

These are the slides for the Advanced C Programming Course on Udemy. They are provided free of charge to all students.

More information about the course: <https://lpa.dev/u1advcp/>

If you have any questions or queries, please add your feedback in the Q&A section of the course on Udemy.

Best regards,

Tim Buchalka
Learn Programming Academy

Advanced C Programming Slides

Main Course Slides.

Welcome to Class!

Jason Fedin

A little bit about myself

- Masters of Science (Computer Science) Binghamton University
- Software Developer (19 Years)
- Online Instructor (15 years)
 - Instructed over 20 different classes, everything from Object-Oriented Programming to Compiler Theory
- Working with the C programming language for over 20 years
- Created beginner C and advanced classes at multiple universities
- Software Developer at Xerox Corporation
- Mainly concentrate on Object Oriented Languages and Mobile Programming
- Focus on writing High Quality Code



Overview

- this class is the second class in my series of C programming classes here at Udemy
 - focuses on more intermediate/advanced concepts of C
- first class was “C programming for Beginners”
 - focused on basic C programming concepts
- it is not required to enroll in the first class if you already have basic knowledge of the C programming language
- if you are new to “C” programming, I would suggest that you first enroll in my “C programming for Beginners” class

Topics

- storage classes
 - auto, register, static, and extern
- advanced data types
 - typedef, variable length arrays, flexible array numbers, complex number types
- type qualifiers
 - const, volatile, and restrict
- bit manipulation
 - binary numbers and bits
 - bitwise operators (logical and shifting)
 - bitmasks and bitfields

Topics (cont'd)

- Advanced control flow
 - goto, null, comma operator
 - setjmp and longjmp
- more on Input and Output
 - getchar, putchar, fgets, etc.
 - puts, sprint, fprintf, fflush
- advanced function concepts
 - variadic functions (variable number of arguments)
 - recursive functions
 - inline functions

Topics (cont'd)

- unions
 - overview, defining and accessing union members
- advanced preprocessor concepts
 - #define, #pragma, #error, #, ##
 - conditional compilation (#ifdef, #endif, #else, #elif, #undef, etc)
 - include guards
- macros
 - overview (vs. functions, when to use)
 - predefined macros
 - creating your own macros

Topics (cont'd)

- advanced debugging and compiler flags
 - debugging with the pre-processor, more on gdb
 - core files, getting the stack trace
 - static analysis and profiling
- static libraries and shared objects
 - overview, creation, dynamic loading
- working with larger programs
 - dividing your program into multiple files and compiling multiple files
- advanced pointers
 - double pointers (pointers to pointers)
 - function pointers
 - more on void pointers

Topics (cont'd)

- useful C libraries
 - the assert library
 - general utilities library (stdlib.h), (exit, atexit, qsort, memcpy, abort)
 - date and time functions
- data structures
 - Linked lists, stacks, queues, and Trees
- Inter-process communication (unix based using Cygwin)
 - overview (message queues, shared memory, piping)
 - forking and signals

Topics (cont'd)

- threads (pthread (posix), not <threads.h> from C11)
 - overview, creating a thread
 - mutexes and semaphores
 - thread management (multi-threading, join, detach)
- networking (unix based using Cygwin)
 - overview (client/server model)
 - creating server and client sockets
- many challenges, solutions, and examples
- organized around theory and many demonstrations
- hands on coding

Course Outcomes

- you will be able to write advanced C programs
- you will be able to write efficient, high quality C code
 - modular
 - low coupling
- master the art of problem solving in programming using efficient, proven methods
- you will understand advanced concepts of the C Programming language
- you will have fun!

Class Organization

Jason Fedin

Class Organization

- Lectures are designed around explaining the why and providing the how
 - A complete learning experience
 - Understand the programming language as a whole
 - Lacking in most Udemy courses
- Powerpoint slides
 - a way to present abstract concepts and definitions that I am not able to demonstrate in code (as easily)
 - Not just reading from slides!
 - Providing insight via experience
 - Providing thorough explanations

Demonstrations of code

- Demonstrations in the IDE (Integrated Development Environment)
 - Code examples (concrete, real-world)
 - More practical/hand-on learning
- Challenges (coding assignments/projects)
 - A way for you to assess your own learning
 - Code alongs – walking through the code of a solution for a particular challenge
 - All source code provided in class

The C99 Standard

Jason Fedin

Overview

- we have already discussed some concepts of the C programming language (beginner class) that were included in the C99 standard
 - bool data type – stdbool.h (added true/false keywords)
 - variable length arrays
 - single line comments
- however, we have mainly focused on the C89 standard in the beginner class
- I want it to be clear that some concepts in this class will include functionality from the C99 standard

History

- C99 is a revised standard for the C programming language
 - refines and expands the capabilities of Standard C
- C99 has not been widely adopted
 - not all popular C compilers support it
 - of the compilers that do offer C99 support, most support only a subset of the language
- so, we do not focus on many of the additions added for C99
 - this is why the focus is mainly on C89
- C89 is almost universally supported

New features added that we will use

- macros with a variable number of arguments
- C99 allows the use of sophisticated numbers (complex.h)
 - `_Complex`
 - used to declare complex floating type variables to store mathematical complex numbers
 - `_Imaginary`
 - declare imaginary floating type variables store mathematical imaginary numbers
- designated initializers
 - allow you to initialize the elements of an array, union, or struct explicitly by subscript or name
- restricted pointers
 - a type qualifier that can be used only for pointers

New features added that we will use

- new comment techniques
 - C99 allows to put comment using a double front slash (//)
- inline functions
 - supplies a hint for the compiler to perform optimizations
- variable length arrays
 - array dimension has to be declared in C89
 - C99 permits declaration of array dimensions using integer variables or any valid integer expressions
- flexible array members
 - allows us to declare an array of unspecified length as the last member of a struct

New features added that we will use

- declaration of variables
 - it is now legal to declare variables it at any point of the program within the curly braces of main() function
 - very obvious in loops
- required in C89 (all variables be declared at the start of a block)

```
int i;  
for (i = 0; i < 10, i++)
```
- new in C99 (variable can be declared anywhere)

```
for (int i = 0; i < 10; i++)
```

The C11 Standard

Jason Fedin

Overview

- we have not really focused on any C11 concepts up to this point
- features added to C11 were for more advanced concepts
 - multithreading support
 - safer standard libraries
 - better compliance with other industry standards
- the standard also attempts to fix some issues presented in C99
 - some mandatory features are optional (variable length arrays and complex types)
- also focuses on better compatibility with C++ as much as possible
- I want it to be clear that we may touch on some concepts from C11 in this class, but, it will be minimal

Overview (cont'd)

- C11 standardizes many features that have already been available in common contemporary implementations
- defines a memory model that better suits multithreading
- fixes problems with the C99 standard
 - some of its mandatory features proved difficult to implement in some platforms
 - politics also played a role
 - cooperation between the C and C++ standards committees in the late 1990s was lacking,
 - design mistakes of C99 were avoided in C11
- focuses on security (such as the string manipulation functions and input/output)
 - a new set of safer standard functions that aim to replace the traditional unsafe functions
 - bounds checking functions
 - that begin or end with _s (strcat_s, strcpy_s, etc)
 - removal of the gets function
 - optional to the standard (will not focus on due to most compilers not supporting)

Topics we will address in this class

- support for anonymous structs and unions
 - has neither a tag name nor a typedef name
 - useful when unions and structures are nested
- static assertions
 - evaluated during translation at a later phase than #if and #error
 - let you catch errors that are impossible to detect during the preprocessing phase
- no-return functions
 - declares a function that does not return
 - suppresses compiler warnings on a function that doesn't return
 - enables certain optimizations that are allowed only on functions that don't return

Mutithreading

- the biggest change in C11 is its standardized multithreading support
- C has supported multithreading for decades
 - however, all of the popular C threading libraries have thus far been non-standard extensions, and hence non-portable
- the new C11 header file <threads.h>
 - declares functions for creating and managing threads, mutexes, condition variables
- however, it is widely not supported on windows and thus we will be learning about <pthread.h> instead (posix compliant)

Overview

Jason Fedin

Overview

- Windows
 - C compiler (Cygwin)
 - IDE's (Integrated Development Environment)
 - Code::Blocks
 - Visual Studio Code

Overview

- Mac
 - C compiler (developer tools)
 - IDE's (Integrated Development Environment)
 - Code::Blocks does not work
 - Visual Studio Code
 - xcode

Overview

- Linux
 - Compiler comes installed
 - IDE's (Integrated Development Environment)
 - Code::Blocks
 - Visual Studio Code
 - vi, gedit, compile from command line

Installing the C Compiler

Jason Fedin

Overview

- you need to install the GNU gcc compiler, make, and gdb debugger from cygwin.com
 - simulates a unix environment
- download either the 32 or 64 bit version of the setup file
 - <https://cygwin.com/install.html>
 - setup-x86_64.exe or setup-x86_32.exe

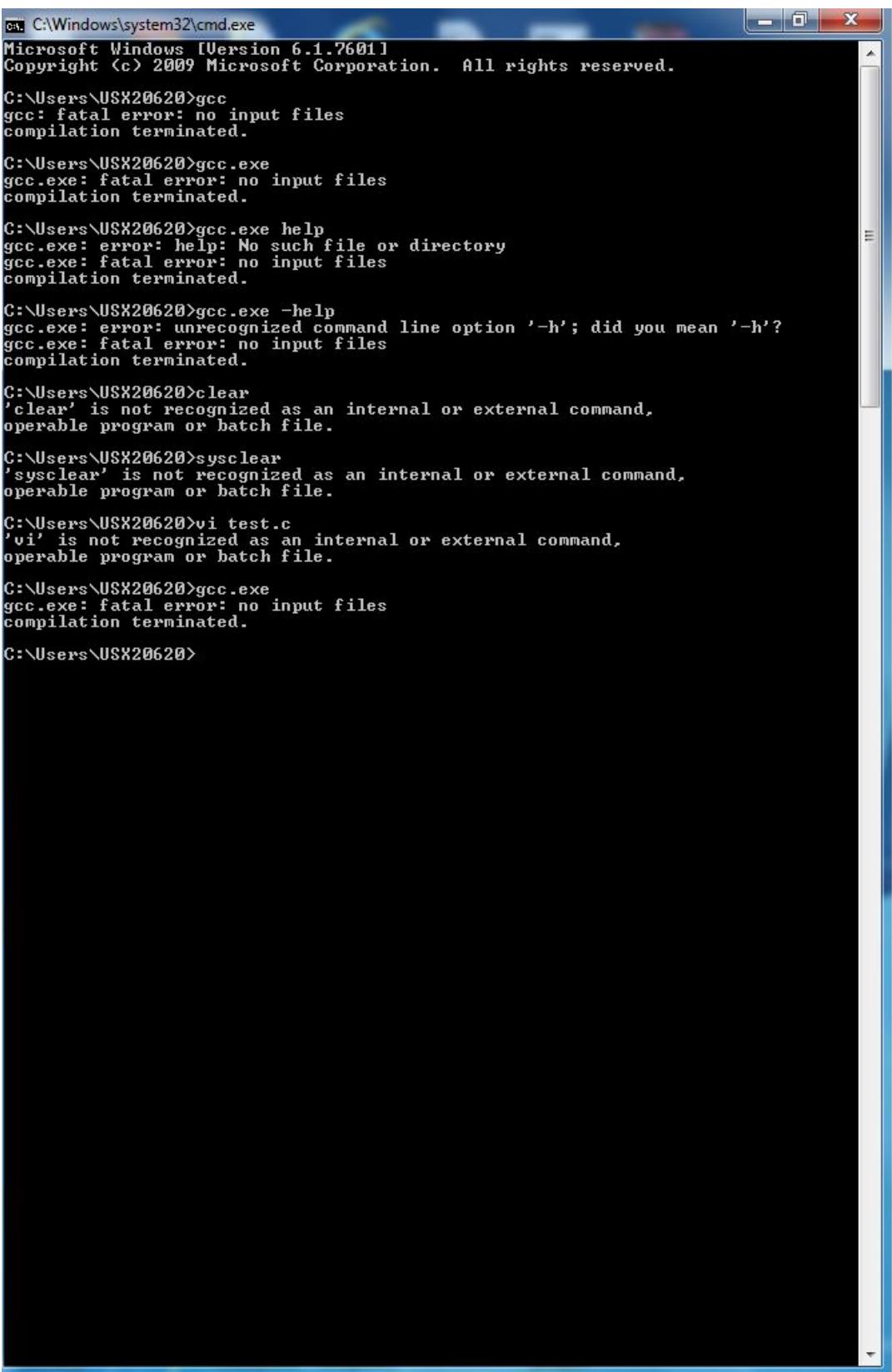
Overview

- run the downloaded Cygwin installer
 - accept the defaults until you reach the Select Your Internet Connection page
 - select the option on this page that is best for you. Click Next.
- on the Choose Download Site page, choose a download site you think might be relatively close to you. Click Next.
- on the Select Packages page you select the packages to download
 - Click the + next to Devel to expand the development tools category
 - You may want to resize the window so you can see more of it at one time
 - Select each package you want to download by clicking the Skip label next to it, which reveals the version number of the package to download
 - at a minimum, select
 - gcc-core: C compiler
 - gdb: The GNU Debugger
 - make: the GNU version of the 'make' utility
 - Packages that are required by the packages you select are automatically selected as well.
- Click Next to connect to the download site and download the packages you selected, and click Finish when the installation is complete

Setting your path up

- Open the Control Panel:
 - On Windows XP select Start > Settings > Control Panel and double-click System.
 - On Windows 7, type var in the Start menu's search box to quickly find a link to Edit the system environment variables
- Select the Advanced tab and click Environment Variables
- In the System Variables panel of the Environment Variables dialog, select the Path variable and click Edit
- Add the path to the cygwin-directory\bin directory to the Path variable, and click OK
 - By default, cygwin-directory is C:\cygwin (for 32 bit Cygwin distribution) or
 - C:\cygwin64 (for 64 bit Cygwin distribution)
 - Directory names must be separated with a semicolon
- Click OK in the Environment Variables dialog and the System Properties dialog.

Verifying the Installation



A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window shows the following command-line session:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\USX20620>gcc
gcc: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe help
gcc.exe: error: help: No such file or directory
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe -help
gcc.exe: error: unrecognized command line option '-h'; did you mean '-h'?
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>clear
'clear' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>sysclear
'sysclear' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>vi test.c
'vi' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>gcc.exe
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>
```

Installing Code::Blocks

Jason Fedin

Code::Blocks Download

- <http://www.codeblocks.org/>

Installing Visual Studio Code

Jason Fedin

Overview

- <https://code.visualstudio.com/docs/editor/whyvscode>
- Required for mac and linux users
- <https://code.visualstudio.com/>
- Optional for windows (Follow the wizard, very straight forward, for installation)

Overview

Jason Fedin

Overview

- Windows
 - C compiler (Cygwin)
 - IDE's (Integrated Development Environment)
 - CodeLite

Overview

- Mac
- C compiler (developer tools)
- IDE's (Integrated Development Environment)
 - CodeLite
 - xcode

Overview

- Linux
 - Compiler comes installed
 - IDE's (Integrated Development Environment)
 - CodeLite
 - vi, gedit, compile from command line

Installing the C Compiler

Jason Fedin

Overview

- you need to install the GNU gcc compiler, make, and gdb debugger from cygwin.com
 - simulates a unix environment
-
- download either the 32 or 64 bit version of the setup file
 - <https://cygwin.com/install.html>
 - setup-x86_64.exe or setup-x86_64.exe

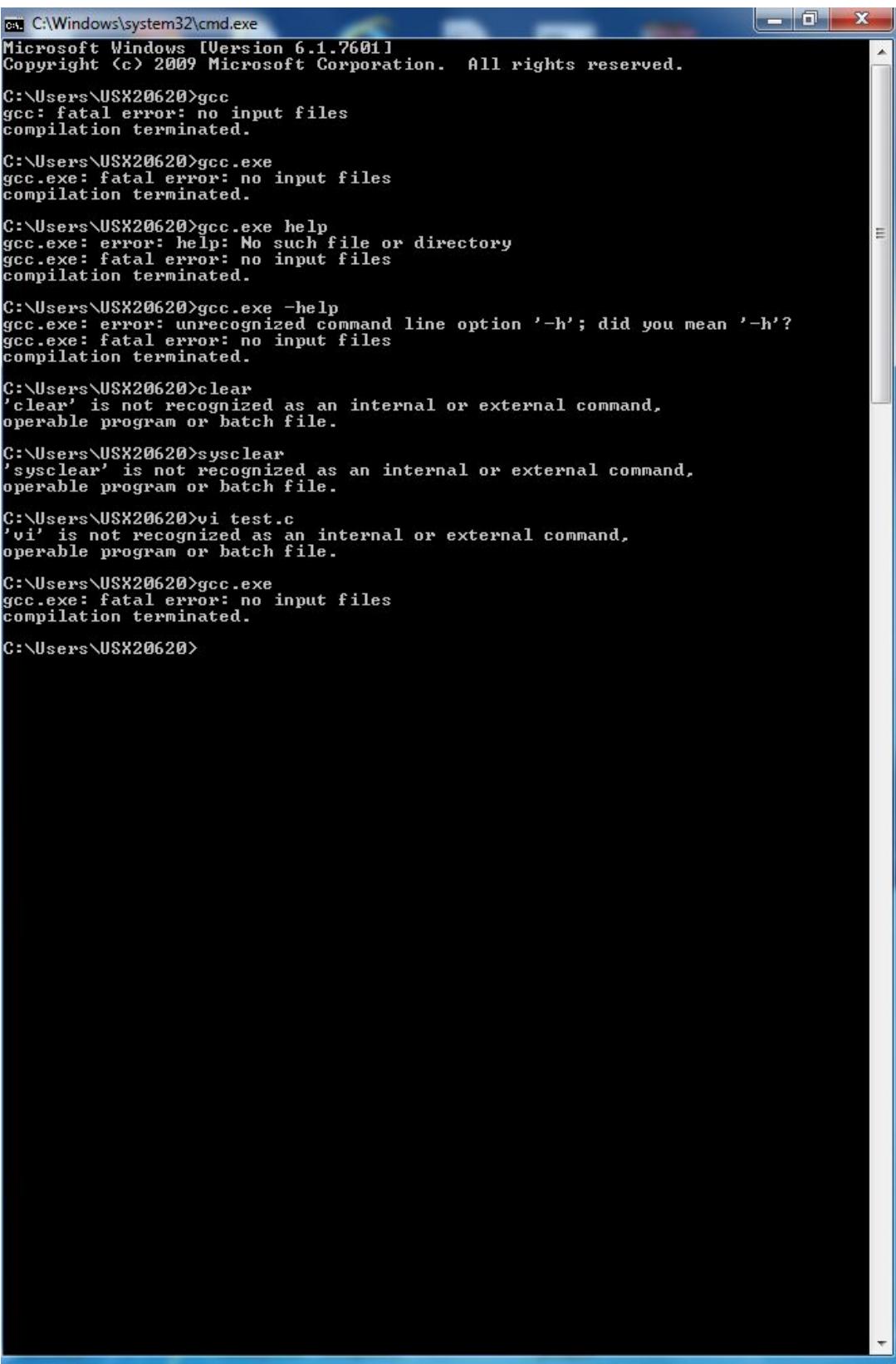
Overview

- run the downloaded Cygwin installer
 - accept the defaults until you reach the Select Your Internet Connection page
 - select the option on this page that is best for you. Click Next.
- on the Choose Download Site page, choose a download site you think might be relatively close to you. Click Next.
- on the Select Packages page you select the packages to download
 - Click the + next to Devel to expand the development tools category
 - You may want to resize the window so you can see more of it at one time
 - Select each package you want to download by clicking the Skip label next to it, which reveals the version number of the package to download
 - at a minimum, select
 - gcc-core: C compiler
 - gdb: The GNU Debugger
 - make: the GNU version of the 'make' utility
 - Packages that are required by the packages you select are automatically selected as well.
 - Click Next to connect to the download site and download the packages you selected, and click Finish when the installation is complete

Setting your path up

- Open the Control Panel:
 - On Windows XP select Start > Settings > Control Panel and double-click System.
 - On Windows 7, type var in the Start menu's search box to quickly find a link to Edit the system environment variables
- Select the Advanced tab and click Environment Variables
- In the System Variables panel of the Environment Variables dialog, select the Path variable and click Edit
- Add the path to the cygwin-directory\bin directory to the Path variable, and click OK
 - By default, cygwin-directory is C:\cygwin (for 32 bit Cygwin distribution) or
 - C:\cygwin64 (for 64 bit Cygwin distribution)
 - Directory names must be separated with a semicolon
- Click OK in the Environment Variables dialog and the System Properties dialog.

Verifying the Installation



A screenshot of a Microsoft Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window shows the following command-line session:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\USX20620>gcc
gcc: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe help
gcc.exe: error: help: No such file or directory
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>gcc.exe -help
gcc.exe: error: unrecognized command line option '-h'; did you mean '-h'?
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>clear
'clear' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>sysclear
'sysclear' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>vi test.c
'vi' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\USX20620>gcc.exe
gcc.exe: fatal error: no input files
compilation terminated.

C:\Users\USX20620>
```

Using The Command Line Interface

Jason Fedin

Using the Command Line Interface

- A text editor (not a Word Processor)
 - A command-prompt or terminal window
 - An installed C compiler
-
- No need for an IDE
 - Simple, efficient workflow
 - Better as you gain experience
 - Can be used if you are overwhelmed by IDEs
 - Useful if hardware resources are limited

Overview

Jason Fedin

Overview

- all of the example programs that have been provided in this class and my beginner class have all been very small and relatively simple
- most programs in the real world that you will have to develop will not be as small or as simple
- it is imperative to learn the proper techniques for dealing with larger programs
 - you need to divide the problem into multiple sub problems and then try to tackle it one by one
- C provides all the features necessary for the efficient development of large programs

Dividing your program into multiple files

- in every program that I have presented so far, it was assumed that the entire program was entered into a single file
- the most common beginner mistake is to jump in directly and try to write all the necessary code into a single file and later try to debug or extend later
 - this approach is doomed to fail and would usually require re-writing from scratch
- all the functions that the program used were included in one file
 - except for the system functions, such as printf() and scanf()
 - standard header files such as <stdio.h> and <stdbool.h> were also included for definitions and function declarations
- when you start dealing with larger programs, they must be organized around multiple files
 - programs that contain more than 100 statements or so

IDE's

- developing with an Integrated Development Environment (IDE) is essential when dealing with large programs
- large programming applications frequently require the efforts of more than one programmer
 - as the number of statements in the program increases, so does the time it takes to edit the program and to subsequently recompile it
 - having everyone work on the same source file, or even on their own copy of the same source file, is unmanageable
- C programs do not require that all the statements for a particular program be contained in a single file
 - you can enter your code for a particular module into one file, for another module into a different file, and so on
- the term module refers either to a single function or to a number of related functions that you choose to group logically
- with Code::Blocks and Visual Studio code, working with multiple source files is easy
 - you simply have to identify the particular files that belong to the project on which you are working, and the software handles the rest for you

Advantages

- the advantages of this approach allows you to achieve Abstraction and promotes the re-usability of code
- teams of programmers can work on programs
 - each programmer works on a different file
- files can contain all functions from a related group
 - e.g. all matrix operations
- well implemented can be re-used in other programs, reducing development time
- when changes are made to a file, only that file need be re-compiled to rebuild the program
- if you check any Open-Source project (GitHub) you can see how the large program is “decentralized” into many numbers of sub-modules
 - each individual module contributes to a specific critical function of the program

How??

- programmers usually start designing a program by dividing the problem into easily managed sections
- each of these sections might be implemented as one or more functions
- all functions from each section will usually live in a single file
- the file contains the definition of each function
 - should create a header file for each of the C files
 - will have the same name as the C file

Compiling multiple source files

Jason Fedin

Overview (Example)

- suppose you have conceptually divided your program into three modules
 - mod1.c, mod2.c, and the main() routine into the file main.c
- to tell the system that these three modules actually belong to the same program
 - you include the names of all three files when you enter the command to compile the program

```
$ gcc mod1.c mod2.c main.c -o dbtest
```

- the above has the effect of separately compiling the code contained in mod1.c, mod2.c, and main.c

Overview (Example)

- errors discovered in mod1.c, mod2.c, and main.c are separately identified by the compiler

mod2.c:10: mod2.c: In function 'foo':

mod2.c:10: error: 'i' undeclared (first use in this function)

mod2.c:10: error: (Each undeclared identifier is reported only once

mod2.c:10: error: for each function it appears in.)

- the above indicates that the compiler identified an error on line 10 of file mod2.c
 - in the function foo
- no messages are displayed for mod1.c and main.c because no errors are found compiling those modules

Overview (Example)

- we already understand that the compiler generates intermediate object files for each source file that it compiles
 - places the resulting object code from compiling mod.c into the file mod.o by default (or obj on windows)
- some C compilers keep these object files around and do not delete them when you compile more than one file at a time
- in the previous example, because mod1.c and main.c had no compiler errors, the corresponding .o files (mod1.o and main.o) would still be around
- replacing the c from the filename mod.c with an o tells the C compiler to use the object file that was produced
- the following command line could be used with a compiler that does not delete the object code files

```
cc mod1.o mod2.c main.o -o dbtest
```

- do you not have to re-edit mod1.c and main.c if no errors are discovered by the compiler, but you also don't have to recompile them

Overview (Example)

- If your compiler automatically deletes the intermediate .o files
 - you can still take advantage of performing incremental compilations
- you can compile each module separately and using the -c command-line option
 - option tells the compiler not to link your file
 - does not produce an executable
 - retains the intermediate object file that it creates

```
$ gcc -c mod2.c
```

- compiles the file mod2.c, placing the resulting executable in the file mod2.o

Overview (Example)

- you can use the following sequence to compile a three-module program using the incremental compilation technique

```
$ gcc -c mod1.c           Compile mod1.c => mod1.o  
$ gcc -c mod2.c           Compile mod2.c => mod2.o  
$ gcc -c main.c           Compile main.c => main.o  
$ gcc mod1.o mod2.o mod3.o -o myProgram    Create executable
```

- the three modules are compiled separately
 - no errors were detected by the compiler
- the last line above lists only object files and no source files
 - the object files are linked together to produce the executable output file 'myProgram'

Overview (Example)

- if you extend the preceding examples to programs that consist of many modules
 - you can see how this mechanism of separate compilations can enable you to develop large programs more efficiently

```
$ gcc -c legal.c
```

Compile legal.c, placing output in legal.o

```
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o -o superchess
```

- the above will compile a program consisting of six modules, in which only the module legal.c needs to be recompiled
- Code::blocks and Visual Studio code have this knowledge of what needs recompilation, and they only recompile files as necessary
- the process of incremental compilation can be automated by using a tool called make which we will discuss in the next lecture

Makefiles

Jason Fedin

Overview

- if you still want to work from the command line, the makefile utility is a tool that you might want to learn how to use
 - not part of the C language
 - very helpful when developing larger programs
 - can help speed your development time
- this powerful utility allows you to specify a list of files and their dependencies in a special file known as a Makefile
- the make program automatically recompiles files only when necessary
 - based on the modification times of a file
- if make finds that your source (.c) file is newer than your corresponding object (.o) file
 - automatically issues the commands to recompile the source file to create a new object file
- you can even specify source files that depend on header files
 - you can specify that a module called datefuncs.o is dependent on its source file datefunc.c as well as the header file date.h
 - if you change anything inside the date.h header file, the make utility automatically recompiles the datefuncs.c file
 - based on the fact that the header file is newer than the source file

Example

SRC = mod1.c mod2.c main.c

OBJ = mod1.o mod2.o main.o

PROG = myProgram\$(PROG): \$(OBJ)

gcc \$(OBJ) -o \$(PROG)

\$(OBJ): \$(SRC)

Example explained

- this Makefile defines the set of source files (SRC)
- this Makefile defines the corresponding set of object files (OBJ)
- this Makefile defines the name of the executable (PROG)
- also defines some dependencies `$(PROG): $(OBJ)`
 - says that the executable is dependent on the object files
 - if one or more object files change, the executable needs to be rebuilt
 - must be typed with a leading tab (`gcc $(OBJ) -o $(PROG)`)
- the last line of the Makefile `$(OBJ): $(SRC)`
 - says that each object file depends on its corresponding source file
 - if a source file changes, its corresponding object file must be rebuilt
 - the make utility has built-in rules that tell it how to do that

Running Make

- here is what happens the first time you run make

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o myProgram
$
```

- make compiled each individual source file and then linked the resulting object files to create the executable

Running Make (cont'd)

- if you instead had an error in mod2.c, here's what the output from make would have looked like

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
mod2.c: In function 'foo2':
mod2.c:3: error: 'i' undeclared (first use in this function)
mod2.c:3: error: (Each undeclared identifier is reported only once
mod2.c:3: error: for each function it appears in.)
make: *** [mod2.o] Error 1
$
```

- make found there was an error in compiling mod2.c and stopped the make process, which is its default action

Running Make (cont'd)

- if you correct mod2.c and run make again

```
$ make
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o myProgram
$
```

- notice that make did not recompile mod1.c
 - the real power and elegance of the make utility
- you can use the Make utility to start using make for your own programs

Communication between modules

Jason Fedin

Overview

- methods can be used so that the modules contained in separate files can effectively communicate
- if a function from one file needs to call a function contained inside another file
 - a function call can be made in the normal fashion
 - arguments can be passed and returned in the usual way
 - in the file that calls the function, you should always make certain to include a prototype declaration
 - so that the compiler knows the function's argument types and the type of the return value
- it is important to remember that even though more than one module might be specified to the compiler at the same time on the command line
 - the compiler compiles each module independently
 - means that no knowledge about structure definitions, function return types, or function argument types is shared across module compilations by the compiler
- it is up to you to ensure that the compiler has sufficient information about such things to correctly compile each module

External Variables

- functions contained in separate files can communicate through external variables
 - an extension to the concept of the global variable
- an external variable is one whose value can be accessed and changed by another module (file)
- inside the module that wants to access the external variable
 - the variable data type is preceded with the keyword extern in the declaration
 - tells the compiler that a globally defined variable from another file is to be accessed
- suppose you want to define an int variable called moveNumber
 - you want to access the value and possibly modify it within a function contained in another file

```
int moveNumber = 0;
```

- declare the above, at the beginning of your program, outside of any function
 - its value could be referenced by any function within that program
 - moveNumber is defined as a global variable

External variables (cont'd)

- this same definition of the variable moveNumber also makes its value accessible by functions contained in other files
- the statement defines the variable moveNumber not just as a global variable, but also as an external global variable
- to reference the value of an external global variable from another module
 - you must declare the variable to be accessed, preceding the declaration with the keyword extern

```
extern int moveNumber;
```

- the value of moveNumber can now be accessed and modified by the module in which the preceding declaration appears
- other modules can also access the value of moveNumber by incorporating a similar extern declaration in the file

Static Versus Extern Variables

- there are situations that arise in which you want to define a variable to be global but not external
- you want to define a global variable to be local to a particular module (file)
 - it makes sense to want to define a variable this way if no functions other than those contained inside a particular file need access to the particular variable
- In these situations, you should define the variable to be static

Static Versus Extern Variables

- if the below declaration is made outside of any function
 - makes the value of the variable accessible from any subsequent point in the file in which the definition appears
 - not from functions contained in other files

```
static int moveNumber = 0;
```

- if you need to define a global variable whose value does not have to be accessed from another file
 - declare the variable to be static
 - a cleaner approach to programming
- the static declaration more accurately reflects the variable's usage
 - no conflicts can be created by two modules that unknowingly both use different external global variables of the same name

Using Header files effectively

Jason Fedin

Overview

- include files are great for grouping all your commonly used definitions inside a file
- you can then simply include the file in any program that needs to use those definitions
- nowhere is the usefulness of the #include facility greater than in developing programs that have been divided into separate program modules
- if more than one programmer is working on developing a particular program, include files provide a means of standardization
 - each programmer is using the same definitions, which have the same values
 - each programmer is also spared the time-consuming and error-prone task of typing these definitions into each file that must use them
- by centralizing the definition of common data structures into one or more include files
 - you eliminate the error that is caused by two modules that use different definitions for the same data structure
 - if a change has to be made to the definition of a particular data structure, it can be done in one place only—inside the include file

the heap and stack

Jason Fedin

Memory Overview

- there are three forms of memory that you can use in your program
 - static
 - stack
 - heap
- in this lecture, we will discuss these three types with a focus on the heap and stack
- each type of memory determines where and how it is stored
- it is very important to understand the differences/advantages/disadvantages of how/where your memory is stored
 - it gives you a strategic advantage for creating scalable programs
- you have to decide when to use memory from the stack vs heap or static memory based on each problem you are trying to solve
- static memory persists throughout the entire life of the program
 - usually used to store things like global variables, or variables created with the static clause

Stack

- the stack is a special region of memory that stores temporary variables
 - used to store variables that are created inside of a function
 - easier to keep track of because the memory is only locally available in the function
- the stack is a "LIFO" (last in, first out) data structure that is managed and optimized by the CPU
 - a linear data structure
 - there is no need to manage the memory yourself
 - variables are allocated and freed automatically
- the stack grows and shrinks as variables are created and destroyed inside a function
 - every time a function declares a new variable, it is "pushed" onto the stack
 - every time a function exits, all of the variables pushed onto the stack by that function, are freed (deleted)
 - once a stack variable is freed, that region of memory becomes available for other stack variables
- there is a limit on the size of variables that can be stored on the stack
- if a program tries to put too much information on the stack, stack overflow will occur
 - happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory
 - stack overflow also occurs in situations where recursion is incorrectly used

Stack

- stack memory is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame
- a key to understanding the stack is the notion that when a function exits, all of its variables are popped off of the stack
 - thus stack variables are local in nature
- stack variables only exist while the function that created them is running
- a common bug is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited)

Heap

- opposite of the stack
- a hierarchical data structure
- the heap is a large pool of memory that can be used dynamically
- memory is not automatically managed
 - more free-floating region of memory
- the heap is managed by the programmer
 - the memory is accessed through the use of pointers
 - you have to explicitly allocate (malloc) and deallocate (free) the memory
 - failure to free the memory when you are finished with it will result in memory leaks
 - memory that is still “being used”, and not available to other processes

Heap

- there are generally no restrictions on the size of the heap (or the variables it creates)
 - other than the physical size of memory in the machine
- variables created on the heap are accessible anywhere in the program
 - essentially global in scope
- heap memory is slightly slower to be read from and written to

When should I use the Heap or Stack?

- use the heap
 - when you need to allocate a large block of memory
 - a large array
 - a big struct
 - when you need to keep that variable around a long time
 - a global
 - when you need variables like arrays and structs that can change size dynamically
 - arrays that can grow or shrink as needed
- use the stack
 - when you are dealing with relatively small variables that only need to persist as long as the function using them is alive
 - easier and faster

Characteristics of the Stack (Summary)

- very fast access
- do not have to explicitly de-allocate variables
- space is managed efficiently by CPU
 - memory is allocated in a contiguous block
 - memory will not become fragmented
- local variables only
- limit on the size of the stack
- variables cannot be resized

Characteristics of the Heap (Summary)

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- you are responsible for managing the memory
- no guaranteed efficient use of space
 - memory is allocated in any random order
 - memory may become fragmented over time as blocks of memory are allocated, then freed
- variables can be resized using realloc()

Auto Variables

Jason Fedin

Storage Classes

- storage classes are used to describe the features of a variable/function
 - include the scope, visibility and life-time
 - help us to trace the existence of a particular variable during the runtime of a program
- the lifetime of a variable is the time period during which variable exist in computer memory
 - some exist briefly, some are repeatedly created and destroyed, and others exist for the entire execution of a program
- the scope of the variable is where the variable can be referenced in a program
 - some can be referenced throughout a program, others from only portions of a program
- a variable's visibility or linkage, determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations

Auto Storage Class

- C provides four storage classes, indicated by their storage class specifiers
 - auto
 - register
 - extern
 - Static
- the four storage-class specifiers can be split into two storage durations
 - automatic storage duration
 - static storage duration
- keyword auto is used to declare variables of automatic storage duration
 - created when the block in which they are defined is entered
 - exist while the block is active
 - destroyed when the block is exited

Local Variables

- local variables are declared within a function body or block of code
- local variables have automatic storage duration by default
- so, these variables are known as automatic local variables
 - they are automatically created each time the function is called
 - their values are local to the function
- the value of a local variable can only be accessed by the function in which the variable is defined
 - Its value cannot be accessed by any other function
 - If an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the function is called

Local Variables

- the C compiler assumes by default that any variable defined inside a function is an automatic local variable
 - the keyword auto is seldom used
- C++ has repurposed the auto keyword for a quite different use, so simply not using auto as a storage-class specifier is better for C/C++ compatibility
- you can, however, make your intentions perfectly clear by explicitly using the keyword auto before the definition of the variable
 - you might do this to document that you are intentionally overriding an external variable definition
 - or that it is important not to change the variable to another storage class

Why use Auto?

- automatic storage is a means of conserving memory
 - automatic variables exist only when they are needed
 - they are created when the function in which they are defined is entered
 - they are destroyed when the function is exited
- automatic storage is an example of the principle of least privilege
 - allowing access to data only when it is absolutely needed
- why have variables stored in memory and accessible when in fact they are not needed?

Syntax

- storage classes precede the type of the variable
- to specify the storage class for a variable, the following syntax is to be followed

storage_class var_data_type var_name;

- the following declaration indicates that double variables x and y are automatic local variables
 - they exist only in the body of the function in which the declaration appears

auto double x, y;

External Variables

Jason Fedin

External Variables

- the extern storage class simply tells us that a variable is defined elsewhere
 - not within the same block where it is used
- an extern variable is a global variable initialized with a legal value where it is declared in order to be used elsewhere
 - an extension to the concept of the global variable
- the main purpose of using extern variables is that they can be accessed between two different files which are part of a large program
 - functions contained in separate files can communicate through external variables
- the extern storage class is used to give a reference of a global variable that is visible to ALL the program files

External Variables

- inside the module that wants to access the external variable
 - the variable data type is preceded with the keyword extern in the declaration
 - tells the compiler that a globally defined variable from another file is to be accessed
- suppose you want to define an int variable called moveNumber
 - you want to access the value and possibly modify it within a function contained in another file

```
int moveNumber = 0;
```

- declare the above, at the beginning of your program, outside of any function
 - its value could be referenced by any function within that program
 - moveNumber is defined as a global variable

External variable

- this same definition of the variable moveNumber also makes its value accessible by functions contained in other files
- the statement defines the variable moveNumber not just as a global variable, but also as an external global variable
- to reference the value of an external global variable from another module
 - you must declare the variable to be accessed, preceding the declaration with the keyword extern
 - the variable cannot be initialized
 - it points the variable name at a storage location that has been previously defined

```
extern int moveNumber;
```

- the value of moveNumber can now be accessed and modified by the module in which the preceding declaration appears
- other modules can also access the value of moveNumber by incorporating a similar extern declaration in the file

External Variables

- you must obey an important rule when working with external variables
 - the variable has to be defined in some place among your source files
- the first way is to declare the variable outside of any function, not preceded by the keyword extern

```
int moveNumber;
```

- the second way to define an external variable is to declare the variable outside of any function, placing the keyword extern in front of the declaration
 - explicitly assigning an initial value to it

```
extern int moveNumber = 0;
```

- these two ways are mutually exclusive

Extern specifier on functions

- when a function is defined, it can be declared to be extern explicitly
- an extern function can be called from a file where it is not defined
 - where it does not need to be defined in a header file

```
extern double foo(double x){ ...}
```

- the definition of the foo function effectively becomes global to any file in the program
 - can be called from outside the file

Static

Jason Fedin

Overview

- the static storage class can be used on local and global variables, as well as functions
- when applied to local variables it instructs the compiler to keep the variable in existence during the life-time of the program
- when applied to global variables, the static modifier causes that variable's scope to be restricted to the file in which it is declared
- when applied to functions, the static function can be called only from within the same file as the function appears

Automatic and static variables (local statics)

- we know that when you normally declare a local variable inside a function
 - you are declaring automatic local variables
 - recall that the keyword auto can, in fact, precede the declaration of such variables (optional)
 - an automatic variable is actually created each time the function is called and destroyed upon exit of the function
- automatic local variables can be given initial values
 - the value of the expression is calculated and assigned to the automatic local variable each time the function is called
 - because an automatic variable disappears after the function completes execution, the value of that variable disappears along with it
 - the value an automatic variable has when a function finishes execution is guaranteed not to exist the next time the function is called
- static variables have a property of preserving their value even after they are out of their scope
 - static variables preserve the value of their last use in their scope.
 - no new memory is allocated because they are not re-declared
 - their scope is local to the function to which they were defined
- making local variables static allows them to maintain their values between function calls
 - does not create and destroy the local variable each time it comes into and goes out of scope

Initialization of local statics

- a static, local variable is initialized only once at the start of overall program execution
 - not each time that the function is called
- the initial value specified for a static variable must be a simple constant or constant expression
- static variables also have default initial values of zero, unlike automatic variables, which have no default initial value
- static variables are allocated memory on the heap, not on the stack

Static Versus Extern Variables (global statics)

- there are situations that arise in which you want to define a variable to be global but not external
- you want to define a global variable to be local to a particular module (file)
 - it makes sense to want to define a variable this way if no functions other than those contained inside a particular file need access to the particular variable
- In these situations, you should define the variable to be static
 - by default, they are assigned the value 0 by the compiler

Static Versus Extern Variables (global statics)

- if the below declaration is made outside of any function
 - makes the value of the variable accessible from any subsequent point in the file in which the definition appears
 - not from functions contained in other files

```
static int moveNumber = 0;
```

- if you need to define a global variable whose value does not have to be accessed from another file
 - declare the variable to be static
 - a cleaner approach to programming
- the static declaration more accurately reflects the variable's usage
 - no conflicts can be created by two modules that unknowingly both use different external global variables of the same name

Static and Extern Variables on functions

- when a function is defined, it can be declared to be extern or static
 - extern case being the default
- a static function can be called only from within the same file as the function appears

```
static double foo(double x){ ...}
```

- the definition of the foo function effectively becomes local to the file in which it is defined
 - cannot be called from outside the file

static and structures

- static variables should not be declared inside a structure
- the C compiler requires the entire structure elements to be placed together
 - memory allocation for structure members should be contiguous
- it is possible to declare a structure
 - inside a function (stack segment)
 - allocate memory dynamically(heap segment)
 - it can be even global
- whatever might be the case, all structure members should reside in the same memory segment
 - the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure
- separating out one member alone to a data segment defeats the purpose of a static variable
- it is possible to have an entire structure as static

Register

Jason Fedin

Overview

- a processor register (CPU register) is one of a small set of data holding places that are part of the computer processor
 - a register may hold an instruction, a storage address, or any kind of data
- the register storage class is used to define local variables that should be stored in a register instead of RAM (memory)
 - makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program
- the register storage class should only be used for variables that require quick access
 - the variables which are most frequently used in a C program
 - if a function uses a particular variable heavily
- the keyword register hints to the compiler that a given variable can be put in a register
 - it is the compiler's choice to put it in a register or not
 - MIGHT be stored in a register depending on hardware and implementation restrictions
 - generally, compilers themselves do optimizations and put the variables in register

Overview

- the keyword register is used to define register storage class
- both local variables and formal parameters can be declared as register variables
- this storage class declares register variables which have the same functionality as that of the auto variables
 - the lifetime of register variable remains only when control is within the block
- the variable stored in a register has a maximum size equal to the register size
- you cannot obtain the address of a register variable using pointers
 - cannot have the unary '&' operator applied to it (as it does not have a memory location)

Summary

Storage Class	Declaration Location	Scope (Visibility)	Lifetime (Alive)
auto	Inside a function/block	Within the function/block	Until the function/block completes
register	Inside a function/block	Within the function/block	Until the function/block completes
extern	Outside all functions	Entire file plus other files where the variable is declared as extern	Until the program terminates
static (local)	Inside a function/block	Within the function/block	Until the program terminates
static (global)	Outside all functions	Entire file in which it is declared	Until the program terminates

<https://www.pinterest.com/pin/195484440055998416/>

(Challenge) Storage Classes

Jason Fedin

Overview

- I have provided a number of small challenges that will help you better understand storage classes in C
 - they are rather “easy”
- Our first challenge is to write a small program that declares the following variables
 - an int variable with block scope and temporary storage
 - a global double variable that is only accessible inside this file
 - a global float variable that is accessible within the entire program
 - a float local variable with permanent storage
 - a register int variable
 - a function that is only accessible with the file it is defined

Challenge #2

- Our second challenge is to write a c program that finds the sum of various numbers
- You cannot pass any variable representing the running total to the sum() function

```
#include <stdio.h>
int sum (int num) {
    /**
     * find sum a number
     */
}

int main() {
    printf("%d ",sum(25));
    printf("%d ",sum(15));
    printf("%d ",sum(30));
    return 0;
}

output
25 40 70
```

Challenge #3

- create a c program that shares a variable amongst two files
- create a source file named main.c
 - this source file should include a global variable used as a loop counter
 - this file includes a main function that uses the global variable to iterate through a loop 5 times
 - inside this loop a function should be invoked (display) that is defined in another file (do not use an include file)
 - the display function should take no parameters
- create another source file named display.c
 - the display function is defined in this file
 - this function displays the global variable from main.c (incremented by 1)

Summary

- review the solution provided for further understanding

The #define statement (constants)

Jason Fedin

Overview

- the #define preprocessor directive (like all preprocessor directives), begins with the # symbol at the beginning of a line
 - can be preceded by spaces or tabs
 - allows for space between the # and the remainder of the directive
- in C programming you can use this directive to define symbolic, or manifest, constants in a program
- preprocessor directives run until the first newline following the #
 - a directive is limited to one line in length
 - the combination backslash/newline is deleted before preprocessing begins
 - you can spread the directive over several physical lines
 - these lines still constitute a single logical line

Overview

- typically, #define statements appear first in the program
 - not required, they can appear anywhere in the program
- most programmers group their #define statements at the beginning of the program (or inside an include file)
 - they can be quickly referenced and shared by more than one source file
- by convention, #define names are defined using only uppercase letters and underscores

Example

- one of the primary uses of the #define statement is to assign symbolic names to program constants

```
#define YES 1
```

- defines the name YES and makes it equivalent to the value 1
 - the name YES can subsequently be used anywhere in the program where the constant 1 could be used
 - it is the same as doing a search and replace with a text editor
 - the preprocessor replaces all occurrences of the defined name with its associated text
 - the #define statement has a special syntax
 - no equal sign used to assign the value 1 to YES
 - a semicolon does not appear at the end of the statement

Example

- whenever this name appears, its defined value of 1 is automatically substituted into the program by the preprocessor

```
gameOver = YES;
```

- this statement assigns the value of YES to gameOver
 - has the effect of assigning 1 to gameOver
- a defined name is not a variable
 - you cannot assign a value to it, unless the result of substituting the defined value is in fact a variable

#define expressions

- a definition for a name can include more than a simple constant value
 - a definition for a name can include an expression
 - the following defines the name TWO_PI as the product of 2.0 and 3.141592654
- ```
#define TWO_PI 2.0 * 3.141592654
```

- you can subsequently use this defined name anywhere in a program where the expression  $2.0 \times 3.141592654$  would be valid
  - you could replace the return statement of a circumference function with the following statement
- ```
return TWO_PI * r;
```

NULL define

- the defined name NULL is frequently used by programmers to represent the null pointer
- NULL is already defined on your system inside a file named <stddef.h>
- by including a definition such as
`#define NULL 0`
- in a program, you can then write more readable statements, such as
`while (listPtr != NULL)`
- to set up a while loop that will execute as long as the value of listPtr is not equal to the null pointer

Benefits of #define

- using a defined name for a constant value helps to make programs more readily extendable and more readable
- e.g.
 - when you define an array, you must specify the number of elements in the array
 - subsequent program statements will likely use the knowledge of the number of elements contained inside the array
- float dataValues[1000];
- there is a good chance that you will see statements in the program that use the fact that dataValues contains 1,000 elements
 - for (i = 0; i < 1000; ++i)
- you would use the value 1000 as an upper bound for sequencing through the elements of the array
- suppose that you had to increase the size of the dataValues array from 1,000 to 2,000 elements
 - would necessitate changing all statements that used the fact that dataValues contained 1,000 elements

Benefits of #define

- a better way of dealing with array bounds, which makes programs easier to extend, is to define a name for the upper array bound

```
#define MAXIMUM_DATAVALUES 1000
```

- you can subsequently define the dataValues array to contain MAXIMUM_DATAVALUES elements

```
float dataValues[MAXIMUM_DATAVALUES];
```

- Statements that use the upper array bound can also make use of this defined name

```
for ( i = 0; i < MAXIMUM_DATAVALUES; ++i )
```

- you can now easily change the size of the dataValues array to 2,000 elements by simply changing the definition

```
#define MAXIMUM_DATAVALUES 2000
```

- if the program is written to use MAXIMUM_DATAVALUES in all cases where the size of the array was used
 - the preceding definition could be the only statement in the program that would have to be changed

Benefits of #define

- another nice use of the #define statement is that it helps to make programs more portable from one computer system to another
- it might be necessary to use constant values that are related to the particular computer on which the program is running
 - might have to do with the use of a particular computer memory address, a filename, or the number of bits contained in a computer word
- if you want to execute a program on a different machine
 - where an int contains 64 bits as opposed to 32
 - can use a define to set this
- in situations in which the program must be written to make use of machine-dependent values
- the #define statement can help isolate machine-dependent values from the program as much as possible
 - would be easier to port to another machine

#define vs. constant

```
#define PI 3.14159f
```

- the above defines PI as a symbol that is to be replaced in the code by the string 3.14159f
- we could have defined PI as a variable, but to tell the compiler that its value is fixed and must not be changed
- you can fix the value of any variable when you declare it by prefixing the type name with the keyword const

```
const float Pi = 3.14159f;           // Defines the value of Pi as fixed
```

- the advantage of defining Pi in this way is that you are now defining it as a constant numerical value with a specified type
- when using #define, PI is just a sequence of characters that replaces all occurrences of PI in your code

Using typedef

Jason Fedin

Overview

- the `typedef` keyword is an advanced feature in C that enables you to create your own name for an existing data type
- to define a new type name with `typedef`
 - write the statement as if a variable of the desired type were being declared
 - where the name of the declared variable would normally appear, substitute the new type name
 - in front of everything, place the keyword `typedef`

```
typedef int Counter;
```

- defines the name `Counter` to be equivalent to the C data type `int`
- variables can subsequently be declared to be of type `Counter`

```
Counter j, n;
```

- the C compiler treats the declaration of the variables `j` and `n` as normal integer variables

Advantages

- one advantage is in the added readability that it lends to the definition of the variables
 - code is more readable when the names of the types lend insight into the meaning and intended use of a type
 - it is clear from the definition of what the intended purpose of certain variables are in the program
- another advantage is maintainability
 - using `typedef`'s instead of explicit types is analogous to using variables instead of explicit numbers (magic numbers)
 - if the type used for a certain purpose needs to be changed, then only the `typedef` line needs to be changed
 - when `typedef`'s are not used, thousands of manual changes to the code may be required to achieve this simple logical operation
- you can also use `typedef` to help make a program more portable
 - used to create synonyms for the basic data types
- a program requiring 4-byte integers may use type `int` on one system and type `long` on another
- programs designed for portability often use `typedef` to create an alias for 4-byte integers, such as `Integer`
 - the alias `Integer` can be changed once in the program to make the program work on both systems

typedef vs. #define

- in many instances, a typedef statement can be equivalently substituted by the appropriate #define statement

#define Counter int; // has the same results as using a typedef as in the previous slide

- we know that the #define is handled by the preprocessor
- the typedef is handled by the C compiler
 - provides more flexibility when it comes to assigning names to derived data types
- remember, the typedef statement does not actually define a new type
 - only a new type name

Best practices

- do not bother using typedefs for structs
 - all they do is save you writing the word “struct”, which is a clue that you probably shouldn’t be hiding anyway
- use typedefs for types that combine arrays, structs, pointers, or functions
- use typedefs for portable types
 - when you need a type that’s at least 20-bits, make it a typedef
 - when you port the code to different platforms, select the right type, short, int, long
 - making the change in just the typedef, rather than in every declaration
- use typedefs for casting
 - a typedef can provide a simple name for a complicated type cast

```
typedef int (*ptr_to_int_fun)(void);
char * p;
..... = (ptr_to_int_fun) p;
```

Variable Length Arrays

Jason Fedin

Overview

- up until this point, all the arrays that you have used have had fixed dimensions that you specify in the code
 - arrays are of constant size
- what if you don't know an array's size at compilation time?
 - to handle this, you would have to use dynamic memory allocation with malloc
- a variable-length array is an array whose length, or size, is defined in terms of an expression evaluated at execution time
 - enable you to work with arrays in your programs without having to give them a constant size
- the term variable in variable-length array does not mean that you can modify the length of the array after you create it
 - once created, a VLA keeps the same size
- the term variable means that you can use a variable when specifying the array dimensions when first creating the array

C11

- a C11-conforming compiler does not have to implement support for variable length arrays because it is an optional feature
- if it does not, the symbol `_STDC_NO_VLA_` must be defined as 1
- you can check for support for variable length arrays using this code

```
#ifdef _STDC_NO_VLA_
    printf("Variable length arrays are not supported.\n");
    exit(1);
#endif
```

Conclusion

- Linus Torvalds is quoted saying

"USING VLA'S IS ACTIVELY STUPID! It generates much more code, and much slower code (and more fragile code), than just using a fixed key size would have done."

