberta.ca
- Office: ATH 337
- Labs start on September 10. Get a lab account from the helpdesk (first floor CSC) BEFORE the labs start
- Id and password for accessing the course material:

  c201 bar201z

- Prerequisite: CMPUT 115 or 175 or 275 or equivalent knowledge of fundamental algorithms and data structures!

Software Engineering Courses

- 201: Small-scale programming
  - learn about UNIX/C/C++ and software libraries
  - get familiar with software development tools
  - know what goes on "under the hood"
  - design and implement interfaces and small programs
  - learn to appreciate software testing, defensive programming, and code profiling
- 301: Team work, object-oriented design
- 401: Large-scale programming

Want to learn more about C++? Take CMPUT 350: Advanced Game Programming, which covers C++ in-depth

CMPUT 201 Topics

1. Introduction, UNIX Tools
2. The C and C++ Programming Languages (including debugging and profiling)
3. Code modularization and abstract data types (including makefiles)
4. Bit manipulation
5. UNIX file I/O
6. Parallel programming with POSIX threads
7. Object oriented and generic programming, C++ Standard Template Library

How to succeed in CMPUT 201?

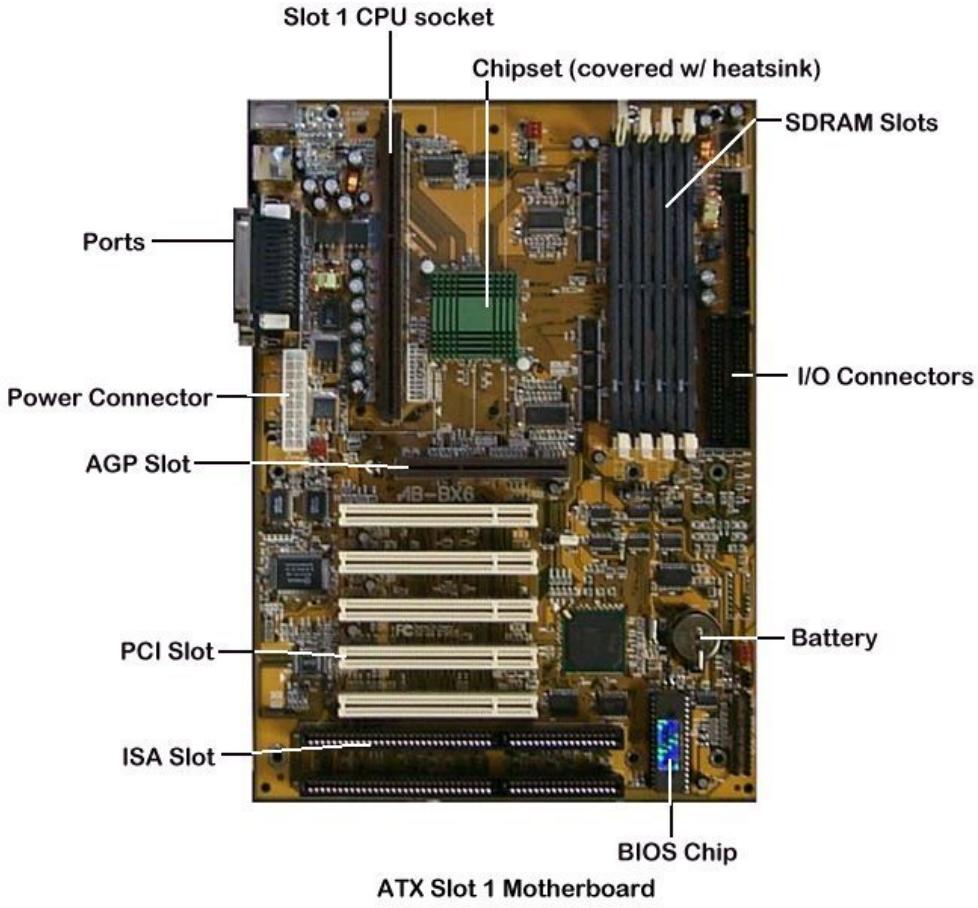
- "Learning by doing"
- Don't hesitate to play around – it's hard to do any permanent damage if you create backups or use a version control system (e.g., GIT)
- Write many small programs to test new concepts
- Learn to find answers for yourself
  - manual pages, online tutorials
  - online textbooks
  - google compiler error messages
- Learn to use debuggers and profilers

How to fail/drown in CMPUT 201?

