Photo: Dennis Ritchie and Ken Thompson

COMP201

Computer Systems & Programming

Lecture #01 – Introduction

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

A little about me…

Koç University
Associate Professor
2020-now

Hacettepe University
Associate Professor
2010-2020

Universitá Ca' Foscari di Venezia
Post-doctoral Researcher
2008-2010

Middle East Technical University
1997-2008
Ph.D., 2008
M.Sc., 2003
B.Sc., 2001

MIT
Fall 2007
Visiting Student

VirginiaTech
Visiting Research Scholar
Summer 2006

- I explore better ways to <u>understand</u>, <u>interpret</u> and <u>manipulate visual data</u>.
- My research interests span a diverse set of topics, ranging from image editing to visual saliency estimation, and to multimodal learning for integrated vision and language.

https://aykuterdem.github.io

Plan For Today

- Course Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

COMP201 during Return to Campus

- The classes will be face-to-face, but also synchronously streamed online using Zoom.
- **It is mandatory for students and faculty to wear masks at all times. Moreover, taking attendance is now mandatory in order to be able to follow the protocol shared in case of detection and follow-up of COVID-19 positive cases.**
- You are encouraged to turn on your cameras in order to keep human touch with others and not to feel alone, but this is clearly not a must!
- I will upload the slides prior to the class so that you can take a look and study the lecture material before the class. **We'll take regular question breaks to address your questions.**
- In Zoom, everyone is muted by default. However, during the question breaks, you can unmute yourself or alternatively use Zoom chat to ask your questions.

Let’s try Qwickly..

Qwickly
ATTENDANCE

Lecture Plan

- Course Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

What is COMP201?

- A new addition to the COMP curriculum, designed from ground-up.
- The third course in the line of COMP's introductory programming courses (COMP100, COMP132, and COMP201)
  - COMP100 teaches you the notion of computational thinking and how to solve problems as a programmer (using Python)
  - COMP132 introduces you object-oriented programming paradigm (using Java)
- COMP201 takes you **behind the scenes**:
  - Not quite down to hardware or physics/electromagnetism (that's for later...)
  - It's how things work **inside C++/Python/Java**, and how your programs map onto the components of computer systems
  - Not only does it just feel good to know how these work, it can also inform projects you work on in the future.

Role within COMP Curriculum

What is COMP201?

Gdb
Make
File
Program
Resource
Grade
Call
Find
See
Material
Syllabus
Ls
Move
Command
Work
Directory
Exam
Line
Bit
Code
Myth
Run
Test
Help
Forum
Fall
Here
Output
Question
Look
Answer
Write
Time
Want
Go
Email
Admin
One
Office
Unix
Assign
Lab
Hour
Need
Up
Back
Course
Set
More
Take
Student
Type
Function
Essential
Faq
Operate
Gcc
Cd
Ssh
Int
Point
Error
Check
Start
Log
Way
Edit
Grep
Flag
Add
Gradebook
Name
Tool
Two
Memory
Miss
Compute
Each
Refer
Key
Emacs
Text
Compile
Change
Sep
Step
Hello
Example
Navigation
Video
Made
Come
Know
Out
Case
Allow
Helloc
Void
Result
Match
Learn
Value
Create
Debug
Bug
Repo
Main
Right
Etc
Print
Many
Number
Note
Once
Such
Without
Path
Follow
Julie
Require
Simply
Different
Keep
Read
Byte
Now
Write
Written
Write
Much
Oull
Output
Valgrind
Search
Offer
Rm
Hg
Txt
Rose
Ingrid
Make
Makefile
Wc
Ie
Tx
Guide
Standard
Header
Pointer
Common
Library
Local
Quit
Show
Source
Lecture
Put
Thing
Option
Sunet
Builds
Through
Exit
Tab
Id
Lot
Copy
Ad
Execute
Rw
Tell
Review
Submit
Current
Sure
Man
Stop
Open
Being
Clear
Bad
Vim
Part
General
Admin
Email
Hello
Example
Emacs
Text
Compile
Change
Sep
Step
Refer
Key
Drwx
Few
Compute
Pwd
Down
Very
Miss
Cp
Gb
Link
Node
Toure
Site
Two
Toggle
Tool
Thing
Painter
Depend
Word
Format
Julie
Require
Simply
Source
Lecture
Put
Show
Quit
Local
Library
Header
Common
Standard
Pointer
Keep
Read
Different
Rm
Lk
Valgrind
Search
Offer
Makefile
Wc
Ie
Rose
Ingrid
Make
Write
Written
Write
Much
Oull
Output
System
Here
Fall
Forum
Help
Test
Run
Question
Try
Myth
Code
Bit
Line
Exam
Name
Gradebook
Directory
Work
Command
Start
Check
Print
Many
Number
Error
Point
Int
Ssh
Cat
Submission
End
Such
Without
Path
Follow

COMP201 Learning Goals

The goals for COMP201 are for students
to gain **mastery** of

- writing C programs with complex use of memory and pointers
- an accurate model of the address space and compile/runtime behavior of C programs

to achieve **competence** in

- translating C to/from assembly
- writing programs that respect the limitations of computer arithmetic
- finding bottlenecks and improving runtime performance
- working effectively in a Unix development environment

and have **exposure** to

- a working understanding of the basics of cache memories

COMP201 Learning Goals

(also learn to identify legitimate programmer scenes in old movies)

Jeff Goldblum's character saving the world by uploading a virus to the alien mothership
Independence Day, 1996