- Torvalds goes on to say that Linux is free of VLAs and he's proud of that fact
- there is a lot of demo/academic code on VLAs and even less material out there in the C training world
- it is possible in C to create a variable length array
 - just not necessary and it's something I would recommend you avoid
 - If you need to dynamically allocate and even reallocate storage buffers, you can always use pointers

Flexible Array Members

Jason Fedin

Overview

- a flexible array member is a feature introduced in the C99 standard of the C programming language
- when using a structure, we can declare an array without a dimension and whose size is flexible in nature
 - a flexible array member's size is variable (can be changed be at runtime)

```
struct s {  
    int arraySize;  
    int array[];  
}; // end struct s
```

- a flexible array member is declared by specifying empty square brackets ([])

Restrictions

- a flexible array member can be declared only as the last member of a struct
- each struct can contain at most one flexible array member
- a flexible array cannot be the only member of a struct
 - the struct must also have one or more fixed members
- any struct containing a flexible array member cannot be a member of another struct
- a struct with a flexible array member cannot be statically initialized
 - it must be allocated dynamically
 - you cannot fix the size of the flexible array member at compile time

Summary

- It is debatable on whether it is good practice to use flexible array members
- some argue that it is not portable
 - since ISO C99 was released in 1999 (20 years ago), striving for ISO C89 compatibility is a weak argument
- some argue that you can just declare an array of size 0 or 1 and reallocate
 - in previous standards of the C language, it was common to declare a zero-sized array member instead of a flexible array member
 - using non-standardized constructs to support flexible array members can yield undefined behavior
 - bad practice and any undefined-behavior should be avoided
- a struct with a flexible array member reduces the number of allocations for it by $\frac{1}{2}$
 - instead of 2 allocations for one struct object you need just 1
 - meaning less effort and less memory occupied
 - you save the storage for one additional pointer
- if you have to allocate a large number of struct instances, you can measurably improve the runtime and memory usage of your program (by a constant factor)

Complex Number Types

Jason Fedin

Overview

- a complex number is a number of the form $a + bi$
 - i is the square root of minus one (imaginary)
 - a and b are real numbers
- a is called the real part, and bi is called the imaginary part of the complex number
- a complex number can also be regarded as an ordered pair of real numbers (a, b)

Operations on Complex Numbers

- you can apply the following operations to complex numbers
- Modulus
 - the modulus of a complex number $a + bi$ is $\sqrt{a^2 + b^2}$
- Equality
 - the complex numbers $a + bi$ and $c + di$ are equal if a equals c and b equals d
- Addition
 - the sum of the complex numbers $a + bi$ and $c + di$ is $(a + c) + (b + d)i$

Operations on Complex Numbers

- Multiplication
 - the product of the complex numbers $a + bi$ and $c + di$ is $(ac - bd) + (ad + bc)i$
- Division
 - the result of dividing the complex number $a + bi$ by $c + di$ is $(ac + bd) / (c^2 + d^2) + ((bc - ad) / (c^2 + d^2))i$
- Conjugate
 - the conjugate of a complex number $a + bi$ is $a - bi$

C programming and Complex numbers

- the C99 standard introduces support for complex numbers and complex arithmetic
- a C11 conforming compiler is not obliged to implement complex arithmetic
- if it does not, it must implement the macro `_STDC_NO_COMPLEX_`
- you can test whether your compiler supports complex arithmetic using preprocessor directives

```
#ifdef _STDC_NO_COMPLEX_
    printf("Complex arithmetic is not supported.\n");
#else
    printf("Complex arithmetic is supported.\n");
#endif
```

- when the code executes, you will see output telling you whether complex arithmetic is supported

Complex and Imaginary types in C

- float _Complex
 - stores a complex number with real and imaginary parts as type float
- double _Complex
 - stores a complex number with real and imaginary parts as type double
- long double _Complex
 - stores a complex number with real and imaginary parts as type long double
- float _Imaginary
 - stores an imaginary number as type float
- double _Imaginary
 - stores an imaginary number as type double
- long double _Imaginary
 - stores an imaginary number as type long double

Declaring complex numbers

- you can declare a variable to store complex numbers like this

```
double _Complex z1; // Real and imaginary parts are type double
```

- the `_Complex` keyword was chosen for the same reasons as type `_Bool`
 - to avoid breaking existing code
- the `complex.h` header defines `complex` as being equivalent to `_Complex`
 - also defines many other functions and macros for working with complex numbers
- with the `complex.h` header included, you can declare the variable `z1` like this

```
double complex z1; // Real and imaginary parts are type double
```

Declaring Imaginary Numbers

- you use the `_Imaginary` keyword to define variables that store purely imaginary numbers
 - there is no real component
- the `complex.h` header defines `imaginary` as a more readable equivalent of `_Imaginary`

```
double imaginary ix = 2.4*I;
```

- casting an imaginary value to a complex type produces a complex number with a zero real part and a complex part the same as the imaginary number
- casting a value of an imaginary type to a real type other than `_Bool` results in 0
- casting a value of an imaginary type to type `_Bool` results in 0 for a zero imaginary value and 1 otherwise

Complex Functions

- the `creal()` function returns the real part of a value of type `double complex` that is passed as the argument
- the `cimag()` function returns the imaginary part

```
double real_part = creal(z1);           // Get the real part of z1  
double imag_part = cimag(z1);          // Get the imaginary part of z1
```

- you append an `f` to these function names when you are working with float complex values (`crealf()` and `cimagf()`)
- you append a lowercase `l` when you are working with long double complex values (`creall()` and `cimagl()`)

Complex Functions

- the `conj()` function returns the complex conjugate of its double complex argument
- the `conjf()` and `conjl()` functions return the complex conjugate for the other two complex types
- you can write arithmetic expressions involving complex and imaginary values using the arithmetic operators `+`, `-`, `*`, and `/`
- the `<complex.h>` header also defines several math functions (`cpow`)
- you can also use the operators `!`, `++`, `--`, `&&`, `||`, `==`, `!=` and unary `&` with complex numbers

Creating Complex Numbers

- to construct complex numbers you need a way to indicate the imaginary part of a number
 - there is no standard notation for an imaginary floating point constant
- complex.h defines two keywords that can be used to create complex numbers
 - a representation of the complex number “0+1i”

const float complex _Complex_I

- complex.h also defines a shorter name for the same constant
 - _Complex_I is a bit of a mouthful

const float complex I

- the above (I) can causes problems if you want to use the identifier I for something else
 - #include <complex.h>
 - #undef I

Designated Initializers

Jason Fedin

Overview

- designated initializers allow you to specify which elements of an array, structure or union are to be initialized by the values following an array index or a field by name (struct and union)
- the C90 standard required the elements of an initializer to appear in a fixed order
 - the same as the order of the elements in the array or structure being initialized
- the c99 standard allows you to initialize the elements in random order
 - specifying the array indices or structure field names they apply to
- can be very useful if you have a struct that contains a large number of fields and you initially just want to set a few of them
- also a good way of making your code more readable

Syntax (array)

- to specify an array index, write '[index] =' or '[index]' before the element value
 - the term in square brackets specifies which element of the array is to be initialized by the value of the term after the '=' sign
 - unspecified elements are default initialized, which means zeros are defined

int a[6] = {[4] = 29, [2] = 15 }; or

int a[6] = {[4]29 , [2]15 };

is equivalent to

int a[6] = { 0, 0, 15, 0, 29, 0 };

- the index values must be constant expressions
- to initialize a range of elements to the same value, write '[first ... last] = value'

int a[] = {[0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };

Type Qualifiers (const)

Jason Fedin

Overview

- type qualifiers can be used in front of variables to give the compiler more information about the intended use of the variable
 - helps with optimization
- we will discuss the const, volatile, and restrict type qualifiers in the upcoming lectures
- C90 added two new type qualifiers
 - constancy and volatility
- these properties are declared with the keywords const and volatile, which create qualified types
- the C99 standard added a third qualifier
 - restrict
 - designed to facilitate compiler optimizations
- type qualifiers are also Idempotent (added in C99)
 - means that you can use the same qualifier more than once in a declaration, and the superfluous ones are ignored

Const

- the compiler allows you to associate the const qualifier with variables whose values will not be changed by the program
 - you can tell the compiler that the specified variables have a constant value throughout the program's execution
- if you try to assign a value to a const variable after initializing it
 - the compiler might issue an error message, although it is not required to do so
- one of the motivations for the const attribute in the language is that it allows the compiler to place your const variables into read-only memory

const vs. define

- #define is pre-processor directive while const is a keyword
- const variables are actual variables like any other normal variable
 - we can pass them around, typecast them and any other thing that can be done with a normal variable
- #define is not scope controlled whereas const is scope controlled
 - #define can be used in anywhere in the program or in other files to by including the related header file
 - a constant can be declared inside a function (function/scope)
- another advantage of using a const over a #define macro is that a const variable provides for type checking by the compiler
- we have also discussed situations when #define cannot be replaced by const

Type Qualifiers (volatile)

Jason Fedin

Overview

- the volatile type qualifier tells the compiler explicitly that the specified variable will change its value
- it is provided so that a program can tell the compiler to suppress various kinds of optimizations
 - prevents the compiler from optimizing away seemingly redundant assignments to a variable
 - prevents the compiler from repeated examination of a variable without its value seemingly changing
- essentially, prevents the compiler from “caching” variables
- the reason for having this type qualifier is mainly because of the problems that are encountered in real-time or embedded systems programming
 - programs that have a lot of threading
 - programs where resources are scarce

Use cases

- a variable should be declared volatile whenever its value could change unexpectedly
- the optimizer must be careful to reload the variable every time it is used instead of holding a copy in a register
- only three types of variables should use volatile
 - memory-mapped peripheral registers
 - global variables (non stack variables) modified by an interrupt service routine
 - global variables accessed by multiple tasks within a multi-threaded application

Syntax

- the syntax is the same as for const

```
volatile int loc1; /* loc1 is a volatile location */  
volatile int * ploc; /* ploc points to a volatile location */
```

- loc1 is a volatile value
- ploc points to a volatile value

Example

```
val1 = x;
```

```
/* some code not using x */
```

```
val2 = x;
```

- a smart (optimizing) compiler might notice that you use x twice without changing its value
 - would temporarily store the x value in a register
 - when x is needed for val2, it can save time by reading the value from a register instead of from the original memory location
- this optimization is not desired if x is changed between the two statements by some other agency
 - you would use the volatile keyword to ensure that the compiler does not optimize and instead has a stored value for each variable
- if the volatile keyword is not used in the declaration, the compiler can assume that a value has not changed between uses, and it can then attempt to optimize the code

Another Example (An I/O port)

- suppose you have an output port that is pointed to by a variable in your program
- if you want to write two characters to the port
 - an O followed by an N

```
*outPort = 'O';  
*outPort = 'N';
```

- a smart compiler might notice two successive assignments to the same location
 - because outPort is not being modified in between, the compiler would remove the first assignment from the program
- to prevent this from happening, you declare outPort to be a volatile pointer

```
volatile char *outPort;
```

using volatile with const

- a value can be both const and volatile
- a hardware clock setting normally should not be changed by the program
 - make it const
- however, it is changed by an agency other than the program
 - make it volatile
- use both qualifiers in the declaration (order does not matter)

```
volatile const int loc;  
const volatile int * ploc;
```

Type Qualifiers (restrict)

Jason Fedin

Overview

- the restrict type qualifier is an optimization hint for the compiler
 - the compiler can choose to ignore it
- used in pointer declarations as a type qualifier for pointers
 - tells the compiler that a particular pointer is the only reference to the value it points to throughout its scope
 - the same value is not referenced by any other pointer or variable within that scope
 - the pointer is the sole initial means of accessing a data object
 - tells the compiler it does not need to add any additional checks
- without the restrict keyword, the compiler has to assume the worse case
 - that some other identifier might have changed the data in between two uses of a pointer
- with the restrict keyword used, the compiler is free to look for computational shortcuts
- if a programmer uses restrict keyword and violate the above condition, result is undefined behavior
- not supported by C++

Syntax

```
int * restrict intPtrA;
```

```
int * restrict intPtrB;
```

- tells the compiler that, for the duration of the scope in which intPtrA and intPtrB are defined
 - they will never access the same value
- their use for pointing to integers inside an array is mutually exclusive

Binary Numbers and Bits

Jason Fedin

Binary Numbers

- a binary number is a number that includes only ones and zeroes
- the number could be of any length
- the following are all examples of binary numbers

0 10101

1 0101010

10 1011110101

01 011010110

111000 000111

Binary Numbers

- every binary number has a corresponding Decimal value (and vice versa)
- Examples:

Binary Number Decimal Equivalent

1	1
10	2
11	3
...	...
1010111	87

Binary Numbers

- each position for a binary number has a value
- for each digit, multiply the digit by its position value
- add up all of the products to get the final result
- in general, the "position values" in a binary number are the powers of two
 - The first position value is 2^0 , i.e. one
 - The 2nd position value is 2^1 , i.e. two
 - The 3rd position value is 2^2 , i.e. four
 - The 4th position value is 2^3 , i.e. eight
 - The 5th position value is 2^4 , i.e. sixteen
 - etc.

Example

- The value of binary 01101001 is decimal 105. This is worked out below:

128 64 32 16 8 4 2 1

0 1 1 0 1 0 0 1

	1 X 1	= 1
	0 X 2	= 0
	0 X 4	= 0
	1 X 8	= 8
	0 X 16	= 0
	1 X 32	= 32
	1 X 64	= 64
	0 X 128	= 0

Answer: 105

Another example

- The value of binary 10011100 is decimal 156. This is worked out below:

128 64 32 16 8 4 2 1

1	0	0	1	1	1	0	0	
						0 X 1	= 0	
					0 X 2	= 0		
				1 X 4	= 4			
			1 X 8			= 8		
		1 X 16				= 16		
	0 X 32					= 0		
0 X 64						= 0		
1 X 128						= 128		

Answer: 156

Bits

- a byte consists of eight smaller units called bits
- each 1 and 0 in a binary number represents 1 bit
 - If the number is 1, then the bit is “turned on”
 - If the number is 0, then the bit is “turned off”
- the rightmost bit of a byte is known as the least significant or low-order bit, whereas the leftmost bit is known as the most significant or high-order bit
- the advantage of grouping bits into bytes, words, and so on is that it makes them easier to handle
- changing a bit's value to 1 is referred to as setting the bit
- changing a bit's value to 0 is referred to as resetting a bit

Bits for the basic C data types

- basic C data types consist of the following number of bits

BIT	_Bool	1	0 to 1
Byte	char	8	-128 to 127
Word	short int	16	-32,768 to 32,767
Long	long int	32	-2,147,483,648 to 2,147,483,647

Negative numbers (signed)

- the representation of negative numbers is handled slightly differently
- computers represent such numbers using a “twos complement” notation
 - the leftmost bit represents the sign bit
 - If this bit is 1, the number is negative
 - If the bit is 0, the number is positive
 - the remaining bits represent the value of the number
- In twos complement notation, the value -1 is represented by all bits being equal to 1

11111111

Negative numbers (signed)

- a convenient way to convert a negative number from decimal to binary is to
 - first add 1 to the value
 - express the absolute value of the result in binary
 - “complement” all the bits
 - change all 1s to 0s and 0s to 1s
- to convert -5 to binary, 1 is added, which gives -4
- 4 expressed in binary is 00000100
- complementing the bits produces 11111011
- to convert a negative number from binary back to decimal
 - first complement all of the bits
 - convert the result to decimal
 - change the sign of the result
 - then subtract 1

Bit Operations and Bit fields for packing information

- we need to understand these concepts (binary numbers and bits) in order to use the bit operators and the concept of bit fields provided by the C programming language
 - you can perform all sorts of sophisticated operations on bits
 - you can manipulate the individual bits in a variable
- a hardware device is often controlled by sending it a byte or two in which each bit has a particular meaning
- operating system information about files often is stored by using particular bits to indicate particular items
- many compression and encryption operations manipulate individual bits
- C's ability to provide high-level language facilities while also being able to work at a level typically reserved for assembly language makes it a preferred language for writing device drivers and embedded code

Bitwise Operators (Logical)

Jason Fedin

Overview

- bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word
- computer programming tasks that require bit manipulation include
 - low-level device control
 - error detection
 - correction algorithms
 - data compression
 - encryption algorithms
 - optimization
- a bitwise operation operates on one or more binary numbers at the level of their individual bits
 - used to manipulate values for comparisons and calculations
 - substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition

Bitwise Logical Operators

- C offers bitwise logical operators and shift operators
 - operate on the bits in integer values

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands
	Binary OR Operator copies a bit if it exists in either operand
^	Binary XOR Operator copies the bit if it is set in one operand but not both
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits

- they operate on each bit independently of the bit to the left or right
 - do not confuse them with the regular logical operators (&&, ||, and !), which operate on values
- all of the logical operators listed in the table (with the exception of the ones complement operator ~) are binary operators
 - take two operands

Use case

- bit operations can be performed on any type of integer value in C
 - int, short, long, long long, and signed or unsigned
 - and on characters, but cannot be performed on floating-point values
- a bit mask is data that is used for bitwise operations
 - using a mask, multiple bits in a Byte can be set either on, off or inverted from on to off (or vice versa) in a single bitwise operation
- one major use of the bitwise AND, &, and the bitwise OR, |, is in operations to test and set individual bits in an integer variable
 - can use individual bits to store data that involve one of two choices
- you could use a single integer variable to store several characteristics of a person
 - store whether the person is male or female with one bit
 - use three other bits to specify whether the person can speak French, German, or Italian
 - another bit to record whether the person's salary is \$50,000 or more
 - in just four bits you have a substantial set of data recorded

Ones compliment operator : ~

- useful when you do not know the precise bit size of the quantity that you are dealing with in an operation
 - can help make a program more portable
- to set the low-order bit of an int called w1 to 0, you can AND w1 with an int consisting of all 1s except for a single 0 in the rightmost bit

```
w1 &= 0xFFFFFE;
```

- works fine on machines in which an integer is represented by 32 bits
- if you replace the preceding statement with

```
w1 &= ~1;
```

- w1 gets ANDed with the correct value on any machine because the ones complement of 1 is calculated and consists of as many leftmost one bits as are necessary to fill the size of an int (31 leftmost bits on a 32-bit integer system)

Summary

- one of the features that sets C apart from most high-level languages is its ability to access individual bits in an integer
 - often is the key to interfacing with hardware devices and with operating systems
- C features several bitwise operators
 - operate independently on each bit within a value
- the bitwise negation operator (\sim) inverts each bit in its operand, converting 1s to 0s, and vice versa
- the bitwise AND operator ($\&$) forms a value from two operands
 - each bit in the value is set to 1 if both corresponding bits in the operands are 1
 - otherwise, the bit is set to 0
- the bitwise OR operator ($|$) also forms a value from two operands
 - each bit in the value is set to 1 if either or both corresponding bits in the operands are 1
 - otherwise, the bit is set to 0
- the bitwise EXCLUSIVE OR operator (\wedge) acts similarly, except that the resulting bit is set to 1 only if one or the other, but not both, of the corresponding bits in the operands is 1

Truth Table

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bitwise Operators (Shifting)

Jason Fedin

Overview

- C also has left-shift (`<<`) and right-shift (`>>`) operators
 - each produces a value formed by shifting the bits in a pattern the indicated number of bits to the left or right
- for the left-shift operator, the vacated bits are set to 0
- for the right-shift operator, the vacated bits are set to 0 if the value is unsigned

Operator	Description
<code><<</code>	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand
<code>>></code>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand

Left Shift Operator (<<)

- shifts the bits of the value of the left operand to the left by the number of places given by the right operand
 - the vacated positions are filled with 0s (low-order bit of the value)
 - bits moved past the end of the left operand are lost (the high-order bit of the value)
- if w1 is equal to 3, then the expression

```
w1 = w1 << 1;
```

- results in 3 being shifted one place to the left
 - which results in 6 being assigned to w1

w1 ... 000 011 03

w1 << 1 ... 000 110 06

Left Shift Operator (<<) (another example)

```
(10001010) << 2 // expression, 138  
(00101000)    // resulting value 40
```

```
int x= 1;  
  
int y;  
  
y= x<< 2; /* assigns 4 to y*/  
  
x<<= 2;      /* changes x to 4 */
```

- left shifting has the effect of multiplying the value that is shifted by two

Right Shift Operator (>>)

- the right shift operator `>>` shifts the bits of a value to the right
 - bits shifted out of the low-order bit of the value are lost (past the right end of the left operand)
- right shifting an unsigned value always results in 0s being shifted in on the left (through the high-order bits)
- If `w1` is an unsigned int, which is represented in 32 bits, and `w1` is set equal to 4151832098

`w1 >>= 1;`

- sets `w1` equal to hexadecimal 2075916049

<code>w1</code>	1111 0111 0111 0111 1110 1110 0010 0010	4151832098
<code>w1 >> 1</code>	0111 1011 1011 1011 1111 0111 0001 0001	2075916049

Right Shift Operator (>>) (another example)

```
(10001010) >> 2 // expression, signed value, 138  
(00100010)    // resulting value, some systems 34
```

```
(10001010) >> 2 // expression, signed value, 138  
(11100010)    // resulting value, other systems, 226
```

- for an unsigned value, you have the following:

```
(10001010) >> 2 // expression, unsigned value  
(00100010)    // resulting value, all system
```

- each bit is moved two places to the right, and the vacated places are filled with 0s
- right shifting has the effect of dividing the value that is shifted by two

Undefined results

- if you shift a value to the left or right by an amount that is greater than or equal to the number of bits in the size of the data item you will get a undefined result
 - on a machine that represents integers in 32 bits
 - shifting an integer to the left or right by 32 or more bits is not guaranteed to produce a defined result in your program
- also if you shift a value by a negative amount, the result is also undefined

Bitmasks

Jason Fedin

Overview

- a bit mask is data that is used for bitwise operations
 - a bit pattern with some bits set to on (1) and some bits to off (0)
- a mask can be used to set multiple bits in a byte to either on, off or inverted from on to off (or vice versa) using a single bitwise operator
- imagine you want to create a program that holds a state, which is based on multiple values that are one(true) or zero(false)
 - can store these values in different variables (booleans or integers)
 - or instead use a single integer variable and use each bit of its internal 32 bits to represent the different true and false values

00000101

- the first bit (reading from right to left) is true, which represents the first variable
- the 2nd is false, which represents the 2nd variable. The third true. And so on.
- a very efficient way of storing data and has many usages

Overview

- bit masking allows you to use the C bitwise operators to manipulate the bits of an integer
 - checking if particular bit values are present or not
 - setting bits to off or on
- you apply a mask to a value to set or read the desired states of an integer variable
- to avoid information peeking around the edges, a bit mask should be at least as wide as the value it's masking

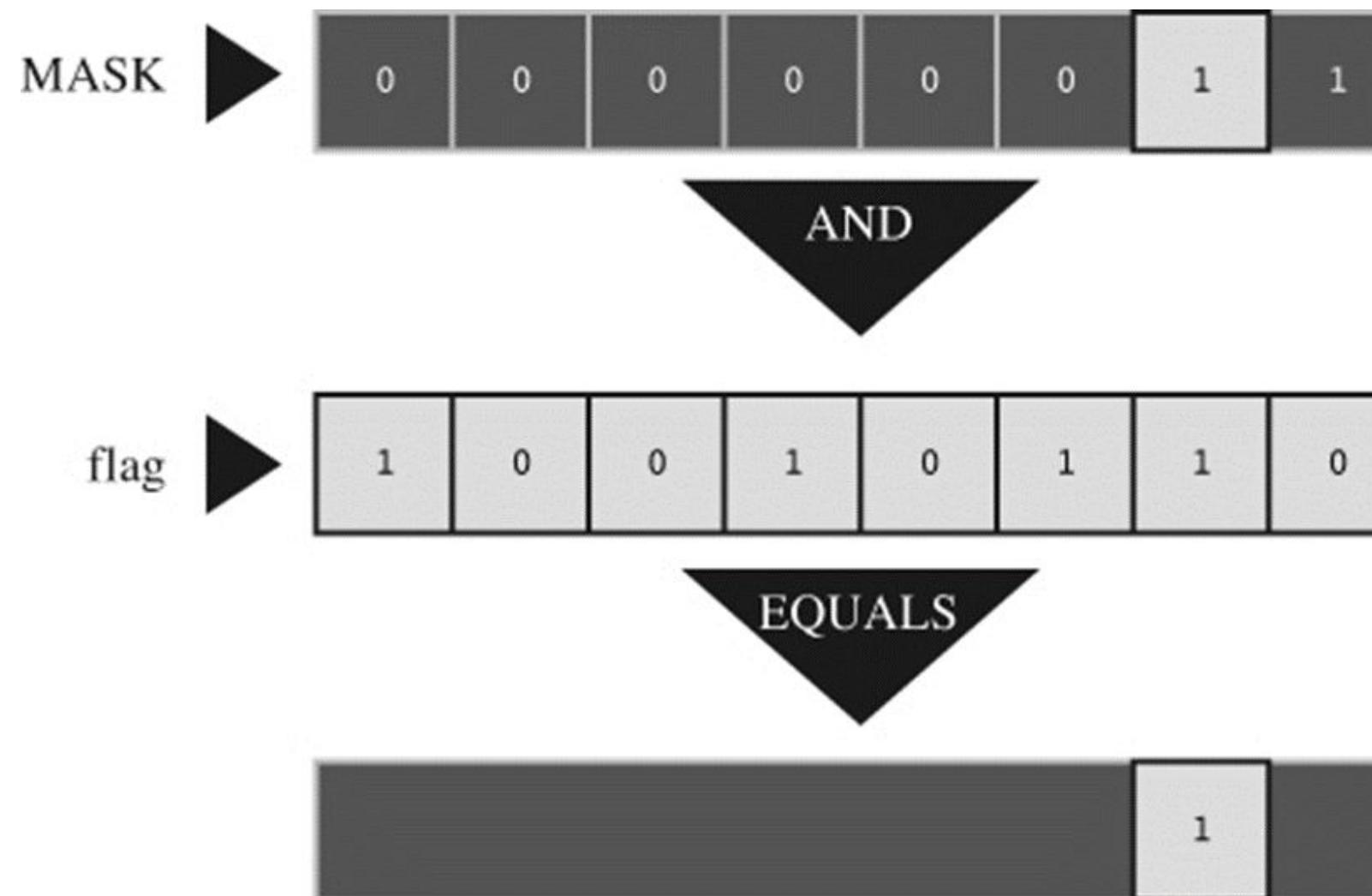
Using a bit mask with AND

- the bitwise AND operator is often used with a mask
- suppose you define the symbolic constant MASK as 2 (binary 00000010)
 - only bit number 1 being nonzero

flags = flags & MASK;

- the above would cause all the bits of flags (except bit 1) to be set to 0
 - any bit combined with 0 using the & operator yields 0
 - bit number 1 will be left unchanged
 - If the bit is 1, 1 & 1 is 1
 - if the bit is 0, 0 & 1 is 0
- we are “using a mask” because the zeros in the mask hide the corresponding bits in flags

bit mask with AND Example



C Primer Plus, Sixth Edition by Stephen Prata

- you can think of the 0s in the mask as being opaque and the 1s as being transparent
- the expression `flags & MASK` is like covering the flags bit pattern with the mask
 - only the bits under MASK's 1s are visible

Turning Bits On (Setting Bits) using OR

- you often need to turn on particular bits in a value while leaving the remaining bits unchanged
 - an IBM PC controls hardware through values sent to ports
 - to turn on the internal speaker, you might have to turn on the 1 bit while leaving the others unchanged
 - you can do this with the bitwise OR operator
- consider the MASK, which has bit 1 set to 1

flags = flags | MASK;

- sets bit number 1 in flags to 1 and leaves all the other bits unchanged
 - any bit combined with 0 by using the | operator is itself
 - any bit combined with 1 by using the | operator is 1

Turning Bits Off (Clearing Bits) using AND

- you often need to turn off particular bits in a value while leaving the remaining bits unchanged
- suppose you want to turn off bit 1 in the variable flags
 - MASK has only the 1 bit turned on
 - MASK is all 0s except for bit 1
 - \sim MASK is all 1s except for bit 1

flags = flags & \sim MASK;

- a 1 combined with any bit using & is that bit
- the above leaves all the bits other than bit 1 unchanged
- a 0 combined with any bit using & is 0
- bit 1 is set to 0 regardless of its original value

Toggling Bits using Exclusive Or

- toggling a bit means turning it off if it is on, and turning it on if it is off
 - you can use the bitwise EXCLUSIVE OR operator to toggle a bit
- if b is a bit setting (1 or 0), then $1 \wedge b$ is 0 if b is 1 and is 1 if b is 0
- $0 \wedge b$ is b, regardless of its value
- if you combine a value with a mask by using ^
 - values corresponding to 1s in the mask are toggled
 - values corresponding to 0s in the mask are unaltered
- to toggle bit 1 in flags

`flags = flags ^ MASK;`

Checking the Value of a Bit

- suppose you want to check the value of a bit
 - does flags have bit 1 set to 1?

```
if (flags == MASK)  
    puts("Wow!"); /* doesn't work right */
```

- even if bit 1 in flags is set to 1, the other bit setting in flags can make the comparison untrue
- you must first mask the other bits in flags so that you compare only bit 1 of flags with MASK

```
if ((flags & MASK) == MASK)  
    puts("Wow!");
```

Using Bit Operators to pack data

Jason Fedin

Overview

- we understand that you can perform all sorts of sophisticated operations on bits
 - often performed on data items that contain packed information
- you can pack information into the bits of a byte if you do not need to use the entire byte to represent the data
 - flags that are used for a boolean true or false condition can be represented in a single bit on a computer
- two methods are available in C that can be used to pack information together to make better use of memory
 - bit fields and bitwise operators
- you could use an unsigned int/long variable to hold the same information
- OR you could use a structure the same size as unsigned int to hold state information
- we will discuss the first option
 - represent the data inside a normal int and then access the desired bits of the int using the bit operators with a bitmask
 - this is a bit more awkward to do (than bit fields)

pack information into an int/long variable

- if you need to store many flags inside a large table, the amount of memory that is wasted could become significant
- an int or a long can be used to conserve memory space
- flags that are used for a boolean true or false condition can be represented in a single bit on a computer
 - each bit in the int can be set to 1 (true) or 0 (false)

10111001

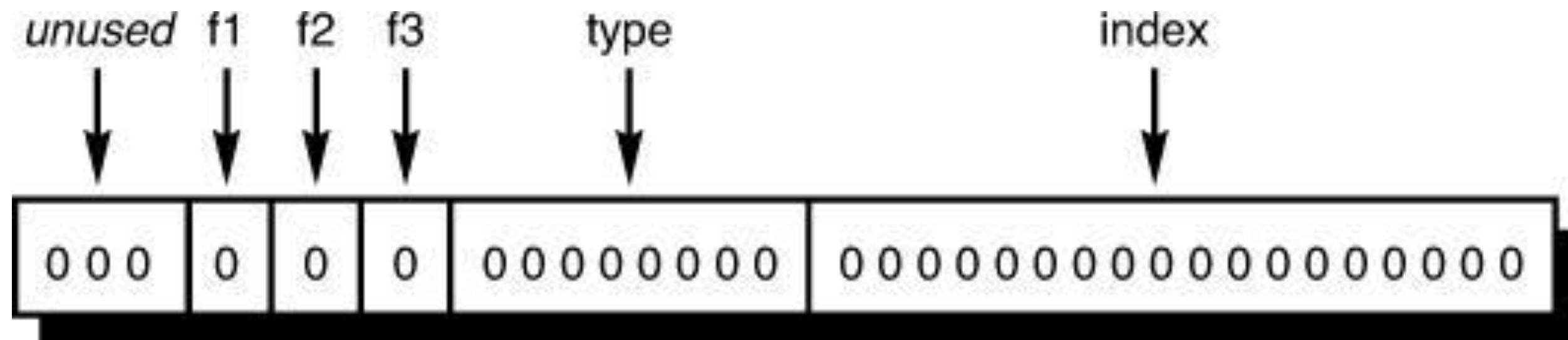
- we can access the desired bits of the int using the bit operators provided by C
 - first bit is true, second bit is false, third bit is true (each bit represents a flag)
 - we are essentially storing eight different values in a single int

Example

- suppose you want to pack five data values into a word because you have to maintain a very large table of these values in memory
 - assume that three of these data values are flags (f_1 , f_2 , and f_3)
 - the fourth value is an integer called type, which ranges from 1 to 255
 - the final value is an integer called index, which ranges from 0 to 100,000
- storing the values of the flags f_1 , f_2 , and f_3 only requires three bits of storage
 - one bit for the true/false value of each flag
- storing the value of the int type requires eight bits of storage
- storing the value of the integer index requires 18 bits
- the total amount of storage needed to store the five data values is 29 bits
- you could define an integer variable that could be used to contain all five of these values

Example (cont'd)

- `unsigned int packed_data; // 32 bits on most systems`
- you can then arbitrarily assign specific bits or fields inside `packed_data` to be used to store the five data values
 - `packed_data` has three unused bits



Programming in C, Fourth Edition by Stephen G. Kochan

Example, setting bits

- you can apply the correct sequence of bit operations to packed_data to set and retrieve values to and from the fields of the int
- to set the type field of packed_data to 7
 - shift the value 7 to the left the appropriate number of places and then OR it into packed_data

```
packed_data |= 7 << 18;
```

- to set the type field to the value n, where n is between 0 and 255
 - to ensure that n is between 0 and 255, you can AND it with 0xff before it is shifted

```
packed_data |= n << 18;
```

Example, setting bits (cont'd)

- the statements on the previous slide only work if you know that the type field is zero
 - to zero out the type field, you need to first AND it with a value (bitmask) that consists of 0s in the eight bit locations of the type field and 1s everywhere else

```
packed_data &= 0xfc03ffff;
```

- to save yourself of having to calculate the bitmask and also to make the operation independent of the size of an integer
 - you could instead use the below statement to set the type field to zero
- packed_data &= ~(0xff << 18);
- combining the statements described previously, you can set the type field of packed_data to the value contained in the eight low-order bits of n

```
packed_data = (packed_data & ~(0xff << 18)) | ((n & 0xff) << 18);
```

- you can see how complex the above expression is for accomplishing the relatively simple task of setting the bits in the type field to a specified value

Example, reading bits (cont'd)

- extracting a value from one of these fields is not as complicated as setting bits
 - the field can be shifted into the low-order bits of the word and then AND'ed with a mask of the appropriate bit length
- to extract the type field of packed_data and assign it to n

```
n = (packed_data >> 18) & 0xff;
```

Another example

- you can use an unsigned long value to represent color values
 - the low-order byte holding the red intensity
 - the next byte holding the green intensity
 - the third byte holding the blue intensity
- you store the intensity of each color in its own unsigned char variable and use a bit mask to set/read

```
#define BYTE_MASK 0xff  
unsigned long color = 0x002a162f;  
unsigned char blue, green, red;  
red = color & BYTE_MASK;  
green = (color >> 8) & BYTE_MASK;  
blue = (color >> 16) & BYTE_MASK;
```

- the code uses the right-shift operator to move the 8-bit color value to the low-order byte
 - then uses the mask technique to assign the low-order byte to the desired variable
- the long color variable can be used to calculate three other colors (without using additional variables)

Using Bit Fields to pack data

Jason Fedin

Overview

- we understand that you can perform all sorts of sophisticated operations on bits
 - often performed on data items that contain packed information
- you can pack information into the bits of a byte if you do not need to use the entire byte to represent the data
 - flags that are used for a boolean true or false condition can be represented in a single bit on a computer
- two methods are available in C that can be used to pack information together to make better use of memory
 - bit fields and bitwise operators
- you could use an unsigned int/long variable to hold the same information
- OR you could use a structure the same size as unsigned int to hold state information
- we will discuss the second option in this lecture
 - you can use bit fields in a structure to address individual bits or groups of bits in a value
 - details are implementation independent

Bit Fields

- a bit field allows you to specify the number of bits in which an int member of a structure is stored
 - uses a special syntax in the structure definition that allows you to define a field of bits and assign a name to that field
 - should use the explicit declarations signed int or unsigned int to avoid problems with hardware dependencies
 - C99 and C11 additionally allow type _Bool bit fields
- bit fields enable better memory utilization by storing data in the minimum number of bits required
 - format enables you to allocate a specified number of bits for a data item
 - can easily set and retrieve its value without having to use masking and shifting

Bit Fields

- a bit field is declared by following an unsigned int member name with a colon (:)
 - an integer constant is placed after the colon which represents the width of the field
 - the number of bits in which the member is stored
 - the constant representing the width must be an integer between 0 and the total number of bits used to store an int (on your platform)
- bit-field members of structures are accessed exactly as any other structure member
- it is possible to specify an unnamed bit field to be used as padding in the structure
- an unnamed bit field with a zero width is used to align the next bit field on a new storage unit boundary.

Example

```
struct packed_struct {  
    unsigned int :3;  
    unsigned int f1:1;  
    unsigned int f2:1;  
    unsigned int f3:1;  
    unsigned int type:8;  
    unsigned int index:18;  
};
```

- the first member is not named
 - :3 specifies three unnamed bits
- the second member, called f1, is also an unsigned int
 - :1 follows the member name and specifies that this member is to be stored in one bit
- the flags f2 and f3 are similarly defined as being a single bit in length
- the member type is defined to occupy eight bits
- the member index is defined as being 18 bits long

Example (setting bits)

- the C compiler automatically packs the preceding bit field definitions together
- the fields of a variable defined to be of type packed_struct can now be referenced in the same convenient way normal structure members are referenced

```
struct packed_struct packed_data;
```

- you can easily set the type field of packed_data to 7 with the simple statement

```
packed_data.type = 7;
```

- or you could set this field to the value of n with the similar statement

```
packed_data.type = n;
```

- you do not need to worry about whether the value of n is too large to fit into the type field
 - only the low-order eight bits of n will be assigned to packed_data.type

Example (reading bits)

- extraction of the value from a bit field is also automatically handled

```
n = packed_data.type;
```

- extracts the type field from packed_data
 - automatically shifts the type field into the low-order bits as required and assigns it to n
- bit fields can be used in normal expressions and are automatically converted to integers

```
i = packed_data.index / 5 + 1;
```

```
if ( packed_data.f2 )
```

Misc. information about bit fields

- you can also include normal data types within a structure that contains bit fields

```
struct table_entry {  
    int      count;  
    char     c;  
    unsigned int f1:1;  
    unsigned int f2:1;  
};
```

- a bit field cannot be dimensioned
 - you cannot have an array of fields, such as flag:1[5]
- you cannot take the address of a bit field
 - no such thing as a type “pointer to bit field”
- you can initialize a bit-field structure by using the same syntax regular structures use

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

The goto statement

Jason Fedin

Overview

- you have probably previously heard about the goto statement and its bad reputation
 - virtually every computer language has such a statement
- a goto statement causes program control to jump to a particular line of code in your program
 - this branch is made immediately and unconditionally upon execution of the goto
- to identify where in the program the branch is to be made, a label is needed
- a label is a name that is formed with the same rules as variable names
- the label is placed directly before the statement to which the branch is to be made and must appear in the same function as the goto

Syntax

- as mentioned, the goto statement has two parts
 - the goto
 - a label name

Form:

goto label; // ends in a semicolon

.

.

.

label : statement // must have a semicolon after the label name

Example

goto someLabel;

- causes the program to branch immediately to the statement that is preceded by the label “someLabel”
- this label can be located anywhere in the function
 - before or after the goto, and might be used as shown

someLabel: printf ("Unexpected end of data.\n");

Problems with goto

- programmers who are lazy frequently abuse the goto statement to branch to other portions of their code
- the goto statement interrupts the normal sequential flow of a program
 - as a result, programs are harder to follow and thus harder to maintain
- using many gotos in a program can make it impossible to decipher
- this style of programming is often derisively referred to as “spaghetti code.”
- for these reasons, goto statements are not considered part of good programming style

Example of spaghetti code using goto

one:

```
for (i = 0; i < number; ++i) {  
    test += i;  
    goto two;  
}
```

two:

```
if (test > 5) {  
    goto three;  
}
```

...