- Skipping lectures or labs and office hours
- Ignoring reading assignments
- Not taking advantage of asking questions in labs
- Starting with programming prior to thinking about the problem and trying to make programs work by applying random changes
- Wasting considerable time by not learning how to use tools like editors, debugger, and profilers
- Writing lots of code before compiling it for the first time. You will be swamped by pages of error messages which are hard to decypher. Instead, write and test code top down, compiling frequently in the process

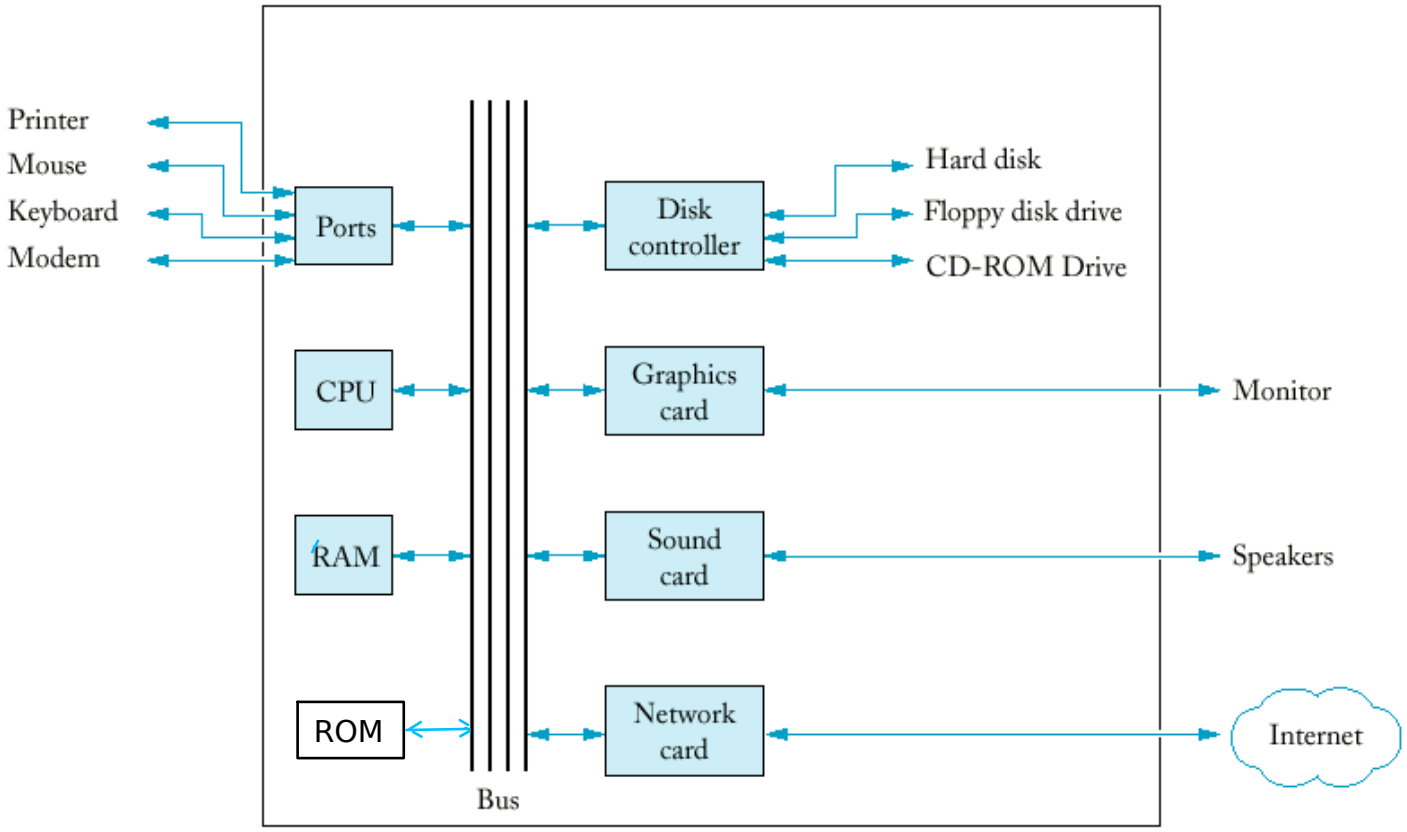
Motivation

Typical Personal Computer (PC) Mainboard