Elliot creating a malicious PDF file, which contains some sort of shellcode that will allow him to take over any Linux computer that opens that file in Evince.
Mr. Robot, S3, Ep9 - eps3.8_stage3.torrent (2017)

Course Overview

1. **Bits and Bytes -** *How can a computer represent integer numbers?*
2. **Chars and C-Strings -** *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap –** *How can we effectively manage all types of memory in our programs?*
4. **Generics -** *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. **Assembly -** *How does a computer interpret and execute C programs?*
6. **The Memory Hierarchy -** *How does the memory system is organized as a hierarchy of different storage devices with unique capacities*
7. **The Heap Allocators -** *How do core memory-allocation operations like malloc and free work?*

COMP201 - The Hybrid Edition

- This quarter, we will try our best to make your COMP201 experience as close as the face-to-face teaching version.
- We are working to emphasize community and connection.
- We understand that the pandemic poses unique challenges for each and every one of us involved.
- We will constantly evaluate and listen to ensure the class is going as smoothly as possible for everyone.
- Please don't hesitate to reach out if you want our support. We are always here for you.

COMP201 Slack Workspace

https://join.slack.com/t/comp201/signup

COMP2 e

Slack Developer Community Code of Conduct

This code of conduct governs Slack Platform's Community events and discussions.

Introduction

- Diversity and inclusion make our community strong. We encourage participation from the most varied and diverse backgrounds possible and want to be very clear about where we stand.
- Our goal is to maintain a safe, helpful and friendly community for everyone, regardless of experience, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, nationality, or other defining characteristic.
- This code and related procedures apply to unacceptable behavior occurring in all community venues, including behavior outside the scope of community activities — online and in-person— as well as in all one-on-one communications, and anywhere such behavior has the potential to adversely affect the safety and well-being of community members.

Expected Behavior

- Be welcoming.
- Be kind.
- Look out for each other.

Unacceptable Behavior

- Conduct or speech which might be considered sexist, racist, homophobic, transphobic, ableist or otherwise discriminatory or offensive in nature.
  - Do not use unwelcome, suggestive, derogatory or inappropriate nicknames or terms.
  - Do not show disrespect towards others. (Jokes, innuendo, dismissive attitudes.)
- Intimidation or harassment (online or in-person). Please read the Citizen Code of Conduct for how we interpret harassment.
- Disrespect towards differences of opinion.
- Inappropriate attention or contact. Be aware of how your actions affect others. If it

https: nup

New in this semester: Backchanneling

- [Dory.App link]

Teaching Team

Aykut Erdem

Mustafa Orkun Acar

Samet Demir

Mert Çökelek

Nafiseh Jabbari Tofighi

Yagmur Akarken

Doğa Demirtürk

Pınar Erbil

Aslıhan Gülseren

Doğa Ege İnhanlı

Atakan Kara

Ahmet Uyar

Barış Samed Yakar

Course Website

https://aykuterdem.github.io/classes/comp201/

*lecture videos on Panopto – can be accesses through Blackboard or from the course webpage

Question Break!

Lecture Plan

- Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

Textbooks

- *Computer Systems: A Programmer's Perspective* by Bryant & O'Hallaron, **3rd Edition**
  - **3rd edition matters** – important updates to course materials
- A C programming reference of your choice
  - *The C Programming Language* by Kernighan and Ritchie
  - Other C programming books, websites, or reference sheets

Course Structure

- **Lectures:** understand concepts, see demos
- **Labs:** learn tools, study code, discuss with peers
- **Assignments:** build programming skills, synthesize lecture/lab content

Monday	Wednesday	Friday
Lecture	Lecture	Lab-A-F

- **assg0**: out later on Wednesday, due Feb 25
- **C bootcamp**: this week (details will be announced soon)

Grading

38%	7 Programming assignments
21%	9 Labs
20%	Midterm exam
21%	Final exam

Grading

38%	Programming assignments
21%	Labs
20%	Midterm exam
21%	Final exam

Assignments

- 7 programming assignments completed individually using **Unix command line tools**
  - Free software, pre-installed on linuxpool cluster dedicated to COMP students
  - GitHub Classroom
  - We will give out starter projects for each assignment
- Graded on **functionality** (*behavior*) and **style** (*elegance*)
  - Functionality graded using *automated tools,* given as point score
  - Style graded via *automated tests* and TA code review,
  - Grades returned via Blackboard

GitHub Classroom

Late Policy

- **Start out with 7 *grace days***: each late day allows you to submit an assignment without penalty if you have free grace days left.
- **Hard deadline:** No submissions will be accepted **48 hours** after the original due date of an assignment
- Penalty per day after grace days are exhausted
  - 1 day: 20% off
  - 2 days: 40% off

- Submissions made within
  - 0-24 hours after the deadline will be counted as using 1 grace day,
  - 24-48 hours after the deadline will be counted as using 2 grace days.

Grading

38%	Programming assignments
21%	Labs
20%	Midterm exam
21%	Final exam

Lab Sections

- Weekly 100-minute labs led by a TA, starting *next* week, offered on Fridays.
- Hands-on practice with lecture material and course topics.

  GitHub Classroom + repl.it + `linuxpool.ku.edu.tr`
- Graded on attendance + participation *(verified by submitting lab work)*
  - Your lowest 2 scores will be dropped, hence there will be no make-up

Grading

38%	Programming assignments
21%	Labs
20%	Midterm exam
21%	Final exam

Midterm and Final Exams

- Face-to-face exams
  - Midterm: At week 10, exact date and time will be announced later
  - Final: Date and time will be announced later

Grading

38%	Programming assignments
21%	Labs
20%	Midterm exam
21%	Final exam

Question Break!