Avoiding goto

- in principle, you never need to use the goto statement in a C program, however, some programmers develop bad habits
 - or have learned programming languages where a goto was necessary, i.e. fortran
- most gotos are used for helping ifs, simulating if elses, controlling loops, or are just there because you have programmed yourself into a corner
 - instead of skipping to the end of a loop and starting the next cycle using a goto statement, use continue
 - instead of leaving a loop using a goto statement, use break
 - actually, break and continue are specialized forms of a goto statement
 - the advantages of using them are that their names tell you what they are supposed to do
 - also they do not use labels and thus, there is no danger of putting a label in the wrong place
 - the alternative forms are clearer than the goto forms
- If you are used to using goto statements, try to train yourself not to

When to use the goto statement

- there is one situation in which the goto statement can be very useful because it provides a way to avoid a lot of complicated logic
 - can be used to exit deeply nested control structures efficiently

```
for(int i = 0 ; i < 10 ; ++i) {  
    for(int j = 0 ; j < 20 ; ++j) {          // Loop executed 10 times  
        for(int k = 0 ; k < 30 ; ++k) {      // Loop executed 10x20 times  
            // Loop body executed 10x20x30 times  
            /* Do something useful */  
            if(must_escape)  
                goto out;  
        }  
    }  
}  
out: /*Statement following the nested loops */
```

- you have a direct exit from the complete nest of loops without any complicated decision making in the outer loop levels

The null statement

Jason Fedin

Overview

- the "null statement" is an expression statement with the expression missing
- C permits a solitary semicolon to be placed wherever a normal program statement can appear
- the null statement has the effect of doing nothing, but exists for syntactical reasons
- although the null statement might seem useless, it is often used by C programmers in while, for, and do loops
- it is useful when the syntax of the language calls for a statement but no expression evaluation

Examples

- the purpose of the following statement is to store all the characters read in from the standard input into the character array pointed to by text until a newline character is encountered

```
while ( (*text++ = getchar ()) != '\n' )  
    ;
```

- all of the operations are performed inside the looping-conditions part of the while statement
- the null statement is needed because the compiler takes the statement that follows the looping expression as the body of the loop
- without the null statement, whatever statement that follows in the program is treated as the body of the program loop by the compiler

Examples

- the following for statement copies characters from the standard input to the standard output until the end of file is encountered

```
for ( ; (c = getchar ()) != EOF; putchar (c) )  
    ;
```

- the next for statement counts the number of characters that appear in the standard input

```
for ( count = 0; getchar () != EOF; ++count )  
    ;
```

- the following loop copies the character string pointed to by from to the one pointed to by to

```
while ( (*to++ = *from++) != '\0' )  
    ;
```

Examples

- when you want to find the index of first occurrence of a certain character in a string

```
int a[50] = "the empire strikes back";
```

```
int i;
```

```
for(i = 0; a[i] != 't'; i++)
```

```
    ;//null statement
```

```
//as no operation is required
```

Examples

```
if (condition1)
    if (condition2)
        dosomething();
else
    /* null statement */
else
    dosomethingelse();
```

- in this case the inner else and null statement keeps the outer else from binding to the inner if

The comma operator

Jason Fedin

Overview

- C supports the use of a comma that can be used in expressions as an operator
 - has the lowest precedence of any C operator
 - acts as a sequence point
- a binary operator that evaluates its first operand and discards the result
 - then evaluates the second operand and returns this value (and type)
 - because all operators in C produce a value, the value of the comma operator is that of the rightmost expression
- the comma operator can be used to separate multiple expressions anywhere that a valid C expression can be used
- the comma operator exists because there are times when you do not want to separate expressions with semicolons

setjmp and longjmp functions

Jason Fedin

Overview

- `setjmp()` and `longjmp()` are functions that let you perform complex flow-of-control in C
- normal program flow in C follows function calls and branching constructs (if, while, etc.)
 - functions `setjmp` and `longjmp` introduce another kind of program flow
- mainly used to implement exception handling in C (error recovery situations)
 - `setjmp` can be used like `try` (in languages like C++ and Java)
 - `longjmp` can be used like `throw`
- suppose there is an error deep down in a function nested in many other functions and error handling makes sense only in the top level function
 - would be very tedious and awkward if all the functions in between had to return normally and evaluate return values
 - would be very tedious if you used a global error variable to determine that further processing doesn't make sense or even would be bad
- the above is a situation where `setjmp/longjmp` makes sense
- you can use `setjmp` and `longjmp` for error handling so that you can jump out of deeply nested call chain without needing to deal with handling errors in every function in the chain

How it works

- setjmp saves a copy of the program counter and the current pointer to the top of the stack
- int setjmp(jmp_buf j)
 - use the variable j to remember where you are now
 - must be called first
- longjmp is then invoked after setjmp (longjmp(jmp_buf j, int i))
 - says go back to the place that the j is remembering
 - restores the process in the state that it existed when it called setjmp
 - return the value of i so the code can tell when you actually got back here via longjmp()
 - the contents of the j are destroyed when it is used in a longjmp()
- often referred to as “unwinding the stack,” because you unroll activation records from the stack until you get to the saved one
- the header file <setjmp.h> needs to be included in any source file that uses setjmp or longjmp

Similarity to goto statement

- although it causes a branch, longjmp differs from a goto
- a goto can't jump out of the current function in C
 - a "longjmp" can jump a long way away, even to a function in a different file
- you can only longjmp back to somewhere you have already been, where you did a setjmp, and that still has a live activation record
- setjmp is more like a "come from" statement than a "go to"

char I/O functions

Jason Fedin

Overview

- we already learned from the previous class that all I/O operations in C must be carried out through function calls
 - I/O operations transport information to and from your program
 - these functions are contained in the standard C library

#include <stdio.h>

- this include file contains function declarations and macro definitions associated with the I/O routines from the standard library
- I would like to cover the following I/O functions in this section so that you have a complete understanding of input and output in C

Char Functions	String Functions	Formatting functions
getc	gets	sprint
getchar	fgets	fprintf
fgetc	puts	fflush
ungetc	fputs	fscanf
putc	getline	sscanf
putchar		
fputc		

Overview

- originally, input/output functions were not part of the definition of C
 - their development was left to C implementations
 - the Unix implementation of C has served as a model for these functions
 - these are standard functions that must work in a wide variety of computer environments
 - they seldom take advantage of features peculiar to a particular system
- we focus on the standard I/O functions available on all systems
 - they enable you to write portable programs that can be moved easily from one system to another
 - also generalize to programs using files for input and output
- when a C program is executed, three files are automatically opened by the system for use by the program
 - stdin, stdout, and stderr (defined in <stdio.h>)
 - stdin identifies the standard input of the program and is normally associated with your terminal window
 - all standard I/O functions that perform input and do not take a FILE pointer as an argument get their input from stdin
 - stdout refers to the standard output, which is normally also associated with your terminal window
 - stderr identifies the standard error file
 - where most of the error messages produced by the system are written and is also normally associated with your terminal window

input functions - getc

- the function getc() enables you to read in a single character from a file
 - takes a FILE pointer as an argument that identifies the file from which the character is to be read
 - must first call fopen() in read mode

```
int getc(FILE *stream);
```

```
int c = getc (inputFile);
```

- has the effect of reading a single character from the file data
 - subsequent characters can be read from the file simply by making additional calls to the getc() function
- this function returns the corresponding integer value (ASCII value of read character) on success
- this function will return the value EOF (special int value used to indicate failure) when the end of file is reached

input functions – getc example

```
#include <stdio.h>
int main() {
    char ch;
    FILE *fp;

    if (fp = fopen("someFile.c", "rw"))
    {
        ch = getc(fp);
        while (ch != EOF)
        {
            // write ch somewhere
            ch = getc(fp);
        }
        fclose(fp);
        return 0;
    }
    return 1;
}
```

input functions – getchar

- this function is very similar to the getc() function described previously
 - reads one character at a time from stdin where getc can read from any input stream
 - getchar() is equivalent to calling getc() with stdin as the argument
- the getchar() function requires no arguments, and it returns the code for the character read from the input stream as type int

```
int getchar(void);
```

```
#include <stdio.h>
int main()
{
    printf("%c", getchar());
    return(0);
}
```

Input: g (press enter key)

Output: g

input functions - getchar with EOF example

- you can use EOF in a program by comparing the return value of getchar() with EOF
 - if they are different, you have not yet reached the end of a file
 - you do not have to define EOF because it is already defined in stdio.h

```
while ((ch = getchar()) != EOF)
```

- when reading from the keyboard as opposed to the file, most systems have a way to simulate an end-of-file condition from the keyboard
 - you cannot just type the letters E O F, and you can't just type -1
 - pressing Ctrl+D at the beginning of a line causes the end-of-file signal to be transmitted in Linux
 - some interpret a Ctrl+Z anywhere as an end-of-file signal

input functions - getchar with EOF example

```
#include <stdio.h>
int main(void) {
    int ch;

    while ((ch = getchar()) != EOF)
        // doSomething
    return 0;
}
```

- the value returned by getchar() is assigned to an int and not a char variable
 - works fine because C allows you to store characters inside ints
- always remember to store the result of getchar() inside an int so that you can properly detect an end-of-file condition
 - if you store the result of the getchar() function inside a char variable, the results are unpredictable
 - on systems that do sign extension of characters, the code might still work okay

input functions - fgetc

- fgetc is a file handling function which is used to read a character from a file
 - reads a single character at a time
 - moves the file pointer position to the next location to read the next character

`int fgetc(FILE *fp)`

`fgetc (fp);`

`fp` – file pointer

input functions – fgetc example

```
/* Open, Read and close a file: Reading char by char */
```

```
# include <stdio.h>
int main( ) {
    FILE *fp ;
    char c ;

    fp = fopen ( "myFile.c", "r" ) ; // opening an existing file

    if ( fp == NULL ) {
        printf ( "Could not open file myFile.c" ) ;
        return 1;
    }

    printf( "Reading the file myFile.c" ) ;
```

input functions – example (cont'd)

```
/* Open, Read and close a file: Reading char by char */
```

```
# include <stdio.h>
int main( ) {
    FILE *fp ;
    char c ;

    fp = fopen ( "myFile.c", "r" ) ; // opening an existing file

    if ( fp == NULL ) {
        printf ( "Could not open file myFile.c" ) ;
        return 1;
    }

    printf( "Reading the file myFile.c" ) ;
```

ungetc

- the ungetc() function enables you to put a character you have just read back into an input stream
 - we already have read from and written to individual characters from/to standard input/standard output and file I/O
 - we cannot determine what character is on the stream until we have read it
 - we would read one character beyond the character we wanted
- this function requires two arguments
 - the first is the character to be pushed back into the stream
 - the second is the identifier for the stream
 - stdin for the standard input stream
- this function returns a value of type int that corresponds to the character pushed back into the stream, or a special character, EOF, if the operation fails

```
int ungetc(int ch, FILE * stream);
```

- you can push a succession of characters back into an input stream, but only one character is guaranteed
 - you should check for EOF this if you are attempting to return several characters to a stream
- this function is useful when you are reading input character by character and do not know how many characters make up a data unit
 - might be reading an integer value, but do not know how many digits there are

Ungetc example

- here is a function that ignores spaces and tabs from the standard input stream using the getchar() and ungetc() functions

```
void eatspaces(void)
{
    char ch = 0;
    while(isspace(ch = (char)getchar())); // Read as long as there are spaces
        ungetc(ch, stdin);           // Put back the nonspace character
}
```

- the while loop continues to read characters as long as they are whitespace characters, storing each character in ch
- the first nonwhitespace character that is read will end the loop, and the character will be left in ch
- the call to ungetc() returns the nonwhitespace character back to the stream for future processing

char output functions

Jason Fedin

output functions - putc

- the function putc() enables you to write a single character to a file (or to stdout)
 - takes two arguments
 - the first argument is the character that is to be written into the file
 - the second argument is the FILE pointer that identifies the file from which the character is to be written to
 - must first call fopen() in write or append mode

int putc(int char, FILE *fp)

putc('\n', stdout);

OR

putc ('\n', outputFile);

- has the effect of writing a newline character into the file identified by the FILE pointer outputFile

String I/O Functions

Jason Fedin

gets

- the gets function stands for get string and reads a line from standard input into a buffer
 - reads until a terminating newline or end-of-file (EOF) is found
 - takes one argument, a pointer to an array of chars where the string is stored
 - returns str on success, and NULL on error or when end of file occurs

`char *gets(char *str)`

- this function is deprecated and should never be used
 - removed from C11
 - as a result, we are not going to discuss it much in this class
 - two options instead (C99) are to use fgets() or getchar()
 - we will discuss fgets in the upcoming slides
- NEVER INVOKE THE gets() FUNCTION
 - no check for buffer overflow is performed
 - it is impossible to tell without knowing the data in advance how many characters gets() will read
 - gets() will continue to store characters past the end of the buffer

fgets

- fgets is used for reading entire lines of data from a file/stream (file get string)
 - has similar behavior to gets()
 - accepts two additional arguments
 - the number of characters to read
 - an input stream (when specified as stdin, same behavior as gets())

`char *fgets(char *buffer, int n, FILE *stream)`

- buffer is a pointer to a character array where the line that is read in will be stored
- n is an integer value that represents the maximum number of characters to be stored into buffer, including the null character
- stream is the pointer to object that identifies the stream where characters are read from
 - usually used with a file stream, however, standard input stream is also acceptable
- reads characters from the specified file until a newline character has been read or until n-1 characters have been read (whichever occurs first)
 - a null character is written immediately after the last character read into the array
 - returns the value of buffer if the read is successful
 - returns the value NULL if an error occurs on the read or if an attempt is made to read past the end of the file

fgets

- the fgets() function retains the newline character (unlike gets)
- it is possible to read a partial line when using fgets
 - truncation of user input can be detected because the input buffer will not contain a newline character
- the fgets function protects against overflowing the string and creating a security hazard
 - not recommended for performance reasons
- fgets is deprecated because the function cannot tell whether a null character is included in the string it reads
 - if a null character is read it will be stored in the string along with the rest of the characters read
 - since a null character terminates a string, this will end your string prematurely, right before the first null character
- only use fgets if you are certain the data read cannot contain a null character
 - otherwise, use getline

getline

- the latest function for reading a string of text is `getline()`
 - a new C library function, having appeared around 2010 or so
- the `getline` function is the preferred method for reading lines of text from a stream (including standard input)
 - the other standard functions, including `gets`, `fgets`, and `scanf`, are too unreliable
- the `getline` function reads an entire line from a stream
 - up to and including the next newline character and takes three parameters

getline

```
ssize_t getline(char **buffer, size_t *size, FILE *stream);
```

- the first parameter is a pointer to a block allocated with malloc or calloc (type char **)
 - the address of the first character position where the input string will be stored
 - this pointer type (a pointer-pointer) causes massive confusion
 - will automatically enlarge the block of memory as needed (realloc)
 - there is never a shortage of space (why getline is so safe)
 - will contain the line read by getline when it returns
- the second parameter is a pointer to a variable of type size_t
 - specifies the size in bytes of the block of memory pointed to by the first parameter
 - the address of the variable that holds the size of the input buffer, another pointer
- the third parameter is simply the stream from which to read the line
- if an error occurs, such as end of file being reached without reading any bytes, getline returns -1
 - otherwise, returns the number of characters read (up to and including the newline, but not the final null character)

getline (example explained) (demo)

- the string input is stored at the memory location referenced by pointer buffer
- uses the `size_t` variable type, which is a special type of integer
 - required by the `getline()` function
 - buffer size is 32 characters, must be referenced as a pointer, not a literal value
- 32 bytes of storage are assigned to memory location buffer via the `malloc()` function
- handles the (rare) condition when memory isn't available. Odds are low that would happen in this program, but it's good programming practice to check
- the `getline()` function uses the address of buffer, `bufsize`, and then `stdin` for standard input
- because variable `characters` is a `size_t` variable type, the `%zu` placeholder is used in the `printf()` function
- as with the `fgets()` function, `getline()` reads and stores the newline character as part of the string
- so if the pointer thing bothers you, just use `fgets()` instead

puts

- the puts() function is used to write a line to the output screen
 - the most convenient function for printing a simple message on standard output
 - automatically appends a newline
- It is simpler than printf, since you do not need to include a newline character
- the difference between puts and printf is that when using printf the argument is interpreted as a formatting string
 - result will be often the same (except for the added newline) if the string doesn't contain any control characters (%)
 - if you cannot rely on that you should not use puts
- the puts function is safe and simple, but not very flexible as it does not give us an option of formatting our string

int puts(const char *string)

formatting functions

Jason Fedin

sprintf

- the sprintf() ("string print formatted") function is used to write formatted output to a string
- when using sprintf(), you can combine several data variables into a character array
 - instead of printing on the console, you store output data to a char buffer (convert)

```
int sprintf(char *string, const char *format, ...)  
sprintf( string, "%d %c %f", value, c, flt );
```

- the first parameter is a char pointer that specifies where to send output (buffer to put the data in)
 - terminates the string with a null character
- the function returns the number of characters stored in the string, not including the terminating null
- this function will behave unpredictably if the string to which it is printing overlaps any of its arguments
 - you need to make sure the size of the buffer to be written to is large enough to avoid buffer overruns
- sprintf is unsafe because it doesn't check the length of the destination buffer
 - can cause the function to overflow the destination buffer when the result of the format string is unexpectedly long
 - leads to security issues and application instability
 - overflows can cause unexpected results

fprintf

- fprintf() is provided to perform the same operation as printf(), but, on a file
 - takes an additional argument
 - the FILE pointer that identifies the file to which the data is to be written to

```
int fprintf(FILE *stream, const char *format, ...)
```

```
fprintf (outFile, "Hello, how are you?\n");  
fprintf(outFile, "my number is: %d\n", 555);
```

fprintf and stderr

- the reason stderr exists is so that error messages can be logged to a device or file other than where the normal output is written
 - particularly desirable when the program's output is redirected to a file
 - the normal output is written into the file, but any system error messages still appear in your window
 - you might want to write your own error messages to stderr for this reason

```
if ( (inFile = fopen ("data", "r")) == NULL )  
{  
    fprintf (stderr, "Can't open data for reading.\n");  
    ...  
}
```

- writes the indicated error message to stderr if the file data cannot be opened for reading
 - if the standard output has been redirected to a file, this message still appears in your window

fscanf

- fscanf() is provided to perform the same operations as the scanf() function, but, on a file
 - used to read formatted input (set of characters) from a file
 - most of the arguments of this function are same as scanf() function
 - takes an additional argument, which is the FILE pointer that identifies the file to which the data is to be read

```
int fscanf(FILE *fp, const char *format [, argument, ...] );
```

- the additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the format string
- returns the number of arguments that are successfully read and assigned (on success)
- returns the value EOF, if the end of the file is reached before any of the conversion specifications have been processed

```
fscanf (myFile, "%i", &i);
```

- reads in the next integer value from the file “myFile” and stores it in the variable i

sscanf

- the sscanf() function allows you to read formatted data from a string rather than standard input or keyboard

```
int sscanf(const char *str, const char * control_string [ arg_1, arg_2, ... ]);  
sscanf(buffer,"%s %d",name,&age);
```

- the first argument is a pointer to the string from where we want to read the data
- the rest of the arguments of sscanf() is same as that of scanf()
- returns the number of items read from the string and -1 if an error is encountered

using fscanf() vs. fgets()/sscanf

- if you use fgets() + sscanf(), you must enter both values on the same line
- if you only use fscanf() on stdin, it will read them off different lines if it does not find the second value on the first line you entered
- if you read a line that you are unable to parse with sscanf() after having read it using fgets() your program can simply discard the line and move on
- If you read a line using fscanf(), when it fails to convert fields, it leaves all the input on the stream
 - so, if you failed to read the input you wanted, you would have to go and read all the data you want to ignore yourself
- you can use fscanf() by itself, however, you may be able to avoid some headaches by using fgets() + sscanf()

using fscanf() vs. fgets()/sscanf

- there is no difference between fscanf() versus fgets()/sscanf() when
 - input data is well-formed
 - two types of errors occur
 - I/O and format
 - fscanf() simultaneously handles these two error types in one function but offer few recovery options
 - fgets() and sscanf() allow logical separation of I/O issues from format ones and thus better recovery
 - only 1 parsing path
 - separating I/O from scanning allows multiple sscanf() options
 - if a given scanning of a buffer does not produce the desired results, other sscanf() calls with different formats are available
 - no embedded '\0'
 - rarely does '\0' occur, but should one occur, sscanf() will not see it as scanning stops with its occurrence, whereas fscanf() continues
- in all cases, check results of all three functions
- **the below is a common question asked by many students in the Q&A (so mention this, another tidbit, unrelated)**
 - **when scanf() converts an int for example, it will leave behind a \n on the input stream (assuming there was one, like from pressing the enter key)**
 - **will cause a subsequent call to fgets() to return immediately with only that character in the input**
 - **a really common issue for new programmers**

fflush

- the fflush() function is used to flush/clean a file or buffer
 - causes any unwritten data in the output buffer to be sent to the output file
 - this process is called flushing a buffer
 - cleans the buffer (making empty) if it has been loaded with any other data already

```
int fflush(FILE *fp);  
fflush(buffer);
```

- buffer is a temporary variable or pointer which loads/points to the data
- if the file is a null pointer, all output buffers are flushed
- the effect of using fflush() on an input stream is undefined
- you can use it with an update stream (any of the read-write modes), provided that the most recent operation using the stream was not input

Variadic functions (variable arguments)

Jason Fedin

Overview

- the word variadic tells us that there is some kind of change or variation involved
 - the variation or change here is that we are dealing with unknown number of arguments for a function
- we typically use a variadic function when we do not know the total number of arguments that will be used for a function
 - one single function could potentially have n number of arguments
 - a variadic function will contribute to the flexibility of the program that you are developing
- the concept of a variadic function is already used in several C's built-in functions
 - in printf when you want to print one number or many numbers
 - `printf(" the one number = %d", nOneNumber);`
 - `printf(" the first number = %d, the second number =%d ", nOneNumber, nSecondNumber);`
- if you look at the stdio.h header, you can see that this was implemented using variadic functions
- you may come up with a need to do this yourself from time to time, so the standard library stdarg.h provides you with routines to write some of your own variadic functions

Using a variadic function

- a variadic function has two parts
 - mandatory arguments
 - at least one is required and is the first one listed
 - order is very important
 - optional arguments
 - listed after mandatory arguments
- for the printf function
 - the first parameter is mandatory ("the one number = %d")
 - the optional part comes second and it could be different(nOneNumber), depending on the situation you are in
- common practice is to have some number that will tell us how many arguments there are (as the first argument)

Creating a variadic function

- when creating a variadic function, you must understand how to reference the variable number of arguments used inside the function
 - you do not know how many there are and you cannot possibly give them names
 - you can solve this problem indirectly, through pointers
- the stdarg.h library header provides you with routines that are implemented as macros (look and operate like functions)
 - you need to use these when implementing your own function with a variable number of arguments
- **va_list**
 - used in situations in which we need to access optional parameters and it is an argument list
 - represents a data object used to hold the parameters corresponding to the ellipsis part of the parameter list
- **va_start**
 - will connect our argument list with some argument pointer
 - the list specified in va_list is the first argument and the second argument is the last fixed parameter

Creating a variadic function

- **va_arg**

- will fetch the current argument connected to the argument list
- we would need to know the type of the argument that we are reading

- **va_end**

- used in situations when we would like to stop using are variable argument list (cleanup)

- **va_copy**

- used in situations for which we need to save our current location
- discussed in next lecture

Creating a variadic function (step 1)

- to create a function with a variable number of arguments, perform the following steps
- provide a function prototype using an ellipsis (three dots)
 - the ellipsis indicates that a variable number of arguments may follow any number of fixed arguments
 - must have at least one fixed argument

```
void f1(int n, ...);           // valid
int f2(const char * s, int k, ...); // valid
char f3(char c1, ..., char c2);  // invalid, ellipsis not last
double f3(...);                // invalid, no parameter
```

Creating a variadic function (Steps 2 and 3)

- create a va_list type variable in the function definition
- initialize the variable to an argument list
 - need to copy the argument list to the va_list variable using va_start

```
double average(double v1, double v2,...) {  
    va_list parg;          // Pointer for variable argument list  
    // More code to go here...  
    va_start( parg, v2);  
    // More code to go here...  
}
```

- first declare the variable parg of type va_list
- call va_start() with this as the first argument and specify the last fixed parameter v2 as the second argument
 - effect of the call to va_start() is to set the variable parg to point to the first variable argument that is passed to the function when it is called
 - still do not know what type of value this represents

Creating a variadic function (step 4)

- access the contents of the argument list using `va_arg()`
 - takes two arguments: a type `va_list` variable and a type name
 - the first time it is called, it returns the first item in the list
 - the next time it is called, it returns the next item, and so on
 - the type argument specifies the type of value returned
 - has to match the specification

```
double someFunction(int lim,...) {  
    va_list ap;          // declare object to hold arguments  
    va_start(ap, lim);   // initialize ap to argument list
```

```
    double tic = va_arg(ap, double); // retrieve first argument  
    int toc = va_arg(ap, int);     // retrieve second argument
```

- If the first argument is 10.0, the above code for `tic` works fine
 - if the argument is 10, the code may not work
 - the automatic conversion of `double` to `int` that works for assignment doesn't take place here

Creating a variadic function (Step 5)

- you should clean up by using the `va_end()` macro as your last step
 - essential to tidy up loose ends left by the process
 - takes a `va_list` variable as its argument
 - resets the `ap` pointer to `NULL`
 - If you omit this call, your program may not work properly
 - variable may not be usable unless you use `va_start` to reinitialize it

```
va_end(ap);           // clean up
```

Summary of rules for variable-length argument lists

- there must be at least one fixed parameter
- must call `va_start()` to initialize the value of the variable argument list pointer in your function
 - this pointer must be declared as type `va_list`
- there needs to be a mechanism to determine the type of each argument
 - either there can be a default type assumed or there can be a parameter that allows the argument type to be determined
 - for example, you could have an extra fixed argument in the `average()` function that would have the value 0 if the variable arguments were double and 1 if they were long
- you must have a way to determine when the list of arguments is exhausted
 - for example, the last argument in the variable argument list could have a fixed value called a sentinel value that can be detected because it's different from all the others
 - OR the first argument could specify the count of the number of arguments in total or in the variable part of the argument list
- you must call `va_end()` before you exit a function with a variable number of arguments
 - If you fail to do so, the function will not work properly

va_copy

Jason Fedin

va_copy

- va_arg() does not provide a way to back up to previous arguments
- it is possible that you may need to process a variable argument list more than once
 - may be useful to preserve a copy of the va_list type variable
 - use va_copy() - two arguments are both type va_list variables, copies the second argument to the first

```
va_list parg;  
va_list parg_copy;  
va_copy(parg_copy, parg);
```

- the first statement creates a new va_list variable, parg_copy
- the next statement copies the contents of parg to parg_copy
- you can then process parg and parg_copy independently to extract argument values using va_arg() and va_end()
- the va_copy() routine copies the va_list object in whatever state it is in
 - if you have executed va_arg() with parg to extract argument values from the list prior to using the va_copy() routine, parg_copy will be in an identical state to parg with some argument values already extracted
- do not use the va_list object parg_copy as the destination for another copy operation before you have executed va_end() for parg_copy

Recursion

Jason Fedin

Overview

- the programs we have discussed are generally structured as functions that call one another in a hierarchical manner
- for some types of problems, it is useful to have functions call themselves
- a recursive function is a function that calls itself either directly or indirectly
- recursive functions can be effectively used to succinctly and efficiently solve problems
 - commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem
- you are unlikely to come across a need for recursion very often
 - provides considerable simplification of the code needed to solve particular problems
- it takes a great deal of practice writing recursive programs before the process will appear natural

Overview

- recursion can be confusing and tricky at first
 - when a function calls itself, there is the immediate problem of how the process stops

```
void Looper(void) {  
    printf("Looper function called.\n");  
    Looper();           // Recursive call to Looper()  
}
```

- calling this function would result in an indefinite number of lines of output
 - after executing the printf() call, the function calls itself
 - there is no mechanism in the code that will stop the process
 - similar to the problem you have with an infinite loop
- a function that calls itself must contain a conditional test (base case) that terminates the recursion

Example

```
int factorial(int n) {  
    //Factorial of 0 is 1 (base case is 0, return 1)  
    if(n==0) return(1);  
  
    return(n*factorial(n-1));  
}
```

- when a recursive function is called with a base case, the function simply returns a result
- when the function is called with a more complex problem
 - the function divides the problem into two conceptual pieces
 - a piece that the function knows how to do
 - a slightly smaller version of the original problem
- the recursion step can result in many more such recursive calls as the function keeps working on the smaller problem
- for recursion to terminate, the sequence of smaller and smaller problems must converge on the base case
 - when the function recognizes the base case, the result is returned to the previous function call
 - a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result

Example

- recursive functions are most commonly illustrated by an example that calculates the factorial of a number
 - the factorial of a positive integer n , written $n!$, is simply the product of the successive integers 1 through n
 - the factorial of 0 is a special case and is defined equal to 1

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

- in the general case, the factorial of any positive integer n greater than zero is equal to n multiplied by the factorial of $n - 1$

$$n! = n \times (n - 1)!$$

- the expression of the value of $n!$ in terms of the value of $(n-1)!$ is called a recursive definition
 - the definition of the value of a factorial is based on the value of another factorial

Example explanation

- the sequence of operations that is performed in the evaluation of factorial (3) can be conceptualized as follows

$$\begin{aligned}\text{factorial (3)} &= 3 * \text{factorial (2)} \\ &= 3 * 2 * \text{factorial (1)} \\ &= 3 * 2 * 1 * \text{factorial (0)} \\ &= 3 * 2 * 1 * 1 \\ &= 6\end{aligned}$$

- it would be a good idea for you to trace through the operation of the factorial() function with a pencil and paper
 - assume that the function is initially called to calculate the factorial of 4
 - list the values of n and result at each call to the factorial() function

Recursion vs. Iteration

- any problem that can be solved recursively can also be solved iteratively (non-recursively using loops)
- both iteration and recursion are based on a control structure
 - iteration uses a repetition structure
 - recursion uses a selection structure
- both iteration and recursion involve repetition
 - iteration explicitly uses a repetition structure
 - recursion achieves repetition through repeated function calls
- iteration and recursion each involve a termination test
 - iteration terminates when the loop-continuation condition fails
 - recursion terminates when a base case is recognized

Recursion vs. Iteration

- iteration with counter-controlled repetition and recursion each gradually approach termination
 - iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail
 - recursion keeps producing simpler versions of the original problem until the base case is reached
- both iteration and recursion can occur infinitely
 - an infinite loop occurs with iteration if the loop-continuation test never becomes false
 - infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case
- a recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem
 - results in a program that is easier to understand and debug

Recursion Pros and Cons

- recursion sometimes offers the simplest solution to some programming problems
- recursive functions can rapidly exhaust a computer's memory resources
 - it repeatedly invokes the mechanism, and consequently the overhead, of function calls
 - expensive in both processor time and memory space
 - each recursive call causes another copy of the function (only the function's variables) to be created (can consume considerable memory)
- avoid using recursion in performance situations
- recursion can be difficult to document and maintain

Tail Recursion

- tail recursion is the simplest form of recursion
 - the recursive call is at the end of the function, just before the return statement
 - the recursive call comes at the end
 - acts like a loop
- tail recursive functions can be optimized by compiler
 - since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use

```
void print(int n) {  
    if (n < 0) return;  
  
    // The last executed statement is recursive call  
    print(n-1);  
}
```

Inline functions

Jason Fedin

Overview

- normally, a function call has overhead when being invoked
 - it takes execution time to set up the call, pass arguments, jump to the function code, and return
- C99 added the concept of inline functions to try and avoid the amount of overhead that comes along with invoking a function
- the point of making a function inline is to hint to the compiler that it is worth making some form of extra effort to call the function faster than it would otherwise
 - usually the compiler will substitute the code of the function into its caller (eliminating the need for a call and return sequence)
 - the program no longer calls that function, the compiler replaces every call to an inline function with the code body of that function
- the inline declaration is only advice to the compiler, which can decide to ignore it
 - may cause the compiler to replace the function call with inline code and/or perform some other sorts of optimizations, or it may have no effect

Overview

- inline functions can improve the runtime performance of a program
 - not guaranteed
 - whether an inline function serves your purpose in a positive or in a negative way depends purely on your code design and is largely debatable
- it is a common misconception that in-lining always equals faster code
 - if there are many lines in inline function or there are more function calls, then in-lining can cause wastage of space
- inline functions can increase the size of the program
 - it increases file size as the same function code is copied again and again in the program wherever it is called
- making a function inline has no effect on the logic of the program from the user's perspective
- it is suggested to only declare functions as inline if they are short and called frequently
 - for a long function, the time consumed in calling the function is short compared to the time spent executing the body of the function
 - no great savings in time using an inline version

declaring inline functions (single file program)

- you can declare an inline function by placing the keyword inline before the function declaration

```
inline void randomFunction();
```

- there are different places to create inline function definitions (same file or header file)
- for the compiler to make inline optimizations, it has to know the contents of the function definition
- the inline function definition has to be in the same file as the function call (internal linkage)
- should always use the inline function specifier along with the static storage-class specifier (using extern less portable)
 - inline functions are usually defined before their first use in a file (definition also acts as a prototype)

```
inline static void foo() // inline definition/prototype
{
    // do something
}
```

declaring inline functions (multi-file program)

- if you have a multi-file program, you need an inline definition in each file that calls the function
 - the simplest way to accomplish this is to put the inline function definition in a header file
 - include the header file in those files that use the function

```
// foo.h
#ifndef FOO_H_
#define FOO_H_
```

```
inline static void foo() {
    // do something
}
#endif
```

- an inline function is an exception to the rule of not placing executable code in a header file
 - because the inline function has internal linkage, defining one in several files does not cause problems

Noreturn functions (C11)

Jason Fedin

Overview

- C11 added a second function specifier (in addition to inline) named `_Noreturn`
- the purpose of this specifier is to inform the user and the compiler that a particular function will not return control to the calling program when it completes execution
 - informing the user helps to prevent misuse of the function
 - informing the compiler may enable it to make some code optimizations
- just like the `inline` function specifier, the `_Noreturn` function specifier is a hint to the compiler
 - using the `_Noreturn` function specifier does not stop a function from returning to its caller
 - only a promise made by the programmer to the compiler to allow it some more degree of freedom to generate optimized code
 - the degree of acceptance is implementation defined

Overview

- the `exit()` function is an example of a `_Noreturn` function
 - once `exit()` is called, the calling function never resumes
- note that this specifier is different from the `void` return type
 - a typical `void` function does return to the calling function
 - it just does not provide an assignable value
- if a function specified with the `_Noreturn` function specifier violates its promise and eventually returns to its caller (by using an explicit `return` statement or by reaching end of function body)
 - the behavior is undefined
 - You MUST NOT return from the function
- compilers are encouraged, but not required, to produce warnings or errors when a `_Noreturn` function appears to be capable of returning to its caller

Using _Noreturn

- the `_Noreturn` keyword appears in a function declaration
- the `_Noreturn` specifier may appear more than once in the same function declaration
 - the behavior is the same as if it appeared once
- this specifier is typically used through the convenience macro `noreturn`
 - provided in the header file `<stdnoreturn.h>`
 - using `_Noreturn` or `noreturn` is fine and equivalent

Example

```
_Noreturn void f () {  
    abort(); // ok  
}
```

```
_Noreturn void g (int i) {  
    // causes undefined behavior if i <= 0  
    if (i > 0) abort();  
}
```

Another example (using noreturn macro)

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

noreturn void my_exit(void);

/* calls exit() and doesn't return to its caller. */
void my_exit(void) {
    printf("Exiting...\n");
    exit(0);
}

int main(void) {
    my_exit();
    return 0;
}
```

Example (Undefined behavior)

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

noreturn void func(void);

void func(void) {
    printf("In func()...\n");
} /* Undefined behavior as func() returns */
```

```
int main(void) {
    func();
    return 0;
}
```

compiler error:[Warning] function declared 'noreturn' has a 'return' statement

Example (Undefined behavior)

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

_Noreturn void foo() {
    return 10;
}

int main(void) {
    printf("Ready\n");
    foo();

    printf("NOT over till now\n");
    return 0;
}
```

compiler error:[Warning] function declared 'noreturn' has a 'return' statement

Overview

Jason Fedin

Overview

- a union is a derived type (similar to a structure) with members that share the same storage space
 - sometimes the same type of construct needs different types of data
- used mainly in advanced programming applications where it is necessary to store different types of data in the same storage area
 - can be used to save space and for alternating data
 - a union does not waste storage on variables that are not being used
- each element in a union is called member
- you can define a union with many members
 - only one member can contain a value at any given time, so only one access of a member at a given time

Overview

- the members of a union can be of any data type
- in most cases, unions contain two or more data types
- it is your responsibility to ensure that the data in a union is referenced with the proper data type
 - referencing data in a union with a variable of the wrong type is a logic error
- the operations that can be performed on a union are
 - assigning a union to another union of the same type
 - taking the address (&) of a union variable
 - accessing union members

Examples

- unions are particularly useful in embedded programming
 - situations where direct access to the hardware/memory is needed
- you could use a union to represent a table that stores a mixture of types in some order
- you could create an array of unions that store equal-sized units
 - each of which can hold a variety of data types
- a union could represent a file containing different record types
- a union could represent a network interface containing different request types

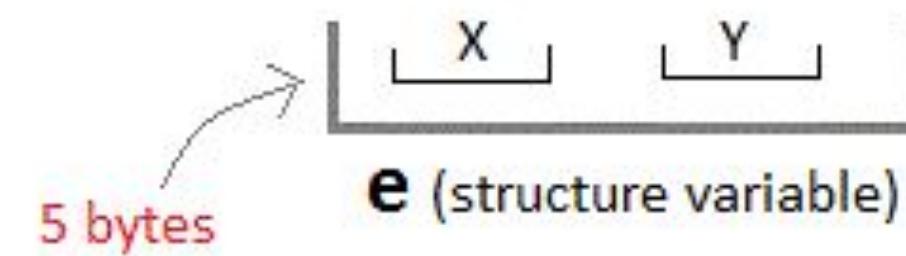
memory allocation for a union

- although structs are similar to unions, the memory allocated for a union is quite different than for a struct
- every time you create an instance of a struct, the computer will lay out the fields in memory, one after the other
 - allocates storage space for all its members separately
- with a union, all the members have an offset of zero (union)
 - one common storage space for all its members
- a union is created with enough space for its largest field
 - the programmer then decides which value will be stored there
- if you have a union called quantity, with fields called count, weight, and volume
 - whether you set the count, weight, or volume field, the data will go into the same space in memory

memory allocation for a union

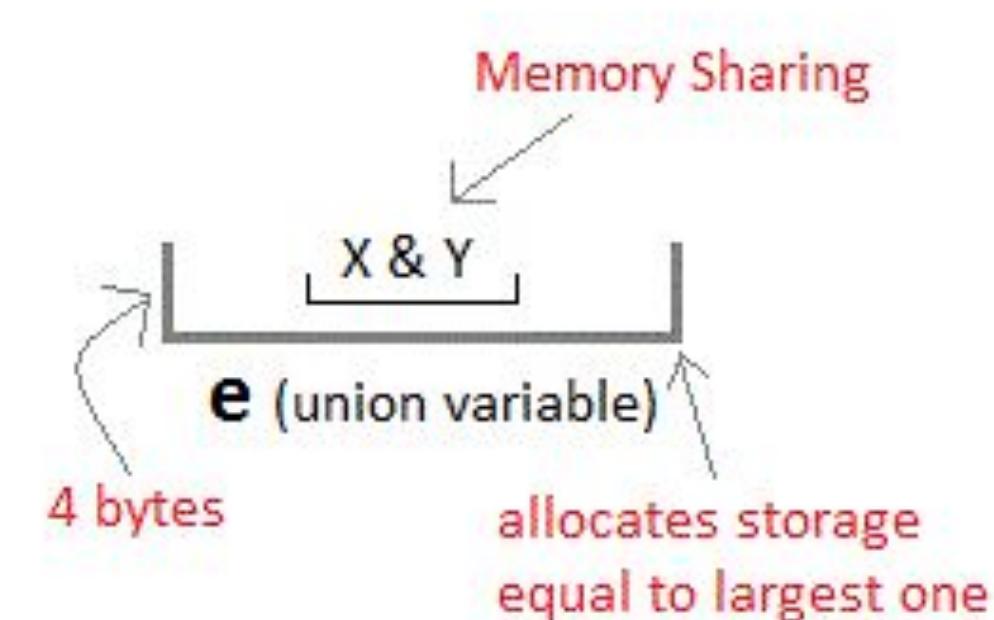
Structure

```
struct Emp  
{  
    char X;      // size 1 byte  
    float Y;     // size 4 byte  
} e;
```



Unions

```
union Emp  
{  
    char X;  
    float Y;  
} e;
```



<https://www.studytonight.com/c/unions-in-c.php>

Example

- if you want to keep track of a quantity of something
 - quantity might be a count, a weight, or a volume
- you could create a struct like this

```
typedef struct {  
    ...  
    short count;  
    float weight;  
    float volume;  
    ...  
} fruit;
```

- there are a few reasons why this is not a good idea
 - it will take up more space in memory
 - someone might set more than one value
 - there is nothing called “quantity”
- a union should be used in this situation
 - you could specify something called quantity in a data type
 - you can decide for each particular piece of data whether you are going to record a count, a weight, or a volume against it

structs vs unions

- although unions are similar to structures, they are used for entirely different situations
- you should use a structure when your construct should be a group of other things
- you should use a union when your construct can be one of many different things but only one at a time
- unions are typically used in situations where space is premium but more importantly for exclusively alternate data
- unions ensure that mutually exclusive states remain mutually exclusive
- unions share a common storage space where structures store several data types simultaneously
 - a structure can hold an int and a double and a char
 - a union can hold an int or a double or a char

structs vs unions

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

<https://www.geeksforgeeks.org/difference-structure-union-c/>

Defining a union

Jason Fedin

Overview

- the declaration for a union is identical to that of a structure
 - except the keyword union is used where the keyword struct is otherwise specified
- a union has the general form

```
union [union tag] {  
    type_1 identifier_1;  
    type_2 identifier_2;  
    ...  
    type_N identifier_N;  
} optional_variable_definitions;
```

- the union tag is optional and each member definition is a normal variable definition
- at the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional
- the union definition is normally placed in a header and included in all source files that use the union type

Defining a union (example)

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

- the above does not define data to contain three distinct members called i, f, and str
 - it defines data to contain a single member that is called either i, f, or str
- a variable of Data type can store an integer, a floating-point number, or a string of characters
 - a single variable (same memory location) can be used to store multiple types of data
- the memory occupied by a union will be large enough to hold the largest member of the union
 - Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string

Creating union variables

- when a union is defined, it creates a user-defined type
 - no memory is allocated
 - to allocate memory for a given union type and work with it, we need to create variables
- a union can be defined to contain as many members as desired
- structures can be defined that contain unions, as can arrays
- pointers to unions can also be declared
 - their syntax and rules for performing operations are the same as for structures

Creating union variables (example)

```
union car {  
    int i_value;  
    float f_value;  
};  
  
int main() {  
    union car car1, car2, *car3;  
    return 0;  
}
```

- another way of creating union variables is:

```
union car {  
    int i_value;  
    float f_value;  
} car1, car2, *car3;
```

Example (displaying the total memory size occupied)

```
union Data {  
    int i;  
    float f;  
    char str[20];  
};  
  
int main() {  
    union Data data;  
    printf( "Memory size occupied by data : %d\n", sizeof(data));  
    return 0;  
}
```

Memory size occupied by data : 20

Anonymous Unions (C11)

- anonymous unions work much the same as anonymous structures
 - an anonymous union is an unnamed member union of a structure or union

```
struct owner {  
    char socsecurity[12];
```

```
};
```

```
struct leasecompany {  
    char name[40];  
    char headquarters[40];  
};
```

```
struct car_data {  
    char make[15];  
    int status; /* 0 = owned, 1 = leased */  
    union {  
        struct owner owncar;  
        struct leasecompany leasecar;  
    };  
};
```

- now, if flits is a car_data structure, we can use flits.owncar.socsecurity

defining a struct containing a union

- the use of a union enables you to define arrays that can be used to store elements of different data types

```
struct {  
    char      *name;  
    enum symbolType  type;  
    union {  
        int   i;  
        float f;  
        char   c;  
    }  data;  
} table [entries];
```

- the above sets up an array called table, consisting of 'entries' elements
 - each element of the array contains a structure consisting of a character pointer called name, an enumeration member called type, and a union member called data
 - each data member of the array can contain either an int, a float, or a char
- the member type might be used to keep track of the type of value stored in the member data
 - you could assign it the value INTEGER if it contained an int
 - FLOATING if it contained a float
 - CHARACTER if it contained a char
 - this information would enable you to know how to reference the particular data member of a particular array element

Using a union and accessing its members

Jason Fedin

Overview

- we can access/assign data to members of a union just like we access/assign data to members of a structure
 - we use the dot (.) operator to access members of a union
 - to access pointer variables, we use the -> operator (indirection operator)
- the dot operator is used with a union name to specify a member of that union

name.member

- the type of name.member is the type specified for member

```
union {  
    int code;  
    float cost;  
} item;
```

```
item.code = 1265; // assigns a value to the code member of the structure item
```

Access pointer members of a union

- you can use the `->` operator with pointers to unions in the same fashion that you use the operator with pointers to structures
- the indirection operator is used with a pointer to a union to identify a member of that union

```
union {  
    int code;  
    float cost;  
} item, * ptrst;
```

```
ptrst = &item;  
ptrst->code = 3451; // assigns an int value to the code member of item
```

- the following three expressions are equivalent
 - `ptrst->code`
 - `item.code`
 - `(*ptrst).code`

Overview

Jason Fedin

Overview (Review)

- the process of creating a C program involves many different steps
 - preprocessor
 - compilation
 - assembler
 - linker
 - loader
- a program called the preprocessor is invoked before any code gets compiled in the C programming language
 - a separate step in the compilation process
 - not part of the compiler
- the C Preprocessor is essentially a text substitution tool
 - instructs the compiler to do required pre-processing before the actual compilation

Overview (Review)

- the C preprocessor mainly performs three tasks on your code
1. removes all the comments
 - a comment is written only for other engineers who need to understand your code
 - it is of no use to a machine
 - the preprocessor removes all comments as they are not required in the execution of the program
 2. includes all of the files from various libraries that the program needs to compile
 - #include directive (includes the contents of the library file specified)
 3. expansion of macro definitions
 - we will discuss macros in the next section
 - small functions that do not contain as much overhead to process as a regular function

Preprocessor Directives

- commands used by the preprocessor are called preprocessor directives and they begin with “#” symbol
 - must be the first nonblank character
 - should begin in the first column
- we have already learned some of them
 - #define (simplest form) and #include
- in the upcoming lectures, we will learn about more of them
 - conditional compilation commands
 - #ifdef, #ifndef, #if, #elif, #else, #endif
 - other directives
 - #undef, #pragma, and #error

Preprocessor operators

- the preprocessor offers operators which can help in creation of macros
 - we will discuss macros in our next section
- these operators are used in the context of the #define directive
- we will learn about the following operators in the next section
 - continuation operator (\)
 - concatenation operators
 - (#) when used within a macro definition, converts a macro parameter into a string constant
 - (##) within a macro definition combines two arguments
 - permits two separate tokens in the definition to be joined into a single token
 - defined()
 - simplifies the writing of compound expressions in certain macro directives

Conditional Compilation

Jason Fedin

Overview

- the C preprocessor offers a feature known as conditional compilation
 - often used to create one program that can be compiled to run on different computer systems
- if you had a large program that had many dependencies on specific hardware or software
 - you might end up with many defines whose values would have to be changed when the program was moved to another computer system
 - you can help reduce this problem by incorporating the values of these defines for each different machine into the program by using the conditional compilation capabilities of the preprocessor
- it is also used to switch on or off various statements in the program
 - debugging statements that print out the values of various variables
 - trace the flow of program execution
- conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code
 - each of the conditional preprocessor directives evaluates an identifier or a constant integer expression
 - cast expressions, sizeof expressions and enumeration constants cannot be evaluated in preprocessor directives

Conditional Compilation commands

- includes a set of commands that tell the compiler to accept or ignore blocks of information or code according to conditions at the time of compilation
- the conditional preprocessor construct is much like the if selection statement
- every #if construct ends with an #endif
- directives #ifdef and #ifndef are provided as shorthand for
 - #if defined(name)
 - #if !defined(name)
- multiple-part conditional preprocessor constructs may be tested with directives #elif and #else

#ifdef

- this directive checks whether an identifier is currently defined
 - identifiers can be defined by a #define directive or on the command line
- to set an identifier, here is an example using the #define directive

```
#define UNIX 1 OR      #define UNIX
```

- most compilers also permit you to define a name to the preprocessor when the program is compiled
 - use the special option -D to the compiler command

```
gcc -D UNIX program.c
```

#ifdef and #endif

- the general form of the #ifdef directive is

#ifdef identifier

- identifiers cannot be any keywords or enumeration constants
- if the identifier specified has been defined by the preprocessor
 - compile all the code up to the next #else or #endif, whichever comes first
 - if there is an #else, all code from the #else to the #endif is compiled if the identifier is not defined
- the #endif directive ends the scope of the #if , #ifdef , #ifndef , #else , or #elif directives
- the #endif directive has the following syntax:

#endif newline

#ifndef

- this directive checks to see if an identifier is not currently defined
 - #ifndef is the negative of #ifdef
 - often used to define a constant if it is not already defined
- the #ifndef directive has the following syntax

#ifndef identifier

- Example

```
#ifndef SIZE
```

```
    #define SIZE 100
```

```
#endif
```

#if

- you can use the #if directive to test the value of a constant expression
 - a constant expression is specified through a #define statement or via the command line when the program is compiled
- the general form of the #if directive

#if constant_expression

- the operand must be a constant integer expression that does not contain any increment (++), decrement (- -), sizeof , pointer (*), address (&), and cast operators
 - you can also use relational and logical operators with the #if directive
- the constant expression is subject to text replacement and can contain references to identifiers defined in previous #define directives
- if an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant zero

#else

- to complement the #ifdef/#ifndef and #if directives, you have the #else directive
 - works exactly the same way as the else statement does
 - identifies a group of directives to be executed or statements to be included if the #ifdef/#ifndef or #if condition fails
- the form #ifdef/#ifndef/#if #else is much like that of the C if else
 - main difference is that the preprocessor does not recognize the braces ({}) syntax of marking a block
 - it uses the #else (if any) and the #endif (which must be present) to mark blocks of directives
 - these conditional structures can be nested
- with an #else directive, everything from the #else to the #endif is done if the identifier is not defined

#else (example in codeblocks)

```
#ifdef UNIX  
# define DATADIR "/uxn1/data"  
#else  
# define DATADIR "\usr\data"  
#endif
```

- the above has the effect of defining DATADIR to "/uxn1/data" if the symbol UNIX has been previously defined and to "\usr\data" otherwise
- you are allowed to put one or more spaces after the # that begins a preprocessor statement
- a value can also be assigned to the defined name on the command line

```
gcc -D DATADIR=/c/my_data
```

#elif

- this directive is used for multiple-choice selections
 - similar to the combined use of the else-if statements in C
 - optional
- the #elif directive has the general form

#elif constant_expression

Include guards and #undef

Jason Fedin

include guards using #ifndef

- the #ifndef directive is commonly used to prevent multiple inclusions of a file
- many include files include other files, so you may include a file explicitly that another include file has already included
- this is a problem because some items that appear in include files, such as declarations of structure types, can appear only once in a file
 - prevents multiple definitions of the same variable/function/macro
- the standard C header files uses the #ifndef technique to avoid multiple inclusions
- one problem is to make sure the identifier you are testing has not been defined elsewhere
 - use the filename as the identifier (using uppercase, replacing periods with an underscore, and using an underscore)

```
#ifndef _STDIO_H  
#define _STDIO_H  
// contents of file  
#endif
```

include guards using #ifndef (example)

```
#ifndef THINGS_H_
#define THINGS_H_
/* rest of include file */
#endif
```

- the definition in the first header file included becomes the active definition and subsequent definitions in other header files are ignored
 - the first time the preprocessor encounters this include file, THINGS_H_ is undefined, so the program proceeds to define THINGS_H_ and to process the rest of the file
 - the next time the preprocessor encounters this file, THINGS_H_ is defined, so the preprocessor skips the rest of the file
 - ensures that the contents of a header file cannot be included more than once into a source file
- using an include directive makes it impossible for the contents of MyHeader.h to appear more than once in a source file
- you should always protect code in your own header files in this way

#undef directive

- another flexibility that you have with preprocessor directives is the ability to undefine an identifier you have previously defined
- on some occasions, you might need to cause a defined name to become undefined
 - cancels an earlier #define definition
- you can accomplish this using the #undef directive

```
#undef name
```

```
#undef LINUX
```

- the above removes the definition of LINUX
 - subsequent #ifdef LINUX statements will evaluate as FALSE

(Other preprocessor directives) #pragma and #error

Jason Fedin

#pragma

- the #pragma directive lets you place compiler instructions in your source code
 - lets you request special behavior from the compiler
- this directive is most useful for programs that are unusually large or that need to take advantage of the capabilities of a particular compiler
- #pragmas can be used
 - to control the amount of memory set aside for automatic variables
 - to set the strictness of error checking
 - to enable nonstandard language features
- the pragma (pragmatic information) directive is part of the standard
 - the meaning of any pragma depends on the software implementation of the standard that is used
 - generally, each compiler has its own set of pragmas
- for example, while C99 was being developed, it was referred to as C9X, and one compiler used the following pragma to turn on C9X support

#pragma c9x on

#pragma

- the syntax of the #pragma directive is:

```
#pragma token_name
```

- #pragma is usually followed by a single token
 - represents a command for the compiler to obey
- there are only a limited list of supported tokens for each standard/compiler
 - the set of commands that can appear in #pragma directives is different for each compiler
 - need to reference the compiler documentation
 - a pragma not recognized by the implementation is ignored
- we will be studying the following #pragmas (which are available in the gcc compiler)
 - #pragma GCC dependency
 - #pragma GCC poison
 - #pragma GCC system_header
 - #pragma once
 - #pragma GCC warning
 - #pragma GCC error
 - #pragma message

#pragma GCC dependency

- #pragma GCC dependency
 - allows you to check the relative dates of the current file and another file
 - If the other file is more recent than the current file, a warning is issued
- this pragma is useful if the current file is derived from the other file, and should be regenerated

#pragma GCC dependency "parse.y"

#pragma GCC dependency "/usr/include/time.h" rerun fixincludes

#pragma GCC poison

- this directive is used to remove an identifier completely from the program
- sometimes, there is an identifier that you want to remove completely from your program, and make sure that it is never used
- to enforce this, you can poison the identifier with this pragma
 - followed by a list of identifiers to poison
 - if any of those identifiers appear anywhere in the source after the directive, an error will be displayed by the compiler

