Week 4 - Friday

COMP 2400

Last time

- What did we talk about last time?
- Recursion

Questions?

Project 2

Scope

Scope

- The **scope** of a name is the part of the program where that name is visible
- In Java, scope could get complex
  - Local variables, class variables, member variables,
  - Inner classes
  - Static vs. non-static
  - Visibility issues with **`public`**, **`private`**, **`protected`**, and default
- C is simpler
  - Local variables
  - Global variables

Local scope

- Local variables and function arguments are **in scope** for the life of the function call
- They are also called **automatic variables**
  - They come into existence on the stack on a function call
  - Then disappear when the function returns
- Local variables can **hide** global variables

Global scope

- Variables declared outside of any function are **global variables**
- They exist for the life of the program
- You can keep data inside global variables between function calls
- They are similar to static members in Java

```
int value;

void change() {
     value = 7;
}

int main() {
     value = 5;
     change();
     printf("Value: %d\n", value);
     return 0;
}
```

Use of global variables

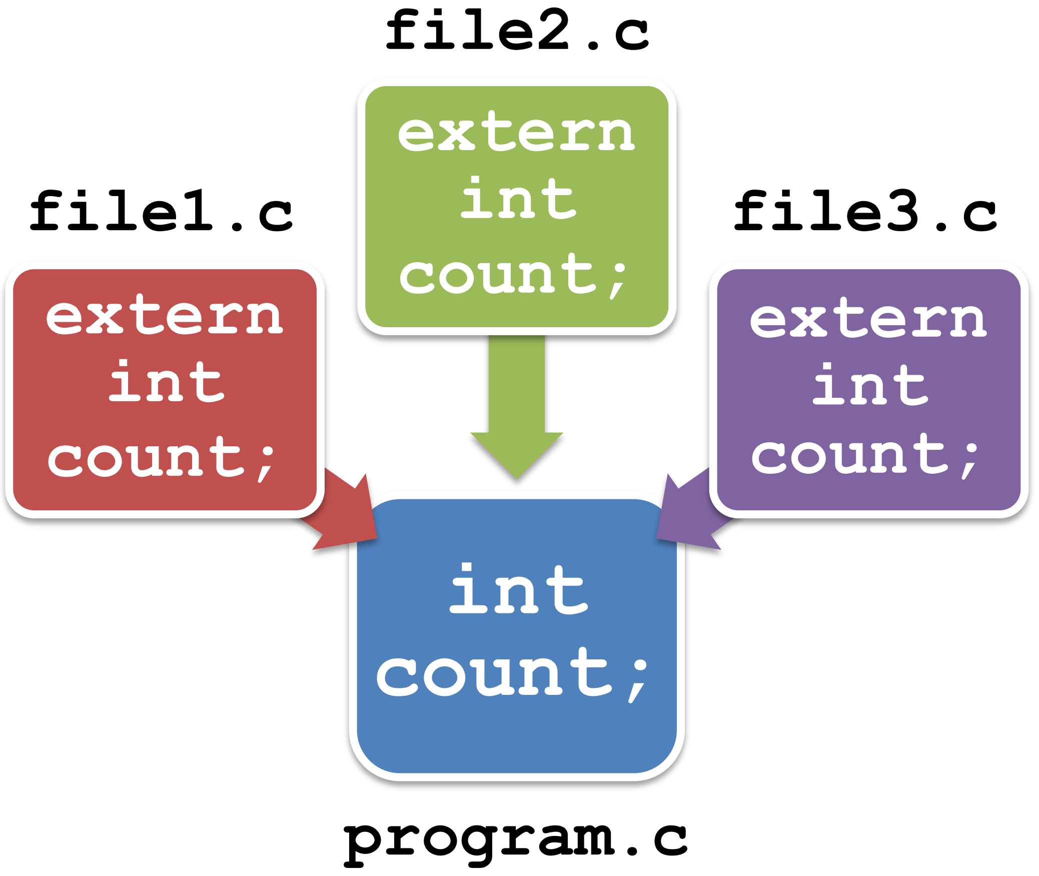
- Global variables should **rarely** be used
- Multiple functions can write to them, allowing inconsistent values
- Local variables can hide global variables, leading programmers to think they are changing a variable other than the one they are
- Code is much easier to understand if it is based on input values going into a function and output values getting returned

Hiding

- If there are multiple variables with the same name, the one declared in the current block will be used
- If there is no such variable declared in the current block, the compiler will look outward one block at a time until it finds it
- Multiple variables can have the same name if they are declared at different scope levels
  - When an inner variable is used instead of an outer variable with the same name, it **hides** or **shadows** the outer variable
- Global variables are used only when nothing else matches
- Minimize variable hiding to avoid confusion

`extern` declarations

- What if you want to use a global variable declared in another file?
- No problem, just put **`extern`** before the variable declaration in your file
- There should only be one true declaration, but there can be many **`extern`** declarations referencing it
- Function prototypes are implicitly **`extern`**