Getting Help

- Option 1: Post on the **Dory**
  - A backchanneling tool to spark engaging interactions between student and attendees; ask questions during the lectures vote and comment on those of others
  - Use Dory before, during, or after the lectures
- Option 2: Post on the **Discussion Forum at Blackboard**
  - Online discussion forum for students; post questions, answer other students' questions
  - Best for general assignment questions (DON'T POST ASSIGNMENT CODE!)
- Visit **Online Office Hours**
  - More info to come soon!
- **Email** the Course Staff
  - Best for **private matters** (e.g. grading questions).

Koç University Honor Code

- For assignments students should be required to digitally add and approve a version of the agreement below.

I hereby declare that I have completed this examination individually, without support from anyone else.

I hereby accept that only the below listed sources are approved to be used during this open-source examination:

(i) Coursebook,

(ii) All material that is made available to students via Blackboard for this course, and

(iii) Notes taken by me during lectures.

I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me.

Honor Code and COMP201

- Please help us ensure academic integrity:
  - Indicate any assistance received on HW (books, friends, etc.).
  - Do not look at other people's solution code or answers
  - Do not give your solutions to others or post them on the web or to the forum.
  - Report any inappropriate activity you see performed by others.
- Assignments are checked regularly for similarity with help of automated software tools.
- If you realize that you have made a mistake, you may retract your submission to any assignment at any time, no questions asked. Come to use before we come for you.
- If you need help, please contact us and we will help you.
  - We do not want you to feel any pressure to violate the Honor Code in order to succeed in this course.

Question Break!

Poll Time

Lecture Plan

- Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

What is Unix?

- **Unix:** a set of standards and tools commonly used in software development.
  - **macOS** and **Linux** are operating systems built on top of Unix
- You can navigate a Unix system using the **command line** ("terminal")
- Every Unix system works with the same tools and commands

What is the Command Line?

- The **command-line** is a text-based interface (i.e., **terminal** interface) to navigate a computer, instead of a Graphical User Interface (GUI).

Graphical User Interface

Text-based interface

Command Line vs. GUI

Just like a GUI file explorer interface, a terminal interface:

- shows you a **specific place** on your computer at any given time.
- lets you go **into folders** and **out of folders**.
- lets you **create new** files and **edit** files.
- lets you **execute programs**.

Graphical User Interface

Command-line interface

Why Use Unix / the Command Line?

- You can navigate almost any device using the same tools and commands:
  - Servers
  - Laptops and desktops
  - Embedded devices (Raspberry Pi, etc.)
  - Mobile Devices (Android, etc.)
- Used frequently by software engineers:
  - **Web development:** running servers and web tools on servers
  - **Machine learning:** processing data on servers, running algorithms
  - **Systems**: writing operating systems, networking code and embedded software
  - **Mobile Development**: running tools, managing libraries
  - And more…
- We'll use Unix and the command line to implement and execute our programs.

Demo: Using Unix and the Command Line

Unix Commands Recap

- **`cd`** – change directories (..)
- **`ls`** – list directory contents
- **`mkdir`** – make directory
- **`emacs`** – open text editor
- **`vi`** – open text editor
- **`rm`** – remove file or folder
- **`man`** – view manual pages

Lab 1:
The Linux Shell
(*next week*)

See the Resources page of the course website for more commands, and a complete reference.

Learning Unix and the Command Line

- Using Unix and the command line can be intimidating at first:
  - It looks retro!
  - How do I know what to type?

- It's like learning a new language:
  - At first, you may have to constantly look things up (**Resources** page on course website!)
  - It's important to spend as much time as possible (during labs and assignments) building muscle memory with the tools

Question Break!

Additional Reading

spectrum.ieee.org

IEEE SPECTRUM

Engineering Topics | Special Reports | Blogs | Multimedia | The Magazine | Professional Resources | Search

Feature | History | Cyberspace

28 Nov 2011 | 21:24 GMT

The Strange Birth and Long Life of Unix

The classic operating system turns 40, and its progeny abound

By Warren Toomey

They say that when one door closes on you, another opens. People generally offer this bit of wisdom just to lend some solace after a misfortune. But sometimes it's actually true. It certainly was for Ken Thompson and the late Dennis Ritchie, two of the greats of 20th-century information technology, when they created the Unix operating system, now considered one of the most inspiring and influential pieces of software ever written.

https://spectrum.ieee.org/tech-history/cyberspace/the-strange-birth-and-long-life-of-unix

Lecture Plan

- Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

The C Language

C was created around 1970 to make writing Unix and Unix tools easier.

- Part of the C/C++/Java family of languages (C++ and Java were created later)
- Design principles:
  - Small, simple abstractions of hardware
  - Minimalist aesthetic
  - Prioritizes efficiency and minimalism over safety and high-level abstractions

C vs. C++ and Java

They all share:

- Syntax
- Basic data types
- Arithmetic, relational, and logical operators

C doesn't have:

- More advanced features like operator overloading, default arguments, pass by reference, classes and objects, ADTs, etc.
- Extensive libraries (no graphics, networking, etc.) – this means not much to learn C!
- many compiler and runtime checks (this may cause security vulnerabilities!)