```
#pragma GCC poison printf sprintf fprintf  
sprintf(some_string, "hello"); // will produce an error
```

#pragma GCC system_header

- the #pragma GCC system_header tells the compiler to consider the rest of the current include file as a system header
 - code that comes before the '#pragma' in the file is not affected
 - this pragma takes no arguments
- system headers are header files that come with the OS or compiler
- GCC gives code found in system headers special treatment
- this pragma can affect the severity of some diagnostic messages
 - all warnings are suppressed while GCC is processing a system header
 - macros defined in a system header are immune to a few warnings wherever they are expanded

#pragma warning, errors, and message

- #pragma once
 - specifies that the header file containing this directive is included only once even if the programmer includes it multiple times during a compilation
 - works similar to using include guards
 - a less-portable alternative to using '#ifndef' to guard the contents of header files against multiple inclusions
- #pragma GCC warning "message"
 - causes the preprocessor to issue a warning diagnostic with the text 'message'
 - message contained in the pragma must be a single string literal
- #pragma GCC error "message"
 - causes the preprocessor to issue an error with the text 'message'
 - message contained in the pragma must be a single string literal
- #pragma message "message"
 - prints string as a compiler message on compilation
 - the message is informational only (not a warning or an error)

#error directive

- one of the least used but potentially most useful features of the C preprocessor is the ANSI-specified #error directive
- the #error directive causes the preprocessor to issue an error message that includes any text in the directive
 - error message is a sequence of characters separated by spaces
 - you do not have to enclose the text in quotes
 - the message is optional

Example

```
#if __STDC_VERSION__ != 201112L // should fail if compiler used is an older standard and succeed when it uses C11
#error Not C11
#endif
```

- gcc test.c
- test.c:8:2: error: #error Not C11
- ```
#error Not C11
```

# #error directive

---

- this directive can be used for code that compiles, but, still never the less is incorrect
- Useful in the following situations

## Incomplete code

```
#error *** Jason - Function incomplete. Fix before using ***
```

## Compiler-dependent code

```
#include <float.h>
#include <limits.h>
#if (INT_MAX != 32767)
#error *** This file only works with 16-bit int.
Do not use this compiler! ***
```

## Conditionally-compiled code

```
#ifdef OPT_1
/* Do option_1 */
#elif defined OPT_2
/* Do option_2 */
#else
#error *** You must define one of OPT_1 or OPT_2 ***
#endif
```

# Other directives

---

- many compilers support a `#warning` directive
  - when one is encountered, the compiler emits a diagnostic containing the remaining tokens in the directive

`#warning` message

- the `#line` preprocessor directive is used to set the file name and the line number of the line following the directive to new values
  - used to set the `_FILE_` and `_LINE_` macros

`#line` linenum

`#line` linenum filename

---

# Overview

Jason Fedin

# Overview

---

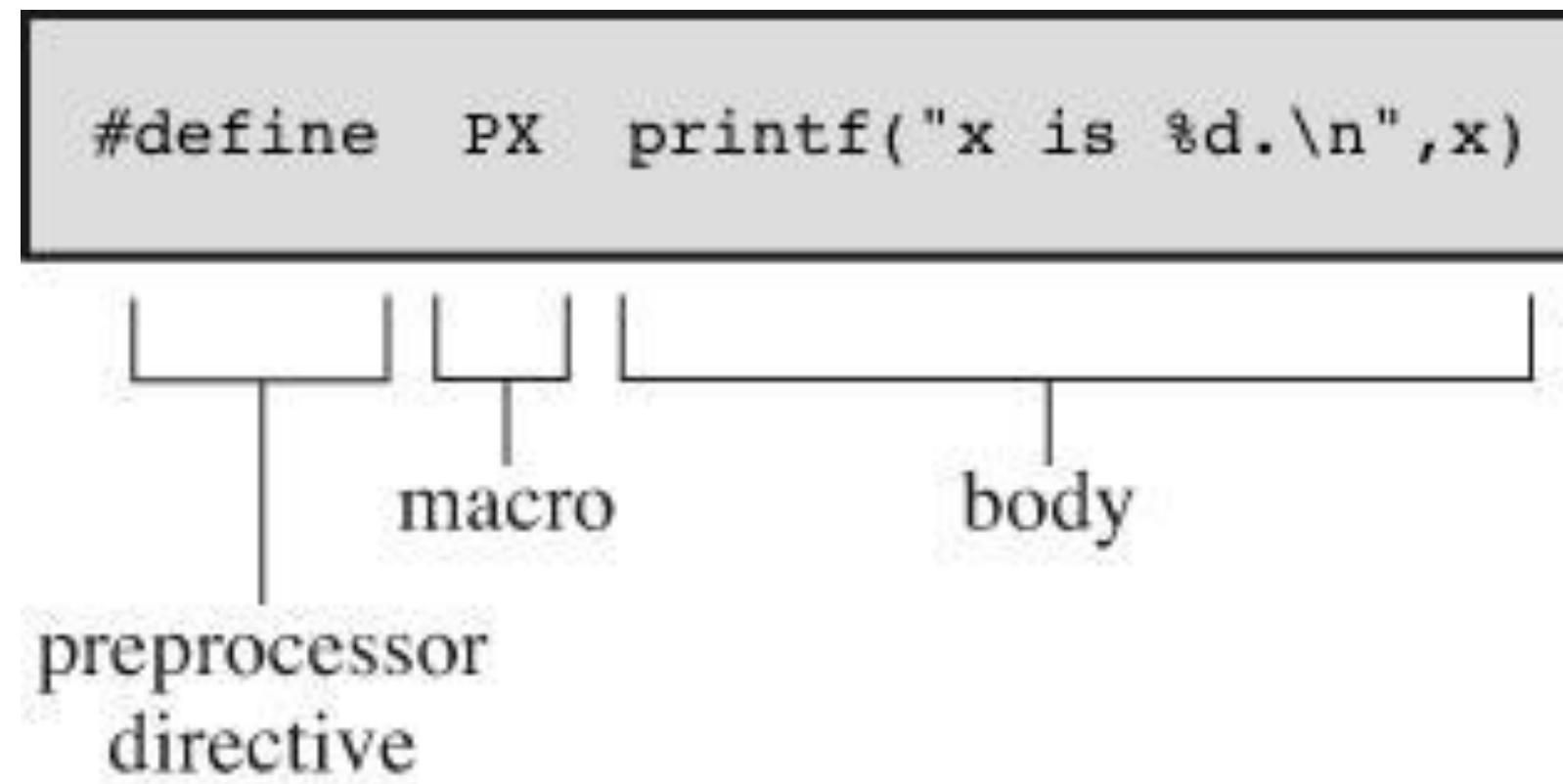
- a macro is essentially a piece of code based on the #define directive
  - technically, we have already learned about macro's when we learned about symbolic constants
- when you first learn about macros, you probably think that they are nothing more than a function call with some strange syntax
  - and you would mostly be right, they “behave” similar to normal functions
- macros are a text processing feature and are “expanded” and replaced by macro definitions
- today, macros in C are considered outdated in terms of modern programming practices
  - however, they are still widely used because they make things easier for the programmer

# Syntax

---

```
#define MACRO macro_body
```

- each #define line has three parts
  - the first part is the #define directive itself
  - the second part is your chosen abbreviation, known as a macro name
  - the third part (the remainder of the line) is termed the replacement list or body
    - preprocessor replaces macro name with the macro body



C Primer Plus, Sixth Edition by Stephen Prata

# Convention

---

- you should use capital letters for macro function names
  - not as widespread as that of using capitals for macro constants
  - one good reason for using capitals is to remind yourself to be alert to possible macro side effects
- there are no spaces in the macro name, however, spaces can appear in the replacement string (macro\_value)
- macros are also not terminated by a semicolon
- macros run until the first newline following the #
  - limited to one line in length by default unless you use the backslash operator (\) for continuation

---

# Macros vs. Functions

Jason Fedin

# Overview

---

- many tasks can be done by using a macro with arguments or by using a function
- as we have learned in the last lecture, macros are essentially functions, but, with different syntax
  - behave just like a function
- however, there are some huge differences (under the hood) and without understanding these differences, you might be using either one when you should not be
- you must understand that macros are pre-processed which means that all the macros would be processed before your program compiles
  - functions are not preprocessed, they are compiled
- so, now the question becomes, what are the differences and when should I use one vs. the other
  - there is no hard-and-fast rule
- the main consideration becomes a trade off between time and space

# Speed

---

- a macro is faster than a function
  - program control must shift to where the function is and then return to the calling program
  - takes longer than inline code
- if you intend to use a macro instead of a function primarily to speed up a program
  - first try to determine whether it is likely to make a significant difference
- a macro that is used once in a program probably will not have any noticeable improvement in running time
- a macro inside a nested loop is a much better candidate for speed improvements
  - many systems offer program profilers to help you pin down where a program spends the most time

# Space

---

- when you call a function, it has to allocate some data (a newly allocated stack frame)
  - macros do not have this overhead
  - macros insert code directly into the program (textual substitution)
- a macro produces inline code (a statement in your program)
  - if you use the same macro 20 times, you get 20 lines of code inserted into your program
  - if you use a function 20 times, you have just one copy of the function statements in your program (less space is used)
- functions are preferred over macros when writing large chunks of code

# Other considerations

---

- macros have an advantage in that they do not have to worry about variable types
  - deal with character strings, not with actual values
  - a macro can be used equally well with an int or float
    - just substitute the argument you pass them
- functions give you type checking
  - if a function expects a string, but you give it an int, you will get an error
- macros are somewhat trickier to use than regular functions because they can have odd side effects if you are not careful
  - some compilers limit the macro definition to one line, and it is probably best to observe that limit, even if your compiler does not
- it is much harder to debug a macro than when you use a function
  - a function can be stepped through by the debugger and a macro cannot
  - when a macro fails, the only way you can find out what the problem is, is by looking at its definition and trying to find out what happened
    - you can tell gcc to tell you how everything expands using `gcc -E source_file.c`

# Alternatives

---

- inline functions are the best alternative to macros
- when you add the inline keyword in front of a function, you are hinting to the compiler to embed the function body inside the caller (just like a macro)
- inline functions can be debugged
- inline functions also have type checking
- however, the inline keyword is merely a hint to the compiler, it is not a strict rule and the compiler can decide to ignore the hint
- macros will always have their place and are not going away

---

# Creating your own Macro

Jason Fedin

# Symbolic constants

---

- as mentioned previously, there are two ways of defining macros, one of which looks like a function and one which does not
  - symbolic constants (constants represented as symbols)
  - function macros (operations defined as symbols)
- the below definition does not look like a function (symbolic constant)

```
#define NONFMAC some text here
```

- defines a macro and some replacement text
  - the replacement text for a macro is any text on the line after the identifier in the #define directive
- after the definition, the macro can be used as follows

```
NONFMAC
/* some text here */
```

- leading or trailing white space around the replacement text is discarded
- its name is simply replaced by its replacement text

# Function Macros

---

- the below definition looks more like a function
  - function macro (operations defined as symbols)

```
#define FMAC(a,b) a here, then b
```

```
#define macro_name(list_of_identifiers) substitution_string
```

- defines a macro and some replacement text
  - the list of identifiers separated by commas (a, b) appears between parentheses following the macro\_name (FMAC)
  - each of these identifiers can appear one or more times in the substitution string
- after the definition, the macro can be used as follows

```
FMAC(first text, some more)
```

```
/* first text here, then some more */
```

- leading or trailing white space around the replacement text is discarded
- it gets a bit trickier with the function macro because of the (identifiers) or formal parameters

# macros with arguments

---

- you can create function-like macros that look and act much like functions when you pass data to them (arguments)
  - looks very similar to a function because the arguments are enclosed within parentheses
- function-like macro definitions have one or more arguments in parentheses, and these arguments then appear in the replacement portion
- to create a macro with arguments, put them in parentheses separated by commas after the macro name

```
#define SQUARE(X) X*X
```

- can be used in program like the below

```
z = SQUARE(2);
```

- looks like a function call, but it does not necessarily behave identically

# macros with arguments

---

```
#define macro_name(list_of_identifiers) substitution_string
```

- a macro that contains arguments has its arguments substituted in the replacement text when the macro is expanded
  - the replacement text for a macro is any text on the line after the identifier in the #define directive
  - the replacement-text replaces the identifier and argument list in the program
- use should use parentheses around each argument and around the definition as a whole
  - ensures that the enclosed terms are grouped properly in an expression (avoid operator precedence)

---

# Preprocessor Operators

Jason Fedin

# Overview

---

- the preprocessor includes the following four operators which can make it easier to create macros
  - \
  - defined
  - #
  - ##
- the backslash (\) operator allows for the continuation of a macro to the next line when the macro is too long for a single line
  - a macro is always a single, logical line
- the defined() operator is used in constant expressions to determine if an identifier is defined using #define

# Overview

---

- the # and ## operators are used for concatenation
  - often useful to merge two tokens into one while expanding macros, called token pasting or token concatenation
- the (#) operator is used within a macro definition
  - causes a replacement text token to be converted to a string surrounded by quotes
- the ('##') operator performs token pasting
  - concatenates two tokens
  - when a macro is expanded, the two tokens on either side of each '##' operator are combined into a single token

# (\ ) operator

---

- a macro is normally confined to a single line
  - the macro continuation operator (\ ) is used to continue a macro that is too long for a single line

```
#define min(x, y) \
((x)<(y) ? (x) : (y))
```

- the above macro definition continues on the second physical line with the first nonblank character found
  - you can position the text on the second line to wherever you feel is the most readable
  - the \ must be the last character on the line, immediately before you press Enter
  - result is seen by the compiler as a single, logical line

# defined() operator

---

- the defined operator is used in constant expressions to determine if an identifier is defined using #define
- if the specified identifier is defined, the value is true (non-zero)
- if the symbol is not defined, the value is false (zero)
- It can also be used in #if statements

```
#if defined (DEBUG)
...
#endif
```

```
// the same as the above, but, using #ifdef
#ifdef DEBUG
...
#endif
```

# defined() operator (example)

---

```
#if defined(WINDOWS) || defined(WINDOWSNT)
define BOOT_DRIVE "C:/"
#else
define BOOT_DRIVE "D:/"
#endif

int main(void) {
 printf("Here is the boot drive path: %s\n", BOOT_DRIVE);
 return 0;
}
```

- the above will display as output "C:/" if either WINDOWS or WINDOWSNT is defined
  - output will be "D:/" otherwise

# (# and ##) operators

---

- these operators are often useful to merge two tokens into one while expanding macros
  - token pasting or token concatenation
- the # operator causes a replacement text token to be converted to a string surrounded by quotes (String expansion)
- the ‘##’ preprocessing operator performs token pasting
  - The two tokens on either side of each ‘##’ operator are combined into a single token
  - replaces the ‘##’ and the two original tokens in the macro expansion

# the # operator

---

- if you place a # in front of a parameter in a macro definition
  - creates a constant string out of the macro argument when the macro is invoked
  - literally inserts double quotation marks around the actual macro argument

```
#define str(x) # x
causes the subsequent invocation
str (testing)
```

to be expanded into  
"testing"

- `printf (str (Programming in C is fun.\n));`  
is therefore equivalent to
- `printf ("Programming in C is fun.\n");`

# the # operator

---

```
#define HELLO(x) printf("Hello, " #x "\n");
```

when HELLO(John) appears in a program file, it is expanded to

```
printf("Hello, " "John" "\n");
```

- the string "John" replaces #x in the replacement text
  - strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to

```
printf("Hello, John\n");
```
- the # operator must be used in a macro with arguments because the operand of # refers to an argument of the macro

# a more practical example

---

```
#define printint(var) printf (# var " = %i\n", var)
```

- this macro is used to display the value of an integer variable
  - if count is an integer variable with a value of 100, the statement

printint (count);

is expanded into

```
printf ("count" " = %i\n", count);
```

- which, after string concatenation is performed on the two adjacent strings, becomes

```
printf ("count = %i\n", count);
```

- the # operator gives you a means to create a character string out of a macro argument
  - a space between the # and the parameter name is optional

# the ## Operator

---

- this operator is used in macro definitions to join two tokens together
  - preceded (or followed) by the name of a parameter to the macro
  - takes the actual argument to the macro that is supplied when the macro is invoked and creates a single token out of that argument and whatever token follows (or precedes) the ##
- usually both will be identifiers, or one will be an identifier and the other a preprocessing number
  - when pasted, they make a longer identifier
- two tokens that don't together form a valid token cannot be pasted together
  - you cannot concatenate x with + in either order
- this operator is most useful when one or both of the tokens comes from a macro argument
  - if either of the tokens next to an '##' is a parameter name
    - replaced by its actual argument before '##' executes

```
#define TOKENCONCAT(x, y) x ## y // TOKENCONCAT(O, K) is replaced by OK in the program
```

---

# another example

---

```
#define concatenate(x, y) x ## y
```

...

```
int xy = 10;
```

...

will make the compiler turn

```
printf("%d", concatenate(x, y));
```

into

```
printf("%d", xy);
```

which will, of course, display 10 to standard output

# another example

---

- it is possible to concatenate a macro argument with a constant prefix or suffix to obtain a valid identifier as in

```
#define make_function(name) int my_ ## name (int foo) {}
make_function(bar)
```

- will define a function called my\_bar()

```
#define eat(what) puts("I'm eating " #what " today.")
eat(fruit)
```

which the macro-processor will turn into  
puts( "I'm eating " "fruit" " today." )

- which in turn will be interpreted by the C parser as a single string constant

# another example

---

- consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command {
 char *name;
 void (*function) (void);
};
```

```
struct command commands[] = {
 { "quit", quit_command },
 { "help", help_command },
 ...
};
```

- it would be cleaner not to have to give each command name twice, once in the string constant and once in the function name
- a macro which takes the name of a command as an argument can make this unnecessary
  - the string constant can be created with stringizing, and the function name by concatenating the argument with '\_command'

```
#define COMMAND(NAME) { #NAME, NAME ## _command }
struct command commands[] = {
 COMMAND (quit),
 COMMAND (help),
 ...
};
```

---

# Predefined Macros

Jason Fedin

# Overview

---

- there are usually a considerable number of standard preprocessing macros defined by the compiler (defined in the documentation)
- standard C provides predefined symbolic constants
  - begin and end with two underscores.
  - cannot be used in #define or #undef directives
- \_FILE\_
  - represent the current file name (string)
- \_LINE\_
  - represent the current line number of the current source code (an integer constant)
- \_func\_
  - the name of any function when placed inside a function of the current file
  - not part of the standard

# Overview

---

- DATE
  - the date the source file was compiled (a string of the form "Mmm dd yyyy" such as "Jan 19 2002")
- TIME
  - the time the source file was compiled (a string literal of the form "hh:mm:ss")
- STDC
  - used to indicate if the compiler supports standard C by returning the value 1

# \_\_FILE\_\_ and \_\_LINE\_\_

---

- the `__FILE__` macro represents the name of the current source file as a string literal
  - typically a string literal comprising the entire file path
    - "C:\\Projects\\Test\\MyFile.c"
- the `__LINE__` macro results in an integer constant corresponding to the current line number
  - can use this in combination with the `__FILE__` macro to identify where in the source code a particular event or error has been detected

```
if(fopen(&myfile, filename, "rb")) // Open for binary read
{
 fprintf(stderr, "Failed to open file in %s line %d\n", __FILE__, __LINE__);
 return -1;
}
```

- if `fopen()` fails, there will be a message specifying the source file name and the line number within the source file where the failure occurred

## \_\_func\_\_

---

- you can always obtain the name of any function in the code that represents the function body by using the identifier `__func__`

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

```
void my_function_name(void) {
 printf("%s was called.\n", __func__);
}
```

- this function just outputs its own name within the format string

my\_function\_name was called

# \_\_DATE\_\_ and \_\_TIME\_\_

---

- the \_\_DATE\_\_ macro generates a string representation of the date in the form Mmm dd yyyy
  - Mmm is the month in characters, such as Jan, Feb, and so on
  - dd is the day in the form of a pair of digits 1 to 31, where single-digit days are preceded by a space
  - yyyy is the year as four digits—2012
- a similar macro, \_\_TIME\_\_, provides a string containing the value of the current time in the form hh:mm:ss
  - digits for hours, minutes, and seconds, separated by colons
  - the time is when the compiler is executed, not when the program is run
- you could use this macro to record in the output when your program was last compiled with this statement:

```
printf("Program last compiled at %s on %s\n", __TIME__, __DATE__);
```

Program last compiled at 13:47:02 on Nov 24 2012

# \_STDC\_

---

- this macro expands to the constant 1 if the current compiler being used conforms to the ISO standard C compiler

```
#include <stdio.h>

int main(void) {
 #if (_STDC_ == 1)
 printf("Implementation is ISO-conforming.\n");
 #else
 printf("Implementation is not ISO-conforming.\n");
 #endif
 /* ... */

 return 0;
}
```

---

# GCC Compiler Options

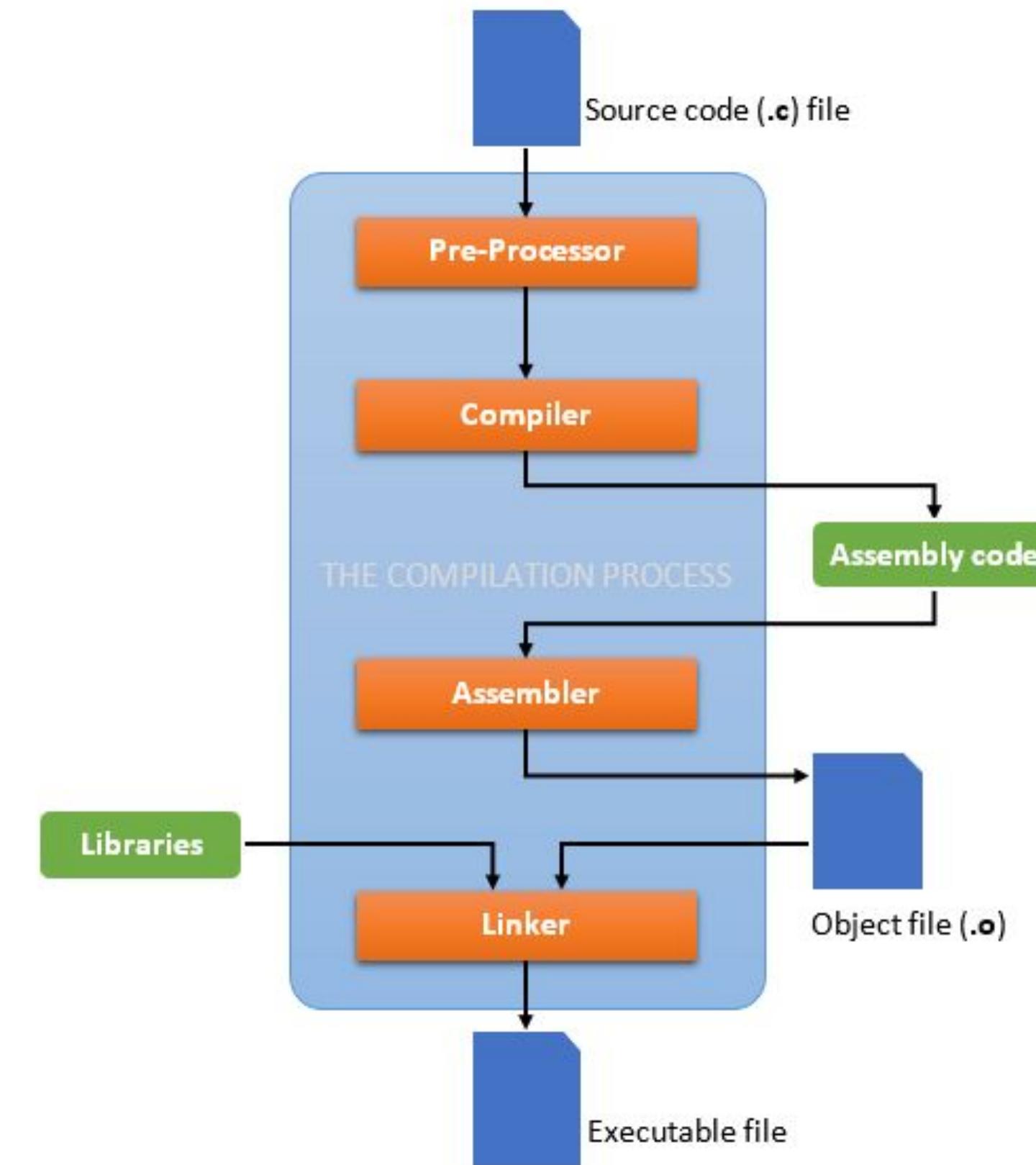
Jason Fedin

# Overview

---

- up to this point, we should all understand the compilation process and how to compile and link a program written in the C programming language
- in this lecture, I would like to describe more details about specific options that you can pass to the compiler that will help with the process of debugging, optimization and other enhancements
- the compiler that we are using is the gcc compiler (everyone should have installed this)
  - a very powerful and popular C compiler for various Linux distributions
  - able to use on windows because of Cygwin
    - a Unix-like environment and command-line interface for Windows (also includes the bash shell)
  - able to use a front end version named clang on mac
- when you invoke GCC, it normally does preprocessing, compilation, assembly and linking
  - some of the options to the compiler allow you to stop this process at an intermediate stage
    - for example, the -c option says not to run the linker
  - other options are passed on to one stage of processing
    - some options control the preprocessor and others the compiler itself
    - other options control the assembler and linker

# Compilation steps



<https://codeforwin.org/2017/08/c-compilation-process.html>

# Overview

---

- the gcc program accepts options and file names as operands
  - many options have multi-letter names
    - therefore multiple single-letter options may not be grouped
      - -dr is very different from -d -r
  - you can mix options and other arguments
  - for the most part, the order you use does not matter
  - order does matter when you use several options of the same kind
    - for example, if you specify -L more than once, the directories are searched in the order specified
  - many options have long names starting with -f or with -W--for example
    - -fforce-mem, -fstrength-reduce, -Wformat and so on

---

# GCC Compiler Options

Jason Fedin

# Optimization flags

---

- there are options to control various sorts of optimizations
- without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results
- turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program
- the compiler performs optimization based on the knowledge it has of the program
  - compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them
- not all optimizations are controlled directly by a flag
- most optimizations are completely disabled
- you can invoke GCC with -Q --help=optimizers to find out the exact set of optimizations that are enabled at each level

# Optimization flags

---

- -O
  - with -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
- -O1
  - optimizing compilation takes somewhat more time, and a lot more memory for a large function
- -O2
  - optimize even more
  - GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff
  - ss compared to -O, this option increases both compilation time and the performance of the generated code
- -O3
  - O3 turns on all optimizations specified by -O2 and also turns on more
- -Ofast
  - Disregard strict standards compliance. -Ofast enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on -ffast-math, -fallow-store-data-races and the Fortran-specific -fstack-arrays, unless -fmax-stack-var-size is specified, and -fno-protect-parens.
- -Og
  - Optimize debugging experience. -Og should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience. It is a better choice than -O0 for producing debuggable code because some compiler passes that collect debug information are disabled at -O0

# GCC Environment Variables

---

- GCC uses the following environment variables
  - PATH
    - for searching the executables and run-time shared libraries (.dll, .so)
  - CPATH
    - for searching the include-paths for headers
    - searched after paths specified in -I<dir> options
    - C\_INCLUDE\_PATH can be used to specify C headers if the particular language was indicated in pre-processing
  - LIBRARY\_PATH
    - for searching library-paths for link libraries
    - it is searched after paths specified in -L<dir> options

# Misc. Tools

---

- "nm" is commonly-used to check if a particular function is defined in an object file
  - a 'T' in the second column indicates a function that is defined
  - a 'U' indicates a function which is undefined and should be resolved by the linker
- "ldd" Utility (List Dynamic-Link Libraries)
  - examines an executable and displays a list of the shared libraries that it needs

ldd a.out

linux-vdso.so.1 (0x00007ffccddde000)

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f9b389cf000)

/lib64/ld-linux-x86-64.so.2 (0x00007f9b38fc2000)

---

# Debugging with the preprocessor

Jason Fedin

# Overview

---

- we have previously discussed preprocessor directives in detail
  - specifically the conditional compilation directives (if, ifdef, else)
- by using these directives, you can arrange for blocks of code to be included in your program to assist in debugging
- the addition of tracing code of your own can still be very useful
- you have complete control of the formatting of data to be displayed for debugging purposes
  - can even arrange for the kind of output to vary according to conditions or relationships within the program
- in this lecture, I will show you how to use the preprocessor to allow for the conditional inclusion of debugging statements in your program

# Overview

---

- the C preprocessor can be used to insert debugging code into your program
- by appropriate use of #ifdef statements, the debugging code can be enabled or disabled at your discretion
  - when a preprocessor identifier DEBUG is defined, the debugging code is compiled with the rest of the program
  - when DEBUG is not defined, the debugging code is left out

---

# Debugging with gdb

Jason Fedin

# Overview

---

- GDB (GNU Project debugger) allows you to see what is going on 'inside' another program while it executes
  - or what another program was doing at the moment it crashed
- gdb is a powerful interactive debugger that is frequently used to debug programs compiled with GNU's gcc compiler
- GDB can do four main kinds of things to help you catch bugs in the act
  - start your program, specifying anything that might affect its behavior
  - make your program stop at a predetermined location
  - examine what has happened, when your program has stopped (display variables)
  - change things in your program (set variables)
    - so you can experiment with correcting the effects of one bug and go on to learn about another
- you can trace your program's execution and even execute it one line at a time

# Overview

---

- the programs that you use gdb on may be executing on the same machine as GDB (native), on another machine (remote), or on a simulator
- GDB can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X
- your C program must be compiled with the gcc compiler using the -g option to make full use of gdb's features
  - causes the C compiler to add extra information to the output file
    - variable and structure types, source filenames, and C statement to machine code mappings

---

# Debugging with gdb

Jason Fedin

# Inserting Breakpoints

---

- the break command can be used to set breakpoints in your program
- a breakpoint is just as its name implies
  - a point in your program that, when reached during execution, causes the program to “break” or pause
  - program’s execution is suspended, which allows you to do things such as look at variables and determine precisely what’s going on at the point
- a breakpoint can be set at any line in your program by simply specifying the line number to the command
- If you specify a line number but no function or filename, the breakpoint is set on that line in the current file
  - if you specify a function, the breakpoint is set on the first executable line in that function

---

# Core files

Jason Fedin

# Overview

---

- a core dump is generated when a program crashes or is terminated abnormally because of segmentation fault
  - possibly division by zero or attempts to access past the end of an array (illegal memory access)
- a segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you”
  - a piece of code tries to do read and write operation in a read only location in memory or freed block of memory
- a core dump is also called a memory dump, storage dump or dump
- a core dump results in the creation of a file
  - this core file contains a snapshot of the contents of the process's memory at the time it terminated
    - this file is used for debugging purposes (used to assist in diagnosing and fixing errors)
  - core dumps allow a user to save a crash for later or off-site analysis, or comparison with other crashes
  - core dumps can be used to capture data freed during dynamic memory allocation and may thus be used to retrieve information from a program that is no longer running

# Overview

---

- a core dump is contained in a file named something similar to "core.<process\_id>"
  - created in current working directory
- on windows which uses Cygwin
  - the dump may not be named core
  - It will likely be named something like "mybadprog.exe.stackdump"
- your system might be configured to disable the automatic creation of this core file, often due to the large size of these files
  - to enable writing core files you use the ulimit command
    - ulimit -c unlimited
    - may have to type in "bash" to make sure you are in a shell
    - otherwise command may not be recognized
- to "view" a core dump you will need a debugger
  - it will allow you to examine the state of the process
  - includes listing the stack traces for all the threads of the process
  - you can also print the values of variables and registers
- analyzing a core file does not work well without debugging information added to the executable (gcc -g)
- we will use gdb to analyze our core files

---

# Profiling

Jason Fedin

# Overview

---

- profiling is a form of dynamic program analysis that measures
  - space (memory)
  - time complexity of a program (efficiency)
  - usage of particular instructions
  - the frequency and duration of function calls
- profiling information serves to aid program optimization
- profiling is achieved by instrumenting either the program source code or its binary executable form using a specific profiling tool
- profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods

# gprof

---

- in this lecture, we will discuss the gnu profiling tool gprof and valgrind
- the GNU profiler gprof uses a hybrid approach of compiler assisted instrumentation and sampling
- Instrumentation is used to gather function call information (e.g. to be able to generate call graphs and count the number of function calls)
- to gather profiling information at runtime, a sampling process is used
  - the program counter is probed at regular intervals by interrupting the program with operating system interrupts
  - the resulting profiling data is not exact but are rather a statistical approximation gprof statistical inaccuracy

# conclusion and comparison

---

- gprof is the dinosaur among the evaluated profilers
  - goes back into the 1980's
  - it was widely used and a good solution during the past decades
  - limited support for multi-threaded applications, the inability to profile shared libraries make it unsuitable for using it in today's real-world projects
- Valgrind delivers the most accurate results and is well suited for multi-threaded applications
  - slow execution of the application under test disqualifies it for larger, longer running applications
- there is another tool available called gperftools (from google)
  - CPU profiler has a very little runtime overhead
  - provides some nice features like selectively profiling certain areas of interest and has no problem with multi-threaded applications

---

# Static Analysis

Jason Fedin

# Static analysis

---

- static analysis is a method of debugging by automatically examining source code before a program is run
  - do not actually run the program
  - contrast with dynamic analysis (profiling), analysis performed on programs while they are executing
- the analysis is performed on some version of the source code
- the process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards
- this type of analysis addresses weaknesses in source code that might lead to vulnerabilities
  - could also be achieved through manual code reviews
- analysis is often performed by an automated tool
  - much more effective than code reviews
- the sophistication of the analysis performed by tools varies
  - from those that only consider the behavior of individual statements and declarations
  - to those that include the complete source code of a program in their analysis
- software metrics and reverse engineering can be described as forms of static analysis

# Static Analysis vs. Dynamic Analysis

---

- both types of analysis detect defects
- the big difference is where they find defects in the development lifecycle
- static analysis identifies defects before you run a program (e.g., between coding and unit testing)
- dynamic analysis identifies defects after you run a program (e.g., during unit testing)
  - some coding errors might not surface during unit testing
- there are defects that dynamic testing might miss that static code analysis can find

# Benefits of the Best Static Code Analysis Tools

---

- the best static code analysis tools offer speed, depth, and accuracy.
- Speed
  - It takes time for developers to do manual code reviews
  - automated tools are much faster
- Static code checking addresses problems early on
  - pinpoints exactly where the error is in the code
  - you will be able to fix those errors faster
  - coding errors found earlier are less costly to fix
- Depth
  - testing cannot cover every possible code execution path, but a static code analyzer can
- Accuracy
  - manual code reviews are prone to human error, however, automated tools are not
  - they scan every line of code to identify potential problems
    - helps you ensure the highest-quality code is in place (before testing begins)
- tools can check the code as you work on your program
  - You will get an in-depth analysis of where there might be potential problems in your code, based on the rules you've applied

# Limitations of Static Analysis

---

- limitations include
- no understanding of developer intent
- may detect a possible overflow in this calculation
  - it can not determine that function fundamentally does not do what is expected
- rules that are not statically enforceable
  - some coding rules depend on external documentation
  - some rules are open to subjective interpretation
- possible defects lead to false positives and false negatives
- in some situations, a tool can only report that there is a possible defect

# Some tools available

---

- there are a ton of tools available that perform static analysis
- the two tools that I have experience with are coverity and codesonar
- Coverity (by Synopsis)
  - reasonably fast and returns few false positives
  - latest versions supports an option to choose between three levels of aggressiveness, with the number of reports increasing (and the number of possible false positives going up) at the higher levels
- CodeSonar (by Grammatech)
  - performs a whole-program, interprocedural analysis on C
  - identifies programming bugs and security vulnerabilities
  - also provides rules checkers for the Power of Ten coding rules
  - can be slow on large code bases
- the U.S. Food and Drug Administration uses CodeSonar to detect defects in fielded medical devices
- NASA uses CodeSonar in its Study on Sudden Unintended Acceleration in the electronic throttle control systems of Toyota vehicles

# How CodeSonar works

---

- employs a unified dataflow and symbolic execution analysis that examines the computation of the complete application
- by not relying on pattern matching or similar approximations, CodeSonar's static analysis engine is extraordinarily deep, finding 3-5 times more defects on average than other static analysis tools
- like a compiler, CodeSonar does a build of your code using your existing build environment, but instead of creating object code, CodeSonar creates an abstract model of your entire program
  - from the derived model, CodeSonar's symbolic execution engine explores program paths
    - reasoning about program variables and how they relate
    - advanced theorem-proving technology prunes infeasible program paths from the exploration
- checkers perform static code analysis to find common defects, violations of policies, etc
  - operate by traversing or querying the model, looking for particular properties or patterns that indicate defects
- an astronomical number of combinations of circumstances must be modeled and explored, so CodeSonar employs a variety of strategies to ensure scalability
  - procedure summaries are refined and compacted during the analysis, and paths are explored in an order that minimizes paging

---

# Double Pointers (pointer to a pointer)

Jason Fedin

# Overview

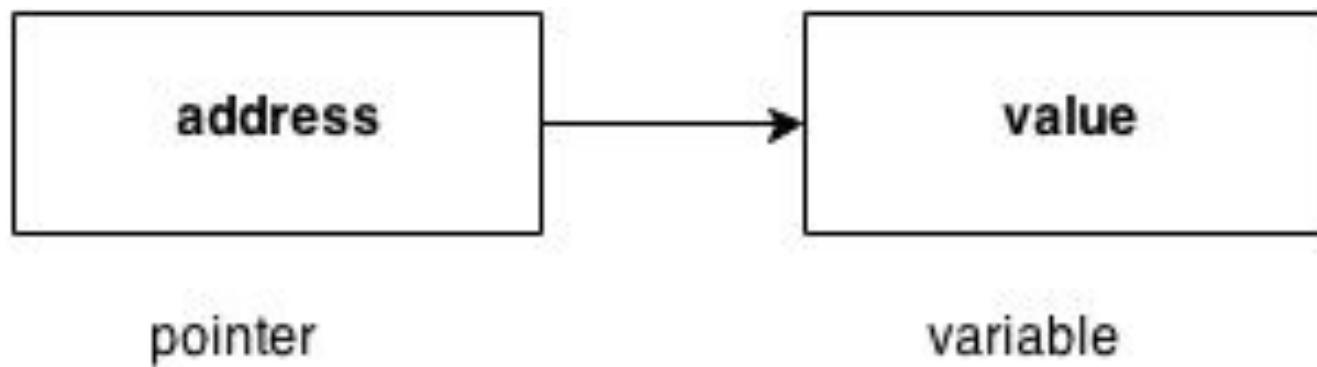
---

- we have a good understanding of the concept of pointers (indirection)
  - we can create pointers to int, pointers to a char, pointers to structures, really any data type
- in this lecture, we will discuss the concept of pointers to other pointers
  - a form of multiple indirection, or a chain of pointers
- we understand the distinction between the pointer itself and what it points to
  - this will help with understanding the concept of pointers to pointers
  - we now have to distinguish between the pointer, what it points to, and what the pointer that it points to points to
- when a pointer holds the address of another pointer, this is known as a pointer-to-pointer or double pointer
- in theory, we might also end up with pointers to pointers to pointers, or pointers to pointers to pointers to pointers
  - although triple and more pointers are hardly ever used and should not be

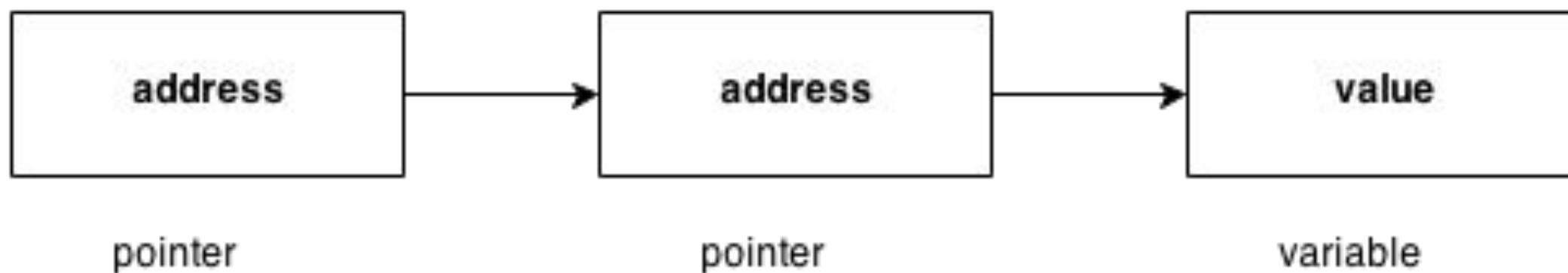
# Overview

---

- a pointer points to a location in memory and thus used to store the address of variables



- with a double pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value



# Syntax of a double pointer

---

- a variable that is a pointer to a pointer is declared by placing an additional asterisk in front of its name
  - two asterisks indicate that two levels of pointers are involved
- the following declaration declares a pointer to a pointer of type int

```
int **var;
```

- when a value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice

```
int x = **var;
```

# Conceptual example

---

```
// declaring a double pointer
```

```
int **ipp;
```

```
// declaring some ints
```

```
int i = 5, j = 6; k = 7;
```

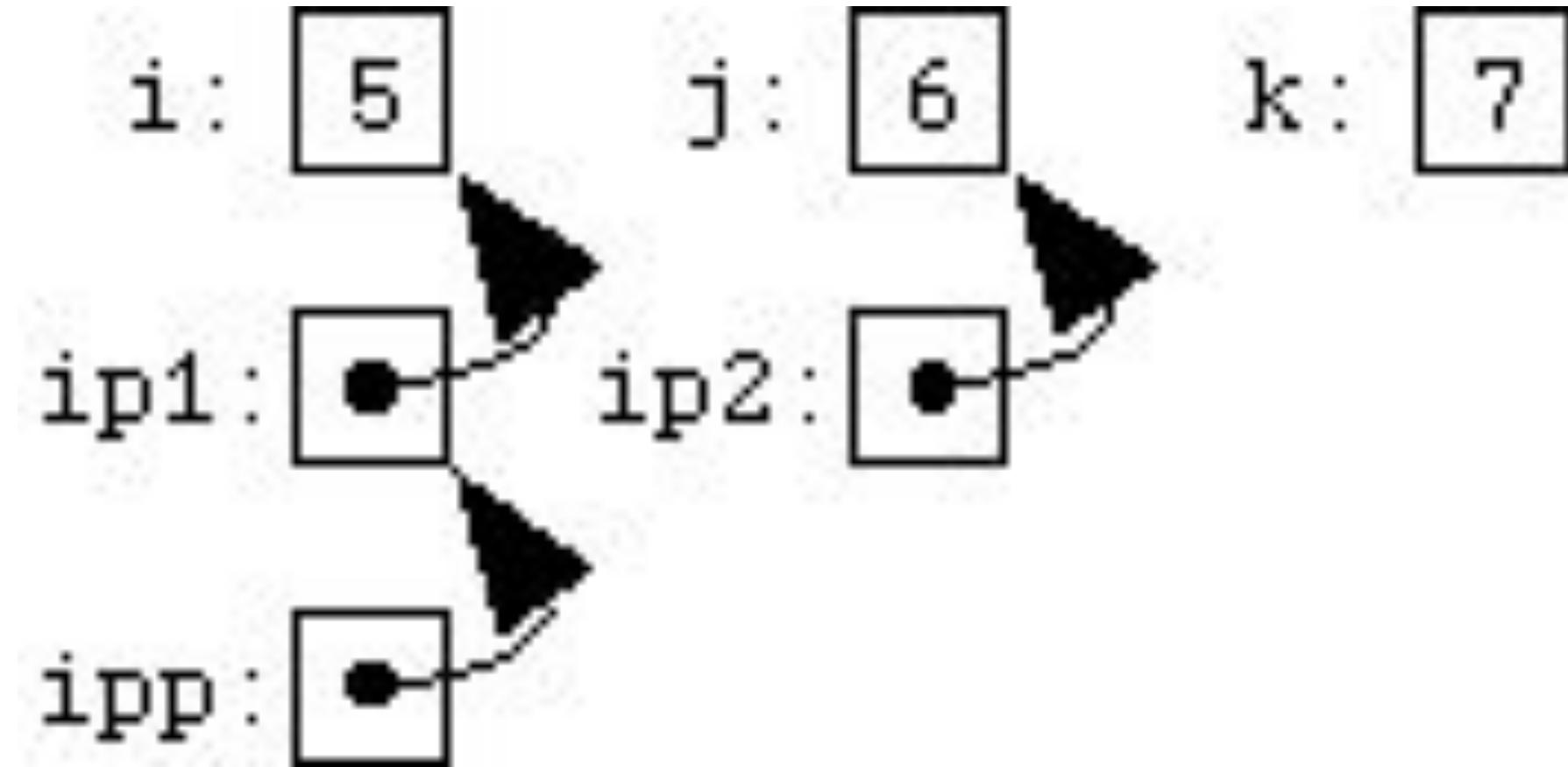
```
// initializing some pointers
```

```
int *ip1 = &i, *ip2 = &j;
```

```
// assigning our double pointer
```

```
ipp = &ip1;
```

- ipp points to ip1 which points to i
- \*ipp is ip1
- \*\*ipp is i (5)

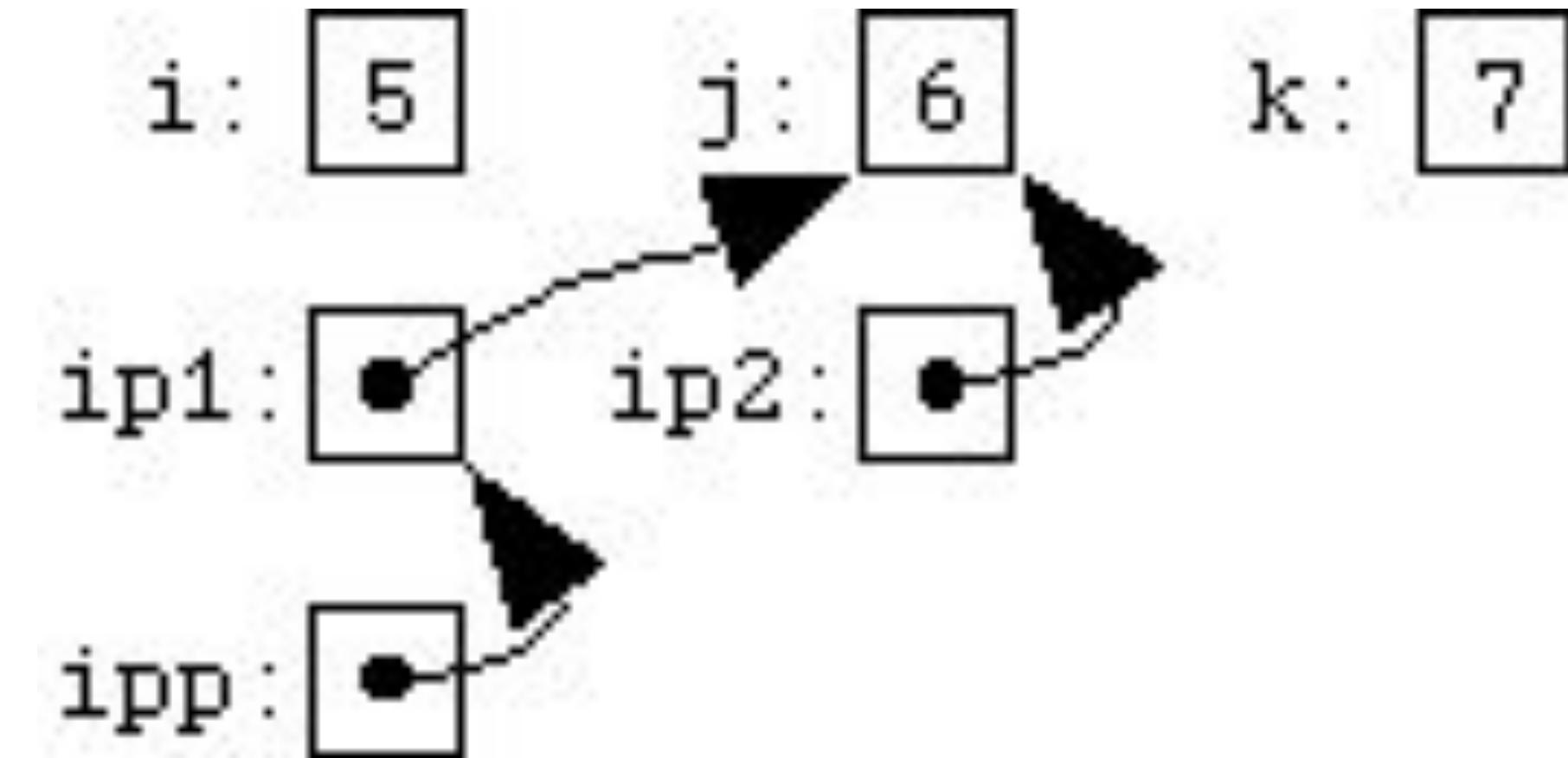


# Conceptual example

---

- if we now change ipp

`*ipp = ip2;`



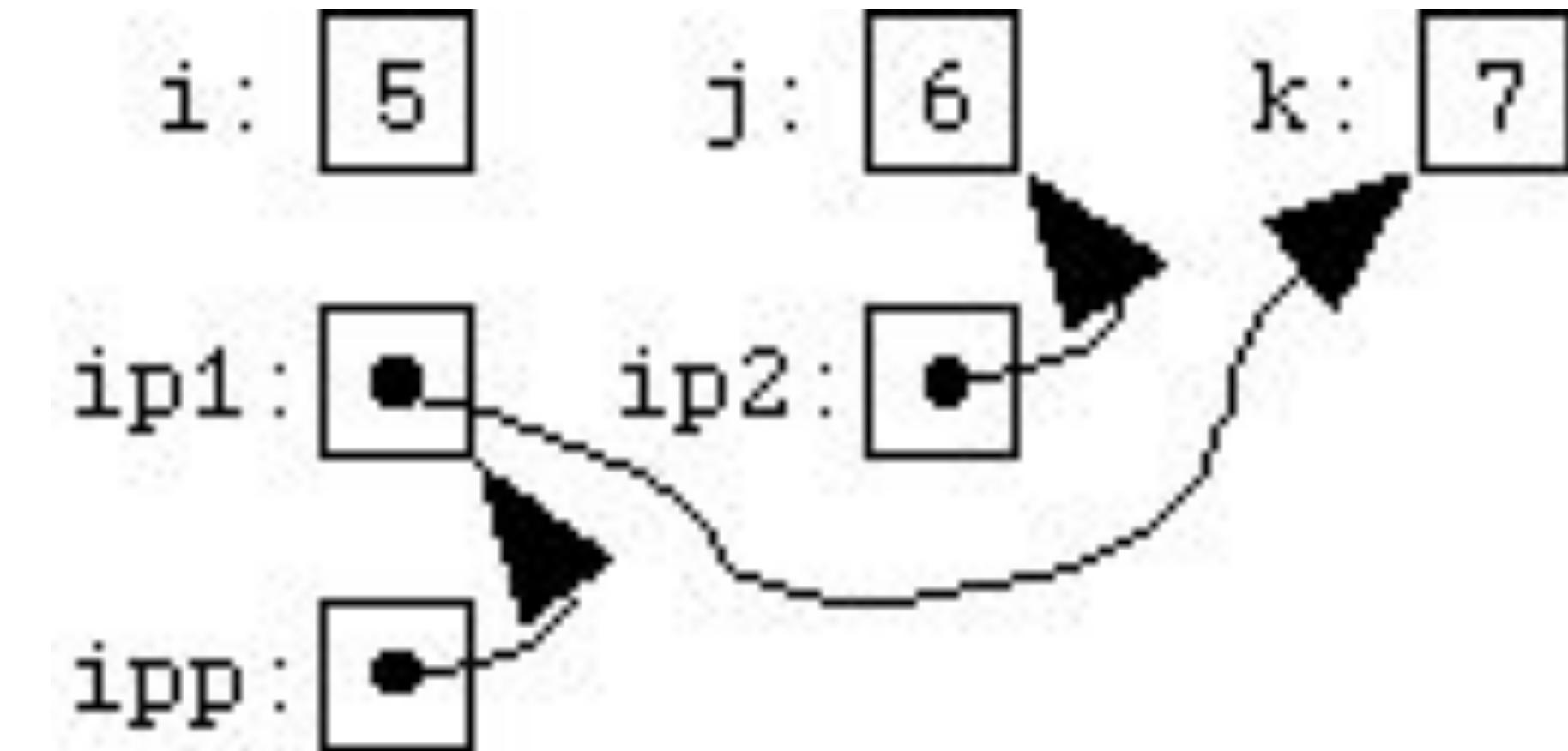
- we have changed the pointer pointed to by ipp (that is, ip1) to contain a copy of ip2
  - ip1 now points at j