`static` declarations

- The `static` keyword causes confusion in Java because it means a couple of different (but related) things
- In C, the `static` keyword is used differently, but also for two confusing things
  - Global `static` declarations
  - Local `static` declarations

Global `static` variables

- When the **`static`** modifier is applied to a global variable, that variable cannot be accessed in other files
- A global **`static`** variable cannot be referred to as an **`extern`** in some other file
- If multiple files use the same global variable, each variable must be **`static`** or an **`extern`** referring to a single real variable
  - Otherwise, the linker will complain that it's got variables with the same name
- A **`static`** function is one that is also only visible in its own file

Local `static` variables

- You can also declare a **`static`** variable local to a function
- These variables exist for the lifetime of the program, but are only visible inside the function
- Some people use these for bizarre tricks in recursive functions
- Try not to use them!
  - Like all global variables, they make code harder to reason about
  - They are not thread safe

Local static example

```
#include <stdio.h>

void unexpected() {
     static int count = 0;
     count++;
     printf("Count: %d", count);
}

int main() {
     unexpected(); //Count: 1
     unexpected(); //Count: 2
     unexpected(); //Count: 3
     return 0;
}
```

The `register` modifier

- You can also use the **`register`** keyword when declaring a local variable

```
register int value;
```

- It is a sign to the compiler that you think this variable will be used a lot and should be kept in a register
- It's only a suggestion
- You can not use the reference operator (which we haven't talked about yet) to retrieve the address of a register variable
- Modern compilers are usually better at register allocation than humans

Systems Programming

Kernel

- When people say OS, they might mean:
  - The whole thing, including GUI managers, utilities, command line tools, editors and so on
  - Only the central software that manages and allocates resources like the CPU, RAM, and devices
- For clarity, people use the term **kernel** for the second meaning
- Modern CPUs often operate in kernel mode and user mode
  - Certain kinds of hardware access or other instructions can only be executed in kernel mode

What does the kernel do?

- Manages processes
  - Creating
  - Killing
  - Scheduling
- Manages memory
  - Usually including extensive virtual memory systems
- File system activities (creation, deletion, reading, writing, etc.)
- Access to hardware devices
- Networking
- Provides a set of system calls that allow processes to use these facilities

Shells

- A **shell** is a program written to take commands and execute them
  - Sometimes called a **command interpreter**
  - This is the program that manages input and output redirection
- By default, one of the shells is your **login shell**, the one that automatically pops up when you log in (or open a terminal)
- It's a program like any other and people have written different ones with features they like:

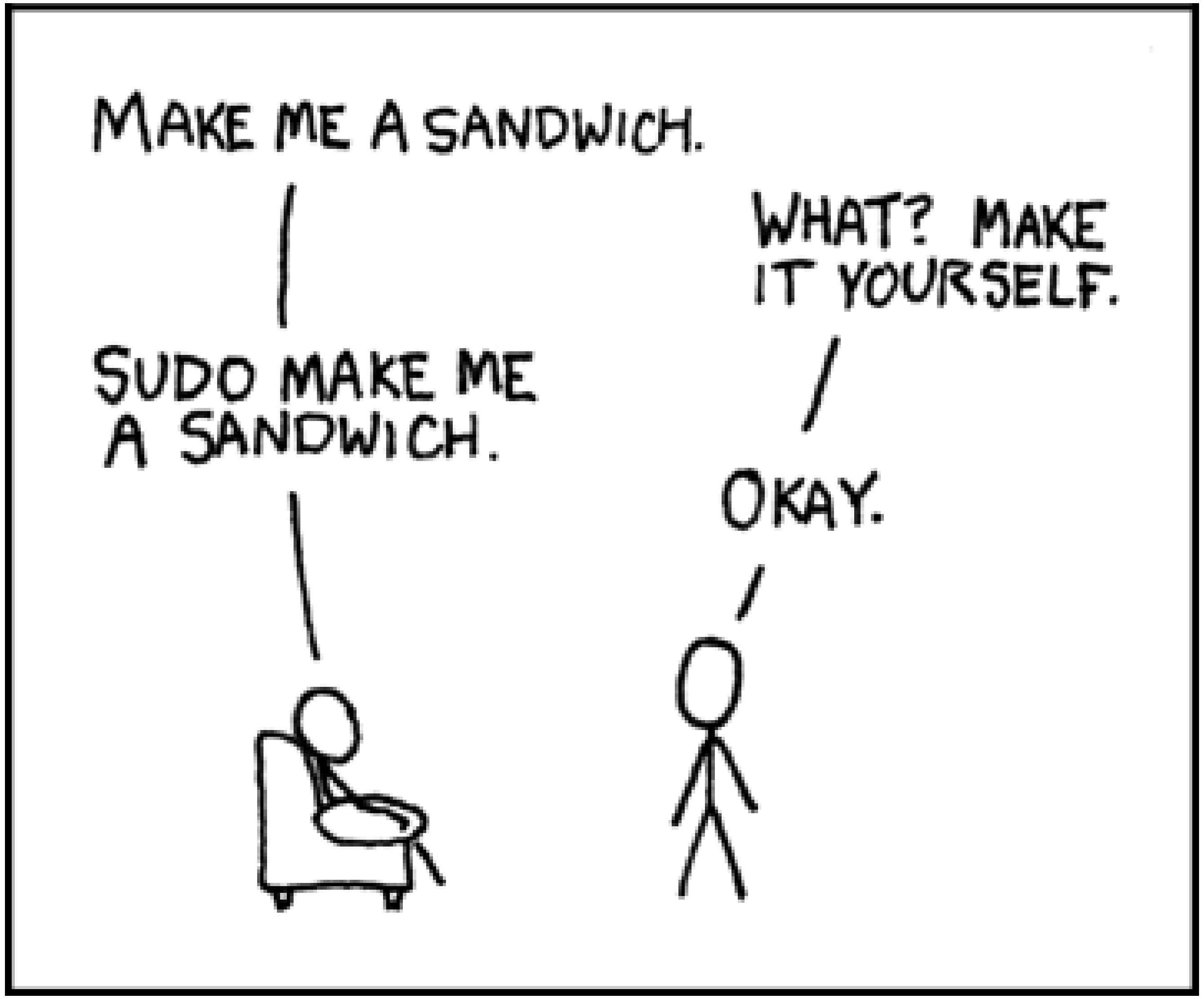
`sh`	The original Bourne shell
`csh`	C shell
`ksh`	Korn shell
`bash`	Bourne again shell, the standard shell on Linux