Flexible design: slots for central processing unit (CPU), memory (RAM), and expansion cards such as video or high-end audio cards

Schematic Design of a PC

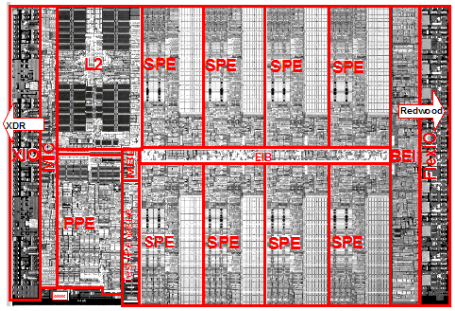


CPU = Central Processing Unit ("Brain of the computer", manages data flows from connected devices to and from memory)

ROM = Read-Only Memory (doesn't lose content when power switched off, contains start-up code)

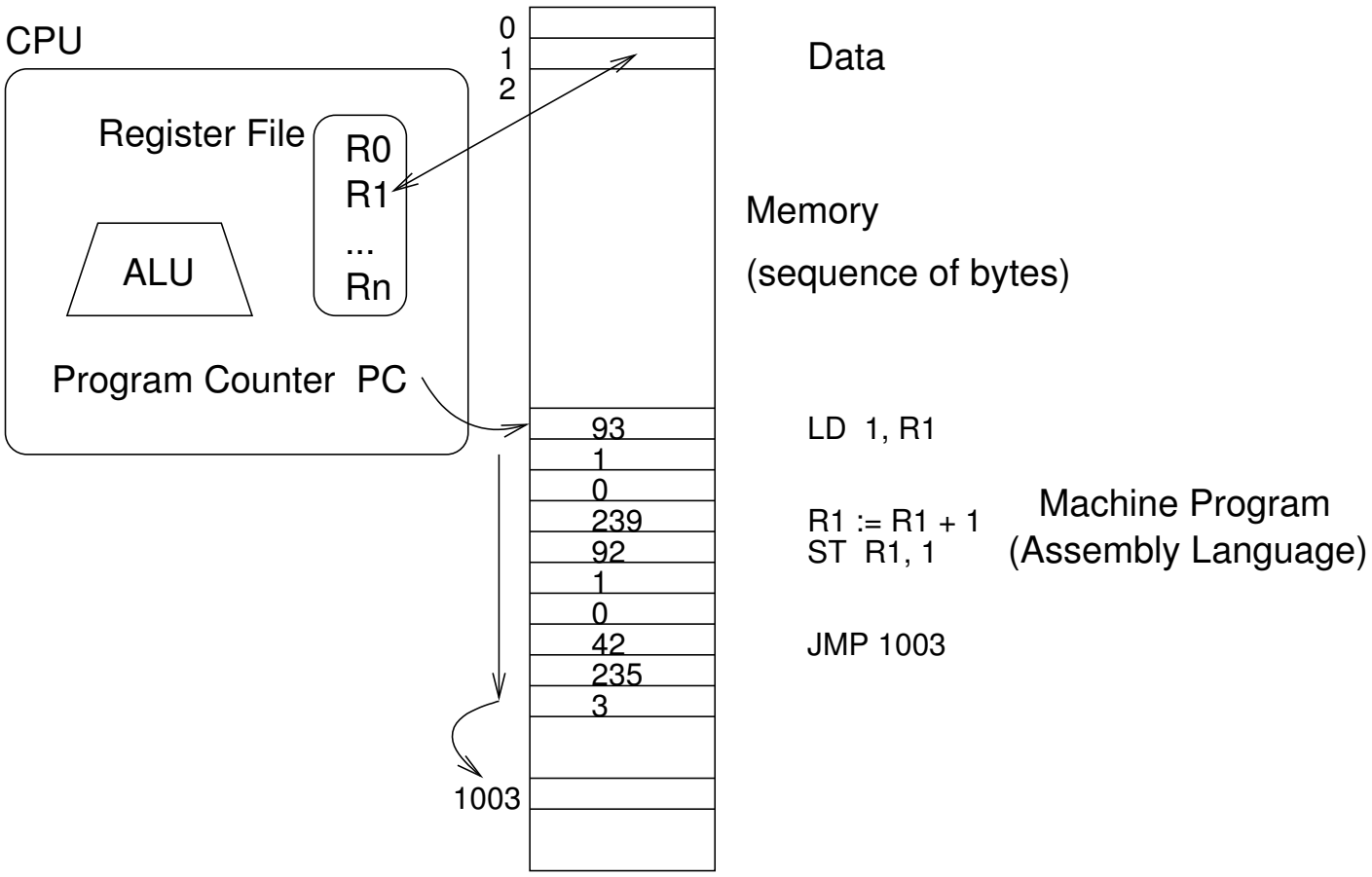
RAM = "Random Access Memory" (misnomer: better would be "read/write" memory, loses content when switching power off, is filled with code/data from harddrive when booting and launching applications)