Programming Language Philosophies

- **C is procedural:** you write functions, rather than define new variable types with classes and call methods on objects. C is small, fast and efficient.
- **C++ is procedural, with objects:** you write functions, and define new variable types with classes, and call methods on objects.
- **Python is also procedural, but dynamically typed:** you still write functions and call methods on objects, but the development process is very different.
- **Java is object-oriented:** virtually everything is an object, and everything you write needs to conform to the object-oriented design pattern.

Why C?

- Many tools (and even other languages, like Python!) are built with C.
- C is the language of choice for fast, highly efficient programs.
- C is popular for systems programming (operating systems, networking, etc.)
- C lets you work at a lower level to manipulate and understand the underlying system.

Programming Language Popularity

https://www.tiobe.com/tiobe-index/

Programming Language Popularity

TIOBE Programming Community Index

Sep 2021	Sep 2020	Change	Programming Language	Ratings	Change
1	1		C	11.83%	-4.12%
2	3	^	Python	11.67%	+1.20%
3	2	v	Java	11.12%	-2.37%
4	4		C++	7.13%	+0.01%
5	5		C#	5.78%	+1.20%

https://www.tiobe.com/tiobe-index/

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>  // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Program comments

You can write block or inline comments.

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Import statements

C libraries are written with angle brackets.
Local libraries have quotes:
#include "lib.h"

Our First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

`main` function – entry point for the program
Should always return an integer (0 = success)

Our First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Main parameters – main takes two parameters, both relating to the command line arguments used to execute the program.

argc is the number of arguments in **argv**
argv is an array of arguments **(char * is C string)**

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

printf – prints output to the screen

Familiar Syntax

```
int x = 42 + 7 * -5;                // variables, types
double pi = 3.14159;
char c = 'Q';                       /* two comment styles */

for (int i = 0; i < 10; i++) {      // for loops
    if (i % 2 == 0) {               // if statements
        x += i;
    }
}

while (x > 0 && c == 'Q' || b) {    // while loops, logic
    x = x / 2;
    if (x == 42) { return 0; }
}

binky(x, 17, c);                   // function call
```

Boolean Variables

To declare Booleans, (e.g. **`bool b = _____`**), you must include **`stdbool.h`**:

```
#include <stdio.h>     // for printf
#include <stdbool.h>   // for bool

int main(int argc, char *argv[]) {
    bool x = 5 > 2 && binky(argc) > 0;
    if (x) {
        printf("Hello, world!\n");
    } else {
        printf("Howdy, world!\n");
    }
    return 0;
}
```

Boolean Expressions

C treats a nonzero value as true, and a zero value as false:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    if (x) {   // true
       printf("Hello, world!\n");
    } else {
       printf("Howdy, world!\n");
    }
    return 0;
}
```

Console Output: printf