# Conceptual example

---

- If we now change ipp

`*ipp = &k;`



<http://c-faq.com/~scs/cclass/int/sx8.html>

- we have now changed the pointer pointed to by ipp (that is, `ip1` again) to point to `k`

---

# Double Pointers (pointer to a pointer)

Jason Fedin

# Use cases

---

- the biggest reason to use a double pointer is when you need to change the value of the pointer passed to a function as the function argument
  - simulate pass by reference (remember, everything in C is pass by value/copy)
- If you pass a single pointer in as an argument
  - you will be modifying local copies of the pointer, not the original pointer in the calling scope
  - with a pointer to a pointer, you modify the original pointer
- use a double pointer as an argument to a function when you want to preserve the memory-allocation or assignment even outside of the function

# Other use cases

---

- there are other uses too, like the main() argument of every C program has a pointer to a pointer for argv
  - each element holds an array of chars that are the command line options
- you must be careful though when you use pointers of pointers to point to 2 dimensional arrays
  - better to use a pointer to a 2 dimensional array instead

```
void test() {
 double **a;
 int i1 = sizeof(a[0]); //i1 == 4 == sizeof(double*)

 double matrix[ROWS][COLUMNS];
 int i2 = sizeof(matrix[0]); //i2 == 240 == COLUMNS * sizeof(double)
}
```

- here is an example of a pointer to a 2 dimensional array done properly:

```
int (*myPointerTo2DimArray)[ROWS][COLUMNS]
```

---

# Function Pointers

Jason Fedin

# Overview

---

- functions contain addresses and thus, we can use pointers to point to functions
  - not as common as other pointer use cases
- pointers to functions can be
  - passed to functions
  - returned from functions
  - stored in arrays
  - assigned to other function pointers
- a function pointer can be used directly as the function name when calling the function
- they are less error prone than normal pointers cause you will never allocate or deallocate memory with them
  - you just need to understand what they are and to learn their syntax

# Common uses of function pointers

---

- a function pointer can be used as an argument to another function
  - telling the second function which function to use
- sorting an array involves comparing two elements to see which comes first
  - the qsort() function from the C library is designed to work with arrays of any kind as long as you tell it what function to use to compare elements
  - takes a pointer to a function as one of its arguments
  - uses that function to sort the type (whether it be integer, string, or structure)
- another common application for function pointers is to create what is known as dispatch tables
  - you can create tables that contain pointers to functions to be called
  - you might create a table for processing different commands that will be entered by a user
    - each entry in the table could contain both the command name and a pointer to a function to call to process that particular command
    - whenever the user enters a command, you can look up the command inside the table and invoke the corresponding function to handle it

# Common uses of function pointers

---

- menu-driven systems are also a common use of function pointers
- you can use them to replace switch/if-statements
- you can use them to realize your own late-binding
- you can use them to implement callbacks
- function pointers are useful when alternative functions may be used to perform similar tasks on data

# Syntax

---

- a pointer to a function stores the address for the start of the function code
  - however, the address by itself is not enough information to invoke it
  - we need to know the number and type of the arguments to be supplied and the type of return value to be expected
    - compiler can not deduce this information just from the address of the function
- this means that declaring a pointer to a function is going to be a little more complicated than declaring a pointer to a data type
  - a function pointer holds an address and must also define a prototype
- just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function

# Declaration

---

- function pointers can be declared, assigned values and then used to access the functions they point to
- the declaration for a pointer to a function looks a little strange and can be confusing
- when you declare a data pointer, you have to declare the type of data to which it points
- when declaring a function pointer, you have to declare the type of function pointed to
  - to specify the function type, you specify the function signature (the return type for the function and the parameter types for a function)

```
int (*pfunction) (int);
```

# Declaration

---

```
int (*pfunction) (int);
```

- declares a variable that is a pointer to a function
  - does not point to anything
  - just defines the pointer variable
- the name of the pointer is pfunction
- the parentheses are essential in the declaration because of the operators' precedence
  - the declaration without the parentheses
    - `int *pfunction(int);`
    - will declare a function pfunction that returns an integer pointer that is not our intention in this case

# Declaration

---

```
int (*pfunction) (int);
```

- the declaration is the de-referenced value of pfunction (\*pfunction)
- points to functions that have one parameter of type int and that return a value of type int to the calling function
  - you can only use this particular pointer to point to functions with these parameters and return value
- if you want a pointer to functions that accepts a float argument and returns float values
  - you need to declare another pointer with the required characteristics

# Assignment

---

```
int (*pfunction) (int);
```

- to set your function pointer pointing to an existing specific function, you simply assign the name of the function to it

```
int lookup(int);
pfunction = lookup; or pfunction = &lookup;
```

- stores a pointer to this function inside the function pointer variable pfunction
- writing a function name without a subsequent set of parentheses is treated in an analogous way to writing an array name without a subscript
  - compiler automatically produces a pointer to the specified function
  - an ampersand is permitted in front of the function name, but it's not required
- if the lookup() function has not been previously defined in the program, it is necessary to declare the function before the preceding assignment can be made

# Invocation

---

```
int (*pfunction) (int);
```

- you can call the function by applying the function call operator to the pointer
  - listing any arguments to the function inside the parentheses

```
int value = pfunction(5);
```

- calls the function pointed to by pfunction, storing the returned value inside the variable value
- you use the function pointer name just like a function name to call the function that it points to
  - no dereference operator is required

# qsort example

---

- the qsort standard library function is very useful function
  - designed to sort an array by a key value of any type into ascending order
  - as long as the elements of the array are of fixed type
- qsort is prototyped in (stdlib.h)

```
void qsort(void *base, size_t num_elements, size_t element_size, int (*compare)(void const *, void const *));
```

- the argument base points to the array to be sorted and num\_elements indicates how long the array is, with element\_size the size in bytes of each array element
- the final argument, compare is a pointer to a function
- qsort calls the compare function which is user defined to compare the data when sorting
  - qsort maintains its data type independence by giving the comparison responsibility to the user
- the compare function must return certain (integer) values according to the comparison result
  - less than zero, if first value is less than the second value
  - zero, if first value is equal to the second value
  - greater than zero, if first value is greater than the second value

# qsort example

---

- to sort the following structure by integer key

```
typedef struct {
 int key;
 struct other_data;
} record;
```

- we can write a compare function

```
int record_compare(void const *a, void const *b){
 return (((record *)a)->key - ((record *)b)->key);
}
```

- assuming that we have an array of array\_length records suitably filled with data we can call qsort like this

```
qsort(array, arraylength, sizeof(record), record_compare);
```

# Pointer to a function vs. function returning a pointer

---

```
/* function returning pointer to int */
int *func(int a, float b);
```

```
/* pointer to function returning int */
int (*func)(int a, float b);
```

- the difference between the above two declarations is only in the parentheses
  - be very careful about placing the parentheses in the right place

---

# void pointers

Jason Fedin

# Overview

---

- we have previously learned that if a pointer is of type pointer to int or (int \*) then it can hold the address of the variable of type int only
  - we cannot assign an address of a float variable to a pointer of type pointer to int
- the void type of pointer is a special type of pointer
  - void represents the absence of type (a generic pointer)
- void pointers are pointers that point to a value that has no type
- a void pointer can point to a variable of any data type
  - from an integer value or a float to a string of characters

# Syntax

---

- a void pointer declaration is similar to the normal pointer, but the difference is that instead of data types we use the void keyword

```
void * pointer_name;
```

- the size of a void pointer is similar to the size of the character pointer
  - has the same representation and alignment requirements as a pointer to a character type

# example

---

```
void *vp;
```

```
int a = 100, *ip;
float f = 12.2, *fp;
char ch = 'a';
```

- vp is a void pointer, so you can assign the address of any type of variable to it

```
vp = &a; // ok
vp = ip; // ok
vp = fp; // ok
```

```
ip = &f; // wrong since type of ip is pointer to int
fp = ip; // wrong since type of fp is pointer to float
```

# dereferencing a void pointer

---

- the data pointed to by a void pointer cannot be directly dereferenced
  - we have no type to dereference to
  - we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it

```
void *vp;
int a = 100;
```

```
vp = &a;
printf("%d", *vp); // wrong
```

- the above will not work
  - because a void pointer has no data type
  - before you dereference a void pointer it must be type casted to the appropriate pointer type

# dereferencing a void pointer

---

```
void *vp;
int a = 100;
vp = &a;
```

- void pointer vp is pointing to the address of integer variable a
  - vp is acting as a pointer to int or (int \*)
  - the proper typecast in this case is (int\*)

(int \*)vp

- the type of vp temporarily changes from void pointer to pointer to int or (int\*)
  - we already know how to dereference a pointer to int, just precede it with indirection operator (\*)

\*(int \*)vp

- type casting changes type of vp temporarily until the evaluation of the expression, everywhere else in the program vp is still a void pointer

---

# Overview

Jason Fedin

# Overview

---

- you have already seen the power of standard libraries in C
  - standard header files (stdio.h, stdlib.h)
  - math, String, and utility functions
- now it is time to use that power for your own code
  - you will learn how to create your own libraries and reuse the same code across several programs
  - you will learn how to share code at runtime with dynamic libraries (shared objects)
- you will be able to write code that you can scale and manage simply and efficiently

# Libraries

---

- a library is a collection (group) of header files and implementation files, exposed for use by other programs
  - interface expressed in a header file
  - implementation expressed in a .c file
- a library consists of one or more object files, which consist of object code
- C functions that can be shared by more than one application should be broken out of an application's source code and compiled and bundled into a library
- libraries may call functions in other libraries such as the Standard C or math libraries to do various tasks
- a programmer has the interface file (headers) to figure out how to use a library

# Advantages of Libraries

---

- libraries are useful because they allow for fast compilation times
  - you do have to compile all sources of your dependencies when compiling your application
- allows application vendors a way to simply release an API to interface with an application
- libraries also allow for modular development and separation of components
- software reuse is a huge benefit of libraries
- version management
  - libraries can cohabitate old and new versions on a single system
- component specialization
  - specialized developers can focus on their core competency on a single library

# Linking

---

- when a C program is compiled, the compiler generates object code (.o or .obj)
- after generating the object code, the compiler also invokes the linker
- linking is the processes of collecting and combining multiple object files in order to create a single executable file
- one of the main tasks for linker is to make the code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program
- a linker can accomplish this task in two ways
  - by copying the code of library function to your object code
  - by making some arrangements so that the complete code of library functions is not copied, but made available at run-time
- static linking is the result of the linker making copy of all used library functions to the executable file
- dynamic linking does not require the code to be copied, it is done by just placing the name of the library in the binary file
  - linking happens when the program is run, when both the binary file and the library are in memory

# Static Linking

---

- when you link your application to another library at compile time, the library code is part of your application
- the primary advantage of static linking is the speed
  - there will be no symbol (a program entity) resolution at runtime
  - every piece of the library is part of the binary image (executable)
- once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system
- static linking creates larger binary files that utilize more disk space and main memory
- once the library is linked and the program is created, you have no way of changing any of the library code without rebuilding the whole program
  - more time consuming to fix bugs

# Dynamic Linking

---

- dynamic linking defers much of the linking process until a program starts running
  - performs the linking process “on the fly” as programs are executed in the system
- libraries are loaded into memory by programs when they start
- during compilation of the library, the machine code is stored on your machine
  - when you recompile a program that uses this library, only the new code in the program is compiled
  - does not recompile the library into the executable file like in static linking
- the main reason for using dynamic linking of libraries is to free your software from the need to recompile with each new release of a library

# Dynamic Linking

---

- dynamic linking is the more modern approach, and has the advantage of a much smaller executable size
- dynamic linking trades off more efficient use of the disk and a quicker linking phase for a small runtime penalty
- dynamic linking helps overall performance in two ways
  - it saves on disk and virtual memory
    - libraries are only mapped in to the process when needed
    - all executables dynamically linked to a particular library share a single copy of the library at runtime
      - ensures that libraries mapped into memory are shared by all processes using them
      - provides better I/O and swap space utilization and is sparing of physical memory, improving overall system throughput

# Library Types

---

- there are two C library types which can be created
- static libraries
  - uses static linking (compile-time, becomes part of the application)
  - each process gets its own copy of the code and data
  - known as an archive
- shared object libraries
  - dynamically linked at run time
    - shared objects are not included into the executable component but are tied to the execution
  - code shared, data is specific to each process
- there is also the concept of a dynamically loaded/unloaded shared object library that is linked during execution using the dynamic linking loader system functions

# Library Conventions

---

- dynamic libraries are called lib something .so
  - have a filename of the form libname.so (version numbers may be appended to the name)
    - the library of thread routines is called libthread .so
- static libraries are called lib something .a
  - has a filename of the form libname.a
  - ".a" library is conceptually the same as the windows ".lib" libraries
- you can identify your libraries by looking at the header files you have used
  - a good hint is to study the #includes that your program uses
  - each header file that you include potentially represents a library against which you must link

# Summary

---

- static Libraries have the advantage of speed
  - all the code to execute the file is in one executable file, with little to virtually zero compatibility issues
- static libraries are larger in size because the file has to be recompiled every time when adding new code to the executable
- for shared objects, only one copy of the shared library is kept in memory
  - making it much faster to compile programs and significantly reducing the size of the executable program
- shared objects have a slower execution time compared to static libraries
- shared objects also may have compatibility issues if a library is changed without recompiling the library into memory

---

# Creating a Static Library (archive)

Jason Fedin

# Overview

---

- a static library is an archive
  - a bunch of object files wrapped up into a single file
  - similar to a .zip or a .tar file
  - a file that contains other files
- created and updated by the ar (for archive) utility
  - a program that takes files and stores them in a bigger file without regard to compression
- standard convention is to name static libraries
  - lib<something>.a
    - names begin with lib and end with a .a extension
    - libc.a file contains the Standard C library
    - "libm.a" contains mathematics functions
- once you have an archive, you can store it in a library directory
  - /usr/local/lib (system directory)
  - /my\_lib (your own director)

# linking to a static library (notes)

---

- the compiler expects to find the libraries in certain directories
  - the compiler looks in a few special places such as /usr/lib/ for libraries
- the compiler option -Lpathname is used to tell the linker a list of other directories in which to search for libraries
  - if you put your archive somewhere else, like /my\_lib?
- the name that follows the -l option needs to match part of the archive name
- if your archive is called libawesome.a, you can compile your program with the -lawesome switch

# Misc

---

- the order of the statically linked libraries on the compiler command line is significant
  - linker is fussy about where libraries are mentioned, and in what order, since symbols are resolved looking from left to right
- this makes a difference if the same symbol is defined differently in two different libraries
- another problem occurs if you mention the static libraries before your own code
  - won't be any undefined symbols yet, so nothing will be extracted
  - when your object file is processed by the linker, all its library references will be undefined

---

# Creating a Shared Object (dynamic library)

Jason Fedin

# Overview

---

- every program linked against this library shares the same one copy
  - contrast to static linking, in which everyone is (wastefully) given their own copy of the contents of the library
- a dynamically linked library (shared object) is created by the link editor
  - the name of the link editor is the command ld
- standard convention is to name dynamic libraries
  - lib<something>.so (shared object)
    - names begin with lib and end with a .so extension
    - similar to windows extension .dll and mac extension .dylib
- you can use the ldd command on linux to list all of the shared objects for a given binary/executable
  - ldd name-of-executable

# linking to a dynamic library (notes)

---

- the compiler expects to find the libraries in certain directories
  - the compiler looks in a few special places such as /usr/lib/ for libraries
  - threads library is in /usr/lib/libthread.so
- the compiler option -Lpathname is used to tell the linker a list of other directories in which to search for libraries
  - LD\_LIBRARY\_PATH and LD\_RUN\_PATH can also be used to provide this information
  - better to use the -Lpathname -Rpathname options at linktime instead
- the name that follows the -l option needs to match part of the library name
- if your archive is called libawesome.so, you can compile your program with the -lawesome switch

---

# Dynamic Loading

Jason Fedin

# Overview

---

- up until now, we have learned about the concepts of static linking and dynamic linking when creating and using a library
- we link against a library at compile time or at run-time
  - the linker at compile time resolves all external symbols or makes sure everything is in place when creating our executable file
    - either by copying the code into the executable or placing the name of the library in the executable so that it can use a shared object
  - creates a dependency
- loading of a library is of two types
  - static loading
  - dynamic loading
- do not confuse the terms static loading vs static linking and dynamic loading vs dynamic linking

# Overview

---

- whenever we run our executable, the loader will try to resolve against all the symbols at program start up
  - loading will store every required library into memory, along with our executable itself
- what if we didn't want to link libraries at compile time (via copy or sharing) and load symbols at program start-up time
  - instead, load them ourselves as needed during runtime
- instead of a predefined dependency on a library, we could make its presence optional and adjust our program's functionality accordingly
  - loading of a library on demand (dynamic loading), not at start up
  - the programmer determines when/if the library is loaded

# Overview

---

- dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory
  - can retrieve the addresses of functions and variables contained in the library
  - can execute those functions or access those variables
  - can unload the library from memory
- it is one of the 3 mechanisms by which a computer program can use some other software
  - the other two are static linking and dynamic linking (both load at program start up)
- unlike static linking and dynamic linking, dynamic loading allows a computer program to start up in the absence of these libraries
  - can discover available libraries, and to potentially gain additional functionality
- the main difference of dynamic linking to a shared object is that the libraries are not automatically loaded at program start-up

# Examples

---

- they are particularly useful for implementing plugins or modules
  - they permit waiting to load the plugin until it is needed
- the Pluggable Authentication Modules (PAM) system uses DL libraries to permit administrators to configure and reconfigure authentication
- (DL) libraries are also useful for implementing interpreters that wish to occasionally compile their code into machine code and use the compiled version for efficiency purposes
  - all without stopping
- the Apache Web Server's \*.dso "dynamic shared object" plugin files are libraries which are loaded at runtime with dynamic loading
- also used in implementing computer programs where multiple different libraries may supply the requisite functionality
  - where the user has the option to select which library or libraries to provide

# Dynamic Loading API (dlopen() )

---

- there is an API for opening a library, looking up symbols, handling errors, and closing the library
  - need to include the header file <dlfcn.h>
- the first function we will discuss is the dlopen() function
  - opens a library and prepares it for use
- if libraries depend on each other (e.g., X depends on Y), then you need to load the dependees first (load Y first, and then X)

```
void * dlopen(const char *filename, int flag);
```

- if filename begins with ``/'' (i.e., it's an absolute path), dlopen() will just try to use it
  - otherwise, it will search for the library in the LD\_LIBRARY\_PATH environment variable and other directories

# Dynamic Loading API (dlopen() )

---

```
void * dlopen(const char *filename, int flag);
```

- the value of flag must be either RTLD\_LAZY or RTLD\_NOW
  - RTLD\_LAZY means to resolve undefined symbols as code from the dynamic library is executed
  - RTLD\_NOW means to resolve all undefined symbols before dlopen() returns and fail if this cannot be done (used if debugging)
- the return value of dlopen() is a "handle" that should be used by the other DL library routines
  - will return NULL if the attempt to load does not succeed (should always check for this)
  - if the same library is loaded more than once with dlopen(), the same file handle is returned

# Dynamic Loading API (dlerror() and dlsym())

---

- errors can be reported by calling dlerror()
  - returns a string describing the error from the last call to dlopen(), dlsym(), or dlclose()\
- the main routine for using a DL library is the dlsym() function
  - looks up the value of a symbol in a given (opened) library
  - no point in loading a DL library if you cannot use it

```
void * dlsym(void *handle, char *symbol);
```

- the handle is the value returned from dlopen, and symbol is a NIL-terminated string
  - do not store the result of dlsym() into a void\* pointer
  - you will have to cast it each time you use it (and you'll give less information to other people trying to maintain the program)
- dlsym() will return a NULL result if the symbol was not found

# Dynamic Loading API (dlerror() and dlsym())

---

- convention is to call dlerror() first
  - to clear any error condition that may have existed
  - then to call dlsym() to request a symbol
  - then call dlerror() again to see if an error occurred

```
dlerror(); /* clear error code */
```

```
s = (actual_type) dlsym(handle, symbol_being_searched_for);
```

```
if ((err = dlerror()) != NULL) {
 /* handle error, the symbol wasn't found */
} else {
 /* symbol found, its value is in s */
}
```

# Dynamic Loading API (dlclose() )

---

- the converse of dlopen() is dlclose()
  - closes a DL library
- the dl library maintains link counts for dynamic file handles
  - a dynamic library is not actually deallocated until dlclose has been called on it
  - need to call as many times as dlopen has succeeded on it
- it is not a problem for the same program to load the same library multiple times

---

# Assert

Jason Fedin

# Overview

---

- the assert library is designed to help with debugging programs
  - defined in assert.h
- the assert() macro enables you to insert tests of arbitrary expressions in your program
  - a run-time check
- takes as its argument an integer expression
  - the argument is a relational or logical expression
- the program will be terminated with a diagnostic message if a specified expression is false during execution
  - error message is written to the standard error stream (stderr)
    - displays the test that failed, the name of the file containing the test, and a line number
  - abort() function is invoked which terminates the program
- assertions are often used for critical conditions in a program
  - if certain conditions are not met, disaster will occur
  - you would want to be sure that the program does not continue when this error occurs
- assertions are not meant as a substitute for error handling during normal runtime conditions
  - use should be limited to finding logic errors

# Example

---

```
assert(some_variable < some_count);
```

- if `some_variable` is not less than `some_count`
  - the expression will be false, so the program will assert
- typical output from the assertion looks like this

*Assertion failed: file c:\jfedin\my\_program.c, func main line 44,  
some\_variable<some\_count abort -- terminating*

- you can see that the function name and the line number of the code are identified as well as the condition that was not met

# Switching Off Assertions

---

- assertions can be switched off by defining the symbol NDEBUG before the #include directive for assert.h

```
#define NDEBUG // Switch off runtime assertions
#include <assert.h>
```

- this code snippet will cause all assert() macros in your code to be ignored
- with some nonstandard systems, assertions are disabled by default, in which case you can enable them by undefining NDEBUG

```
#undef NDEBUG // Switch on assertions
#include <assert.h>
```

- by including the directive to undefine NDEBUG, you ensure that assertions are enabled for your source file
  - #undef must appear before the #include directive for assert.h to be effective

# Compile-Time Assertions

---

- C11 added a feature that does a compile-time check
  - can cause a program not to compile
- the assert.h header makes static\_assert an alias for the C keyword \_Static\_assert
  - more compatible with C++, which uses static\_assert as its keyword for this feature
- static\_assert is treated as a declaration statement
  - unlike most kinds of C statements, it can appear either in a function or external to a function
- the static\_assert() macro enables you to output an error message during compilation
  - message includes a string literal that you specify
  - whether or not the output is produced depends on the value of an expression that is a compile time constant
- takes two arguments
  - first is a constant integer expression
  - second is a string

`static_assert(constant_expression, string_literal);`

- when the constant expression evaluates to zero, compilation stops and the error message is output

# Compile-Time Assertions

---

- the `static_assert()` enables you to build checks into your code about your implementation
- suppose your code assumes that type `char` is an unsigned type
  - you could include this static assertion in the source file

```
static_assert(CHAR_MIN == 0, "Type char is a signed type. Code won't work.");
```

- `CHAR_MIN` is defined in `limits.h` and is the minimum value for type `char`
  - when `char` is an unsigned type, `CHAR_MIN` will be zero
  - when it is a signed type it will be negative.
- the above will cause compilation to be halted and an error message that includes your string to be produced when type `char` is signed

---

# stdlib.h

Jason Fedin

# Overview

---

- there are several dozens of functions which facilitate every C programmer in writing programs for different requirements
- programming in C requires any C program to use these various different functions
  - we have discussed many of these functions
    - strcpy, printf, scanf, etc.
- all these functions are put together in the standard library
  - functions performing somewhat similar tasks are put together in same header file
    - functions performing input and output are put together in 'stdio.h' header
    - functions performing string manipulations are defined in 'string.h' header
- these standard library functions help make your code more portable and more efficient

# Overview

---

- in this lecture and the next, we will focus on the general utilities library (stdlib.h)
  - random-number generators
  - program termination functions
  - searching and sorting functions
  - conversion functions
  - memory-management functions
- More specifically
  - exit
  - atexit
  - abort
  - qsort

# The exit() function

---

- we understand that program execution is automatically terminated whenever the last statement in main() is executed
  - or when executing a return from main()
- at times, you might want to force the termination of a program earlier than the above
  - when an error condition is detected by a program
    - input error
    - a file to be processed by the program cannot be opened
- the exit function forces a program to terminate as if it executed normally
  - no matter from what point you are executing
  - invoked automatically upon return from main

```
int exit(int status);
```

# The exit function

---

```
int exit(int status);
```

- the above statement has the effect of terminating (exiting from) the current program
- any open files are automatically closed by the system
  - flushes all output streams
  - closes all open streams
  - closes temporary files created by calls to the standard I/O function tmpfile()
- the integer value status is called the exit status, and has the same meaning as the value returned from main()
  - EXIT\_FAILURE (symbolic constant) represents an integer value that you can use to indicate the program has failed (non-zero)
  - EXIT\_SUCCESS (symbolic constant) represents an integer value that you can use to indicate it has succeeded (0)
- when a program terminates simply by executing the last statement in main, its exit status is undefined
  - you should exit or return from main() with a defined exit status

# the atexit() function

---

- the atexit() function allows you to specify particular functions to be called when exit() executes
  - must register the functions to be called on exit
- takes as an argument a pointer to a function (the function name)

```
int atexit(void (*function)(void));
```

- to use the atexit() function
  - pass it the address of the function you want called on exit
  - registers this function in a list of functions to be executed when exit() is called
    - up to 32 functions may be registered for execution at program termination
- the functions registered by atexit() should be type void functions taking no arguments
  - cannot have arguments and cannot return a value
- typically, the functions invoked, would perform housekeeping tasks
  - updating a program-monitoring file
  - resetting environmental variables

# the abort() function

---

- we just mentioned that the atexit and exit functions relate to normal termination of a program
- the abort function causes abnormal program termination
  - raises the SIGABRT signal
  - returns an implementation defined code indicating unsuccessful termination
- the abort function follows the philosophy of “fail hard and fail often”
- this function causes a noticeable failure because it will dump core and generate a core file

# the abort() function

---

- the following may occur
  - file buffers are not flushed
  - streams are not closed
  - temporary files are not deleted
- functions passed to atexit() are not called
- **note:** will not cause the program to terminate if SIGABRT is being caught by a signal handler passed to signal and the handler does not return

```
void abort(void);
```

# The qsort() function

---

- the “quick sort” method is one of the most effective sorting algorithms
  - particularly for larger arrays
  - partitions arrays into ever smaller sizes until the element level is reached
- first, the array is divided into two parts, with every value in one partition being less than every value in the other partition
  - process continues until the array is fully sorted
- the name for the C implementation of the quick sort algorithm is qsort()
  - sorts a data array pointed to by a function pointer (comparison function) passed into the qsort function
- qsort() and the comparison function use void pointers for generality
  - you have to tell qsort() explicitly how large each element of the array is
  - you have to convert its pointer arguments to pointers of the proper type for your application

# Syntax

---

```
void qsort (void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

- the first argument is a pointer to the beginning of the array to be sorted
  - can refer to any kind of array because of use of void pointer
- the second argument is the number of items to be sorted
  - converts this value to type size\_t
- the third argument is the size of each element
  - because qsort() converts its first argument to a void pointer, qsort() loses track of how big each array element is
    - to compensate, you must tell qsort() explicitly the size of the data object (this argument)

# Syntax

---

- the final argument is the address of the function to be used for comparing elements
- the signature of the comparison function is

```
int (*compar)(const void *, const void *)
```

- a pointer to a function that returns an int and that takes two arguments
  - each of which is a pointer to type const void
  - these two pointers point to the items being compared
- the comparison function should return the following
  - a positive integer if the first item should follow the second value
  - a zero if the two items are the same
  - a negative integer if the second item should follow the first

---

# stdlib.h (part II)

Jason Fedin

# Overview

---

- in this lecture we will focus on the general utilities library (stdlib.h) and the string library (string.h)
  - random-number generators
  - conversion functions
  - memory and string functions
- more specifically
  - atoi, atof, atol, strtod, strtodf, strtol, itoa
  - rand and srand
  - system
  - getenv
- and from the string library
  - memcpy and memmove (string.h)
  - strdup and strndup (string.h)

# String to number conversion

---

- the following functions convert character strings to numbers
- all routines skip leading whitespace characters in the string and stop their scan upon encountering a character that is invalid for the type of value being converted

```
int atoi(char const *s);
```

- converts the string pointed to by s into an int, returning the result

```
char a[10] = "100";
int value = atoi(a);
printf("Value = %d\n", value);
```

# random numbers

---

- C provides the rand() and srand() functions in order to create random numbers
- random numbers can be used for many types of applications from security to use in games
  - can be used to generate UUID, Certificates, Passwords etc.
- in order to generate random numbers, you generally need a hardware-based source which generates random signals
  - these signals can then be used to generate a random number
- creating randomness is a very hard job
  - usually a seed is provided for every random function execution
- seed values are used to make a random start from the application point of view
  - we generally use functions like time or network traffic as a seed value because they are changing regularly

# rand()

---

```
int rand (void)
```

- used to create random numbers without any parameters
- returns a random number in the range [0, RAND\_MAX]
  - RAND\_MAX is defined in <stdlib.h> and has a minimum value of 32767

# srand

---

void srand (unsigned seed)

- sets the starting point for producing a series of pseudo-random integers
  - if srand() is not called, the rand() seed is set as if srand(1) were called at program start
  - any other value for seed sets the generator to a different starting point
- the pseudo-random number generator should only be seeded once
  - before any calls to rand(), and the start of the program
  - it should not be repeatedly seeded
  - it should not be reseeded every time you wish to generate a new batch of pseudo-random numbers
- standard practice is to use the result of a call to srand(time(0)) as the seed
  - time() returns a time\_t value which varies every time and hence the pseudo-random number varies for every program call

---

# stdlib.h (part III)

Jason Fedin

# Overview

---

- in this lecture we will focus on the general utilities library (stdlib.h) and the string library (string.h)
  - misc. functions
  - memory and string functions
- more specifically
  - system
  - getenv
- and from the string library
  - memcpy and memmove (string.h)
  - strdup and strndup (string.h)

# The system() function

---

- the system function passes the command name or program name specified as an argument to the host environment
  - to be executed by the command processor and returns after the command has been completed
  - allows you to execute programs on the console/shell

```
int system(const char *command)
```

- gives the command contained in the character array pointed to by command to the system for execution
  - the value returned is -1 on error, and the return status of the command otherwise

```
system ("mkdir /usr/tmp/data");
```

- causes the system to create a directory called /usr/tmp/data (assuming you have the proper permissions to do so)

# The getenv() function

---

- an 'Environment' is an implementation-defined list of NAME/VALUE pairs maintained by the operating system
- the operating system defines specific meanings to "Environment Variables"
- not all implementations necessarily define same set of Environment Variables
  - they must define same meanings to whatever Environment Variables they have
- the function getenv searches for the environment string that is passed in
  - returns a null-terminated string with the value of the requested environment variable, or NULL if that environment variable does not exist

```
char *getenv(char const *NAME);
```

# memcpy() and memmove()

---

- you cannot assign one array to another, so often programmers use loops to copy one array to another, element by element
  - one exception is using strcpy() and strncpy() functions for character arrays
- the memcpy() and memmove() functions offer you almost the same convenience for other kinds of arrays

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

- both of these functions copy n bytes from the location pointed to by s2 to the location pointed to by s1
  - both return the value of s1

# memcpy() and memmove()

---

- the difference between the two (as indicated by the keyword restrict)
  - memcpy() is free to assume that there is no overlap between the two memory ranges
- the memmove() function does not make that assumption
  - copying takes place as if all the bytes are first copied to a temporary buffer before being copied to the final destination
- if you use memcpy() when there are overlapping ranges then the behavior is undefined
  - it might work or it might not
- these functions are designed to work with any data type
  - so, the two pointer arguments are type pointer-to-void
- these functions have no way of knowing what type of data is being copied
  - so, they use the third argument to indicate the number of bytes to be copied
- remember, for an array, the number of bytes is not, in general, the number of elements
  - if you are copying an array of 10 double values
  - you would use  $10 * \text{sizeof(double)}$ , not 10, as the third argument

# strup and strndup

---

- these function are used to duplicate a given string
  - a non standard function which may not available in standard library in C
- returns a pointer to a null-terminated byte string, which is a duplicate of the string pointed to by s
- the memory obtained is done dynamically using malloc and hence it can be freed using free()

```
char *strup(const char *string);
char *strndup(const char *s, size_t n);
```

- strndup is similar to strup(), but copies at most n bytes
  - If s is longer than n, then only n bytes are copied, and a NULL ("") is added at the end

# size\_t and NULL

## size\_t

- the unsigned integral type and is the result of the sizeof keyword

## NULL

- macro is the value of a null pointer constant

---

# Date and Time functions

Jason Fedin

# Overview

---

- the standard library gives a nice collection of functions which allow us to deal with date and time related computations
  - the time.h header declares functions that produce the time and date when you call them
- these functions provide output in various forms from the hardware timer in your PC
  - you can use these functions to obtain the current time and date
  - you can use these functions to calculate the time elapsed between two events
  - you can use these functions to measure how long the processor has been occupied performing a calculation
- examples include the 'clock()' function which returns amount of processor time used by the program
- the 'time()' function returns the number of seconds from an arbitrary epoch
- the 'ctime()' and 'asctime()' functions give current date and day of time as function value
- the 'gmtime()' converts time value into Coordinated Universal Time (UTC)

# prototypes

---

```
clock_t clock(void);
time_t time(time_t *returned value);
char *ctime(time_t const *time_value);
double difftime(time_t tm1, time_t tm2);
char *asctime(struct tm const *tm_ptr);
struct tm *localtime(time_t const *time_value);
struct tm *gmtime(time_t const *time_value);
```

- 'localtime()' returns structure of type struct tm
  - can be passed as an argument to asctime() function
- 'asctime()' and 'ctime()' functions return current date and time of day
  - the 'ctime()' function may call 'asctime()' to perform its work

# Getting Time Values

---

- lets talk about the clock function
  - the simplest function returning a time value

```
clock_t clock(void);
```

- returns the processor time (not the elapsed time) used by the program since some implementation-defined reference point
  - often since execution began
  - return value is of type `clock_t`, an integer type that is defined in `time.h`
- you typically call the `clock()` function at the start and end of some process in a program
  - the difference is a measure of the processor time consumed by the process
- the processor time is the total time the processor has been executing on behalf of the process that called the `clock()` function
  - the value that is returned by the `clock()` function is measured in clock ticks
  - to convert this value to seconds
    - you divide it by the value that is produced by the macro `CLOCKS_PER_SEC` (`time.h`)
      - `CLOCKS_PER_SEC` is the number of clock ticks in 1 second
    - returns -1 if an error occurs

# Getting Time Values

---

- to determine the processor time used in executing a process
  - you need to record the time when the process starts executing and subtract this from the time returned when the process finishes

```
clock_t start = 0, end = 0;
double cpu_time = 0.0;
start = clock();
// Execute the process for which you want the processor time...
end = clock();
cpu_time = (double)(end-start)/CLOCKS_PER_SEC; // Processor time in seconds
```

- stores the total processor time used by the process in cpu\_time
  - the cast to type double is necessary in the last statement to get the correct result

# Getting Time Values

---

- the time() function returns the calendar time as a value of type time\_t
  - the calendar time is the current time usually measured in seconds since a fixed time on a particular date
  - the fixed time and date is often 00:00:00 GMT on January 1, 1970
  - typical of how time values are defined

```
time_t time(time_t *timer);
```

- if the argument is not NULL
  - the current calendar time is also stored in timer
- type time\_t is defined in the header file and is often equivalent to type long
- to calculate the elapsed time in seconds between two successive time\_t values returned by time()
  - you use the function difftime()

```
double difftime(time_t T2, time_t T1);
```

- will return the value of T2 - T1 expressed in seconds as a value of type double
  - the time elapsed between the two time() function calls that produce the time\_t values, T1 and T2

# Getting the Date

---

- you can get today's date as a string using the ctime function

```
char *ctime(const time_t *timer);
```

- accepts a pointer to a time\_t variable as an argument that contains a calendar time value returned by the time() function
- returns a pointer to a 26-character string containing the day, the date, the time, and the year, which is terminated by a newline and '\0'
- a typical string returned might be the following  
"Mon Aug 25 10:45:56 2003\n\0"
- the ctime() function has no knowledge of the length of the string you have allocated to store the result
  - makes this an unsafe operation

- you might use the ctime() function like this

```
char time_str[30] = {'\0'};
time_t calendar = time(NULL);
printf("%s", ctime(&calendar));
```

# Getting the Date

---

- you can also get at the various components of the time and date from a calendar time value by using the `localtime()` function

```
struct tm *localtime(const time_t *timer);
```

- accepts a pointer to a `time_t` value
- returns a pointer to a structure of type `tm` (`time.h`)
  - the same structure each time you call it
  - structure members are overwritten on each call
- returns `NULL` if timer cannot be converted
- the time that `localtime()` produces is local to where you are

# struct tm

---

- all the members are of type int
- tm\_sec
  - seconds (0 to 60) after the minute on 24-hour clock
  - value goes up to 60 for positive leap-second support
- tm\_min
  - minutes after the hour on 24-hour clock (0 to 59)
- tm\_hour
  - the hour on 24-hour clock (0 to 23)
- tm\_mday
  - day of the month (1 to 31)

# struct tm

---

- tm\_mon
  - month (0 to 11)
- tm\_year
  - year (current year minus 1900)
- tm\_wday
  - weekday (Sunday is 0; Saturday is 6)
- tm\_yday
  - day of year (0 to 365)
- tm\_isdst
  - daylight saving flag
    - positive for daylight saving time
    - 0 for not daylight saving time
    - negative for not known

# Getting the Date

---

- the `asctime()` generates a string representation of a `tm` structure

```
char *asctime(const struct tm *time_data);
```

- the `asctime()` returns a `char` pointer
  - must be an array with at least 26 elements
  - works for `tm` structures where the year is from 1000 to 9999

```
time_t time_ptr;
time(&time_ptr);
printf("Current date and time = %s", asctime(localtime(&time_ptr)));
```

# Getting the Day for a Date

---

- you can use the mktime() function to determine the day of the week for a given date

```
time_t mktime(struct tm *ptime);
```

- you pass the address of a tm structure object to the function
  - the tm\_mon, tm\_mday, and tm\_year members set to values corresponding to the date you are interested in
  - the values of the tm\_wday and tm\_yday members of the structure will be ignored
  - if the operation is successful, the values will be replaced with the values that are correct for the date you have supplied
- returns the calendar time as a value of type time\_t if the operation is successful
- returns -1 if the date cannot be represented as a time\_t value
  - causing the operation to fail

# gmtime

---

- there is one more function which converts a 'time\_t' value into 'struct tm'