Central Processing Unit (CPU)



CPUs nowadays contain multiple execution cores which run programs in parallel at a speed of over a billion machine instructions per second each

von Neumann Computer Architecture



Memory is a sequence of numbers between 0 and 255 (8-bit values called "bytes")

Each byte in memory has a unique address

Machine code and data are indistinguishable — both are just sequences of bytes

CPUs contain registers (fast temporary memory) and execute machine code that is pointed to by a so-called program counter (PC), which advances after each step

Each machine instruction by itself doesn't do much, but execution is FAST (billions of instructions per second)

Typical machine instructions read values from memory into a register, manipulate register values using an Arithmetic Logic Unit (ALU), store values back, or change the PC (jump instructions)

For instance, in the example shown earlier

```
LD 1, R1    // loads a word stored at address 1 into register R1
R1 := R1+1  // adds 1 to R1 and writes the result back to R1
ST R1, 1    // stores the content of R1 in memory at address 1
JMP 1003    // program execution resumes at address 1003
```

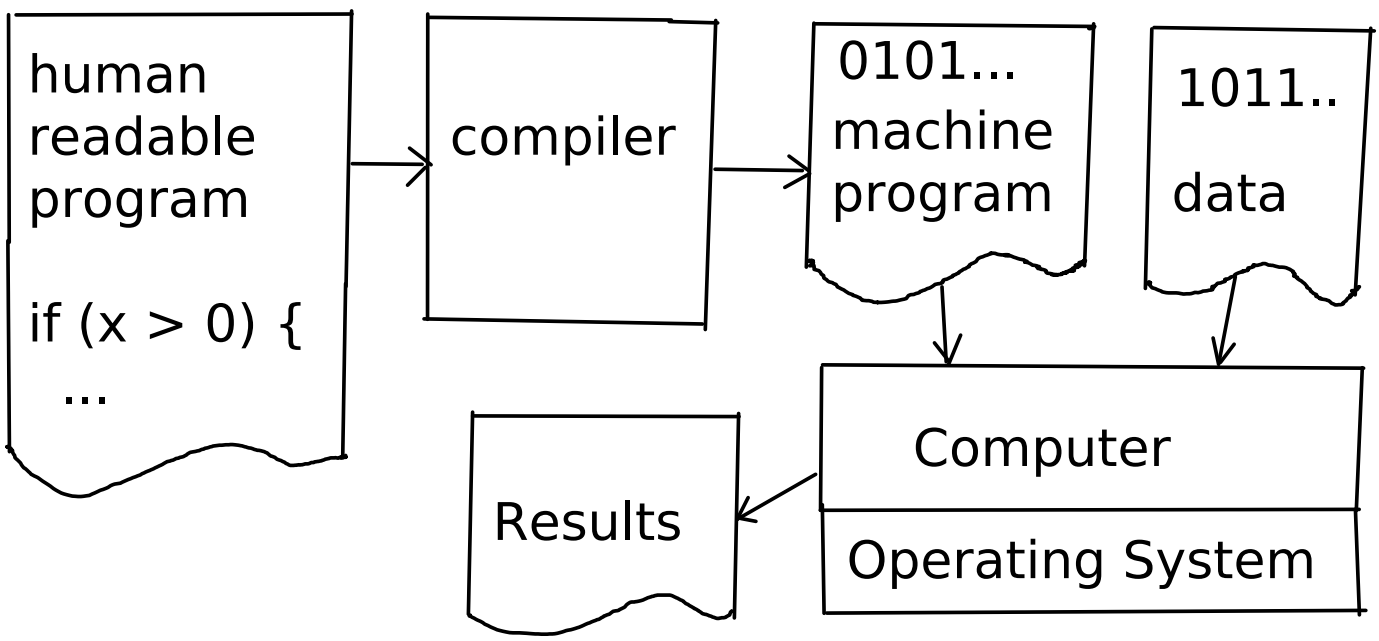
Problems:

How to deal with a large variety of machine architectures and machine codes?

We want to write programs at a more abstract level that is easier to read

Also, programs ought to run on different machines without changes

Solution:

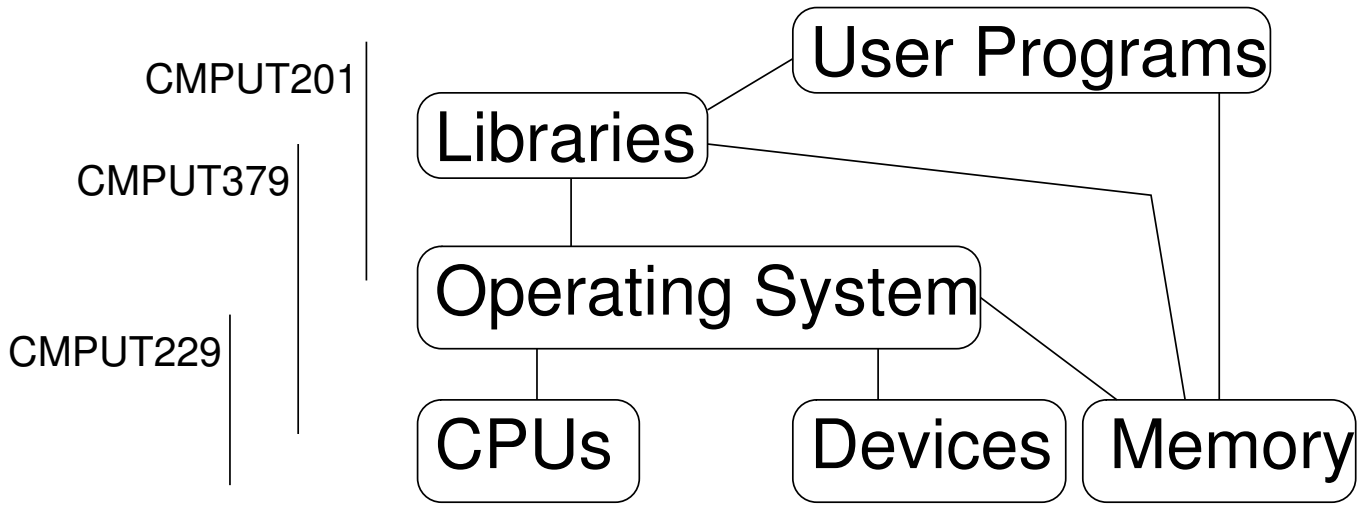


Design operating systems that abstract hardware peculiarities by means of device drivers that interact with hardware and provide a uniform programming interface