```
printf(text, arg1, arg2, arg3);
```

```
// Example
char *classPrefix = "COMP";
int classNumber = 201;
printf("You are in %s%d", classPrefix, classNumber);    // You are in COMP201
```

`printf` makes it easy to print out the values of variables or expressions.

If you include *placeholders* in your printed text, `printf` will replace each placeholder *in order* with the values of the parameters passed after the text.

`%s (string)` `%d (integer)` `%f (double)`

Question Break!

Writing, Debugging and Compiling

We will use:

- the **vi/emacs** text editor to write our C programs
- the **make** tool to compile our C programs
- the **gdb** debugger to debug our programs
- the **valgrind** tools to debug memory errors and measure program efficiency

Demo: Compiling And Running A C Program

`args.c`

Working On C Programs Recap

- **ssh** – remotely log in to `linuxpool` computers (*later*)
- **Vi/Emacs** – text editor to write and edit C programs
  - Use the mouse to position cursor, scroll, and highlight text
  - :w / Ctl-x Ctl-s to save, :q / Ctl-x Ctl-c to quit
- **make** – compile program using provided Makefile
- **./myprogram** – run executable program (optionally with arguments)
- **make clean** – remove executables and other compiler files
- Lecture codes are accessible at Blackboard

Recap

- COMP201 is a programming class, which uses C to teach you about what goes on under the hood of programming languages and software.
- We'll use Unix and command line tools to write, debug and run our programs.
- Please regularly visit the course website, https://aykuterdem.github.io/classes/comp201 and follow the announcements on Blackboard.
- We're looking forward to an exciting quarter!
- **Next time:** *How a computer represents integer numbers? What are the limitations?*

Photo: Dennis Ritchie and Ken Thompson

COMP201

Computer Systems & Programming

Lecture #02 – Bits and Bytes, Representing and Operating on Integers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Frame: Professor Farnsworth (Futurama)

Good news, everyone!

- Assg0 will be out today (due Oct 7)
- We're dropping Dory.app for Slack (#back-channel)
- C bootcamp discussion on Friday!
- My office hour will be on Tuesdays btw 11:00-12:00.

COMP201 Topic 1: How can a computer represent integer numbers?

wsj.com

DJIA Futures 28069 0.26% ▲ U.S. 10 Yr 1/32 Yield 0.783% ▲ Euro 1.1797 0.08% ▲

THE WALL STREET JOURNAL.

SHARE

US Air, Comair Scramble To Get Back to Normal

A Wall Street Journal NEWS ROUNDUP
Dec. 27, 2004 12:01 am ET

SAVE PRINT TEXT

Air travelers on two airlines continued to face canceled flights and lost luggage after weather, worker absences and computer glitches left thousands stranded in airports over the holiday weekend. Comair Inc., a regional carrier owned by Delta Air Lines that canceled its entire schedule Saturday, resumed limited flights yesterday but said it wouldn't return to normal until midweek.

US Airways Group Inc. blamed more than 400 canceled flights and thousands of pieces of stranded luggage on large numbers of workers who called in sick, as well as on a heavy winter storm. A spokesman said the carrier had no evidence of a concerted job action, but the troubles underscore the problems low morale could cause the carrier as it struggles to emerge from bankruptcy-court protection.

It was unclear how many holiday travelers were affected, though the major disruptions appeared to be limited to US Airways and Comair. UAL Corp.'s United Airlines and Northwest Airlines reported weather difficulties in Chicago and Detroit, respectively. AMR Corp.'s American Airlines said it experienced problems due to unusual snowfall at its Dallas-Fort Worth hub over the weekend.

Demo: Unexpected Behavior

airline.c

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

0

1

Bits

- Computers are built around the idea of two states: “on” and “off”. Transistors represent this in hardware, and bits represent this in software!

One Bit At A Time

- We can combine bits, like with base-10 numbers, to represent more data. **8 bits = 1 byte.**
- Computer memory is just a large array of bytes! It is *byte-addressable*; you can't address (store location of) a bit; only a byte.
- Computers still fundamentally operate on bits; we have just gotten more creative about how to represent different data as bits!
  - Images
  - Audio
  - Video
  - Text
  - And more...

Base 10

5 9 3 4

Digits 0-9 (0 to base-1)

Base 10

5 9 3 4

thousands hundreds tens ones

= **5***1000 + **9***100 + **3***10 + **4***1

Base 10

5	9	3	4
10^3	10^2	10^1	10^0

Base 10

	5	9	3	4
10^X:	3	2	1	0

Base 2

	1	0	1	1
$\mathbf{2}^{X}$:	3	2	1	0

Digits 0-1 (*0* to *base-1*)

Base 2

1	0	1	1
2^3	2^2	2^1	2^0

Base 2

Most significant bit (MSB)

Least significant bit (LSB)

1 0 1 1

eights fours twos ones

$= \mathbf{1}*8 + \mathbf{0}*4 + \mathbf{1}*2 + \mathbf{1}*1 = 11_{10}$

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 $\leq$ 6?

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? $2^2=4$

2^3	2^2	2^1	2^0
0	1		

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**
  - Now, what is the largest power of 2 ≤ 6 – 2^2?

0	1		
2^3	2^2	2^1	2^0

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **$2^2=4$**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **$2^1=2$**

0	1	1	_
2^3	2^2	2^1	2^0

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **$2^2=4$**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **$2^1=2$**
  - $6 - 2^2 - 2^1 = 0$!

0	1	1	_
2^3	2^2	2^1	2^0

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **$2^2=4$**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **$2^1=2$**
  - $6 - 2^2 - 2^1 = 0$!

0	1	1	0
2^3	2^2	2^1	2^0

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **2^1=2**
  - 6 – 2^2 – 2^1 = 0!

0	1	1	0
2^3	2^2	2^1	2^0

= **0***8 + **1***4 + **1***2 + **0***1 = 6

Practice: Base 2 to Base 10

What is the base-2 value 1010 in base-10?

a) 20
b) 101
c) 10
d) 5
e) Other

Practice: Base 10 to Base 2

What is the base-10 value 14 in base 2?

a) 1111
b) 1110
c) 1010
d) Other

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store?

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store? **minimum = 0** **maximum = ?**

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store? **minimum = 0** **maximum = ?**

	1	1	1	1	1	1	1	1
2^x:	7	6	5	4	3	2	1	0

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store? **minimum = 0** **maximum = ?**

	1	1	1	1	1	1	1	1
2^x:	7	6	5	4	3	2	1	0

- **Strategy 1:** $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store? **minimum = 0** **maximum = 255**

	1	1	1	1	1	1	1	1
2^x:	7	6	5	4	3	2	1	0

- **Strategy 1:** $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$
- **Strategy 2:** $2^8 - 1 = 255$

Multiplying by Base

$$1450 \times 10 = 1450\underline{0}$$

$$1100_2 \times 2 = 1100\underline{0}$$

Key Idea: inserting 0 at the end multiplies by the base!

Dividing by Base

$$1450 \ / \ 10 = 145$$

$$1100_2 \ / \ 2 = 110$$

Key Idea: removing 0 at the end divides by the base!

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal.**

$$\underbrace{0110}_{0\text{-}15}\ \underbrace{1010}_{0\text{-}15}\ \underbrace{0011}_{0\text{-}15}$$

Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal.**

Each is a base-16 digit!

Hexadecimal

- Hexadecimal is *base-16*, so we need digits for 1-15. How do we do this?

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
										10	11	12	13	14	15

Hexadecimal

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

- We distinguish hexadecimal numbers by prefixing them with **`0x`**, and binary numbers with **`0b`**.
- E.g. **`0xf5`** is **`0b11110101`**

$$0x\ \underbrace{f}_{1111}\ \underbrace{5}_{0101}$$

Practice: Hexadecimal to Binary

What is **`0x173A`** in binary?

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

Practice: Hexadecimal to Binary

What is **`0b1111001010`** in hexadecimal? (*Hint: start from the right)*

Binary	**11**	**1100**	**1010**
Hexadecimal	**3**	**C**	**A**

Question Break!

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Number Representations

- **Unsigned Integers:** positive and 0 integers. (e.g. 0, 1, 2, … 99999…
- **Signed Integers:** negative, positive and 0 integers. (e.g. …-2, -1, 0, 1,… 9999…)

- **Floating Point Numbers:** real numbers. (e,g. 0.1, -12.2, 1.5×10^{12})

Number Representations

- **Unsigned Integers:** positive and 0 integers. (e.g. 0, 1, 2, … 99999…
- **Signed Integers:** negative, positive and 0 integers. (e.g. …-2, -1, 0, 1,… 9999…)

- **Floating Point Numbers:** real numbers. (e,g. 0.1, -12.2, 1.5×10^{12})

More on this next week!

Number Representations

C Declaration	Size (Bytes)
`int`	4
`double`	8
`float`	4
`char`	1
`char *`	8
`short`	2
`long`	8

In The Days Of Yore…

C Declaration	Size (Bytes)
`int`	4
`double`	8
`float`	4
`char`	1
`char *`	4
`short`	2
`long`	4

Transitioning To Larger Datatypes

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32}-1, equaling **2^{32} bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024 GB** of memory (RAM)!

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- We have already discussed converting between decimal and binary, which is a nice 1:1 relationship. Examples:

```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

- The range of an unsigned number is 0 → 2^w - 1, where *w* is the number of bits. E.g. a 32-bit integer can represent 0 to 2^{32} – 1 (4,294,967,295).

Unsigned Integers

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

Signed Integers

- A **signed** integer is a negative integer, 0, or a positive integer.
- *Problem:* How can we represent negative *and* positive numbers in binary?

Signed Integers

- A **signed** integer is a negative integer, 0, or a positive integer.
- *Problem:* How can we represent negative *and* positive numbers in binary?

Idea: let's reserve the *most significant bit* to store the sign.

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\underbrace{011}_{3}$$

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{000}_{0}$$

$$\underbrace{1}_{\text{negative}}\underbrace{000}_{0}$$

Sign Magnitude Representation

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

- We've only represented 15 of our 16 available numbers!

Sign Magnitude Representation

- **Pro:** easy to represent, and easy to convert to/from decimal.
- **Con:** +-0 is not intuitive
- **Con:** we lose a bit that could be used to store more numbers
- **Con:** arithmetic is tricky: we need to find the sign, then maybe subtract (borrow and carry, etc.), then maybe change the sign. This complicates the hardware support for something as fundamental as addition.

Can we do better?

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0101 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0101 \\
+\,1011 \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0011 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0011 \\
+\,1101 \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0000 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0000 \\
+0000 \\
\hline
0000
\end{array}
$$

A Better Idea

Decimal	Positive	Negative
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

Decimal	Positive	Negative
8	1000	1000
9	1001 (same as -7!)	NA
10	1010 (same as -6!)	NA
11	1011 (same as -5!)	NA
12	1100 (same as -4!)	NA
13	1101 (same as -3!)	NA
14	1110 (same as -2!)	NA
15	1111 (same as -1!)	NA

There Seems Like a Pattern Here…

$$\begin{array}{r} 0101 \\ +\,1011 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0011 \\ +\,1101 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0000 \\ +\,0000 \\ \hline 0000 \end{array}$$

- The negative number is the positive number **inverted, plus one!**

There Seems Like a Pattern Here…

A binary number plus its inverse is all 1s.

$$\begin{array}{r} 0101 \\ +\,1010 \\ \hline 1111 \end{array}$$

Add 1 to this to carry over all 1s and get 0!

$$\begin{array}{r} 1111 \\ +\,0001 \\ \hline 0000 \end{array}$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\ ?????? \\
\hline
000000
\end{array}
$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\ ???100 \\
\hline
000000
\end{array}
$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\ 011100 \\
\hline
000000
\end{array}
$$

Two’s Complement

Two's Complement

- In **two's complement**, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself.**
- The **two's complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!

Two's Complement

- **Con:** more difficult to represent, and difficult to convert to/from decimal and between positive and negative.
- **Pro:** only 1 representation for 0!
- **Pro:** all bits are used to represent as many numbers as possible
- **Pro:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!

Two's Complement

- Adding two numbers is just...adding! There is no special case needed for negatives. E.g. what is 2 + -5?

$$
\begin{array}{r}
0010 \\
+\,1011 \\
\hline
1101
\end{array}
\qquad
\begin{array}{c}
2 \\
-5 \\
\\
-3
\end{array}
$$

Two's Complement

- Subtracting two numbers is just performing the two's complement on one of them and then adding. E.g. 4 – 5 = -1.

0100	4		0100	4
-0101	5	→	+1011	-5
			1111	-1

Practice: Two's Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)

d) -8 (1000)

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

```
0b1111 + 0b1 = 0b0000
```

- If you go below the **minimum** value of your bit representation, you *wrap around* or *overflow* back to the **largest** bit representation.

```
0b0000 - 0b1 = 0b1111
```

Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap arou*

`0b111`

- If yo *wrap arou*

https://xkcd.com/571 **Can't Sleep**

`0b0000 - 0b1 = 0b1111`

Title text: If androids someday DO dream of electric sheep, don't forget to declare sheepCount as a long int.