Users and groups

- On Linux, every user has a unique login name (user name) and a corresponding numerical ID (UID)
- A file (**`/etc/passwd`**) contains the following for all users:
  - Group ID: first group of which the user is a member
  - Home directory: starting directory when the user logs in
  - Login shell
- Groups of users exist for administrative purposes and are defined in the **`/etc/group`** file

Superusers

- The **superuser** account has complete control over everything
- This account is allowed to do anything, access any file
- On Unix systems, the superuser account is usually called **`root`**
- If you are a system administrator, it is recommended that you do not stay logged in as root
  - If you ever get a virus, it can destroy everything
- Instead, administrators should log in to a normal account and periodically issue commands with elevated permission (often by using **`sudo`**)


MAKE ME A SANDWICH.
WHAT? MAKE IT YOURSELF.
SUDO MAKE ME A SANDWICH.
OKAY.

Single file system

- In Windows, each drive has its own directory hierarchy
  - `C:` etc.
- In Linux, the top of the file system is the **root directory /**
  - Everything (including drives, usually mounted in `/mnt`) is under the top directory
  - `/bin` is for programs
  - `/etc` is for configuration
  - `/usr` is for user programs
  - `/boot` is for boot information
  - `/dev` is for devices
  - `/home` is for user home directories

Files

- There are regular files in Linux which you can further break down into data files and executables (although Linux treats them the same)
- A **directory** is a special kind of file that lists other files
- **Links** in Linux are kind of like shortcuts in Windows
  - There are **hard links** and **soft links** (or **symbolic links**)
- File names can be up to 255 characters long
  - Can contain any ASCII characters except **/** and the null character **\0**
  - For readability and compatibility, they *should* only use letters, digits, the hyphen, underscore, and dot
- Pathnames describe a location of a file
  - They can start with **/** making them absolute paths
  - Or they are relative paths with respect to the current working directory

File permissions

- Every file has a UID and GID specifying the user who owns the file and the group the file belongs to
- For each file, permissions are set that specify:
  - Whether the owner can read, write, or execute it
  - Whether other members of the group can read, write, or execute it
  - Whether anyone else on the system can read, write, or execute it
- The **`chmod`** command changes these settings (**`u`** is for owner, **`g`** is for group, and **`o`** is everyone else)
- Example that adds the execute (**`x`**) permission to others (**`o`**) on a file called **`script.sh`**:

```
chmod o+x script.sh
```

File I/O

- All I/O operations in Linux are treated like file I/O
- Printing to the screen is writing to a special file called **`stdout`**
- Reading from the keyboard is reading from a special file called **`stdin`**
- When we get the basic functions needed to open, read, and write files, we'll be able to do almost any kind of I/O

Processes

- A **process** is a program that is currently executing
- In memory, processes have the following segments:

Text	The executable code
Data	Static variables
Heap	Dynamically allocated variables
Stack	Area that grows and shrinks with function calls

- A **segmentation fault** is when your code tries to access a segment it's not supposed to
- A process generally executes with the same privileges as the user who started it

Upcoming

Next time…

- Arrays
- More on makefiles

Reminders

- Read K&R chapter 5
- **Finish Project 2**
  - Due **Monday** by midnight!