Write programs in high-level programming languages that get translated into particular machine code by a program called compiler

So, instead of having to produce N machine code versions of our program for N different machine architectures, we only write our program once and let the N different compilers (that somebody else wrote ☺) translate our program

Software/Hardware Layers



In this course we take a look at the operating system UNIX and the high-level programming languages C and C++

Linux is a free software UNIX variant. Android is based on Linux

C is sometimes called the "lingua franca" of computing, because it is ubiquitous

Many other high-level languages such as Perl, Python, and Java have C interfaces, because C allows us to create very efficient programs

C++ is the better C. Among other things it supports object oriented and generic programming

The UNIX Operating System, Shell, Editor

Why UNIX?

- Open standards (e.g. POSIX threads)
- Dominant server operating system
- Free versions available (FreeBSD, OpenBSD, Linux)
- Many free software development tools:
  gcc, emacs, gprof, gdb, gawk, make, etc.
- Multi-tasking
  (multiple programs can run at the same time)
- Multi-user
  (multiple users can work on the same machine)

We will be using Linux in the labs. You need to install a UNIX variant on your computer if you want to work at home or attend CSC B2 labs

Linux

I highly recommend to administer your own Linux system at home. There are many freely downloadable distributions. E.g.

- Fedora (`fedoraproject.org`)
  Frequent updates. Up-to-date software packages.
  Requires a separate partition on your harddrive

- Fedora live CD or USB stick
  (`fedoraproject.org/wiki/FedoraLiveCD`)
  Does not require any changes in your setup!
  Great for checking Linux out

You can also run Linux in a virtual machine under Windows or OS/X, or install Linux subsystem on Windows 10, or simply connect to a lab computer from home using an SSH client

See the Lab 00 section on software installation, and "Virtual Fedora 20 Machine" and "How to work from home?" on page "Course Material" for details

Getting Started

Two ways of accessing a UNIX computer:

1. Sitting in front of it and typing in a command window
2. Connecting to it from a remote machine using ssh ("secure shell")

   ```
   ssh ug01.cs.ualberta.ca
   ```

Both require you to provide a userid and password

In what follows I will present a quick UNIX introduction/refresher. We recommend you spending the first two weeks of the course to get familiar with the commandline terminal, UNIX commands, and a text editor

Command Shell

- In interactive mode, shells are command line interfaces (text window with keyboard attached to it) e.g. "xterm"
- Issue operating system or internal shell commands directly via keyboard input; e.g.
  - ls (list directory contents) `ls -lrt`
  - cd (change directory) `cd workdir`
  - mv (move/rename) `mv old-file new-file`
  - mkdir (make directory) `mkdir AS1`
  - cp (copy file or directory) `cp -r dir backup`
  - rm (remove file or directory) `rm -rf dir`
  - cat (display file) `cat text`
  - echo (display string) `echo hello`
  - exit (quit shell) `exit`
  - man (command info) `man man`
  - apropos (keyword search) `apropos root`

UNIX File System

Data is stored in file systems which are usually located on harddisks or faster solid-state-drives (SSDs)

Persistent: data isn't lost when computer is switched off (unlike RAM)

In UNIX such data is organized as follows:

- Hierarchical structure (tree)
- / represents the root directory
- Directories ("folders") can contain other directories and files (internal nodes)
- Files (leaves) are just sequences of bytes
- Files/directories are uniquely located by a directory path. E.g. `/home/user/AS1/foo.c`
- / is also used as directory separator