Min and Max Integer Values

Type	Size (Bytes)	Minimum	Maximum
`char`	1	-128	127
`unsigned char`	1	0	255
`short`	2	-32768	32767
`unsigned short`	2	0	65535
`int`	4	-2147483648	2147483647
`unsigned int`	4	0	4294967295
`long`	8	-9223372036854775808	9223372036854775807
`unsigned long`	8	0	18446744073709551615

Min and Max Integer Values

`INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX, …`

Overflow

+1
+1
111…111
000…000
111…110
000…001
111…101
000…010
111…100
000…011
…
…
100…010
011…101
100…001
011…110
100…000
011…111
+1

Practice: Overflow

At which points can overflow occur for signed and unsigned int? (assume binary values shown are all 32 bits)

A. Signed and unsigned can both overflow at points X and Y

B. Signed can overflow only at X, unsigned only at Y

C. Signed can overflow only at Y, unsigned only at X

D. Signed can overflow at X and Y, unsigned only at X

E. Other

Unsigned Integers

Signed Numbers

Overflow In Practice: PSY

YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”

Overflow In Practice: Gandhi

- In the game "Civilization", each civilization leader had an "aggression" rating. Gandhi was meant to be peaceful, and had a score of 1.
- If you adopted "democracy", all players' aggression reduced by 2. Gandhi's went from 1 to **255**!
- Gandhi then became a big fan of nuclear weapons.

https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Windows 95 can only run for 49.7 days before crashing,

- Windows 95 was unable to run longer than 49.7 days of runtime!
- There exists `GetTickTime` function – part of the Windows API – which returns the number of milliseconds which has elapsed since the system has started up as a 32-bit uint.
- And there's 86M ms in a day, i.e. 1000 * 60 * 60 * 24 = 86,400,000 and 32 bits is 4,294,967,296 so 4,294,967,296 / 86,400,000 = 49.7102696 days!

cnet.com

Amazon event: Everything announced | Amazon's Astro home robot | Ring's flying Always Home Cam | Amazon's wall-mounted Echo Show 15 | 'Hey, Disney!' coming to Amazon Echo

cnet REVIEWS NEWS TECH FINANCE HEALTH HOME CARS DEALS JOIN / SIGN IN

Windows may crash after 49.7 days

A bizarre and probably obscure bug will crash some Windows computers after about a month and a half of use, Microsoft confirms.

Jan. 2, 2002 4:43 p.m. PT

A bizarre and probably obscure bug will crash some Windows computers after about a month and a half of use.
The problem, which affects both Microsoft Windows 95 and 98 operating systems, was confirmed by the company in an alert to its users last week.

"After exactly 49.7 days of continuous operation, your Windows 95-based computer may stop responding," Microsoft warned its users, without much further explanation. The problem is apparently caused by a timing algorithm, according to the company.

Microsoft has posted a fix for the problem, but cautions that the patch has not yet been completely tested and should only be downloaded by users affected by the problem. However, if you have used your computer for two months straight without a problem, it is probably safe to assume that you are not affected.

Microsoft confirmed the bug warning, but could not be reached to elaborate on how many users the problem will hit, exactly why the glitch occurs, or when a more reliable fix will be available.

Microsoft is in the process of testing a collection of bug fixes for Windows 98,

This classic Porsche 911 is fully... 00:00/09:00

https://youtu.be/tdrRoSdBM5M

Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Comair/Delta airline had to cancel thousands of flights days before Christmas
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen

Demo Revisited: Unexpected Behavior

`airline.c`

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

`printf` and Integers

- There are 3 placeholders for 32-bit integers that we can use:
  - `%d`: signed 32-bit int
  - `%u`: unsigned 32-bit int
  - `%x`: hex 32-bit int
- The placeholder—not the expression filling in the placeholder—dictates what gets printed!

Casting

- What happens at the byte level when we cast between variable types? **The bytes remain the same!** **This means they may be interpreted differently depending on the type.**

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

This prints out: `"v = -12345, uv = 4294954951"`. Why?

Casting

- What happens at the byte level when we cast between variable types? **The bytes remain the same!** **This means they may be interpreted differently depending on the type.**

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

The bit representation for -12345 is
`0b11111111111111111001111111000111`.

If we treat this binary representation as a positive number, it's *huge*!

Casting

0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
two's complement
signed integer
representation

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
unsigned integer
representation

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`			
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	**No!**
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	**No!**
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	**No!**
`-1 > -2`			
`(unsigned)-1 > -2`			

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	No!
`-1 > -2`	Signed	1	yes
`(unsigned)-1 > -2`			

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	No!
`-1 > -2`	Signed	1	yes
`(unsigned)-1 > -2`	Unsigned	1	yes

Type	Size (Bytes)	Minimum	Maximum
`int`	4	`-2147483648`	`2147483647`
`unsigned int`	4	`0`	`4294967295`

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4 - true
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2
u1 > u2
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2 - true
s1 > u3
```

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2 - true
s1 > u3 - true
```

Expanding Bit Representations

- Sometimes, we want to convert between two integers of different sizes (e.g. **`short`** to **`int`**, or **`int`** to **`long`**).
- We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension")
- Note: when doing <, >, <=, >= comparison between different size types, it will *promote to the larger type*.