```
struct tm *gmtime(time_t const *time_value);
```

- converts time value into Coordinated Universal Time (UTC)
- UTC was formerly called Greenwich Mean Time and hence the name gmtime

---

# Abstract Data Types

Jason Fedin

# Overview

---

- in programming, you try to match the data type to the needs of a programming problem
  - you would use the int type to represent the number of pants you own
  - you would use the float or double type to represent your average cost per pair of pants
- in other scenarios, the data could be a list of items
  - no basic C type matches data for a list of items
- we would need to define a structure to represent individual items
  - we would then create a couple methods for tying together a series of structures to form a list
  - we use C's capabilities to design a new data type that matched our needs
- a type specifies two kinds of information
  - a set of properties
    - the int type's property is that it represents an integer value and shares the properties of integers
  - a set of operations
    - you can perform arithmetic operations on ints (add, subtract, multiply, etc)
- when you declare a variable to be an int, you are saying that these and only these operations can affect it

# Abstract Data Type (ADT)

---

- an Abstract Data type (ADT) is a type whose behavior is defined by a set of value and a set of operations
  - sound like a structure in C
- the definition of ADT only mentions what operations are to be performed
  - not how these operations will be implemented
  - does not specify how data will be organized in memory nor what algorithms will be used for implementing the operations
- it is called “abstract” because it gives an implementation-independent view
  - the process of providing only the essentials and hiding the details is known as abstraction
  - a contract for a type of data structure
- examples of abstract data types include
  - lists, stacks, trees, etc.

# creating a new Abstract Data type

---

- suppose you want to define a new data type
  - you have to provide a way to store the data
    - usually by designing a structure
  - you have to provide ways of manipulating the data
    - adding, deleting, etc.
- to define new data types, you want to follow a three-step process (from more abstract to more concrete)
- provide an abstract description of the type's properties and of the operations you can perform on the type
  - not tied to any particular implementation
  - referred to as an abstract data type (ADT)
- develop a programming interface that implements the ADT
  - indicate how to store the data
  - describe a set of functions that perform the desired operations
- write code to implement the interface

# Abstraction example

---

- for a list abstract data type, the data is a number of items
- a list should be able to hold a sequence of items in some kind of order
- the list type should support operations such as adding an item to the list
- here are some useful operations
  - initializing a list to empty
  - adding an item to the end of a list
  - determining whether the list is empty
  - determining whether the list is full
  - determining how many items are in the list
- the above is an abstract definition of a list data type
  - a data object capable of holding a sequence of items and to which you can apply any of the above operations
  - does not state what kind of items can be stored in the list
  - does not specify how to store the items or what algorithms to use

# ADT's we will study

---

- in the next few lectures, we will be studying the following abstract data types
  - Linked Lists
  - Stacks
  - Queues
  - Binary Trees
- to give you a sneak peak, lets quickly look at some of the operations for each of these types of ADT's
  - operation names will be similar (add, insert)

# Linked List ADT

---

- insert – adds an element at the beginning of the list
- delete – deletes an element at the beginning of the list
- deleteAt – remove the element using a given key
- size – return the number of elements in the list
- isEmpty – Return true if the list is empty, otherwise return false
- search – searches an element using a key
- display – displays the complete list

# Stack ADT

---

- push - insert an element at one end of the stack called top
- pop - remove and return the element at the top of the stack, if it is not empty
- peek - return the element at the top of the stack without removing it, if the stack is not empty
- size - return the number of elements in the stack
- isEmpty - return true if the stack is empty, otherwise return false

# Queue ADT

---

- enqueue – insert an element at the end of the queue
- dequeue – remove and return the first element of the queue, if the queue is not empty
- peek – return the element of the queue without removing it, if the queue is not empty
- size – return the number of elements in the queue
- isEmpty – return true if the queue is empty, otherwise return false

# Binary Tree ADT

---

- insert() -- insert a new node into the tree in the correct place
- delete - delete a node in the tree
- search - search for a node in the tree
- size() - count the number of nodes in the tree.
- preorder traversal
- postorder traversal
- inorder traversal

---

# Linked Lists

Jason Fedin

# Overview

---

- a linked list is an example of an abstract data structure
  - it can be used to store a lot of different kinds of data
  - it is the second most used data structure after the array
  - there is no linked list data type in C, so you have to create your own
- it is a linear data structure
- it is a sequence of data structures which are connected together via links/nodes
  - each link contains data items (elements)
- there are different types of linked lists
  - single linked list
    - can only be parsed one-way (forward)
  - doubly linked list
    - previous and next pointers, can be traversed forward and backward

# Overview

---

- linked lists are dynamic
  - the length of a list can increase or decrease as necessary
- a linked list can be used when the number of data elements to be represented in the data structure is unpredictable
- a node/link can contain data of any type including other struct objects
- linked lists can be maintained in sorted order by inserting each new element at the proper point in the list

# Linked lists and pointers

---

- linked lists heavily use pointers in its implementation
  - understanding pointers is crucial to understanding how linked lists work
  - you must also be familiar with dynamic memory allocation and structures
- linked lists work like an array that can grow and shrink as needed, from any point in the array
- linked lists are accessed via a pointer to the first node of the list
- the link pointer in the last node of a list is set to NULL to mark the end of the list

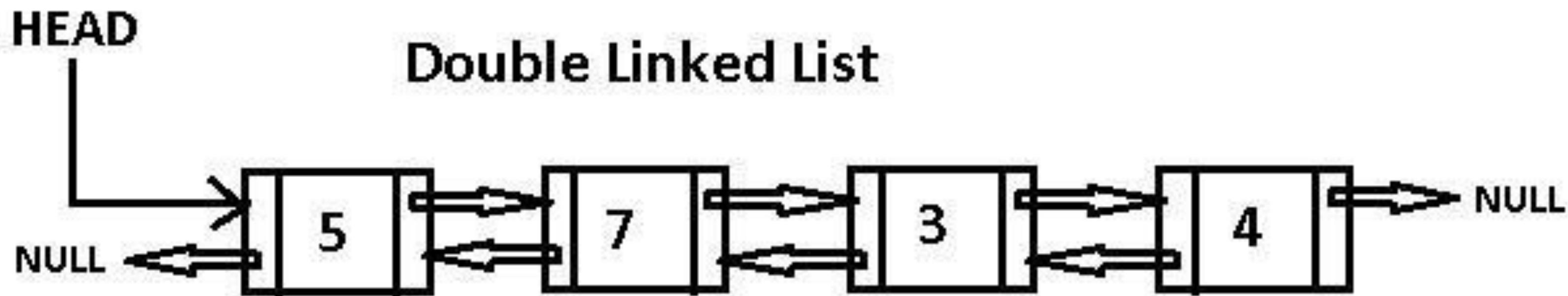
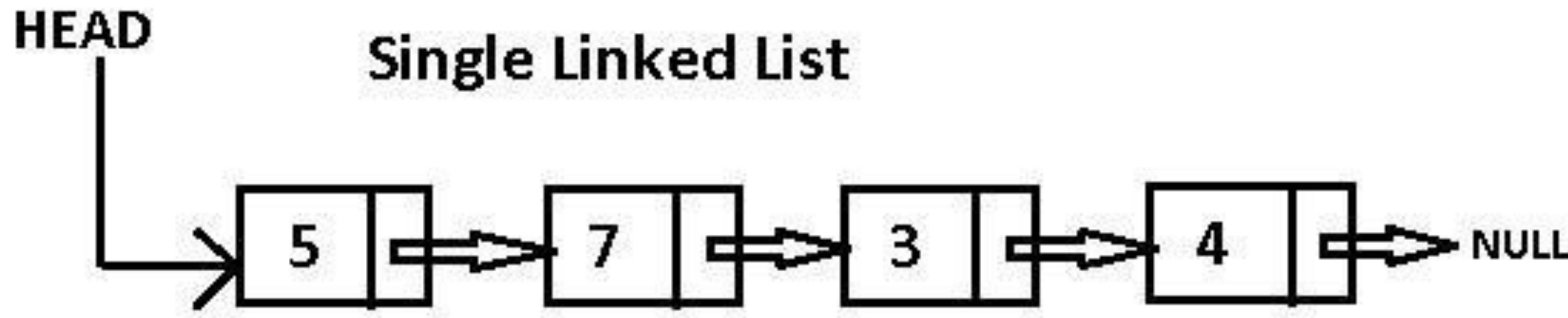
# Links/Nodes

---

- each node contains the following
  - a piece of data
  - the first node/link is often referred to as the head node/link
  - each node/link also contains a connection to another node/link
    - this is referred to as the next node/link
  - the last node/link points to NULL
- in order to traverse the list, you follow the pointer from each node/link to the next
- prepending to a list is very fast
- inserting into a sorted list is very fast

# Illustration

---



<https://medium.com/journey-of-one-thousand-apps/data-structures-in-the-real-world-508f5968545a>

---

# linked lists vs. arrays

---

- arrays are a fixed size
  - must know the upper limit on the number of elements in advance
- an array can be declared to contain more elements than the number of data items expected
  - this can waste memory
- linked lists can provide better memory utilization than arrays
  - using dynamic memory allocation that grow and shrink at execution time
    - however, extra memory space for a pointer is required with each element of the list (more overhead)
    - increases the risk of memory leaks and segment faults
- Insertion and deletion in a sorted array can be time consuming
  - all the elements following the inserted or deleted element must be shifted appropriately
- the elements of an array are stored contiguously in memory
  - allows immediate access to any array element
- linked list elements are not stored at a contiguous location
  - elements are linked using pointers
  - have to access elements sequentially starting from the first node/link

---

# Linked Lists (Implementation)

Jason Fedin

# Overview

---

- in this lecture, we are going to implement a linked list data structure
- we are going to create an application that allows us to perform operations on a linked list of characters
- this application will allow you insert a character at the beginning of a list, at the end, or in alphabetical order or to delete a character from the beginning of the list or a specific one
- the primary functions that we will implement are
  - insertAtBeginning
  - insertAtEnd
  - insert
  - delete
  - deleteAtBeginning
  - isEmpty
  - printList
- we will utilize structures and pointers in our implementation

---

# Stacks

Jason Fedin

# Overview

---

- a stack is a constrained version of a linked list
- all insertions and deletions are only made at the top of the stack
- the last item to be put in to the stack is always the first item to be removed
  - referred to as a last-in, first-out (LIFO) data structure
- a stack is referenced via a pointer to the top element of the stack
  - the link member in the last node of the stack is set to NULL to indicate the bottom of the stack
  - not setting the link in the bottom node of a stack to NULL can lead to runtime errors
- stacks and linked lists are represented identically
  - difference is that insertions and deletions may occur anywhere in a linked list, but only at the top of a stack

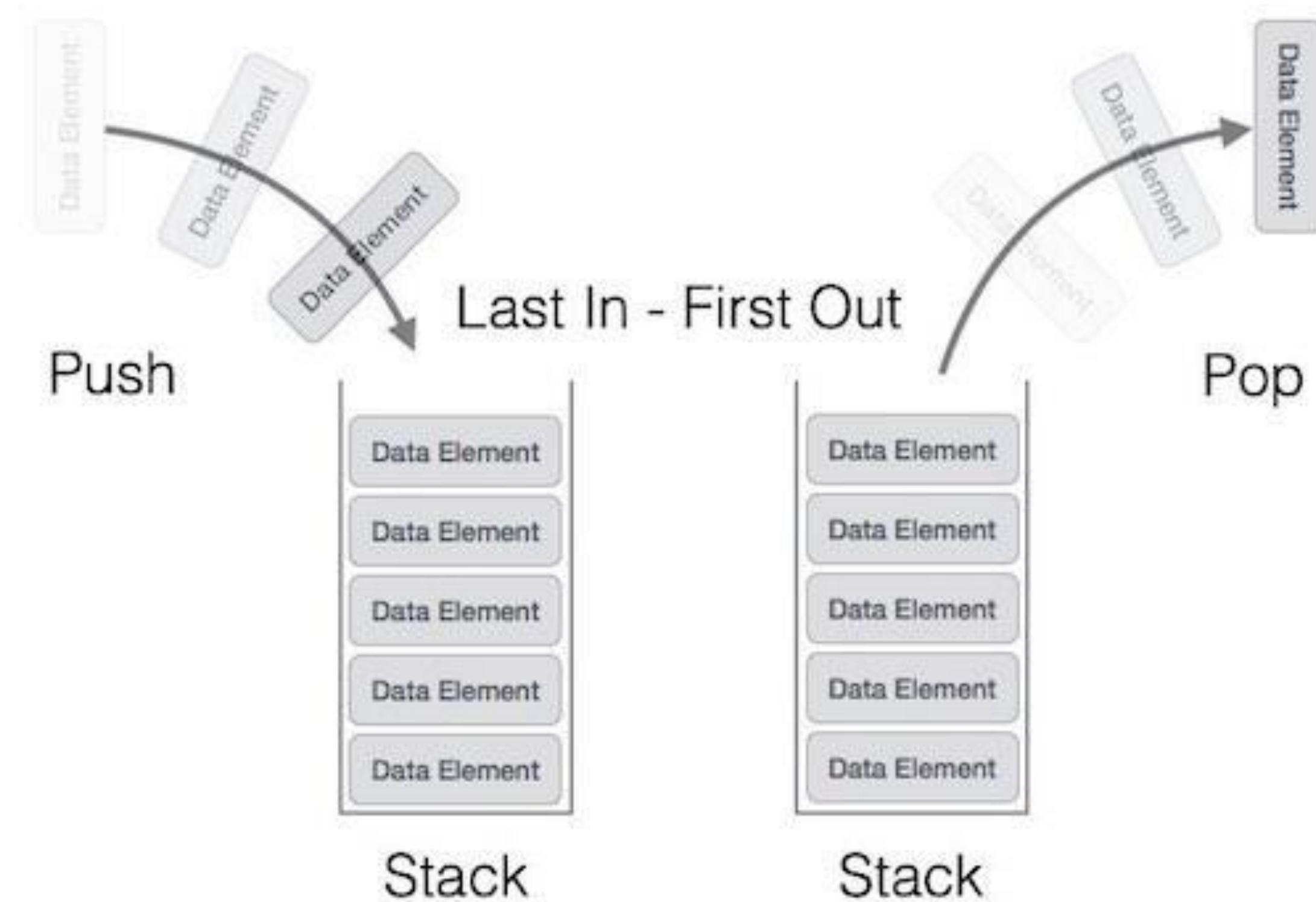
# Basic Operations

---

- the primary functions used to manipulate a stack are the push and pop function
- push inserts a new element and places it on top of the stack
- pop removes an element from the top of the stack
  - frees the memory that was allocated and returns the element
- other operations include
  - peek – looking at an element at the top without removing it
  - isEmpty – checking if the stack is empty

# Illustration

---



[https://www.tutorialspoint.com/data\\_structures\\_algorithms/stack\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)

---

# Applications

---

- stacks support recursive function calls
  - whenever a call is made, the function must know how to return to its caller, so the return address is pushed onto a stack
  - if a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller
- stacks are used to store data in memory
  - contain the space created for automatic variables on each invocation of a function
  - when the function returns the space for those variables is popped off the stack
- the call stack is useful when debugging
  - shows each function call and any nested function calls
- stacks are used by compilers in the process of evaluating expressions and generating machine language code
  - balancing symbols (matching starting and ending brackets, parenthesis)

# Applications

---

- stacks can be used when implementing page visited history in a web browser
- a stack could be used as an “undo” operation in a text editor
- a stack can be used to implement post-fix notation in a computer language ( order of operations and operands)
  - infix to Postfix /Prefix conversion
- used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem
- an application to reverse a string could use a stack
  - push each letter of the string on to the stack
  - then pop them back (string is now reversed)

---

# Stacks (Implementation)

Jason Fedin

# Overview

---

- a stack can be implemented either through an array or a linked list
- I will give a brief description of implementation via a linked list, but, we will mostly focus on the array implementation
- what identifies the data structure as a stack in either case is not the implementation but the interface
  - popping and pushing items onto the array or linked list, with few other helper operations

# Linked List implementation

---

- for the single linked list implementation of a stack, we use a pointer to the top of the list and a counter to keep track of the size of the list
- the push operation is performed by inserting the element at the front of the list
- the pop operation is performed by deleting the element at the front of the list
- some drawbacks of this implementation include speed (all the operations take constant time)
  - calls to malloc and free are expensive especially in comparison to the pointer manipulation routines

# Array Implementation

---

- a simpler way of implementing a stack is to use an array
  - no need for pointers
- push and pop functions are implemented by using the operations available on an array
- the only problem with this implementation is array size must be specified initially.
- the stack cannot grow and shrink dynamically

---

# Queues

Jason Fedin

# Overview

---

- another common data structure is the queue
- a queue is similar to a checkout line in a grocery store
  - the first person in line is serviced first
  - other customers enter the line only at the end and wait to be serviced
- queue elements are removed only from the head of the queue
- queue elements are inserted only at the tail of the queue
- a queue is referred to as a first-in, first-out (FIFO) data structure

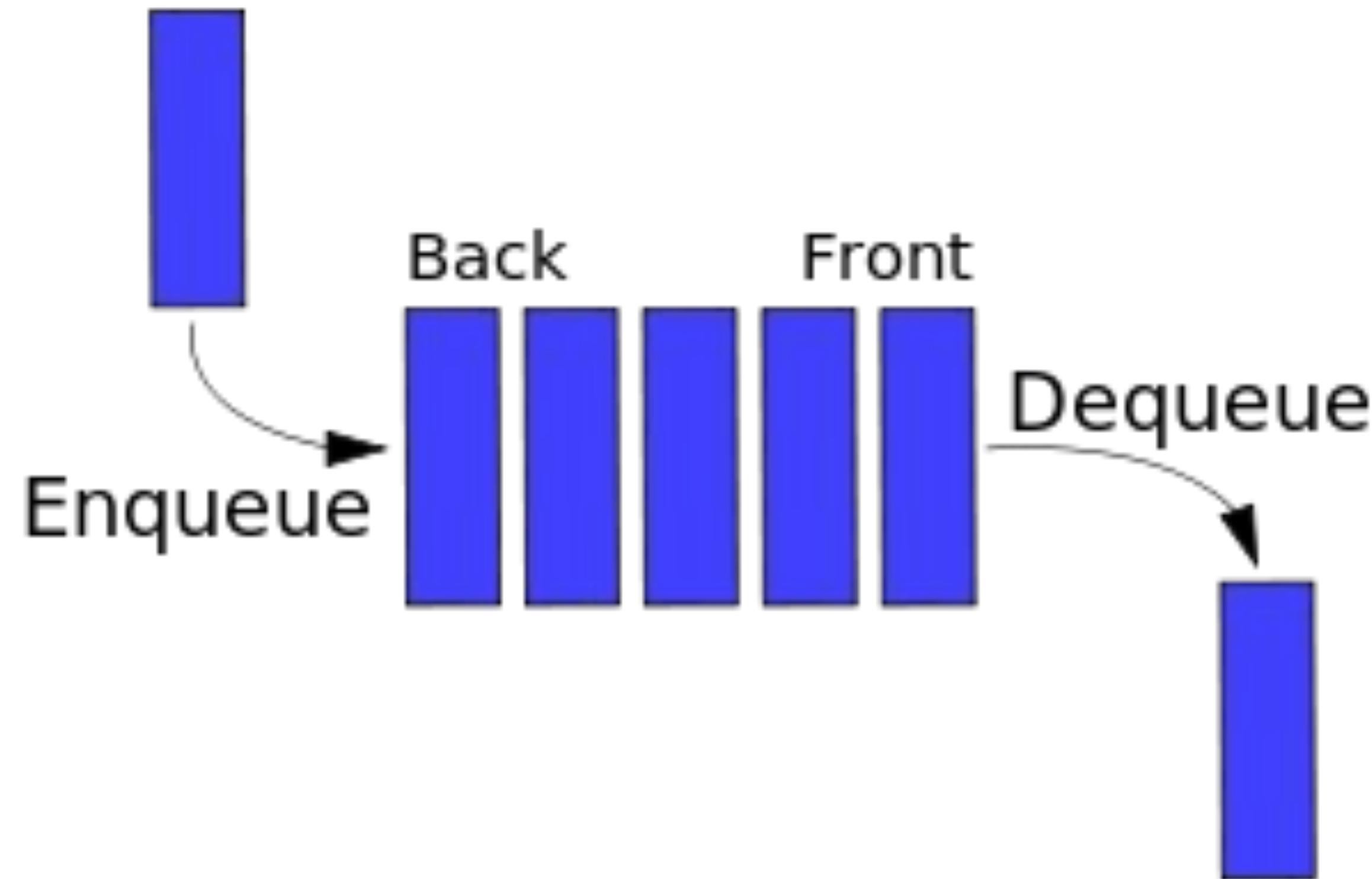
# Overview

---

- try not to confuse a queue with a stack
  - a stack works based on the last-in-first-out (LIFO) principle
  - the difference between stacks and queues is in removing
    - in a stack we remove the item that was most recently added
    - in a queue, we remove the item that was least recently added
- there are two main operations in a queue
  - enqueue
  - dequeue
- enqueue will insert an element into the back of the queue
- dequeue will remove an element from the front of the queue
- other operations
  - IsEmpty - check if queue is empty
  - IsFull - check if queue is full
  - peek - get the value of the front of queue without removing it
  - poll or offer (same as dequeue and enqueue)

# Overview

---



[https://en.wikipedia.org/wiki/Queue\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type))

---

# Queue Applications

---

- some computers have only a single processor, so only one user at a time may be serviced
  - entries for the other users are placed in a queue
  - each entry gradually advances to the front of the queue as users receive service
  - the entry at the front of the queue is the next to receive service
- queues are also used to support print spooling
  - a multiuser environment may have only a single printer
  - many users may be generating outputs to be printed
  - If the printer is busy, other outputs may still be generated
    - these are spooled to disk where they wait in a queue until the printer becomes available
- Information packets also wait in queues in computer networks
  - each time a packet arrives at a network node, it must be routed to the next node on the network
  - the routing node routes one packet at a time
    - additional packets are enqueueued until the router can route them
- basically, when a resource is shared among multiple consumers, queues are often utilized

# Advantages

---

- speed
  - data queues are a fast method of inter-process communication
  - data queues free up jobs from performing some work
    - can lead to a better response time and an overall improvement in system performance
  - data queues serve as the fastest form of asynchronous communication between two different tasks
    - there is less overhead than with database files and data areas
- flexibility
  - queues are flexible, requiring no communications programming
  - the programmer does not need any knowledge of inter-process communication
  - data queues allow computers to handle multiple tasks
  - the queue can remain active when there are no entries and ready to process data entries when necessary

---

# Queues (Implementation)

Jason Fedin

# Overview

---

- a queue can be implemented either through an array or a linked list
- I will give a brief description of implementation via a linked list, but, we will mostly focus on the array implementation
- what identifies the data structure as a queue in either case is not the implementation but the interface
  - enqueue and dequeue onto the array or linked list, with few other helper operations

# Linked List implementation

---

- for the single linked list implementation of a queue, we use a pointer to the front of the list (head) and a pointer to the rear of the list (tail)
- the enqueue operation is performed by inserting the element at the rear of the list
- the dequeue operation is performed by deleting the element at the front of the list
- some drawbacks of this implementation include speed (all the operations take constant time)
  - calls to malloc and free are expensive especially in comparison to the pointer manipulation routines

# Array Implementation

---

- we need to keep track of two indices, front and rear
- we enqueue an item at the rear and dequeue an item from front
- if we simply increment front and rear indices, then there may be problems, front may reach end of the array
- the solution to this problem is to increase front and rear in circular manner

# Array Implementation

---

- two integers named FRONT and REAR are used to keep track of the first and last elements in the queue
- when initializing the queue, we set the value of FRONT and REAR to -1
- on enqueueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR
- on dequeuing an element, we return the value pointed to by FRONT and increase the FRONT index.
- before enqueueing, we check if queue is already full.
- before dequeuing, we check if queue is already empty.
- when enqueueing the first element, we set the value of FRONT to 0.
- when dequeuing the last element, we reset the values of FRONT and REAR to -1.

# Array Implementation

---

- the following steps should be taken to enqueue (insert) data into a queue

Step 1 – Check if the queue is full.

Step 2 – If the queue is full, produce overflow error and exit.

Step 3 – If the queue is not full, increment rear pointer to point the next empty space.

Step 4 – Add data element to the queue location, where the rear is pointing

Step 5 – return success

# Array Implementation

---

Step 1 – Check if the queue is empty.

Step 2 – If the queue is empty, produce underflow error and exit.

Step 3 – If the queue is not empty, access the data where front is pointing.

Step 4 – Increment front pointer to point to the next available data element.

Step 5 – Return success.

---

# Binary Trees

Jason Fedin

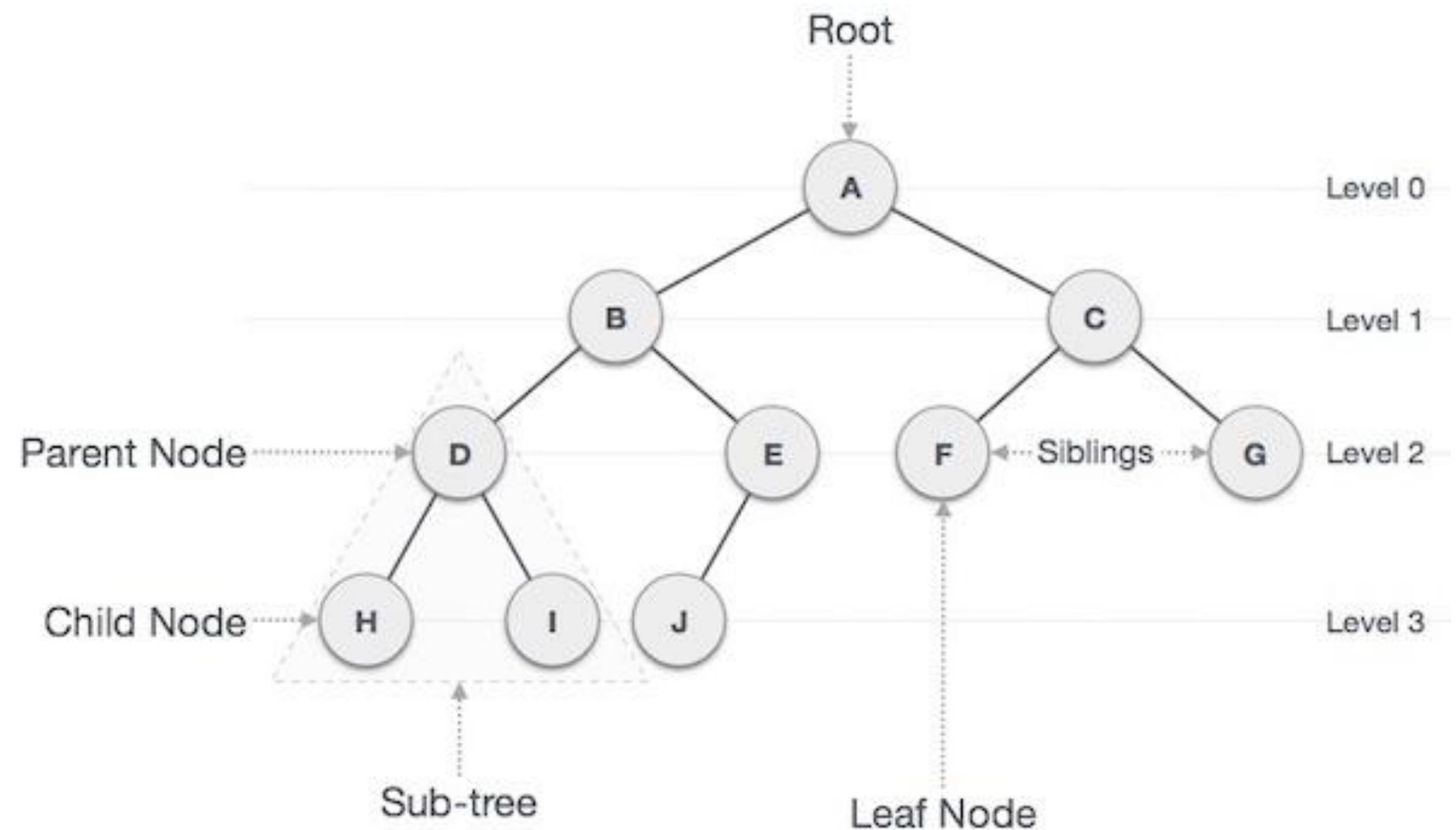
# Overview

---

- linked lists, stacks and queues are linear data structures
- a tree is a nonlinear, two-dimensional data structure with special properties
- trees whose nodes contain a maximum of two links are called binary trees
  - none, one, or both of which may be NULL
- the root node is the first node in a tree
- each link in the root node refers to a child
  - the left child is the first node in the left subtree
  - the right child is the first node in the right subtree
- the children of a node are called siblings
- a node with no children is called a leaf node

# Illustration

---



- computer scientists normally draw trees from the root node down
  - exactly the opposite of trees in nature

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/tree\\_data\\_structure.htm](https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm)

# Applications

---

- often used in search, game logic, autocomplete tasks, and graphics
- the most common use of a binary tree is a binary search tree
  - used in many search applications where data is constantly entering/leaving
  - map and set objects in many libraries
- binary space partition algorithm
  - used in almost every 3D video game to determine what objects need to be rendered
- used in many high-bandwidth routers for storing router-tables
- different forms of binary trees are used by compilers to parse expressions
- data compression algorithms
- database problems like indexing

# Binary search trees

---

- a binary search tree is a linked structure that incorporates the binary search algorithm
  - ordered data structure
  - allows for fast lookup, addition and removal of items
- a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays
- a binary search tree has the following properties
  - values in any left subtree are less than the value in its parent node
  - values in any right subtree are greater than the value in its parent node

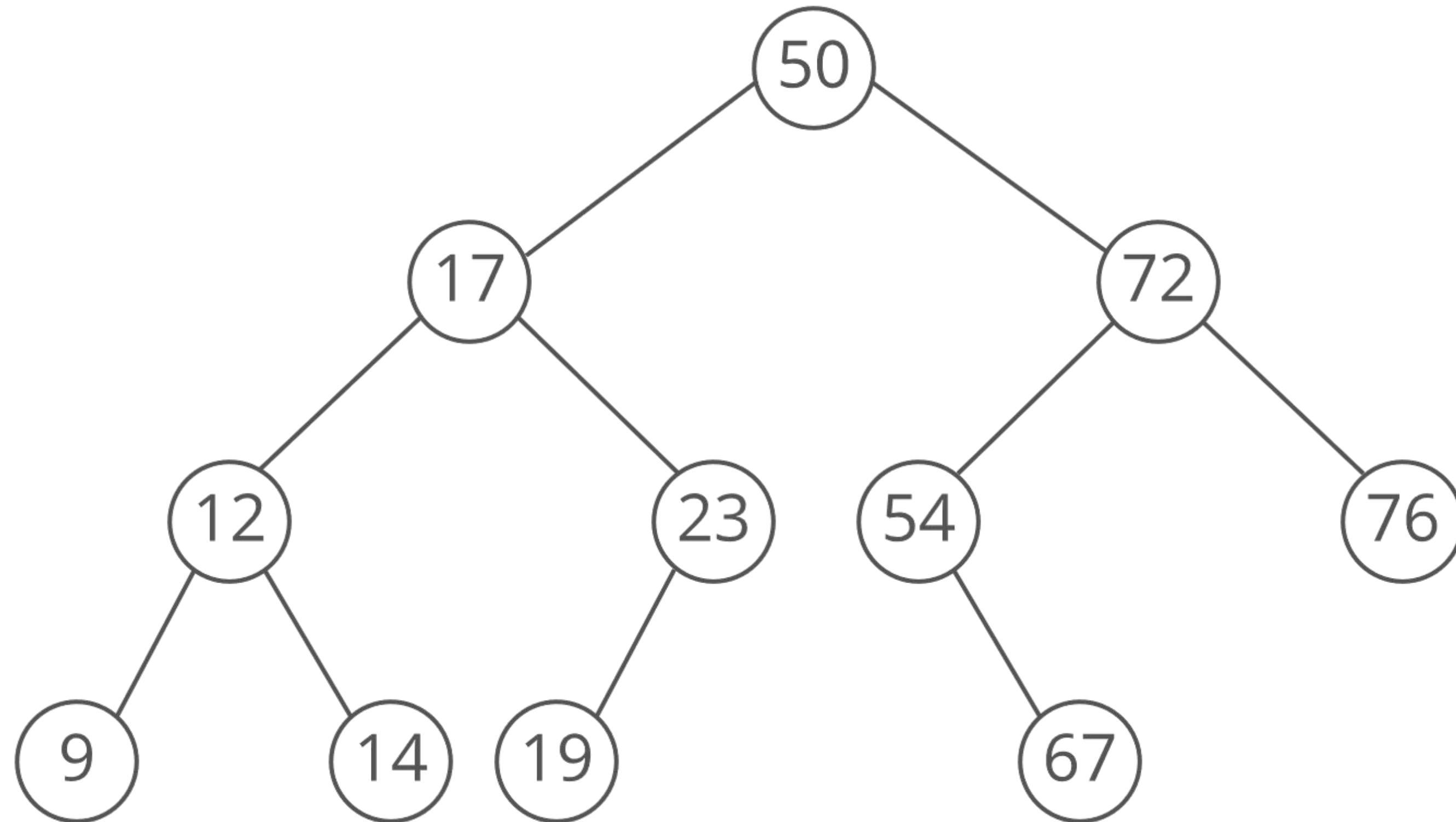
# Binary search trees

---

- is considered balanced if every level of the tree is fully filled with the exception of the last level
  - on the last level, the tree is filled left to right
- a perfect BST is one in which it is both full and complete
  - all child nodes are on the same level and each node has a left and a right child node
- a binary search tree has no duplicate node values
  - an attempt to insert a duplicate value will be recognized when creating the tree
    - a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did
    - the duplicate will eventually be compared with a node in the tree containing the same value
      - duplicate value is discarded at this point
- a node can only be inserted as a leaf node in a binary search tree

# Illustration

---



<https://www.interviewcake.com/concept/java/binary-search-tree>

---

# Binary search tree

---

- searching a binary tree for a value that matches a key value is fast
- if the tree is tightly packed, each level contains about twice as many elements as the previous level
- suppose you want to find an item (target)
  - if the item precedes the root item, you need to search only the left half of the tree
  - if the target follows the root item, you need to search only the right subtree of the root node
  - one comparison eliminates half the tree
- suppose you search the left half
  - means comparing the target with the item in the left child
  - if the target precedes the left-child item, you need to search only the left half of its descendants, and so on
  - each comparison cuts the number of potential matches in half
- a binary search tree combines a linked structure with binary search efficiency

# Basic operations on a binary search tree

---

- **insert**

- Inserts an element in a tree/create a tree

- **search**

- searches an element in a tree

- trees can be traversed in different ways because they are non-linear (inorder, preorder, postorder)

- **inorder traversal**

- gives nodes in non-decreasing order
  - the steps are
    - traverse the left subtree in-order
    - process the value in the node
    - traverse the right subtree in-order
  - the value in a node is not processed until the values in its left subtree are processed
  - prints the node values in ascending order

# Basic operations on a binary search tree

---

- **preorder traversal**

- used to create a copy of the tree
- the value in each node is processed as the node is visited
- the values in the left subtree are then processed
- then the values in the right subtree are processed

- **postorder traversal**

- used to delete the tree the steps are
  - traverse the left subtree post-order
  - traverse the right subtree post-order
  - then process the value in the node
- the value in each node is not processed until the values of its children are processed

InOrder(root) visits nodes in the following order:

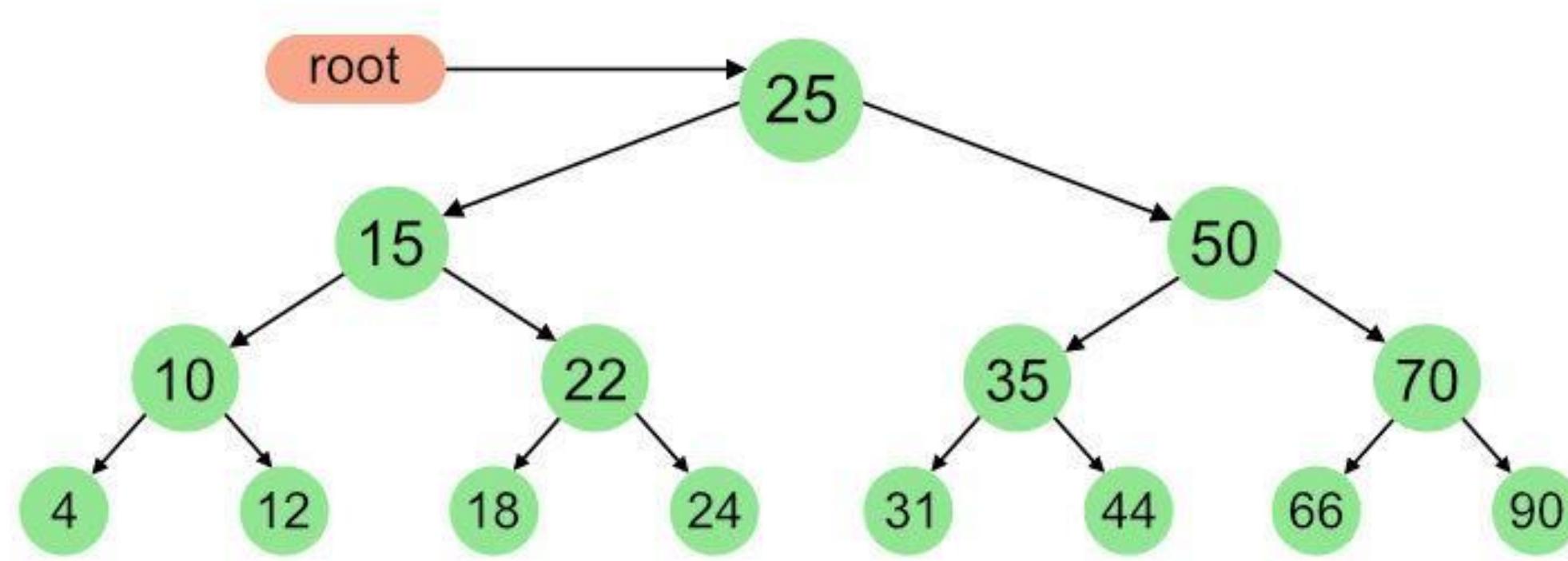
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Inorder (Left, Root, Right)

Preorder (Root, Left, Right)

Postorder (Left, Right, Root)

<https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

---

# Binary Search Tree (Implementation)

Jason Fedin

# Overview

---

- when inserting or searching for an element in a binary search tree
  - the key of each visited node has to be compared with the key of the element to be inserted or found
- the shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate
- after a sequence of random insertion and deletion
  - the expected height of the tree approaches the square root of the number of keys ( $\sqrt{n}$ )
  - grows much faster than  $\log n$
- there has been a lot of research to prevent degeneration of the tree resulting in worst case time complexity of  $O(n)$

---

# Interprocess Communication (IPC)

Jason Fedin

# Processes

---

- a process is a program in execution
- a program is a file containing the information of a process
  - when you start execution of the program, it is loaded into RAM and starts executing
- each process has its own address space and (usually) one thread of control
- you can have multiple processes executing the same program
  - BUT each process has its own copy of the program within its own address space and executes it independently of the other copies
- processes are organized hierarchically
  - each process has a parent process which explicitly arranged to create it
- the processes created by a given parent are called its child processes
  - a child inherits many of its attributes from the parent process

# Processes

---

- a system call is a request for service that a program makes of the kernel (the brain of the operating system)
  - the service is generally something that only the kernel has the privilege to do
  - sometimes called kernel calls
- programmers do not normally need to be concerned with system calls because there are functions in the GNU C Library to do virtually everything that system calls do
  - these functions work by making system calls themselves
- each process is identified with a unique positive integer called a process ID (PID)
- the system call getpid() returns the process ID of the calling process
  - this call is always successful and thus no return value to indicate an error

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

- the system call getppid() returns the Parent PID of the calling process

# Processes (example)

---

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
 int mypid, myppid;
 printf("Program to know PID and PPID's information\n");
 mypid = getpid();
 myppid = getppid();
 printf("My process ID is %d\n", mypid);
 printf("My parent process ID is %d\n", myppid);
 printf("Cross verification of pid's by executing process commands on shell\n");
 system("ps -ef");
 return 0;
}
```

# Interprocess communication

---

- a process can be of two types
  - independent process
    - not affected by the execution of other processes
    - does not interact with other programs
  - co-operating process
    - affected by other executing processes
    - can be used for increasing computational speed, convenience and modularity
- what if you want to send specific data to a process or read its output?
- interprocess communication lets processes work together
  - programs can be much more powerful by letting them talk to other programs on your system
  - often referred to as IPC

# Interprocess communication

---

- to reiterate, each process has its own address space
- if any process wants to communicate with some information from its own address space to other processes
  - it is only possible with IPC (inter process communication) techniques
- the communication between multiple processes can be seen as a method of co-operation between them
- communication can be of two types
  - between related processes initiating from only one process, such as parent and child processes
  - between unrelated processes, or two or more different processes

# Interprocess communication

---

- a process can communicate with another process in many different ways
- pipes (same process)
  - first process communicates with the second process, allows flow of data in one direction only (half duplex)
- named pipes (different processes, FIFO)
  - the first process can communicate with the second process and vice versa at the same time (full duplex)
- message queues
  - processes will communicate with each other by posting a message and retrieving it out of a queue (full duplex)
- shared memory
  - communication between two or more processes is achieved through a shared piece of memory among all processes
- sockets
  - mostly used to communicate over a network between a client and a server
- signals
  - communication between multiple processes by way of signaling
  - a source process will send a signal (recognized by number) and the destination process will handle it accordingly

# Pipes (named and anonymous)

---

- a pipe is a communication mechanism between two or more related or interrelated processes
  - can be either within one process or a communication between the child and the parent processes
- communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.
- can be thought of filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug
  - the filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe
  - implies that one output (water) is input for the other (bucket)
- another name for a named pipe is FIFO (First-In-First-Out)
- system calls are used for creating and using pipes
- steps to utilize pipes for IPC are
  - create a pipe
  - send a message to the pipe
  - retrieve the message from the pipe and write it to the standard output
  - send another message to the pipe
  - retrieve the message from the pipe and write it to the standard output

# Message Queues

---

- all processes can exchange information through access to a common system message queue
- message queues are a sort-of mix between signals and sockets
  - allows you to create a data stream of messages which can allow one or multiple processes to communicate between them
  - sending and receiving can be achieved by using a simple function
- the sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process
  - each message is given an identification or type so that processes can select the appropriate message
  - process must share a common key in order to gain access to the queue in the first place.
- system calls are used for creating and using message queues
- to perform communication using message queues, follow the below steps
  - the ftok() function is used to generate a unique key
  - create a message queue or connect to an already existing message queue (msgget())
  - write into message queue (msgsnd())
  - read from the message queue (msgrcv())
  - perform control operations on the message queue (msgctl())

# Shared Memory

---

- shared memory is memory shared between two or more processes
- communication is done via this shared memory where changes made by one process can be viewed by another process
- communication between processes requires processes to share some variable and it completely depends on how programmer will implement it
- system calls allow you to create a memory segment which can be accessed and modified by multiple processes
  - setting it up is easy (and similar to Message Queues)
  - managing the given data segment might be the only tricky part
- to perform communication using shared memory, follow these steps
  - create the shared memory segment or use an already created shared memory segment (`shmget()`)
  - attach the process to the already created shared memory segment (`shmat()`)
  - detach the process from the already attached shared memory segment (`shmdt()`)
  - control operations on the shared memory segment (`shmctl()`)

---

# Signals

Jason Fedin

# Overview

---

- signals are a limited form of inter-process communication (IPC), typically used in Unix like operating systems
  - a notification to a process indicating the occurrence of an event
- signals are a way to communicate information to a process about the state of other processes, the operating system and hardware, so that the process can take appropriate action
- signals are generated by the system or the user can also generate the signal programmatically
- when a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal
  - a process can receive a particular signal asynchronously (at any time)
    - after receiving the signal, the process will interrupt its current operations
    - the process has to stop whatever it is doing and go deal with the signal
    - it will either handle or ignore the signal, or in some cases terminate
- there are fix set of signals that can be sent to a process defined by the operating system
- a signal is just a short message which contains a single integer value

# Examples

---

- when we are working in the shell and wish to "kill all cat programs" we type the command
  - killall cat - sends a signal to all processes named cat that says "terminate." (SIGTERM)
- signals are also used in the context of terminal signaling which is how programs stop, start and terminate
  - typing Ctrl-c that is the same as sending a SIGINT signal
  - typing Ctrl-z that is the same as sending a SIGTSTP signal
  - typing fg or bg that is the same as sending a SIGCONT signal
- when a program calls the fgets() function
  - the operating system reads the data from the keyboard, and when it sees the user hit Ctrl-C, sends an interrupt signal to the program
- some other signal events include
  - illegal instructions, segmentation violations, termination orders from the operating system
  - floating-point exceptions (division by zero or multiplying large floating-point values)

# Default actions

---

- whenever a signal is raised (either programmatically or system generated signal), a default action is performed for some signals
- **Term** - the process will terminate
- **Core** - the process will terminate and produce a core dump file that traces the process state at the time of termination
- **Ign** - the process will ignore the signal
- **Stop** - the process will stop, like with a Ctrl-Z
- **Cont** - the process will continue from being stopped

# Signal names

---

- each signal has a name, value, and default action that a process will take in response
  - each signal starts with SIG
  - available signals can be checked with the command kill -l (l for Listing signal names)

| Signal  | Value | Action | Comment                                                                 |
|---------|-------|--------|-------------------------------------------------------------------------|
| <hr/>   |       |        |                                                                         |
| SIGHUP  | 1     | Term   | Hangup detected on controlling terminal or death of controlling process |
| SIGINT  | 2     | Term   | Interrupt from keyboard                                                 |
| SIGQUIT | 3     | Core   | Quit from keyboard                                                      |
| SIGILL  | 4     | Core   | Illegal Instruction                                                     |
| SIGABRT | 6     | Core   | Abort signal from abort(3)                                              |
| SIGFPE  | 8     | Core   | Floating point exception                                                |
| SIGKILL | 9     | Term   | Kill signal                                                             |
| SIGSEGV | 11    | Core   | Invalid memory reference                                                |
| SIGPIPE | 13    | Term   | Broken pipe: write to pipe with no readers                              |

# Signal names

---

| Signal  | Value    | Action | Comment                           |
|---------|----------|--------|-----------------------------------|
| <hr/>   |          |        |                                   |
| SIGALRM | 14       | Term   | Timer signal from alarm(2)        |
| SIGTERM | 15       | Term   | Termination signal                |
| SIGUSR1 | 30,10,16 | Term   | User-defined signal 1             |
| SIGUSR2 | 31,12,17 | Term   | User-defined signal 2             |
| SIGCHLD | 20,17,18 | Ign    | Child stopped or terminated       |
| SIGCONT | 19,18,25 | Cont   | Continue if stopped               |
| SIGSTOP | 17,19,23 | Stop   | Stop process                      |
| SIGTSTP | 18,20,24 | Stop   | Stop typed at tty                 |
| SIGTTIN | 21,21,26 | Stop   | tty input for background process  |
| SIGTTOU | 22,22,27 | Stop   | tty output for background process |

---

# Raising a Signal

Jason Fedin

# Overview

---

- signals can be generated by a user programmatically in addition to be sent by the OS (Operating System)
- the signal handling library (<signal.h>) provides the capability to raise specific signals
- a signal can be generated by calling raise() or kill() system calls
  - raise() sends a signal to the current process
  - kill() sends a signal to a specific process.

int raise(int sig)

- if this function is executed successfully, the signal specified in sig is generated
- if the call is unsuccessful, raise() returns a non-zero value
- the function raise() can only send a signal to the program that contains it
  - cannot send a signal to other processes
  - to send a signal to other processes, the system call kill() should be used
- after raising the signal, the execution of the current process is stopped

# alarm()

---

- the alarm function provides a mechanism for a process to interrupt itself in the future
- the alarm() function sets a timer
  - when the timer expires, the process receives a signal (SIGALARM)
- if we ignore or do not catch this signal, the process is terminated
  - the default action for SIGLARM is to terminate the process
- there is only one alarm clock per process
- the alarm() function will return a value if another alarm has been previously set

---

# Handling a Signal using the signal function

Jason Fedin

# Overview

---

- it is possible to handle or catch almost all signals in your program
- sometimes you need to run your own code when receiving a signal (handle/catch)
  - maybe your process has files or network connections open
  - it might want to close things down and tidy up before exiting
- it is also possible to ignore almost all signals
  - neither performing the default action nor handling the signal
- a few signals cannot be ignored or handled/caught
  - SIGKILL, SIGABRT and SIGSTOP (which is why "kill 9" is the ultimate kill statement)
- the actions performed for signals are
  - default action
  - handle the signal
  - ignore the signal

# Handling/Catching a signal

---

- the signal handling library (<signal.h>) provides the capability to catch and handle signals
  - signal handling can be done in two different ways through system calls
    - signal() or sigaction()
- first, let's discuss using the signal function
- this function is used to tell the operating system which function it should call when a signal is sent to a process
  - the function is used to deal with (or handle) a signal that is sent to it
- if you have a function called foo() that you want the operating system to call when someone sends an interrupt signal to your process
  - you need pass the function name foo to the signal function as a parameter
  - the function foo() is called the handler
- handlers are intended to be short, fast pieces of code
  - they should do just enough to deal with the signal that has been received

# Handling/Catching a signal

---

- the signal function receives two arguments
  - an integer signal number and a pointer to the signal handling function
- the signal function returns
  - on success returns the address of a function that takes an integer argument and has no return value
  - on error returns SIG\_ERR in case of error

```
#include <signal.h>
```

```
typedef void (*sighandler_t) (int);
sighandler_t signal(int signum, sighandler_t handler);
```

- the system call signal() would call the registered handler (second parameter) upon generation of signal
- the handler (second parameter) can be either one of
  - SIG\_IGN (Ignoring the Signal)
  - SIG\_DFL (Setting signal back to default mechanism)
  - or user-defined signal handler or function address

---

# Handling a Signal using sigaction

Jason Fedin

# Handling/Catching a signal (sigaction)

---

- you can handle or catch signals in your program using a sigaction
- a sigaction is essentially a function wrapper
  - a struct that contains a pointer to a function
- sigactions are used to tell the operating system which function it should call when a signal is sent to a process
  - the function is used to deal with (or handle) a signal that is sent to it
- if you have a function called foo() that you want the operating system to call when someone sends an interrupt signal to your process
  - you need to wrap the foo() function up as a sigaction
  - the function foo() is called the handler
- handlers are intended to be short, fast pieces of code
  - they should do just enough to deal with the signal that has been received

# sigaction function

---

- the signal handling library (<signal.h>) provides the capability to catch and handle signals via the sigaction function
  - used to either examine or change a signal action

```
int sigaction(int signum, const struct sigaction *newaction, struct sigaction *oldaction)
```

- sigaction() takes three parameters:
  - the signal number
    - the integer value of the signal you want to handle
  - the new action
    - this is the address of the new sigaction you want to register
  - the old action
    - if you pass a pointer to another sigaction, it will be filled with details of the current handler that you are about to replace
    - If you do not care about the existing signal handler, you can set this to NULL
- the sigaction() function will return -1 if it fails and will also set the errno variable
  - you should always check for errors in your own code

# struct sigaction

---

```
int sigaction(int signum, const struct sigaction *newaction, struct sigaction *oldaction)
```

- the sigaction structure contains the following fields

## Field 1 – Handler mentioned either in sa\_handler or sa\_sigaction

```
void (*sa_handler)(int);
```

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

- the handler for sa\_handler specifies the action to be performed based on the signum
  - SIG\_DFL indicating default action
  - SIG\_IGN to ignore the signal or pointer to a signal handling function
- the handler for sa\_sigaction
  - specifies the signal number as the first argument
  - specifies the pointer to siginfo\_t structure as the second argument
  - specifies the pointer to user content as the third argument (out of scope)
- the structure siginfo\_t contains signal information such as the signal number to be delivered, signal value, process id, real user id of sending process, etc.