Shell continued

- Special directories:
  - / root directory, everything is stored beneath
  - . current directory
    `cp ./foo ./bar` = `cp foo bar`
  - .. parent directory `cd ../..` : 2 levels up
  - ~ home directory `cd ~/foo` `cd` = `cd ~`
- Command history/editing
  - use arrow keys to navigate, <delete> or <backspace> keys to remove characters
- Simple programming language
  - variables, functions, command aliases
- Startup code in `~/.bashrc` (when shell=bash)
  - customizations!

```
function ll() { ls -l "$@"; }
```

Launching Programs

- Type program name (+ parameters separated by spaces) and hit the return key `<ret>`

  ```
  ls<ret>
  emacs foo.c<ret>
  ```

- Shell interprets the first word as command name and tries to locate a function definition with this name (see `~/.bashrc`). If this fails it searches in the directories listed in variable `$PATH` (try `echo $PATH`)

- To detach a program from the terminal to run it in background type:

  ```
  command &<ret>
  ```

  which is equivalent to

  ```
  command<ret><ctrl-z>bg<ret>
  ```

  (launches command, then suspends it, and then sends to the background)

Wildcards

`*` matches all strings

`?` matches one character

Examples:

- `wc *.c`
  count the words in all files in the current directory with names that end with `.c`
- `ls foo?bar`
  list all filenames that start with `foo`, followed by an arbitrary character and `bar`

Hidden Files

Files with names starting with `.` are hidden, they are not listed nor matched by wildcards

This is why `ls` does not show `.` nor `..`

Useful for avoiding clutter
(e.g. many resource files `.*rc` in `~`)

`ls -a` reveals them ("all")

Filename Completion

Many shells have a filename completion feature: when hitting the <tab> key the shell tries to complete the filename. Saves typing!

```
cat super<tab>
```

will complete the command to

```
cat supercalifragilisticexpialidocious
```

if this is the only filename starting with `super`

Input/Output Redirection

Output of programs can be stored in a file using >:

```
cat file1 file2 > file3
```

[writes content of files `file1` and `file2` to `file3`]

Generates error message if `file3` already exists

Use `>!` to override

```
cat > foo
```

[copy keyboard input ended by `<ctrl-d>` to file foo]

Input can also be redirected:

```
grep foo < text
```

[display all lines in file `text` that contain `foo`]

Or both: `sort < file > file.sorted`

Pipes

Powerful UNIX feature: output of commands can become input for subsequent commands

```
grep aaa file | wc -l
```

[count the number of lines in file that contain aaa]

```
sort file | uniq | wc -l
```

[count the number of unique lines in file]

Edit Textfiles

- Several good editors exist: `emacs`, `vim`, `...`
- We recommend `emacs`, it's powerful!
- Type `emacs x <ret>` to edit file `x` in a separate window.
- To edit within the terminal window, launch emacs with `emacs -nw x <ret>`
- Large number of commands bound to keys. E.g.
  - `<ctrl-x> <ctrl-s>` : save buffer
  - `<ctrl-x> <ctrl-f>` : load file
  - `<ctrl-x> <ctrl-c>` : exit
  - `<ctrl-s>` : search
  - `<alt-%>` : search and replace
  - `<ctrl-x> 2` : split window; `<ctrl-x> o` : switch buffer
  - `<alt-x>` command : launch external commands such as gdb, gnus
- `man emacs`, emacs reference cards, emacs tutorial (in help menu or on the web, see "Course Resources")
- Highly customizable: `emacs ~/.emacs`

More Details

Lab 00:

- UNIX commands
- emacs and bash customization
- editing and compiling C++ programs

First C Program

- Create file hello.c using emacs and save it

```
#include <stdio.h>

int main()
{
  printf("hello world\n");
  return 0;
}
```

- g++ -o hello hello.c generates executable hello which prints hello world in the terminal window and positions the cursor in the following line after being invoked with ./hello
- Without the -o hello option, g++ creates executable file a.out. man g++ explains g++ options
- g++ is the GNU C++ compiler, which expects C++ programs. It also usually works with C programs, because normally C programs are also C++ programs. gcc only compiles C programs
- You can also use the clang,clang++ compilers. Using both gcc and clang can speed up development because error messages are sometimes cryptic