# struct sigaction

---

## Field 2 – Set of signals to be blocked

```
int sa_mask;
```

- this variable specifies the mask of signals that should be blocked during the execution of signal handler

## Field 3 – Special flags

```
int sa_flags;
```

- this field specifies a set of flags which modify the behavior of the signal

## Field 4 – Restore handler.

```
void (*sa_restorer) (void); // returns 0 on success and -1 in case of failure
```

# Summary

---

- the operating system talks to processes using signals
  - programs are normally stopped using signals
  - for most error signals, the default handler stops the program
- handlers can be replaced with the signal() or sigaction functions
- you can send signals to yourself with raise()
- the interval timer sends SIGALRM signals.
  - the alarm() function sets the interval timer.
  - there is one timer per process
- kill sends signals to a process
  - kill -KILL will always kill a process

---

# The fork() system call

Jason Fedin

# Overview

---

- in multitasking operating systems, processes (running programs) need a way to create new processes
  - a new process is often needed to run other programs or to run a different “branch” of the existing program
- the fork() system call is a function where a process creates a copy of itself
  - creates a new process
  - defined in <unistd.h>
- when a process calls fork, it is deemed the parent process and the newly created process is its child
- the child process has an exact copy of all the memory segments of the parent process
  - the fork operation creates a separate address space for the child
  - updating a variable in one process will not affect the other

# Use Case

---

- fork() is often used with the exec() system call in order to start the execution of a different program
- a typical use case is that a process will first invoke the fork function to create a copy of itself
  - then, the copy (child process), calls the exec system call to overlay itself with the other program
    - it ceases execution of its former program in favor of the other
- after a new child process is created, both processes will execute the next instruction following the fork() system call
  - child and parent processes run in parallel
    - we have to distinguish the parent from the child
      - this can be done by testing the returned value of fork()

# Overview

---

- the fork() function does not take any parameters and returns a process ID

```
pid=fork();
```

- the returned process ID is of type pid\_t defined in sys/types.h
  - the process ID is an integer
  - as mentioned in a previous lecture, a process can use function getpid() to retrieve the process ID assigned to this process
- in the subsequent blocks of code you need to check the value of pid
  - returns a negative value when the creation of a child process was unsuccessful
  - returns a zero value to the newly created child process
  - returns a positive value, the process ID of the child process, to the parent
- when we test the value of pid to find whether it is equal to zero or greater than it we are effectively finding out whether we are in the child process or the parent process

---

# Threads

Jason Fedin

# Overview

---

- programs often need to do several things at the same time
- games need to update the graphics on the screen while also reading input from a game controller
- chat programs will need to read text from the network and send data to the network at the same time
- media players will need to stream video to the display as well as watch for input from the user controls

# Processes

---

- simple processes do one thing at a time
- in the real world, you cannot do all of the tasks at the same time (not by yourself)
  - if someone comes into the shop, you may need to stop stocking the shelves to help a customer
- if you work in a shop alone, you are the same as a simple process
  - you do one thing after another, but always one thing at a time
  - you can switch between tasks to keep everything going
    - but what if there is a blocking operation?
    - what if you are serving someone at the checkout and the phone rings?
- most of the programs we have written so far have had a single path of execution
  - there has only been one task working inside the program's process
  - the main() function is the single path of execution (sequentially, line by line)

# Working on tasks at the same time

---

- so, how can your code perform several different tasks at once?
  - how about creating a process to handle all these tasks (fork)?
    - the answer is no
- processes take time to create
  - creating a process for each task is not efficient
- processes cannot share data easily
  - separate processes have a complete copy of all the data from the parent process
    - if the child needs to send data back to the parent, then you need use some method of IPC (pipes, shared memory, message queues, sockets)
- processes are just plain difficult
  - you need to create a chunk of code to generate processes, and that can make your programs long and messy

# Threads

---

- you need something that starts a separate task quickly, can share all of your current data, and will not require a lot of code to write
  - the answer is to use threads
- threads are a way to divide a single process into sub-runnable parts
  - a separate path of execution inside a program
  - sometimes called lightweight processes
- a thread can also be scheduled independently of the larger program that it is inside of (done by the OS)
  - means that a single program may actually use more than 100% of CPU resources on multi-processor machines
- a thread has its own unique id, program counter (PC), a register set, and a stack space just like a process
- threads share memory across each thread by having the same address space (unlike multi-processes)
  - two threads have access to the same set of variables and can alter each other's variable values
    - if one thread changes a global variable, all of the other threads will see the change immediately
- threads also share OS resources like open files and signals
  - all of the threads will all be able to read and write to the same files and talk on the same network sockets

# Multi-threading

---

- threads are popular way to improve an application through parallelism (simultaneous running of code)
  - several tasks can execute concurrently (many tasks can run in any order and possibly in parallel)
- a multithreaded program is like a shop with several people working in it
  - If one person is running the checkout, another is filling the shelves, and someone else is waxing the surfboards
    - everybody can work without interruptions
    - if one person answers the phone, it won't stop the other people in the shop
- in the same way that several people can work in the same shop, you can have several threads living inside the same process (program)
  - each thread runs independently (a thread of execution)
- multi-threading means you can give each thread a separate task and they will all be performed at the same time
  - in a browser, multiple tabs can be different threads
  - MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

# Advantages of using threads

---

- threads require less overhead than "forking" or spawning a new process
  - the system does not initialize a new system virtual memory space and environment
- threads provide efficient communication and data exchange because they share the same address space
- threaded applications offer potential performance gains and practical advantages over non-threaded applications in several ways
  - the creation of a thread is much faster (much less operating system overhead)
  - faster context switching
  - faster termination of a thread

# Disadvantages of using threads

---

- very easy to overlook the consequences of interactions between concurrently executing threads
  - considerable potential for very obscure errors in your code
- providing for thread synchronization is the biggest issue
  - the potential for two or more threads attempting to access the same data at the same time
- imagine a program with several threads that may access a variable containing salary data
- suppose that two independent threads can modify the value
  - if one thread accesses the value to change it and the second thread does the same before the first thread has stored the modified value
    - you will have inconsistent data

# Multi-threading in C

---

- C is a language that runs on one thread by default (main), which means that the code will only run one instruction at a time
- C does not contain any built-in support for multithreaded applications
  - unless you count C11, but, these threads are not very portable and not widely supported
- threading was traditionally provided via hardware and OS support in the past
  - implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications
- in order to take full advantage of the capabilities provided by threads, a standardized programming interface was required
- In 1995, POSIX became the standard interface for many system calls in UNIX including the threading environment
- we are going to write multi-threaded C programs using POSIX threads
  - also known as pthreads, implementation is available with the gcc compiler
  - the key model for programming with threads in nearly every language (Java, python and other high level languages)

# POSIX thread (pthread) libraries

---

- the POSIX thread libraries are a standards based thread API for C/C++
  - allows one to spawn a new concurrent process flow
  - can be found on almost any modern POSIX-compliant OS
- it is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor
  - thus gaining speed through parallel or distributed processing
- a thread is spawned by defining a function and its arguments which will be processed in the thread
- the purpose of using the POSIX thread library in your software is to execute software faster
- pthread functions are defined in a pthread.h header/include file and implemented in a thread library

# Upcoming lectures

---

- In the upcoming lectures we will discuss
  - creation of a thread
  - passing arguments and returning data to/form a thread
  - common thread functions (cancel, detach, stack management)
  - thread synchronization (mutexes and semaphores)
- it is not possible to cover more than an introduction on pthreads given time constraints
  - an entire 15 hours course could be creating on posix threads
- some pthread concepts such as thread scheduling, thread-specific data, thread canceling, handling signals and reader/writer locks will not be covered

---

# Creating a Thread

Jason Fedin

# pthread API

---

- the functions that comprise the pthreads API can be grouped into three major categories
- thread management
  - routines that work directly on threads - creating, detaching, joining, etc.
  - also include functions to set/query thread attributes (joinable, scheduling etc.)
- synchronization
  - Routines that manage read/write locks and barriers and deal with synchronization
  - mutex functions provide for creating, destroying, locking and unlocking mutexes (mutual exclusion)

# pthreads API

---

- condition variables
  - routines that address communications between threads that share a mutex
  - based upon programmer specified conditions
- operations that can be performed on threads include
  - thread creation
  - termination
  - synchronization (joins,blocking)
  - scheduling
  - data management
  - process interaction

# Creating a thread

---

- the lifecycle of a thread, much like a process, begins with creation
  - threads are not forked from a parent to create a child
  - instead they are simply created with a starting function as the entry point
- on POSIX operating systems, there is a library named pthread.h
  - allows you to create threads and perform many operations on threads
  - must include this library when creating and using threads
- In this lecture, I will describe three functions that are involved in the creation of a thread
  - pthread\_create, pthread\_exit, and pthread\_join

# Creating a thread

---

- the `pthread_create` function is called to create a new thread and make it executable
  - initially, your `main()` program comprises a single, default thread and all other threads must be explicitly created by the programmer
- the maximum number of threads that may be created by a process is implementation dependent
  - once created, threads are peers, and may create other threads
  - there is no implied hierarchy or dependency between threads

# pthread\_create

---

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
```

- `pthread_create()` takes four arguments
  - the first argument is of type `pthread_t`
    - an integer used to identify the thread in the system
    - upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by `thread`
  - the second argument specifies attributes for the thread
    - you can specify a thread attributes object, or `NULL` for the default values
    - examples of attributes that can be specified include detached state, scheduling policy, scope, stack address and stack size
  - the third argument is name of the function that the thread will execute once it is created
  - the fourth argument is used to pass arguments to the function (`start_routine`)
    - a pointer cast of type `void` is required
    - `NULL` may be used if no argument is to be passed
    - to pass multiple arguments, you would need to use a pointer to a structure

# pthread\_join

---

- it is often useful to be able to identify when a thread has completed or exited
- the method for doing this is to join the thread, which is a lot like the wait() call for processes
- a join is performed when one wants to wait for a thread to finish
  - used to link the current thread process to another thread
  - a thread calling routine may launch multiple threads then wait for them to finish to get the results
- a call to pthread\_join blocks the calling thread until the thread with identifier equal to the first argument terminates
  - makes the program stops in order to wait for the end of the selected thread
- typically, only the main thread calls join, but other threads can also join each other
- all threads are automatically joined when the main thread terminates

# pthread\_join

---

- the pthread\_join() also receives the return value of your thread function and stores it in a void pointer variable
  - once both threads have finished, your program can exit smoothly

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- the first argument is the thread id of the thread you want to wait for
- if the second argument is not NULL, this value is passed to pthread\_exit() by the terminating thread

# pthread\_exit

---

- threads can be terminated in a number of ways
  - by explicitly calling pthread\_exit
  - by letting the thread function return
  - by a call to the function exit which will terminate the process including any threads
- typically, the pthread\_exit() routine is called after a thread has completed its work and is no longer required to exist
- if main() finishes before the threads it has created finish, and exits with pthread\_exit(), the other threads will continue to execute
  - otherwise, they will be automatically terminated when main() finishes
- although not explicitly required to call pthread\_exit() at the end of the thread function
  - it is good practice to do so, as you may have the need to return some arbitrary data back to the caller via pthread\_join()

# pthread\_exit

```
void pthread_exit(void *value_ptr);
```

- the first argument makes the value\_ptr available to any successful join with the terminating thread
- sometimes it is desirable for a thread not to terminate (e.g., a server with a worker thread pool)
  - can be solved by placing the thread code in an infinite loop and using condition variables

---

# Passing arguments and returning values to/from threads

Jason Fedin

# Passing Arguments to Threads

---

- the `pthread_create()` function permits the programmer to pass one argument to the thread start routine
- when passing more than one argument, you can create a structure which contains all of the arguments, and then pass a pointer to that structure in the `pthread_create()` function
- all arguments must be cast to `(void *)`

---

# Common Thread functions

Jason Fedin

# pthread\_detach

---

- when a thread is created, one of its attributes defines whether it is joinable or detached
  - by default if you passed NULL as the second argument the thread will be in joinable state
  - only threads that are created as joinable can be joined
    - if a thread is created as detached, it can never be joined
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state
  - some system resources may be able to be freed
- there are two common use cases of when you would want to detach
  - in a cancellation handler for a pthread\_join()
    - nearly essential to have a pthread\_detach() function in order to detach the thread on which pthread\_join() was waiting
  - in order to detach the "initial thread" (as may be desirable in processes that set up server threads)
- the pthread\_detach() routine can be used to explicitly detach a thread even though it was created as joinable

```
int pthread_detach(pthread_t thread);
```

# Stack Management

---

- the POSIX standard does not dictate the size of a thread's stack
  - implementation dependent and varies
- exceeding the default stack limit is often very easy to do
  - results in program termination and/or corrupted data
- safe and portable programs do not depend upon the default stack limit
  - instead, explicitly allocate enough stack for each thread by using the `pthread_attr_setstacksize` function

`pthread_attr_getstacksize (attr, stacksize)`  
`pthread_attr_setstacksize (attr, stacksize)`

# pthread\_equal and pthread\_once

---

- the pthread\_equal function compares two thread IDs
  - if the two IDs are different 0 is returned, otherwise a non-zero value is returned
  - operator == should not be used to compare two thread IDs against each other

pthread\_equal (thread1,thread2)

- the pthread\_once function executes the init\_routine exactly once in a process
  - the first call to this function by any thread in the process executes the given init\_routine, without parameters
  - any subsequent call will have no effect

pthread\_once (once\_control, init\_routine)

- the init\_routine routine is typically an initialization routine
- the once\_control parameter is a synchronization control structure that requires initialization prior to calling pthread\_once
  - `pthread_once_t once_control = PTHREAD_ONCE_INIT;`

---

# Thread Synchronization Concepts

Jason Fedin

# Overview

---

- In this lecture, we will be discussion some of the issues that occur when using threads and concepts that can help prevent these issues
- the issues that we will address are the following:
  - race conditions
  - deadlocks
- our goal is to strive for thread safe code
- all of these issues are related to thread synchronization
- thread synchronization is the concurrent execution of two or more threads that share critical resources
- threads should be synchronized to avoid critical resource use conflicts
  - otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time

# Race conditions

---

- while the code may appear on the screen in the order you wish the code to execute
  - threads are scheduled by the operating system and are executed at random
- it cannot be assumed that threads are executed in the order they are created
  - they may also execute at different speeds
- when threads are executing (racing to complete) they may give unexpected results (race condition)
- a race condition often occurs when two or more threads need to perform operations on the same memory area
  - but the results of computations depends on the order in which these operations are performed
- may occur when commands to read and write a large amount of data are received at almost the same instant
  - the machine attempts to overwrite some or all of the old data while that old data is still being read

# Deadlocks

---

- deadlocks can occur when multiple threads are trying to access a shared resource
- a deadlock is a situation in which two threads are sharing the same resource and effectively preventing each other from accessing this resource
  - causes program execution to halt indefinitely
  - each thread is “waiting” on the other thread
- traffic gridlock is an everyday example of a deadlock situation
- the dining philosophers problem is a common example of a deadlock
  - each philosopher picks up his or her left fork and waits for the right fork to become available, but it never does
- deadlocks occur when one thread “locks” a resource and never “unlocks” that resource
  - caused by poor application of “locks” or joins
  - you have to be very careful when applying two or more “locks” to a section of code

# Thread safe code

---

- because of the issues of race conditions and deadlocks a threaded function must call functions which are "thread safe"
  - code only manipulates shared data structures in a manner that ensures that all threads behave properly without unintended interaction
- "thread safe" means that there are no static or global variables (shared resources) which other threads may corrupt or clobber
  - usually any function that does not use static data or other shared resources is thread-safe
- thread-safe means that the program protects shared data
  - possibly through the use of mutual exclusion, atomic operations or condition variables
  - you want to strive to write "thread-safe" code
- thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured

# Mutual Exclusion

---

- a critical section of code is code that contains a shared resource and is accessible by multiple processes or threads
  - it is important for a thread programmer to minimize critical sections if possible
- mutual exclusion is when a process or a thread is prevented simultaneous access to a critical section
  - a property of concurrency control which is used to prevent race conditions
- mutual exclusion is the method of serializing access to shared resources
- you do not want a thread to be modifying a variable that is already in the process of being modified by another thread
- another scenario is where a value is in the process of being updated and another thread reads an old value

# Mutexes

---

- a mutex is a lock that one can virtually attach to some resource
- one of the primary means of implementing thread synchronization and for protecting shared data
  - used to prevent data inconsistencies due to race conditions
- anytime a global resource is accessed by more than one thread the resource should have a mutex associated with it
  - can apply a mutex to protect a segment of memory ("critical region") from other threads
- if a thread wishes to modify or read a value from a shared resource, the thread must first gain the lock
  - once it has the lock it may do what it wants with the shared resource without concerns of other threads accessing the shared resource because other threads will have to wait
  - once the thread finishes using the shared resource, it unlocks the mutex, which allows other threads to access the resource
- as an analogy, you can think of a mutex as a safe with only one key and the resource it is protecting lies within the safe
  - only one person can have the key to the chest at any time, therefore, is the only person allowed to look or modify the contents of the chest at the time it holds the key

# Atomic operations and Condition variables

---

- atomic operations allow for concurrent algorithms and access to certain shared data types without the use of mutexes
  - one can modify some variable within a multithreaded context without having to go through a locking protocol
- condition variables allow threads to synchronize to a value of a shared resource
  - used as a notification system between threads
- you could have a counter (flag) that once it reaches a certain count
  - a thread will activate and then wait on a condition variable
- active threads will signal on this condition variable to notify other threads waiting/sleeping on this condition variable
  - causing a waiting thread to wake up
- you can also use a broadcast mechanism if you want to signal all threads waiting on the condition variable to wakeup
- when waiting on condition variables, the wait should be inside a loop

# Next Lecture

---

- In the next lecture, I will show you how we can implement thread-safe code by utilizing concepts such as mutexes, atomic operations, and condition variables
- the threads library (pthread.h) provides functions that help with thread synchronization
- we will strive to write code that will handle race condition situations
- we will strive to write code that prevents deadlocks

---

# Mutexes

Jason Fedin

# Overview

---

- the threads library (pthread.h) provides three synchronization mechanisms that we can implement
  - mutexes - locks, block access to variables by other threads
  - joins - makes a thread wait till others are complete (terminated)
  - condition variables - a way to communicate to other threads
- we have already discussed and provided examples of joins
  - a protocol to allow the programmer to collect all relevant threads at a logical synchronization point
- so, we will focus on mutexes first and then condition variables

# Mutexes

---

- as we have discussed, mutexes are one way of synchronizing access to shared resources
- when protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so
  - if four threads are updating the same data, but only one uses a mutex, the data can still be corrupted
- very often the action performed by a thread owning a mutex is the updating of global variables
  - a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update
- a typical sequence in the use of a mutex is as follows
  - create and initialize a mutex variable
  - several threads attempt to lock the mutex
  - only one succeeds and that thread owns the mutex
  - the owner thread performs some set of actions
  - the owner unlocks the mutex
  - another thread acquires the mutex and repeats the process
  - finally the mutex is destroyed
- when several threads compete for a mutex, the losers block at that call
- please understand that a deadlock can occur when using a mutex lock
  - making sure threads acquire locks in an agreed order (i.e. preservation of lock ordering)

# Creating and destroying mutexes

---

- mutex variables must be declared with type pthread\_mutex\_t
  - they must be initialized/created before they can be used
- there are two ways to initialize/create a mutex variable

## 1. statically, when it is declared

- the below is a mutex variable named lock that is initialized to the default pthread mutex values

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

# Creating and destroying mutexes

---

## 2. dynamically, with the pthread\_mutex\_init() function

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

- this function requires a pthread\_mutex\_t variable to operate on as the first argument
- attributes for the mutex can be given through the second parameter
  - to specify default attributes, pass NULL as the second parameter
- the pthreads standard defines three optional mutex attributes
  - protocol - specifies the protocol used to prevent priority inversions for a mutex
  - prioceiling - specifies the priority ceiling of a mutex
  - process-shared - specifies the process sharing of a mutex
- the pthread\_mutex\_destroy(mutex) function should be used to free a mutex object which is no longer needed

# Locking and unlocking mutexes

---

- to perform mutex locking and unlocking, the pthreads library provides the following functions

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- used by a thread to acquire a lock on the specified mutex variable
- if the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- will attempt to lock a mutex, however, if the mutex is already locked, the routine will return immediately with a "busy" error code
- may be useful in preventing deadlock conditions, as in a priority-inversion situation.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- will unlock a mutex if called by the owning thread
- calling this function is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data
- an error will be returned if
  - If the mutex was already unlocked
  - If the mutex is owned by another thread
- anytime a global resource is accessed by more than one thread the resource should have a mutex associated with it
  - the above functions should be used to control access to a "critical section" of code from other threads
  - it is up to the programmer to ensure that the necessary threads all make the mutex lock and unlock calls correctly

# Simple Deadlock example

---

- the order of applying the mutex is also important
  - potential for deadlock

```
void *function1()
{
 ...
 pthread_mutex_lock(&lock1); - Execution step 1
 pthread_mutex_lock(&lock2); - Execution step 3 DEADLOCK!!!
 ...
 ...
 pthread_mutex_lock(&lock2);
 pthread_mutex_lock(&lock1);
 ...
}
```

---

# Condition Variables

Jason Fedin

# Condition variables

---

- condition variables provide yet another way for threads to synchronize
- while mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data
- without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if a condition is met
  - can be very resource consuming since the thread would be continuously busy in this activity
- a condition variable is a way to achieve the same goal without polling
- the condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true
  - used with the appropriate functions for waiting and later, thread continuation
- a condition variable must always be associated with a mutex
  - to avoid a deadlock created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it
  - the thread will be perpetually waiting for a signal that is never sent
- any mutex can be used with a condition variable
  - there is no explicit link between the mutex and the condition variable

# Condition Variables Example

---

- you can think of a condition variable as a "big pillow" in the sense that threads can fall asleep on a condition variable and be woken from that condition variable
- here is an analogy of how a condition variable can work
  - tommy the thread wanted to access shared information
    - he acquired the appropriate lock but then was disappointed to see that the shared information was not ready yet
    - without anything else to do, he decided to sleep (or wait) on a nearby condition variable until another thread updated the shared information and woke him up
  - timmy the thread finally came along and updated the shared information while tommy was asleep on the nearby condition variable
    - timmy noticed tommy asleep, so timmy signaled him to wake up off the condition variable
    - timmy then went on his way and left tommy to play with his new, excited shared information
- one thing that was missed in the above analogy is we know that to access the shared information a thread needs to hold a lock
  - since timmy needs to update the shared information while tommy is asleep and waiting
    - we must have any thread that sleeps on a condition variable release the lock while it is asleep
    - but since tommy had the lock when he fell asleep, he expects to still have the lock when he wakes up
    - so the condition variable semantics guarantee that a thread sleeping on a condition variable will not fully wake up until
      - (1) it receives a wake-up signal
      - (2) it can re-acquire the lock that it had when it fell asleep

# Condition variables functions

---

- condition variables must be declared with type pthread\_cond\_t, and must be initialized before they can be used

## Creating/Destroying:

pthread\_cond\_init (condition,attr) - Dynamically created

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; (Statically declared)

pthread\_cond\_destroy (condition)

## Waiting on condition:

pthread\_cond\_wait - will put the caller thread to sleep on the condition variable condition and release the mutex lock, guaranteeing that when the subsequent line is executed after the caller has woken up, the caller will hold lock

pthread\_cond\_timedwait - place limit on how long it will block

## Waking thread based on condition:

pthread\_cond\_signal

pthread\_cond\_broadcast - wake up all threads blocked by the specified condition variable

- for all of the above functions, the caller must first hold the lock that is associated with that condition variable
  - failure to do this can lead to several problems

---

# Overview

Jason Fedin

# Overview

---

- programs on different machines often need to talk to each other
- we have already learned how to use I/O to communicate with files
- we have learned how processes on the same machine can communicate with each other
- luckily, C is used to write most of the low-level networking code on the Internet
- In this section, we are going to learn how to write C programs that can talk to other programs across the network and across the world
- most networked applications need two separate programs
  - a server and a client
- the goal of this section is to be able to create programs that act as servers and programs that act as clients

# TCP protocol

---

- TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation through which application programs can exchange data
- TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other
- together, TCP and IP are the basic rules defining the Internet
  - all major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP
- TCP is a connection-oriented protocol
  - means a connection is established and maintained until the application programs at each end have finished exchanging messages
- TCP creates a connection between the source and destination node before transmitting the data and keeps the connection alive until the communication is no longer active

# Features of a TCP connection

---

- Connection Oriented
- Reliability
- Handles lost packets
- Handles packet sequencing
- Handles duplicated packets
- Full Duplex
- Flow Control
- Congestion Control

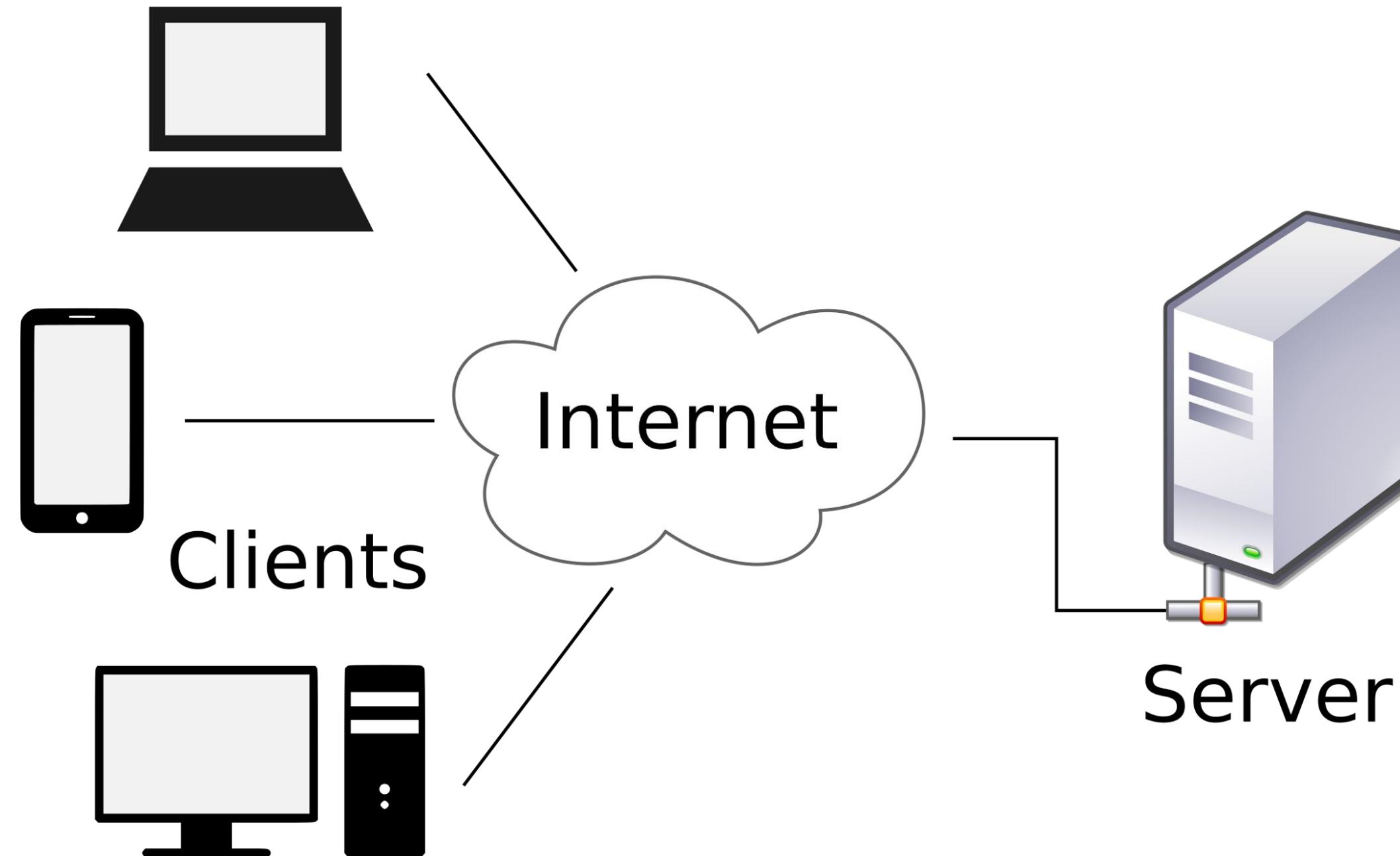
# Client/Server model

---

- the Client-Server model is a relationship in which one program (the client) requests a service or resource from another program (the server)
  - two processes or two applications that communicate with each other to exchange some information
  - one of the two processes acts as a client process, and another process acts as a server
  - there can be multiple clients that talk to one server
- clients typically communicate with servers by using the TCP/IP protocol suite
- computer transactions in which the server fulfills a request made by a client are very common
  - the client-server model has become one of the central ideas of network computing
- the client establishes a connection to the server over a local area network (LAN) or wide-area network (WAN), such as the Internet
  - clients need to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established
- once the server has fulfilled the client's request, the connection is terminated
- because multiple client programs share the services of the same server program, a special server called a daemon may be activated just to await client requests

# Client/Server model

---



[https://en.wikipedia.org/wiki/Client%E2%80%93server\\_model](https://en.wikipedia.org/wiki/Client%E2%80%93server_model)

# Client/Server model

---

- as mentioned, the client process makes a request for information and after getting the response, this process may terminate or may do some other processing
- an example of a client program would be an Internet Browser
  - sends a request to the Web Server to get one HTML webpage
- the server process is takes a request from one or more clients
- after getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client
- once done, it becomes ready to serve another client
- server processes are always alert and ready to serve incoming requests
- an example of a server process would be a Web Server
  - keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser

# Types of Servers

---

- there are two types of servers you can implement
- Iterative Server
  - the simplest form of a server where the server process serves one client at a time
  - after completing the first request, it takes request from another client
  - other client wait until it is their turn
- Concurrent Servers
  - this type of server runs multiple concurrent processes to serve many requests at a time
  - one process may take longer and another client does not need to wait too long
  - the simplest way to write a concurrent server under Unix is to fork a child process to handle each client separately

# Sockets

---

- sockets are the "virtual" endpoints of any kind of network communications done between two computers
- socket programming is a way of connecting two nodes on a network to communicate with each other
- one socket(node) listens on a particular port at an address
- another socket reaches out to the other to form a connection

# Sockets

---

- server forms the listener socket while client reaches out to the server
- when you type [www.google.com](http://www.google.com) in your web browser
  - it opens a socket and connects to google.com to fetch the page and show it to you
  - same with any chat client like gtalk or skype
- all network communication goes through a socket
- sockets are supported by Unix, Windows, Mac, and many other operating systems

# Steps in using sockets to communicate

---

- create a new socket for network communication
- actively attempt to establish a connection (connect)
- attach a local address to a socket (bind)
- send some data over connection (send)
- announce willingness to accept connections (listen)
- receive some data over connection (receive)
- block caller until a connection request arrives (accept)
- release the connection (close)

---

# The Sockets API

Jason Fedin

# Include files

---

- the following header files need to be included to work with sockets for network communication
  - include system calls for sockets and socket data)

<sys/types.h>

- contains definitions of a number of data types used in socket calls

<sys/socket.h>

- the main socket header file
- includes a number of definitions of structures needed for sockets (socket creation, accept, listen, bind, send, recv, etc)

<netinet/in.h>

- contains constants and structures needed for internet domain addresses

<netdb.h>

- defines the structure hostent

<arpa/inet.h>

defines in\_addr structure

# ports

---

- a computer might need to run several server programs at once
  - might be sending out web pages, posting email, and running a chat server all at the same time
- to prevent the different conversations from getting confused, each server uses a different port
- a port is just like a channel on a TV
  - different ports are used for different network services, just like different channels are used for different content
- a port will be defined as an integer number between 1024 and 65535
  - port numbers smaller than 1024 are considered well-known system ports (telnet uses port 23, http uses 80, ftp uses 21) (do not use these)
- the port assignments to network services can be found in the file /etc/services
- if you are writing your own server then care must be taken to assign a port to your server
  - you should make sure that this port should not be assigned to any other server

# Port and Service Functions

---

- the following functions are provided to fetch service name from the /etc/services file
- struct servent \*getservbyname(char \*name, char \*proto)
  - takes a service name and a protocol name and returns the corresponding port number for that service
- struct servent \*getservbyport(int port, char \*proto)
  - takes a port number and a protocol name and returns the corresponding service name
- the return value for each function is a pointer to a structure

```
struct servent {
 char *s_name;
 char **s_aliases;
 int s_port;
 char *s_proto;
};
```

# IP Address Functions

---

- the below functions are for IP Address functionality
- `int inet_aton (const char *strptr, struct in_addr *addrptr)`
  - converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided
- `in_addr_t inet_addr (const char *strptr)`
  - converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address
- `char *inet_ntoa (struct in_addr inaddr)`
  - converts the specified Internet host address to a string in the Internet standard dot notation

# Structures used for support of the main functions

---

- various structures are used in socket programming to hold information about the address and port, and other information
- socket address structures are an integral part of every network program, most socket functions require a pointer to a socket address structure as an argument
  - we allocate them, fill them in, and pass pointers to them to various socket functions
- sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents
- we always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument
- when a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function
- always set the structure variables to NULL, otherwise it may get unexpected junk values in your structure

# Structures used for support of the main functions

---

- sockaddr is a structure that holds the socket information
  - this is a generic socket address structure, which will be passed in most of the socket function calls

```
struct sockaddr {
 unsigned short sa_family;
 char sa_data[14];
};
```

- sa\_family can be the following, AF\_INET, AF\_UNIX, AF\_NS, AF\_IMPLINK
  - represents an address family
  - in most of the Internet-based applications, we use AF\_INET.
- sa\_data is a protocol-specific address
  - we will use port number IP address, which is represented by sockaddr\_in structure

# Structures used for support of the main functions

---

- sockaddr\_in is a structure that helps you to reference to the socket's elements

```
struct sockaddr_in {
 short int sin_family;
 unsigned short int sin_port;
 struct in_addr sin_addr;
 unsigned char sin_zero[8];
};
```

- in\_addr is used only in the above structure as a structure field and holds the 32 bit netid/hostid

```
struct in_addr {
 unsigned long s_addr;
};
```

# Structures used for support of the main functions

---

- hostent is a structure is used to keep information related to host

```
struct hostent {
 char *h_name;
 char **h_aliases;
 int h_addrtype;
 int h_length;
 char **h_addr_list
```

```
#define h_addr h_addr_list[0]
};
```

# Steps in using sockets to communicate

---

- create a new socket for network communication (socket)
- attach a local address to a socket (bind)
- announce willingness to accept connections (listen)
- block caller until a connection request arrives (accept)
- actively attempt to establish a connection (connect)
- send some data over connection (send)
- receive some data over connection (receive)
- release the connection (close)

# Sockets versus File I/O

---

- working with sockets is very similar to working with files
- the socket() and accept() functions both return handles (file descriptor)
  - reads and writes to the sockets requires the use of these handles (file descriptors)
- in Linux, sockets and file descriptors also share the same file descriptor table
- sockets have addresses associated with them whereas files do not
- you cannot randomly access a socket like you can a file with lseek()
- sockets must be in the correct state to perform input or output

# Socket Functions

---

- the most common/important functions are the following:
  - socket()
  - connect()
  - bind()
  - listen() and accept()
  - read(), recv(), recvfrom(), or recvmsg()
  - write(), send(), sendto(), or sendmsg()
  - close()
- most of the functions are used by both the client and the server with the exception of
  - bind() is used particularly by server programs
  - connect() by client programs
- all of the above functions need to include <sys/types.h> and <sys/socket.h> except for read/write/close which are defined in <unistd.h>

# socket function

---

- to perform network I/O, the first thing a process must do is call the socket function
  - specifying the type of communication protocol desired and protocol family

```
int socket (int family, int type, int protocol);
```

- family specifies the protocol family and is usually the constant AF\_INET for IPv4 protocols and AF\_INETc for IPv6 protocols
- type specifies the kind of socket you want, and this is usually set to SOCK\_STREAM for a stream socket or SOCK\_DGRAM for a datagram socket

# socket function

---

- protocol is the third and should be set to the specific protocol type you are using
  - IPPROTO\_TCP - TCP transport protocol
  - IPPROTO\_UDP - UDP transport protocol
  - IPPROTO\_SCTP- SCTP transport protocol
- returns a socket descriptor that you can use in later system calls or -1 on error
- the setsockopt function helps in manipulating options for the socket referred to by the file descriptor sockfd
  - completely optional, but it helps in reuse of address and port
  - prevents error such as: “address already in use”

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

# connect function

---

- the connect function is used by a TCP client to establish a connection with a TCP server

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- sockfd is a socket descriptor returned by the socket function
- serv\_addr is a pointer to struct sockaddr that contains destination IP address and port
- addrlen is set to sizeof(struct sockaddr)
- returns 0 if it successfully connects to the server, otherwise it returns -1 on error

# bind function

---

- when a server starts up, it needs to tell the operating system which port it is going to use
  - called binding the port
- once a server program has created a socket and named it with bind( ) giving it an IP address and port number, should any program anywhere on the network give that same name to the connect( ) function, that program will find the server program and they will link up

```
int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

- sockfd is a socket descriptor returned by the socket function
- my\_addr is a pointer to struct sockaddr that contains the local IP address and port
  - a 0 value for port number means that the system will choose a random port, and INADDR\_ANY value for IP address means the server's IP address will be assigned automatically

```
server.sin_port = 0;
server.sin_addr.s_addr = INADDR_ANY;
```

- addrlen is set to sizeof(struct sockaddr)
- returns 0 if it successfully binds to the address, otherwise it returns -1 on error

# listen function

---

- if your server becomes popular, you will probably get lots of clients connecting to it at once
  - would you like the clients to wait in a queue for a connection?
- the listen() system call tells the operating system how long you want the queue to be
- calling listen() with a queue length of 10 means that up to 10 clients can try to connect to the server at once
  - they will not all be immediately answered, but they will be able to wait
  - the 11th client will be told the server is too busy
- the listen function is called only by a server

```
int listen(int sockfd,int backlog);
```

- sockfd is a socket descriptor returned by the socket function
- backlog is the max number of allowed connections
- returns 0 on success, otherwise it returns -1 on error

# accept function

---

- once you have bound a port and set up a listen queue, you then just have to...wait
- servers spend most of their lives waiting for clients to contact them
- the accept() system call waits until a client contacts the server, and then it returns a second socket descriptor that you can use to hold a conversation on

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

- sockfd is a socket descriptor returned by the socket function
- cliaddr is a pointer to struct sockaddr that contains client IP address and port
- addrlen should be set to sizeof(struct sockaddr)
- returns a non-negative descriptor on success, otherwise it returns -1 on error
  - all read-write operations will be done on this descriptor to communicate with the client

# recv function

---

- the recv function is used to receive data over stream sockets or connected datagram sockets
  - if you want to receive data over unconnected datagram sockets you must use recvfrom()

int recv(int sockfd, void \*buf, int len, unsigned int flags);

- sockfd is a socket descriptor returned by the socket function
- buf is the buffer to read the information into
- len is the maximum length of the buffer
- flags is set to 0
- returns the number of bytes read into the buffer, otherwise it will return -1 on error

# recvfrom function

---

- the recvfrom function is used to receive data from unconnected datagram sockets

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);
```

- sockfd is a socket descriptor returned by the socket function
- buf is the buffer to read the information into
- len is the maximum length of the buffer
- flags is set to 0
- from is a pointer to struct sockaddr for the host where data has to be read
- fromlen should be set it to sizeof(struct sockaddr)
- returns the number of bytes read into the buffer, otherwise it returns -1 on error

# write function

---

- the write function attempts to write nbytes bytes from the buffer pointed by buf to the file associated with the open file descriptor, fildes

```
int write(int fildes, const void *buf, int nbytes);
```

- fildes is a socket descriptor returned by the socket function
- buf is a pointer to the data you want to send
- nbytes is the number of bytes to be written
  - If nbytes is 0, write() will return 0 and have no other results if the file is a regular file
  - otherwise, the results are unspecified
- returns the number of bytes actually written to the file associated with fildes if successful otherwise, -1 is returned

# Send function

---

- the send function is used to send data over stream sockets or connected datagram sockets
  - if you want to send data over unconnected datagram sockets, you must use sendto() function

```
int send(int sockfd, const void *msg, int len, int flags);
```

- sockfd is a socket descriptor returned by the socket function
- msg is a pointer to the data you want to send
- len is the length of the data you want to send (in bytes)
- flags should be set to 0
- returns the number of bytes sent out, otherwise it will return -1 on error

# sendto function

---

- the sendto function is used to send data over unconnected datagram sockets

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
```

- sockfd is a socket descriptor returned by the socket function
- msg is a pointer to the data you want to send
- len is the length of the data you want to send (in bytes)
- flags should be set to 0
- to is a pointer to struct sockaddr for the host where data has to be sent
- tolen should be set it to sizeof(struct sockaddr)
- returns the number of bytes sent, otherwise it returns -1 on error

# read function

---

- the read function attempts to read nbytes bytes from the file associated with the buffer, fildes, into the buffer pointed to by buf

```
int read(int fildes, const void *buf, int nbytes);
```

- fildes is a socket descriptor returned by the socket function
- buf is the buffer to read the information into
- nbytes is the number of bytes to read
- returns the number of bytes actually written to the file associated with fildes, if successful
  - otherwise, -1 is returned

# close function

---

- the close function is used to close the communication between the client and the server

```
int close(int sockfd);
```

- sockfd is a socket descriptor returned by the socket function
- returns 0 on success, otherwise it returns -1 on error

# shutdown function

---

- the shutdown function is used to gracefully close the communication between the client and the server
  - gives more control in comparison to the close function

`int shutdown(int sockfd, int how);`

- sockfd is a socket descriptor returned by the socket function
- How
  - 0 – indicates that receiving is not allowed
  - 1 – indicates that sending is not allowed
  - 2 – indicates that both sending and receiving are not allowed
    - when how is set to 2, it's the same thing as close()
- returns 0 on success, otherwise it returns -1 on error

---

# Server Socket

Jason Fedin

# Steps in using sockets to communicate

---

- create a new socket for network communication
- attach a local address to a socket (bind)
- announce willingness to accept connections (listen)
- block caller until a connection request arrives (accept)
- actively attempt to establish a connection (connect)
- send some data over connection (send)
- receive some data over connection (receive)
- release the connection (close)

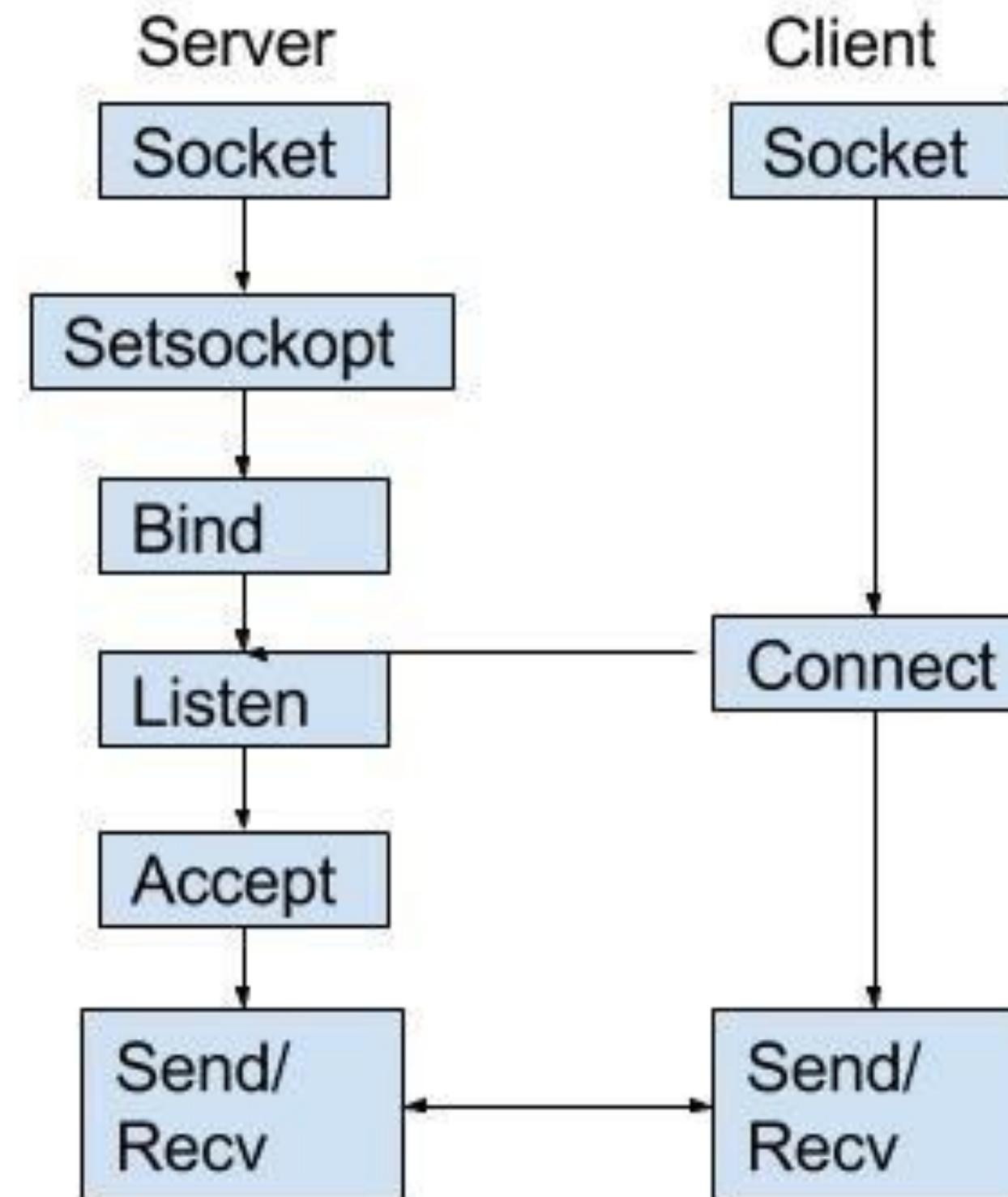
# Steps specifically for a server socket

---

- the steps involved in establishing a socket on the server side are as follows
- create a socket with the `socket()` system call
- bind the socket to an address using the `bind()` system call
  - for a server socket on the Internet, an address consists of a port number on the host machine
- listen for connections with the `listen()` system call
- accept a connection with the `accept()` system call
  - typically blocks the connection until a client connects with the server
- send and receive data using the `read()/recv` and `write()/send` system calls

# Illustration

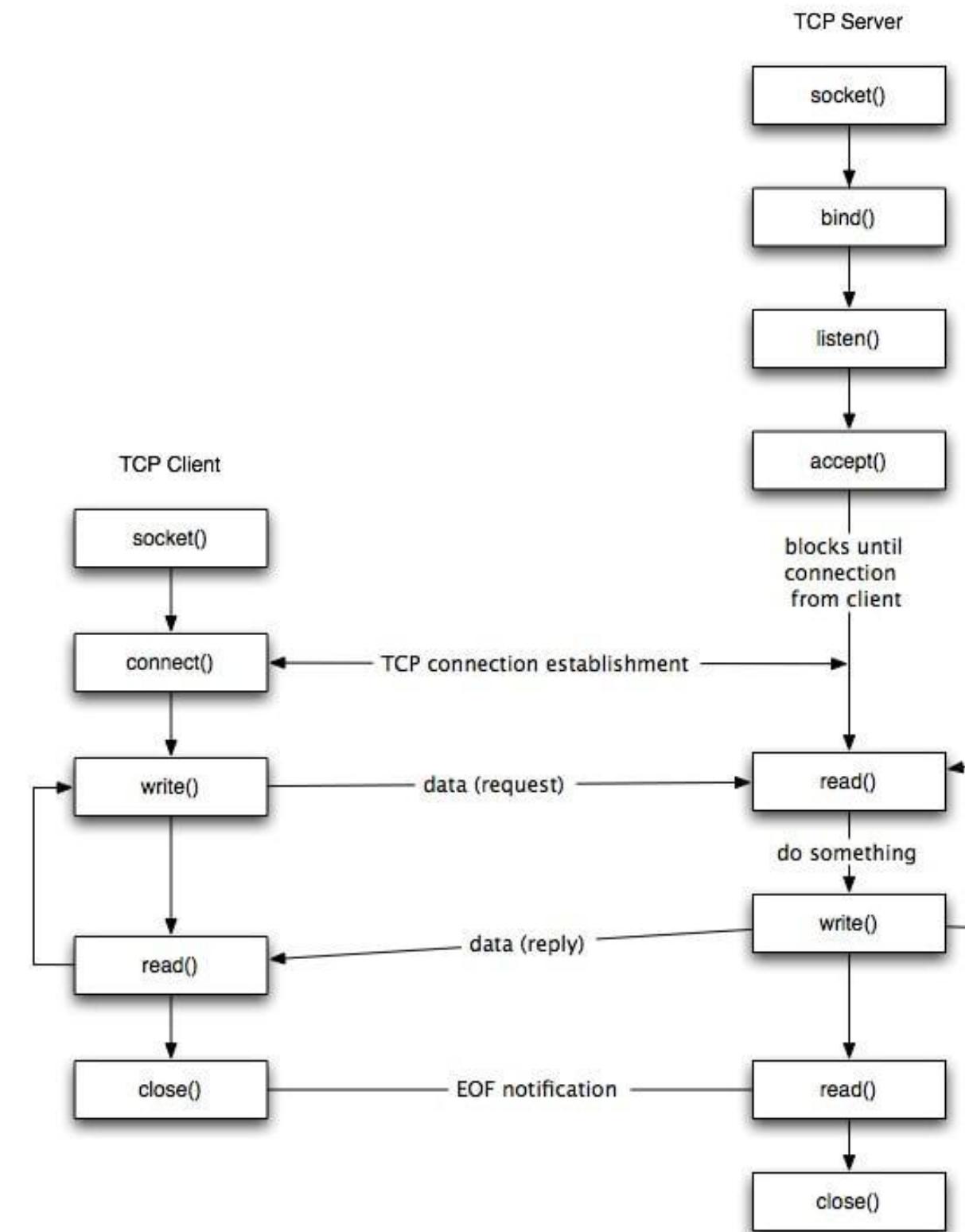
---



<https://www.thecrazyprogrammer.com/2017/06/socket-programming.html>

# Another

<https://aticleworld.com/socket-programming-in-c-using-tcip/>



# simple server steps (pseudocode)

---

my\_sd = socket( )

bind( my\_sd, <local address, mainly a port number> )

listen( my\_sd )

start loop

    his\_sd = accept( my\_sd, <empty address to be filled in with his incoming info> )

    recv( his\_sd, <where to put what you receive> )

    send( his\_sd, <the stuff you want sent> )

    close( my\_sd )

end loop

---

# Client Socket

Jason Fedin

# Steps in using sockets to communicate

---

- create a new socket for network communication
- attach a local address to a socket (bind)
- announce willingness to accept connections (listen)
- block caller until a connection request arrives (accept)
- actively attempt to establish a connection (connect)
- send some data over connection (send)
- receive some data over connection (receive)
- release the connection (close)

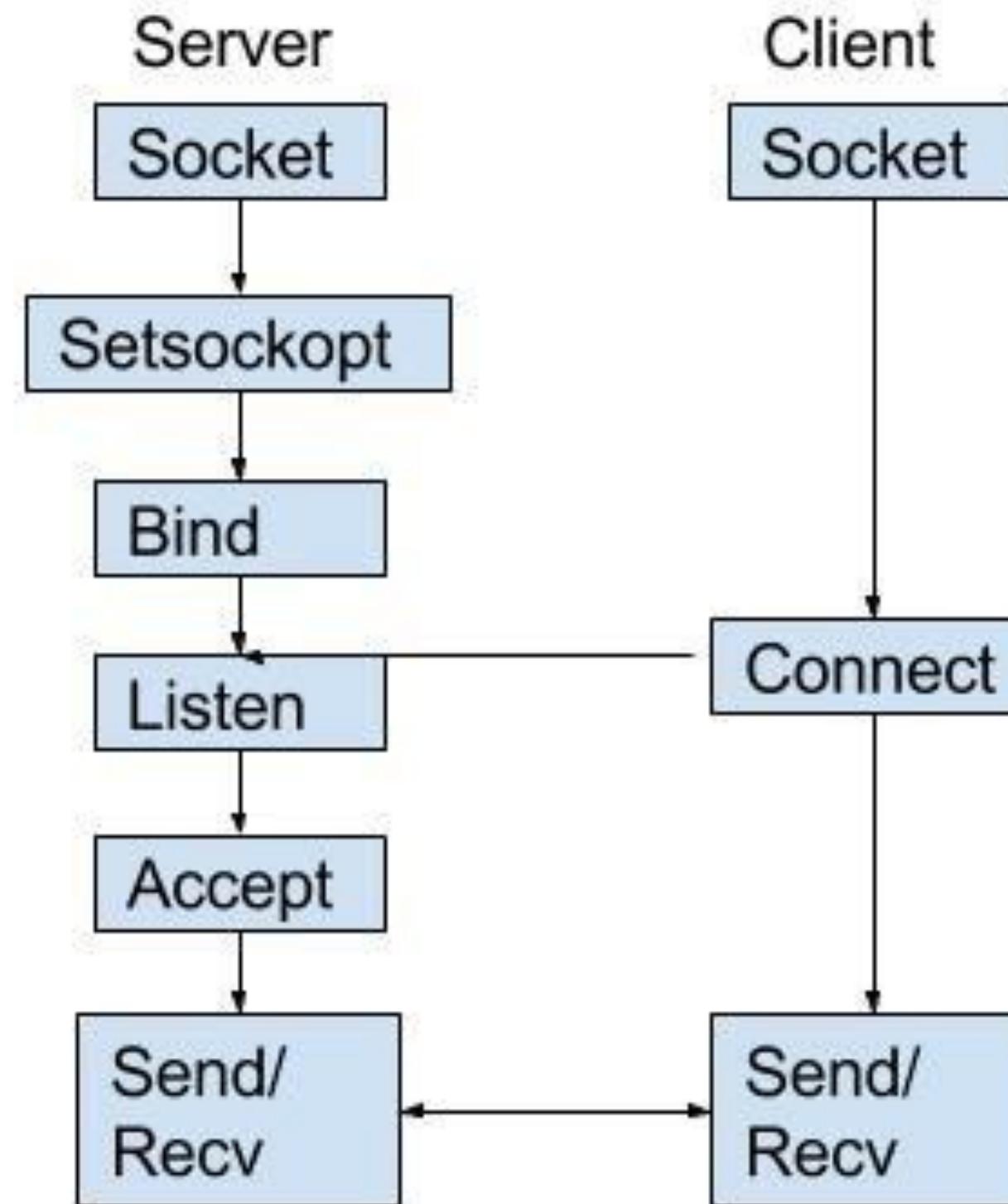
# Steps specifically for a server socket

---

- create a socket with the `socket()` system call
- connect the socket to the address of the server using the `connect()` system call
- send and receive data
  - there are a number of ways to do this, but the simplest way is to use the `read()/recv` and `write()/send` system calls

# Illustration

---

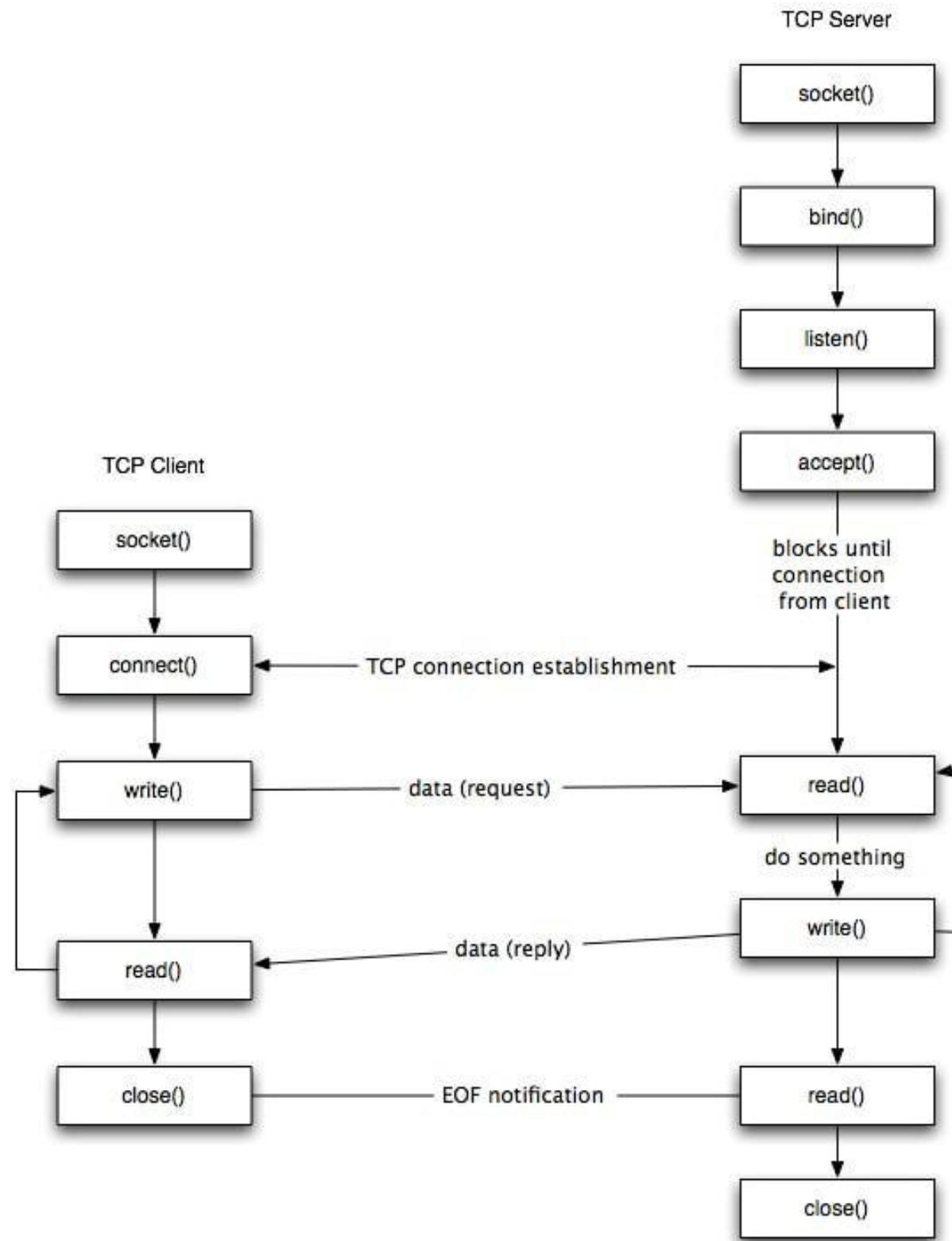


<https://www.thecrazyprogrammer.com/2017/06/socket-programming.html>

---

# Another

<https://aticleworld.com/socket-programming-in-c-using-tcpip/>



# simple client steps (pseudocode)

---

```
my_sd = socket()
```

```
his_sd = connect(my_sd, <presumed address of some
server>)
```

```
send(his_sd, <the stuff you want sent>)
recv(his_sd, <where to put what you receive>)
```

```
close(my_sd)
```

---

# Course Summary

Jason Fedin

# Topics

---

- storage classes
  - auto, register, static, and extern
- working with larger programs
  - dividing your program into multiple files and compiling multiple files
- advanced data types
  - typedef, variable length arrays, flexible array numbers, complex number types
- type qualifiers
  - const, volatile, and restrict
- bit manipulation
  - binary numbers and bits
  - bitwise operators (logical and shifting)
  - bitmasks and bitfields

# Topics (cont'd)

---

- Advanced control flow
  - goto, null, comma operator
  - setjmp and longjmp
- more on Input and Output
  - getchar, putchar, fgets, etc.
  - puts, sprint, fprintf, fflush
- advanced function concepts
  - variadic functions (variable number of arguments)
  - recursive functions
  - inline functions

# Topics (cont'd)

---

- unions
  - overview, defining and accessing union members
- advanced preprocessor concepts
  - #define, #pragma, #error, #, ##
  - conditional compilation (#ifdef, #endif, #else, #elif, #undef, etc)
  - include guards
- macros
  - overview (vs. functions, when to use)
  - predefined macros
  - creating your own macros

# Topics (cont'd)

---

- advanced debugging and compiler flags
  - debugging with the pre-processor, more on gdb
  - core files, getting the stack trace
  - static analysis and profiling
- advanced pointers
  - double pointers (pointers to pointers)
  - function pointers
  - more on void pointers
- static libraries and shared objects
  - overview, creation, dynamic loading

# Topics (cont'd)

---

- useful C libraries
  - the assert library
  - general utilities library (stdlib.h), (exit, atexit, qsort, memcpy, abort)
  - date and time functions
- data structures
  - Linked lists, stacks, queues, and Trees
- Inter-process communication (unix based using Cygwin)
  - overview (message queues, shared memory, piping)
  - forking and signals

# Topics (cont'd)

---

- threads (pthread (posix), not <threads.h> from C11)
  - overview, creating a thread
  - mutexes and semaphores
  - thread management (multi-threading, join, detach)
- networking (unix based using Cygwin)
  - overview (client/server model)
  - creating server and client sockets
- many challenges, solutions, and examples
- organized around theory and many demonstrations
- hands on coding

# Course Outcomes

---

- you will be able to write advanced C programs
- you will be able to write efficient, high quality C code
  - modular
  - low coupling
- master the art of problem solving in programming using efficient, proven methods
- you will understand advanced concepts of the C Programming language
- you will have fun!

# CONGRATULATIONS!!!!

---

# CONGRATULATIONS!!!!

- do not hesitate to ask questions
- provide feedback via the ratings
- offer constructive criticism
  - always looking to improve the course
- will make course updates to improve the class

# Advanced C Programming Slides

---

# Challenge Slides.

---

# (Challenge) Variable Length Arrays

Jason Fedin

# Challenge #1 (variable length arrays)

---

- our challenge is to write a program to read elements in an array and find the sum of array elements

Example

Input elements: 10, 20, 30, 40, 50

Output

Sum of all elements = 150

- you must use a variable length array for the size of your array
- the obvious benefit of allowing variable-length arrays, is that you don't have to know in advance of your program running, how much memory to allocate for your array variables

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Flexible Array Members

Jason Fedin

# Challenge #1 (flexible array members)

---

- our challenge is to write a program that uses a flexible array member inside a structure
- create a structure named myArray that has a length member and a flexible array member named array
- read in from the user the size of the array at runtime
- allocate memory for the structure based on this size read in from the user
- set the length member and fill the array with some dummy data
- print the array elements

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Complex Number Types

Jason Fedin

# Challenge #1 (Complex Numbers)

---

- our third challenge is to write a program that performs some calculations on complex numbers
- create a double complex number with the imaginary number squared using the multiplication operator (`I * I`)
  - display as output the real number and the imaginary number
- create a double complex number with the imaginary number squared using the pow function from the math library (`pow (I, 2)`)
  - display as output the real number and the imaginary number

# Challenge #1 (Complex Numbers)

---

- create a double complex number that performs Euler's formula
  - $\text{PI} = \text{acos}(-1)$
  - the complex number is the exponent of  $I * \text{PI}$
  - display as output the real number and the imaginary number
- create a double complex number that performs a conjugate
  - a complex number that is  $1+2*I$
  - a complex number that is  $1-2*I$
  - display as output the real number and the imaginary number
- Use the complex.h header file and the creal() and cimag() functions

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Binary Numbers and Bits

Jason Fedin

# Challenge #1 (converting binary to decimal)

---

- your first challenge will test your understanding of converting binary numbers to decimal
- write a function that converts a binary number to a numeric value
- if you have

```
long long bin = 01001001;
```

- you can pass bin as an argument to the function and have the function return an int value of 25

# Challenge #2 (converting decimal to binary)

---

- your second challenge will test your understanding of converting decimal numbers to binary
- write a function that converts an integer to a binary value (long long)
  - long long convertDecimalToBinary(int n)
- if you have

```
int decimal = 10;
```

- you can pass decimal as an argument to the function and have the function return a long long value of 1010

# Summary

---

- review the demonstrations and solutions provided for further understanding of all challenges

---

# (Challenge) Bitwise Operators

Jason Fedin

# Challenge (bitwise operators)

---

- your challenge will test your knowledge of the bitwise operators
- write a program that reads two integers from the user
- your program will print the results of applying
  - the `~` operator to each number
  - the `&`, `|`, and `^` operators to the pair
  - the `<<` and `>>` operations to a specific number
  - show the results as binary strings (need to write a `decimalToBinary` function)
- Example run

Enter an integer: 10

Enter another integer: 11

The result of applying the `~` operator on number 10 (1010) is: -1011

The result of applying the `~` operator on number 11 (1011) is: -1100

The result of applying the `&` operator on number 10 (1010) and number 11 (1011) is: 1010

The result of applying the `|` operator on number 10 (1010) and number 11 (1011) is: 1011

The result of applying the `^` operator on number 10 (1010) and number 11 (1011) is: 1

The results of applying the left shift operator `<<` on number 10 (1010) by 2 places is number (101000) is: 40

# Summary

---

- review the demonstration and solution provided for further understanding of all challenges

---

# (Challenge) Setting and Reading bits

Jason Fedin

# Challenge (setting and reading bits)

---

- this challenge will test your knowledge of setting and reading bits
- write a C program to input any number from a user
  - the program should check whether nth bit of the given number is set (1) or not
  - the program should set nth bit of the given number as 1

Enter any number: 10

Enter nth bit to check and set (0-31): 2

The 2 bit is set to 0

Bit set successfully.

Number before setting 2 bit: 10 (in decimal)

Number after setting 2 bit: 14 (in decimal)

# Summary

---

- review the demonstration and solution provided for further understanding of all challenges

---

# (Challenge) Using Bit Fields to pack data

Jason Fedin

# Challenge (bit fields)

---

- this challenge will test your understanding of bitfields
- write a program that contains the following bit fields in a structure (onscreen box)
  - the box can opaque or transparent
  - a fill color is selected from the following palette of colors: black, red, green, yellow, blue, magenta, cyan, or white
  - a border can be shown or hidden
  - a border color is selected from the same palette used for the fill color
  - a border can use one of three line styles—solid, dotted, or dashed
- you need only a single bit to indicate whether the box is opaque or transparent
- you need only a single bit to indicate if the border is shown or hidden
- the eight possible color values can be represented by the eight possible values of a 3-bit unit
- a 2-bit unit is more than enough to represent the three possible border styles
- a total of 10 bits, then, is enough to represent the possible settings for all five properties

# Challenge (bit fields) (cont'd)

---

- use padding to place the fill-related information in one byte
- the border-related information in a second byte
- the padding brings the structure up to 16 bits
  - without padding, the structure would be 10 bits

# Challenge (bit fields)

---

- you can use a value of 1 for the opaque member to indicate that the box is opaque and a 0 value to indicate transparency
- you can do the same for a “show\_border” member
- for colors, you can use a simple RGB (red-green-blue) representation
  - these are the primary colors for mixing light
  - the usual order is for the left bit to represent blue intensity
  - the middle bit green intensity
  - the right bit red intensity

# Challenge (bit fields)

---

| Bit Pattern | Decimal | Color   |
|-------------|---------|---------|
| 000         | 0       | Black   |
| 001         | 1       | Red     |
| 010         | 2       | Green   |
| 011         | 3       | Yellow  |
| 100         | 4       | Blue    |
| 101         | 5       | Magenta |
| 110         | 6       | Cyan    |
| 111         | 7       | White   |

- can be used as values for a “fill\_color” and “border\_color” struct members
- lastly, you can choose to let 0, 1, and 2 represent the solid, dotted, and dashed styles
  - can be used as values for a “border\_style” member

# Challenge (bit fields)

---

- your program should utilize the structure that contains the bitfields as described in the previous slides
- first, create a variable of the structure type and initialize the bitfields with some default values
  - display these default values
- second, modify the structures contents by accessing each bitfield and changing the default value to another value
  - display these modified values

# Challenge (bit fields) (example output)

---

Original box settings:

Box is opaque.

The fill color is yellow.

Border shown.

The border color is green.

The border style is dashed.

Modified box settings:

Box is transparent.

The fill color is white.

Border shown.

The border color is magenta.

The border style is solid.

# Summary

---

- feel free to convert this challenge to use unsigned integer variables, bitwise operators, and bitmasks to store/read/write the bits as opposed to bitfields in a struct
  - how much harder is it???
- review the demonstration and solution provided for further understanding of all challenges

---

# (Challenge)

# The goto statement

Jason Fedin

# Challenge (using goto)

---

- our challenge is to write a program that uses goto statements to simulate a nested looping structure that prints a triangle of asterisks
- the output should resemble something like the below:

```
*
* *
* *
* *

```

- the program should not utilize any looping constructs and instead only use the goto statement
- after writing the program, you should come to the conclusion that using a goto statement is terrible and your code looks like spaghetti code

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) setjmp and longjmp functions

Jason Fedin

# Challenge (setjmp and longjmp)

---

- our challenge is to write a C program that uses setjmp and longjmp to gracefully handle unrecoverable program errors
  - when you discover an unrecoverable error, you should transfer control back to the main input loop and start again from there
- create a function named error\_recovery that prints out an error and then uses longjmp to transfer control back to a main function loop
- your main function should include a forever loop that uses setjmp at the top of the loop before processing
- you can add “dummy” code in your loop that simulates an error (by calling error\_recover) when setjmp returns 0

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) char I/O Functions

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of the common char I/O functions
  - you must choose the correct function to use based on the requirements (sometimes, there are multiple options)
- write a program in C to count the number of words and characters in a file OR from standard input
- this program can take zero command-line arguments or one command-line argument
  - If there is one argument, it is interpreted as the name of a file
  - If there is no argument, the standard input (stdin) is to be used for input

## Example input (from file)

hello, how are you

jason

nope

ok

what

lest

aaaaa

# Challenge #1

---

```
// invoked using the file content
.a.out test.txt
```

The number of words in the file are : 10  
The number of characters in the file are : 39

```
// invoked using standard input
.a.out
```

what  
nope  
ok

The number of words in the file are : 3  
The number of characters in the file are : 10

# Challenge #2

---

- this challenge will test your understanding of the fgetc and fputc functions
- write a c program to convert uppercase to lowercase and vice versa in file
- this program can take a command-line argument for the name of the file or you can ask the user for the name of the file
- you will need to create a temporary file to store the result (converted character)
  - can then rename this temporary file back to the original file
- you will need to use the isupper(ch) to check for upper or lower case characters (can also use tolower and toupper)

# Challenge #2

---

## Example input (from file)

hello, how are you

jason

nope

ok

what

lest

aaaaa

## Example output

HELLO, HOW ARE YOU

JASON

NOPE

OK

WHAT

LEST

AAAAA

---

# (Challenge) String I/O Functions

Jason Fedin

# Challenge

---

- this challenge will test your understanding of the common String I/O functions
  - you must choose the correct function to use based on the requirements (sometimes, there are multiple options)
- write a program that takes two command-line arguments
  - the first is a character
  - the second is a filename
- the requirements of the program are that it should print only those lines in the file containing the given character
  - lines in a file are identified by a terminating '\n'
  - assume that no line is more than 256 characters long
- you are required to use fgets() or getline() (or try both) for reading of the file
  - use puts to display the output

# Challenge example input and output

hello, how are you

jason

nope

ok

what

lest

Aaaaa

Example run

./a.out j test.out

jason

# Summary

---

- review the demonstration and solution provided for further understanding

---

# Challenge (formatting functions)

Jason Fedin

# Challenge

---

- this challenge will test your understanding of the common formatting I/O functions
  - you must choose the correct function to use based on the requirements (sometimes, there are multiple options)
- write a program that takes as input, a set of numbers from a file and write even, odd and prime numbers to standard output
- you can use either fscanf or (fgets and sscanf) to accomplish the above
- I have provided a sample input file (numbers.txt) that you can test with

# Challenge example input from file

---

```
73771782 81296771 79982326 75332246 10128193
81643413 76259734 94432076 50063976 91748657
42311916 -192004290747362 53851612 43498487
73193311 96685173 39019033 8630045 59322952
```

# Challenge example output

---

File opened successfully. Reading integers from file.

Even number found: 73771782

Odd number found: 81296771

Even number found: 79982326

Even number found: 75332246

Prime number found: 10128193

Odd number found: 81643413

Even number found: 76259734

Even number found: 94432076

Even number found: 50063976

Odd number found: 91748657

Even number found: 42311916

Even number found: -1920042

Even number found: 90747362

Even number found: 53851612

Odd number found: 43498487

Prime number found: 73193311

Odd number found: 96685173

Prime number found: 39019033

Odd number found: 8630045

Even number found: 59322952

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Variadic functions

Jason Fedin

# Challenge

---

- this challenge will test your understanding of variadic functions (functions that take a variable number of arguments)
- write a program that creates a variadic function that will allow a programmer to add any amount of numbers (integers) that they would like to be added together
- in order to know how many numbers are being passed to this variadic function, you can use the first argument as the number of arguments

# Example run and output

---

- invoking the function with a variable number of arguments
  - addingNumbers( 2, 10, 20 )
  - addingNumbers( 3, 10, 20, 30 )
  - addingNumbers( 4, 10, 20, 30, 40 )
- Output
  - 30
  - 60
  - 100

# Additional Challenges

---

- feel free to add requirements to this challenge so that instead of using integers to add, you use doubles as the data type and calculate the sum
  - you can also change the function prototype to include two fixed doubles as two fixed arguments and require the user to enter 0.0 as the last argument in their variable list (to indicate the end of the list)
- you can also add to the program, functions that calculate average value and sum of unknown number of numbers, max or min of a list of numbers
- you can also create your own printf function (takes a variable number of arguments)
  - you would need a character to specify the format
    - you can use '%', as it is implemented in printf and scanf or some other character

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Recursion

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of recursion and specifically, recursive functions
- write a program which will calculate the sum of numbers from 1 to n using recursion

## Sample Input/Output:

Input the last number of the range starting from 1 : 5

The sum of numbers from 1 to 5 :

15

# Challenge #2

---

- this challenge will test your understanding of recursion and specifically, recursive functions
- write a program which will find GCD (greatest common denominator) of two numbers using recursion

## Sample Input/Output:

Input 1st number: 10

Input 2nd number: 50

The GCD of 10 and 50 is: 10

# Challenge #3

---

- this challenge will test your understanding of recursion and specifically, recursive functions
- write a program which will find reverse a string using recursion

## Sample Input/Output:

Enter the string: studytonight

The original string is: studytonight  
The reverse string is: thginotyduts

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge) Unions

Jason Fedin

# Challenge

---

- this challenge will test your understanding of unions in c programming
  - how to define/declare and how to access assign
- write a program which will define a union and then use this union to assign and access its members
- you must define a union named student that consists of the following three elements
  - char letterGrade
  - int roundedGrade
  - float exactGrade
- your program should declare two union variables inside the main method (variable 1 and variable 2)
- your program should assign random values to variable 1
  - You then need to display each value for each member of this union
  - You should notice that something is wrong

# Challenge

---

- your program should then assign a value (using variable 2) to its member letter grade and then print it out
- your program should then assign a value (using variable 2) to its to member rounded grade and then print it out
- your program should then assign a value (using variable 2) to its to member exact grade and then print it out
- you should notice the difference in output with variable 1 and variable 2
  - why is there a difference?

# Example output

---

Union record1 values example

Letter Grade :

Rounded Grade : 1118633984

Exact Grade : 86.500000

Union record2 values example

Letter Grade : A

Rounded Grade : 100

Exact Grade: 99.500000

# Optional Additions

---

- Declare the union variables by using the option tag in the declaration
- Use pointers as union variables and practice assigning/accessing members
- Display the total size of the union in bytes
- Initialize a union member by using a designated initializer
- Add a union inside a structure

# Summary

---

- review the demonstration and solution provided for further understanding

---

# Macros (Challenge)

Jason Fedin

# Challenge #1

---

- The following challenges will test your understanding of macros
- Write a program to print the values of the following predefined symbolic constants
  - `_LINE_`
  - `_FILE_`
  - `_DATE_`
  - `_TIME_`
  - `_STDC_`
- Example output

`_LINE_ = 34`

`_FILE_ = fedin.c`

`_DATE_ = Jul 16 2019`

`_TIME_ = 11:12:23`

`_STDC_ = 1`

# Challenge #2

---

- Write a program that defines a macro that accepts two parameters and returns the sum of the given numbers

```
#define MACRO_NAME(params) MACRO_BODY
```

- MACRO\_NAME should be SUM
- params are the parameters passed to macro
- MACRO\_BODY is the code for the actual logic of the macro
- Your program should have the user enter the two numbers
- Your program should then display the sum as output by invoking the above macro

# Challenge #3

---

- Write a C program to find the square and cube of a number using macros
- You should create two macros
  - a SQUARE macro
  - a CUBE macro
- You need to figure out how many parameters there should be
- Your program should have the user enter any number
- Your program should then display the square and cube of the number as output by invoking the above macros

## Example output

Enter any number to find square and cube: 10

SQUARE(10) = 100

CUBE(10) = 1000

# Challenge #4

---

- Write a C program to check whether a character is uppercase or lowercase using macros
- You should create two macros that accept a single argument (character)
  - IS\_UPPER
  - IS\_LOWER
  - both of the macros should return a boolean
    - true (1) or false (0) based on whether they are upper or lower case
- Your macro will need to include a conditional and can use logical operators
- Your program should have the user enter any character ( getchar() )
- Your program should then display whether the character is upper or lower case as output by invoking the above macros

## Example output

Enter any character: C

'C' is uppercase

Enter any character: 8

Entered character is not in the alphabet

# Challenge #4 additional enhancements

---

- For challenge #4, feel free to add the below macros to perform additional checks on characters
  - to further your learning
- logic to check alphabets, digits, alphanumeric, vowels, consonants, special characters, white space etc.
  - some of the above macros can use the other macros
    - an IS\_ALPHANUMERIC macro could use an IS\_ALPHABET and IS\_DIGIT macros

# Summary

---

- review the demonstration and solution provided for further understanding

---

# Challenge

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of compiling from the command line
- write a hello world program
  - Code that contains a single print statement in your main method
- manually compile and run your code
  - Compile many times with multiple options (as described in a lecture in this section)
    - try changing the object name
    - Add debugging information
    - Add warnings
    - Display warnings as errors
- also, go into code::blocks or whatever IDE you are using and also change the build options in there

# Challenge #2, #3, and #4

---

- the next few challenges will test your understanding of debugging
  - you will need to use the preprocessor directives to debug
  - you will need to use gdb to debug
  - you will need to use gdb to analyze a core file

# Overview (challenge #2)

---

- you need to modify main.c to include debugging statements in main() such as
  - `fprintf(stderr, "Number of parameters = %d\n", argc);`
- You need to modify main.c to include debugging statements in sum() such as
  - `fprintf(stderr, "x=%d\n", x);`
  - `fprintf(stderr, "y=%d\n", y);`
  - `fprintf(stderr, "z=%d\n", z);`
  - `fprintf(stderr, "*a=%d\n", *a);`
- now recompile and run the program again
  - you should see some of the output from above (not everything in sum, depending on where you placed your print statements)

# Overview (challenge #2)

---

- add an additional statement immediately after the last one that successfully printed (in the sum function)

```
fprintf(stderr, "a=%ld\n", (long)a);
```

- this should help to explain why the last printf() failed to print
- try running the program with only one parameter, and see how it fails
  - add some more debugging statements to your main() to find exactly where the line is that is causing the problem
- when using printf/fprintf, always end the format string with a \n
  - will ensure that your debugging message is printed to the screen, and not just buffered, waiting for additional characters to be sent
- in the work above, we never actually fixed the problems with main.c
  - this is intentional, so that we could continue to use for your challenge

# Overview (challenge #2)

---

- so that should help make sure that you can use printf() to generate debugging output to identify the problem line when your program crashes
  - what about when it is not crashing?
- lots of debugging output can be annoying
- Lets try adding preprocessor directives
  - allow you to turn on and off debugging code

# challenge #2

---

- this challenge will test your understanding of debugging using conditional compilation
  - you will need to use the preprocessor directives to debug
  - allows you to turn on and off debugging code
- copy the source file in the resources section that has the added debug fprintf statements (main.c)
- modify main.c to use preprocessor directives to easily turn on or off the debugging output on the basis of whether a DEBUG symbol is defined
- manually compile the code to include the -D DEBUG compiler option
  - run the program with 1 parameter or 3
  - you will see your debug output
- manually compile the code without the -D DEBUG compiler option
  - Run the program with 1 parameter or 3
  - You will NOT see your debug output

# challenge #2

---

- optional additions to challenge #2
- create a macro to make the program more readable
  - replace the fprintf statements with your macro
- add various debug levels
  - you can expand on the notion of the DEBUG macro a little further to allow for both compile-time and execution-time debugging control
    - declare a global variable Debug that defines a debugging level
    - all DEBUG statements less than or equal to this level produce output
    - DEBUG now takes at least two arguments; the first is the level
      - or you could pass this level argument as an additional argument to your program

# Challenge #3 (Demo)

---

- debugging statements are simple, and work in all environments
  - but debugging that way is often quite tedious, as you must edit and re-compile many times until the right statements are in place to capture the effect that you need to understand
- from the lecture in this section, we know that a debugger provides an interactive way to watch the execution of your program
  - you can stop the program at any point, examine the contents of variables, look at the calling stack, check function parameters, and more
- for our third challenge, we will use gdb to debug our program (main.c)
  - will need to compile with debugging information (-g)

# Challenge #3

---

- this challenge will test your understanding of using the gdb debugger
- use the source file from your previous challenge (main.c) and compile with the -g option and turn debugging on
- run your program within the GNU debugger and start it  
gdb main  
(gdb) run 1 2 3

# Challenge #3

---

- you will notice that your program has received a SGSEGV, Segmentation fault
  - it shows the function in which the error occurred---sum()

Starting program: main 1 2 3

Number of parameters = 4

i=1

j=2

k=3

x=1

y=2

z=12

a=140737347995280

Program received signal SIGSEGV, Segmentation fault.

0x00005555555482c in sum ()

# Challenge #3

---

- your challenge is to now “play” around with the debugger to figure out the problem
  - use the print command to display the value of variables
  - show the call stack
  - set breakpoints
  - delete breakpoints
  - list source code
  - Use help
- become experienced with using gdb commands

# Challenge #4 (DEMO, debugging with core dumps)

---

- when a program fails as a result of a segmentation fault, or a bus error, or from calling abort(), the OS will attempt to store a copy of the running process image (e.g., exactly what was in memory at the time) to disk
  - called dumping core
- to see how large a core file you are permitted to generate, type ulimit -a
  - a limit of 0 will prevent the generation of core files at all
  - I suggest that you set it (typically in a file called .profile) to a unlimited (e.g., by commenting out the limit that exists there)
  - If your core file size is unlimited, then you can limit it if you want from the command line
- may have to type in “bash” to make sure you are in a shell
  - Otherwise command may not be recognized
- type ulimit -c unlimited to enable core files

# Challenge #4 (debugging with core dumps)

---

- this challenge will test your understanding of using gdb to analyze core files
- use the source file from your previous challenge and compile with the -g option and turn debugging on
- run the program to generate the core file
  - should be in the same directory, named core ....
- use gdb to attach to the core file
- run the backtrace to analyze the core file
  - captures the state of the program in which the error occurs
  - tells you the exact line of the crash of the program

# Optional Challenge

---

- try running the program using valgrind or gprof
  - play around with these tools to see how they can be valuable
- even with a debugger, we might not be able to catch all bugs
- static analyzers and profiling tools can often help with mis-managed memory

---

# Challenge

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of double pointers
    - how to declare a pointer to pointer
    - how to initialize a pointer to a pointer
    - how to access a pointer to a pointer
  - write a program that creates, assigns, and accesses some double pointers
1. create a normal integer variable (non pointer) and assign it a random value
  2. create a single integer pointer variable
  3. create a double integer pointer variable
  4. assign the address of the normal integer variable (step 1) to the single pointer (step 2)
  5. assign the address of the single pointer (step 2) to the double pointer variable (step 3)

# Challenge #1

---

- display the following output using all possible syntax
  - all possible ways to find value of the normal integer variable (step 1)
  - all possible ways to find address of the normal integer variable (step 1)
  - all possible ways to find the value of the single pointer variable (step 2)
  - all the possible ways to find the address of the single pointer variable (step 2)
  - all possible ways to print the double pointer value and address (step 3)

# Challenge #1 Example Output

---

Value of num is: 123

Value of num using singlePointer is: 123

Value of num using doublePointer is: 123

Address of num is: 0xffffcbcc

Address of num using singlePointer is: 0xffffcbcc

Address of num using doublePointer is: 0xffffcbcc

Value of Pointer singlePointer is: 0xffffcbcc

Value of Pointer singlePointer using doublePointer is: 0xffffcbcc

Address of Pointer singlePointer is: 0xffffcbc0

Address of Pointer singlePointer using doublePointer is: 0xffffcbc0

Value of Pointer doublePointer is: 0xffffcbc0

Address of Pointer doublePointer is: 0xffffcbb8

# Challenge #2

---

- this challenge will test your understanding of double pointers used as arguments to a function
- write a program that includes a function that modifies a pointers value
  - not the value that the pointer is pointing to
  - the actual value of the pointer (the address that the pointer is pointing to)
- essentially this program will be simulating “pass by reference” in the C language
  - you want to change the value of the pointer passed to a function as the function argument

# Challenge #2

---

- first, create a function named allocateMemory that takes a single integer pointer as a function parameter
  - void allocateMemory(int \*ptr);
  - this function should allocate memory for this pointer
- now create a main function that does the following:
  - creates an integer pointer and initializes it to NULL
  - Invokes the allocateMemory function, passing in the integer pointer just created
  - assign a value to the integer pointer that it is pointing to (de-reference)
  - print the value of what the pointer is pointing to (de-reference)
  - free the pointer
- What is the output of the program? Why is this the output?

# Challenge #2

---

- now modify your program to use a double pointer
- modify the function named allocateMemory to take a double pointer of type int as a function parameter
  - void allocateMemory(int \*\*ptr);
  - this function should allocate memory for this pointer (use the correct syntax)
- now modify your main function that does the following:
  - creates an integer pointer and initializes it to NULL
  - Invokes the allocateMemory function, passing in the address of the integer pointer just created (double pointer)
  - assign a value to the integer pointer that it is pointing to (de-reference)
  - print the value of what the pointer is pointing to (de-reference)
  - free the pointer
- What is the output of the program? Why is this the output?

# Challenge #2 (reminders)

---

- if you pass a single pointer in as an argument
  - you will be modifying local copies of the pointer, not the original pointer in the calling scope
  - with a pointer to a pointer, you modify the original pointer
- use a double pointer as an argument to a function when you want to preserve the memory-allocation or assignment (value of the pointer) even outside of the function

# Summary

---

- review the demonstration and solution provided for further understanding

---

# Function Pointer Challenge

Jason Fedin

# Challenge

---

- this challenge will test your understanding of
  - how to declare a function pointer
  - how to assign functions to a function pointer
  - how to pass a function pointer as parameter
  - how to invoke a function using its pointer
- write a program that will perform some arithmetic operation (user entry) on an array using function pointers
- demo solution file to students showing how the program works (show the output, similar to next slides)

# Challenge

---

- download the challenge starter file from the resource section of the lecture (Challenge-starter-code.c)
- you need to modify the program to
  - based on user input, the program will perform the specified operation on the two arrays and then display the result
  - you will use an array of function pointers to perform the correct operation
- add code to the starter file:
  - **Task 1:**
    - create an array of function pointers that point to each arithmetic function (add, sub, mult, div)
  - **Task 2:**
    - add a fourth parameter to the performOp function so that it takes a function pointer with the signature of the arithmetic functions

# Challenge

---

- **Task 3:**

- pass in the correct function pointer (array index) to the performOp function

- **Task 4:**

- include the code for the function performOp
    - your code should invoke the function pointed to from the parameter passed in
      - will perform the operation on elements of the two arrays passed in and store the result in a third array
      - function should return the new array containing the results

- **Task 5:**

- include the code for the function display
    - your code should display as output all the elements of the given array in a readable format
- test your code with different user choices

# Summary

---

- review the demonstration and solution provided for further understanding

---

# (Challenge)

# Create a static library

Jason Fedin

# Overview

---

- the goal of this challenge is to create a static library and then write a program that uses that library
- create a C source file named “StringFunctions.c”
- “StringFunctions.c” should contain implementation of various string manipulation functions
  - Find the frequency of characters in a string
  - Remove all the characters in a String except Alphabets
  - Calculate the length of a string without using strlen
  - Concatenate two strings with using strcat
  - Copy a string manually without using strcpy
  - Find the substring of a given string
- these are easy functions to implement
  - the goal of this program is to create a static library of these C String manipulation functions

# Steps

---

- create a header file that contains function prototypes of each string manipulation function in “StringFunctions.h”

```
/*
str - string to search
searchCharacter - character to look for
return type - int : count for the number of times that character was
found
*/
int numberOfCharactersInString(char *str, char searchCharacter);
```

# removeNonAlphaCharacters, lengthOfString

---

```
/*
source - source string
return type - int : 0 on success
*/
```

```
int removeNonAlphaCharacters(char *source);
```

```
/*
source - source string
return type - int : length of string
*/
```

```
int lengthOfString(char *source);
```

# strConcat, strCopy

---

```
/*
str1 - string to concatenate to (resulting string)
str2 - second string to concatenate from
return type - int : 0 on success
*/
int strConcat(char *str1, char *str2);
```

```
/*
source - string to copy from
destination - second string to copy to
return type - int : 0 on success
*/
int strCopy(char *source, char *destination);
```

# substring

---

```
/*
source - source string
from - starting index from where you want to get substring
n - number of characters to be copied in substring
target - target string in which you want to store target string
return type - int : 0 on success
*/
int substring(char *source, int from, int n, char *target);
```

# Steps

---

- create your “StringFunctions.c” source file that implements all of the functions in the header file
- create a static library (.a) containing your string manipulation functions
  - lib\_stringfunctions.a
- create a program that acts as a driver for your string functions
  - test all of your functions in this file
  - statically link this program to your static library (lib\_stringfunctions.a)
- compile and run your program
  - should also try to manually compile
- review the demonstration and solution provided for further understanding

---

# (Challenge)

# Create a shared object

Jason Fedin

# Overview

---

- the goal of this challenge is to create a shared object (dynamic library) and then write a program that uses this library
- create a C source file named “StringFunctions.c”
- “StringFunctions.c” should contain implementation of various string manipulation functions
  - Find the frequency of characters in a string
  - Remove all the characters in a String except Alphabets
  - Calculate the length of a string without using strlen
  - Concatenate two strings with using strcat
  - Copy a string manually without using strcpy
  - Find the substring of a given string
- these are easy functions to implement
  - the goal of this program is to create a shared object library of these C String manipulation functions

# Steps

---

- create a header file that contains function prototypes of each string manipulation function in “StringFunctions.h”

```
/*
str - string to search
searchCharacter - character to look for
return type - int : count for the number of times that character was
found
*/
int numberOfCharactersInString(char *str, char searchCharacter);
```

# removeNonAlphaCharacters, lengthOfString

---

```
/*
```

```
source - source string
```

```
return type - int : 0 on success
```

```
*/
```

```
int removeNonAlphaCharacters(char *source);
```

```
/*
```

```
source - source string
```

```
return type - int : length of string
```

```
*/
```

```
int lengthOfString(char *source);
```

# strConcat, strCopy

---

```
/*
str1 - string to concatenate to (resulting string)
str2 - second string to concatenate from
return type - int : 0 on success
*/
int strConcat(char *str1, char *str2);
```

```
/*
source - string to copy from
destination - second string to copy to
return type - int : 0 on success
*/
int strCopy(char *source, char *destination);
```

# substring

---

```
/*
source - source string
from - starting index from where you want to get substring
n - number of characters to be copied in substring
target - target string in which you want to store target string
return type - int : 0 on success
*/
int substring(char *source, int from, int n, char *target);
```

# Steps

---

- create your “StringFunctions.c” source file that implements all of the functions in the header file
- create a shared object library (.so) containing your string manipulation functions
  - lib\_stringfunctions.so
- create a program that acts as a driver for your string functions
  - test all of your functions in this file
  - link this program to your dynamic library (lib\_stringfunctions.so)
- compile and run your program
  - should also try to manually compile
- review the demonstration and solution provided for further understanding

---

# (Challenge) Dynamic Loading

Jason Fedin

# Overview

---

- the goal of this challenge is to further your understanding of dynamic loading
- you need to dynamically load the symbols from the shared object library (lib\_stringfunctions.so) that you created in the last challenge
  - create another program that uses the dlopen and dlsym functions
  - basically, convert your driver program (main.c) from the last challenge to dynamically load all the string functions in the shared object
- create a C source file named “dynamicLoading.c”
- use dlopen to open the shared object file from the last challenge (lib\_stringfunctions.so)
  - make sure you use an absolute path when specifying the shared object filename (current directory??)
  - otherwise, you will need to setup your LD\_LIBRARY\_PATH environment variable or place the shared object in a certain directory

# Overview

---

- create function pointers for all 6 string functions from the shared object library

- // set up function pointers

```
int (*numChars) (char *, char);
int (*removeNonAlpha) (char *);
int (*lengthStr) (char *);
int (*strConcat) (char *, char *);
int (*strCopy) (char *, char *);
int (*substring) (char *, int, int, char *);
```

# Overview

---

- set the function pointers to point to the right symbol/function using the `dlsym` function
  - use `dlerror` to check for errors
- invoke all of the string manipulation functions using the function pointers
  - display output to the screen to verify that the string functions work
- make sure you call `dlclose` for the shared object handle that you used in `dlopen`
- the output of this program will be the same as in the last challenge
  - the difference is that the symbols/functions are dynamically loaded and not loaded at program start up time
- review the demonstration and solution provided for further understanding

---

# Challenge

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of random numbers
- write a C program that generates 50 random numbers between
  - 0.5 and 0.5
- you should output the random numbers
  - the first line of output should be the number of data
  - the next 50 lines should be the 50 random numbers
- you are required to use the srand function, passing in the time function
  - as a seed to using the rand() function

# Challenge #2

---

- this challenge will test your understanding of the quick sort algorithm (qsort()) from the stdlib.h file
- write a program that will sort an array of doubles from lowest to highest using the qsort function
- create a function that takes a double array and a size parameter which generates some random double values
  - void fillarray(double ar[], int n);
- create a function that displays an array (takes a double array and size)
  - void showarray(const double ar[], int n);
- your main function should
  - create an array
  - fill it with random numbers
  - display it
  - sort it using qsort
  - display the sorted array

# Challenge #2 (example output)

---

Random list:

|        |        |        |        |        |         |
|--------|--------|--------|--------|--------|---------|
| 2.1304 | 0.9808 | 4.6147 | 0.4364 | 0.5014 | 0.7591  |
| 0.7105 | 1.0393 | 0.8863 | 0.2333 | 0.0671 | 0.1706  |
| 0.3908 | 1.1928 | 4.5768 | 0.6113 | 2.0695 | 1.2160  |
| 0.5074 | 0.3792 | 0.6842 | 0.4489 | 0.8038 | 0.8789  |
| 0.0735 | 6.1123 | 0.2898 | 2.5516 | 3.2049 | 7.2541  |
| 0.2455 | 1.0603 | 0.4940 | 0.4935 | 0.7676 | 13.5337 |
| 0.4697 | 1.2911 | 0.3849 | 1.8076 |        |         |

Sorted list:

|        |        |        |         |        |        |
|--------|--------|--------|---------|--------|--------|
| 0.0671 | 0.0735 | 0.1706 | 0.2333  | 0.2455 | 0.2898 |
| 0.3792 | 0.3849 | 0.3908 | 0.4364  | 0.4489 | 0.4697 |
| 0.4935 | 0.4940 | 0.5014 | 0.5074  | 0.6113 | 0.6842 |
| 0.7105 | 0.7591 | 0.7676 | 0.8038  | 0.8789 | 0.8863 |
| 0.9808 | 1.0393 | 1.0603 | 1.1928  | 1.2160 | 1.2911 |
| 1.8076 | 2.0695 | 2.1304 | 2.5516  | 3.2049 | 4.5768 |
| 4.6147 | 6.1123 | 7.2541 | 13.5337 |        |        |

# Challenge #3

---

- this challenge will test your understanding of getting the current time
- write a program to print the current time
  - you should use the time and ctime functions
  - you should handle errors using fprintf and the exit function with the correct failure code

# Challenge #4

---

- this challenge will test your understanding of using the tm structure
- write a program to compute the number of seconds passed since the beginning of the current month
  - you should use the localtime, difftime, and mktime functions
  - you should handle errors using fprintf and the exit function with the correct failure codes

# Summary

---

- review the demonstrations and solutions provided for further understanding

---

# Linked List Challenge

Jason Fedin

# Challenge

---

- this challenge will test your understanding of linked lists
- write a program that performs operations on a linked list
- you need to create a linked list that stores integers and uses pointers
- your program should perform the following operations
  - Insert node at first
  - Insert node at last
  - Insert node at position
  - Delete Node from any Position
  - Update Node Value
  - Search Element in the linked list
  - Display List
  - Exit

# Challenge

---

- your program should create a structure that stores each nodes value and contains a next pointer
- your program can utilize global variables for previous, head, tail, temp, new nodes, etc or you can pass around data to functions
- your program will need to create a menu that allows the user to enter their choices for what operation to perform on the list

---

# Challenge

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of signals
- specifically, your program will:
  - raise signals
  - catch signals
  - use the alarm function to raise a signal
- write a program that will test a user's multiplication skills
  - the program will ask the user to work on an answer to a simple multiplication problem
  - keep track of how many answers are correct
- the program will keep running forever, unless
  - the user presses Ctrl-C OR
  - the user takes more than 5 seconds to answer the question
- when the program ends, it will display the final score (number of answers correct)

# Challenge #1

---

- I will provide some starter code for you that will generate the random multiplication questions and keep track of correct answers (see Challenge1\_starter\_code.c)
- you will need to add code for the following:
  - the program needs to handle the user pressing Ctrl-C
    - will need to handle this signal using the signal or sigaction functions
  - the program needs to raise a signal if the user does not answer a question within 5 seconds
    - use the alarm function and catch the SIGALRM signal

# Challenge #1 (Testing)

---

- to test this program, you will need to run it a couple of times
- the first time, you should answer a few questions and then hit Ctrl-C
  - ctrl-c sends the process an interrupt signal that makes the program display the final score and then exit()
- the second time, wait for at least five seconds on one of the answers and see what happens
  - the alarm signal ( SIGALRM) should occur
    - the program was waiting for the user to enter an answer, but because he took so long, the timer signal was sent
    - program should output “TIME’S UP!” and then raise the SIGINT signal which causes the program to display the final score

# Challenge #2

---

- this challenge will test your understanding of forking a process
- write a program to create one parent with three children processes (four processes)
  - must use the fork() function
- your program should contain output that identifies each parent and each child
  - will need to write if statements to check process id's returned from the fork() call, so that the output information is correct
    - "parent", "first child", "second child", "third child"
    - utilize the getpid() and getppid() functions to display each processes id
- at some instance of time, it is not necessary that child process will execute first or parent process will be first allotted CPU
  - any process may get CPU assigned, at some quantum time
  - also, the process id may differ during different executions

# Challenge #2 (Example output, demo)

---

parent  
13207 13208

  my id is 13194  
  my parentid is 6841

second child  
13207 0

  my id is 13208  
  my parentid is 13194

First child  
0 13209  
  my id is 13207  
  my parentid is 13194

third child  
0 0  
  my id is 13209  
  my parentid is 13207

# Summary

---

- review the demonstration and solution provided for further understanding

---

# Challenge

Jason Fedin

# Challenge #1

---

- this challenge will test your understanding of threads
  - creating and exiting threads, joining threads and passing arguments to thread functions
- write a program in which multiple threads print a message
  - create 10 separate threads and have each thread call one function that takes an argument
    - pass a different number for each thread to your thread function
    - might want to use an array to store each thread and to store each number
- create a global variable named counter and initialize it to 0 (shared variable amongst threads)

# Challenge #1

---

- create one function that accepts one parameter (void \*)
  - function should get invoked for each thread that you create
  - function should increment the global counter
  - function should then print out the message passed into it, its thread id, and the global counter
  - function should print out again the message passed into it, its thread id, and the global counter
- lastly, the main thread should call join and exit on all of its child threads
  - to make the original thread wait for its child threads to complete
- run the program many times, did you notice anything strange about the global counter values for the two print statements in the same thread?

# Challenge #2

---

- this challenge will test your understanding of threads and mutexes
  - you will make sure that data that is shared between threads is mutually exclusive when updated and when read
    - you should have consistent values in critical sections of code
- the goal of this challenge is to write thread-safe code
  - code that will produce the correct result regardless of the order that threads are scheduled to run or if they are accessing shared resources
- you should have noticed in challenge 1 that the global variable count value was a different value when displayed on two consecutive statements in the same thread
  - multiple threads would update the counter between the two print statements

# Challenge #2

---

- the global variable count should be the same value in each print statement for each thread
  - when the data is modified by a thread, the same thread should be able to read that data before it is modified again by another thread
  - data consistency of a shared resource, value is changing from statement to statement by another thread
- the counter variable is shared across multiple threads without any synchronization constructs to protect it
- for this challenge, you will need to edit the code from challenge 1 to create a mutex variable to ensure that multiple threads do not access the variable at the same time and introduce race conditions
- each thread should try to lock and unlock the mutex when they are executing any critical section (shared resources)
  - incrementing and printing the global counter variable
  - after doing this, the counter should display the same value in each print statement for that same thread

# Challenge #3

---

- this challenge will test your understanding of threads and condition variables
- we will modify the program from challenge #2 so that certain threads execute critical sections of code before other threads do
- this program should print messages from threads that pass in an even number first (parameter) and those that pass in an odd number after all the evens have printed
  - since we are not guaranteed that the threads will start in any given order, we must have the odd threads wait until all the even threads have printed
  - we do not care about the order that they print their message
- this program will require you to use condition variables to accomplish the ordering
  - all of the odd threads will sleep until a certain condition is met (all the even threads have finished)
- condition variables are always associated with locks because the shared information that they depend on must be synchronized across threads

# Challenge #3

---

- you will need to use the `pthread_cond_wait` and `pthread_cond_broadcast` functions
- you will also need a global variable that indicates when all even threads have finished
  - `int number_evens_finished = 0;` (shared data amongst threads)
  - you can increment the global variable for each even thread and signal the other threads when this count is equal to `NUM_THREADS / 2`
- you can check if numbers are even by using the modulus operator
  - `number % 2 == 0`
- you can use the `sleep(1)` call after all threads have been created and are running before checking if the even threads have finished

---

# Challenge

Jason Fedin

# Challenge

---

- this challenge will test your understanding of sockets and the client/server model
- you are going to write three programs for this challenge
- two of the programs will be client programs
- one of the programs will be a server program

# Challenge

---

- the functionality of the programs should be as follows
  - Client1 will send an integer to the server process
  - the server will decrement the number and send the result to client2
  - the server should print both the value it receives and the value that it sends
  - Client2 prints the number it receives and then all the processes should terminate
- the server should be listening for numbers on a port known to the clients
  - should handle the client connections sequentially and accept connections from multiple clients
- after servicing one client to completion (Client 1), it should proceed to the next