Expanding Bit Representation

```
unsigned short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

Expanding Bit Representation

```
short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b

int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

— or —

```
short s = -4;
// short is a 16-bit format, so                    s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), 53191:

`0000 0000 0000 0000 1100 1111 1100 0111`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1100 1111 1100 0111`

This is -12345! And when we cast `sx` back an `int`, we sign-extend the number.

`1111 1111 1111 1111 1100 1111 1100 0111` // still -12345

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = -3;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), -3:

`1111 1111 1111 1111 1111 1111 1111 1101`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1111 1111 1111 1101`

This is -3! **If the number does fit, it will convert fine.** `y` looks like this:

`1111 1111 1111 1111 1111 1111 1111 1101` // still -3

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit unsigned `int`), 128000:

`0000 0000 0000 0001 1111 0100 0000 0000`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1111 0100 0000 0000`

This is 62464! **Unsigned numbers can lose info too.** Here is what `y` looks like:

`0000 0000 0000 0000 1111 0100 0000 0000` // still 62464

The `sizeof` Operator

```
long sizeof(type);
```

```
// Example
long int_size_bytes = sizeof(int);     // 4
long short_size_bytes = sizeof(short); // 2
long char_size_bytes = sizeof(char);   // 1
```

`sizeof` takes a variable type as a parameter and returns the size of that type, in bytes.

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 130; // 0b1000 0010
char cx = x;
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x;
```

```
short x = 25; // 0b1 1001
char cx = x;
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 130; // 0b1000 0010
char cx = x; // -126
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x; // 124
```

```
short x = 25; // 0b1 1001
char cx = x; // 25
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 390; // 0b1 1000 0110
char cx = x;
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x;
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 390; // 0b1 1000 0110
char cx = x; // -122
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x; // -15
```

Summary: Basic Rules of Expanding, Truncating

- Expanding (e.g., `short int` to `int`)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., `unsigned` to `unsigned short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Slide credit: R.E. Bryant and D.R. O'Hallaron

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- Byte Ordering

Number Representations

C Declaration	Intel IA32	X86-64
`int`	4	4
`double`	8	8
`float`	4	4
`char`	1	1
`char *`	8	8
`short`	2	2
`long`	8	8

Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**

- Conventions
  - **Big Endian**: Sun (Oracle SPARC), PPC Mac, Internet
    - Least significant byte has highest address
  - **Little Endian**: x86, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address

Byte Ordering Example

- Big Endian: Sun (Oracle SPARC), PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86, ARM processors running Android, iOS, and Linux
  - Least significant byte has lowest address
- Example:
  - Variable `x` has 4-byte value of `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian

Little Endian

The origin of “endian”

“Gulliver finds out that there is a law, proclaimed by the grandfather of the present ruler, requiring all citizens of Lilliput to break their eggs only at the little ends. Of course, all those citizens who broke their eggs at the big ends were angered by the proclamation. Civil war broke out between the Little-Endians and the Big-Endians, resulting in the Big-Endians taking refuge on a nearby island, the kingdom of Blefuscu.”

– Danny Cohen, On Holy Wars and A Plea For Peace (1980)

Illustration by Chris Beatrice

Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

```
int A = 15213;
```

```
int B = -15213;
```

Two's complement representation

```
long int C = 15213;
```

Recap

- Getting Started With C
- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types
- **Next time:** How can we manipulate individual bits and bytes? How can we represent floating point numbers?

photo by unsplash user @swimstaralex

COMP201

Computer Systems & Programming

Lecture #03 –Bits and Bitwise Operators, Floating Point

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

Recap: Unsigned and Signed Integers

Recap: Overflow in Unsigned Integers

Recap: Overflow in Signed Numbers

Bits and Bytes So Far

- all data is ultimately stored in memory in binary
- When we declare an integer variable, under the hood it is stored in binary

```
int x = 5;// really 0b0…0101 in memory!
```

- Until now, we only manipulate our integer variables in base 10 (e.g. increment, decrement, set, etc.)
- Today, we will learn about how to manipulate the underlying binary representation!
- This is useful for: more efficient arithmetic, more efficient storing of data, etc.

Plan For Today

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Expanding Bit Representations

- Sometimes, we want to convert between two integers of different sizes (e.g. **`short`** to **`int`**, or **`int`** to **`long`**).
- We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension")
- **Note:** when doing <, >, <=, >= comparison between different size types, it will *promote to the larger type*.

Expanding Bit Representation

```
unsigned short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

Expanding Bit Representation

```
short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b

int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

— or —

```
short s = -4;
// short is a 16-bit format, so                    s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), 53191:

`0000 0000 0000 0000 1100 1111 1100 0111`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1100 1111 1100 0111`

This is -12345! And when we cast `sx` back an `int`, we sign-extend the number.

`1111 1111 1111 1111 1100 1111 1100 0111` // still -12345

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = -3;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), -3:

`1111 1111 1111 1111 1111 1111 1111 1101`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1111 1111 1111 1101`

This is -3! **If the number does fit, it will convert fine.** `y` looks like this:

`1111 1111 1111 1111 1111 1111 1111 1101` // still -3

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and <u>discards the *more significant bits*</u>.

```
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit unsigned `int`), 128000:

`0000 0000 0000 0001 1111 0100 0000 0000`

When we cast `x` to a `short`, it only has 16-bits, and C *truncates* the number:

`1111 0100 0000 0000`

This is 62464! **Unsigned numbers can lose info too.** Here is what `y` looks like:

`0000 0000 0000 0000 1111 0100 0000 0000` // still 62464

The `sizeof` Operator

```
long sizeof(type);
```

```
// Example
long int_size_bytes = sizeof(int);     // 4
long short_size_bytes = sizeof(short); // 2
long char_size_bytes = sizeof(char);   // 1
```

`sizeof` takes a variable type as a parameter and returns the size of that type, in bytes.

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 130; // 0b1000 0010
char cx = x;
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x;
```

```
short x = 25; // 0b1 1001
char cx = x;
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 130; // 0b1000 0010
char cx = x; // -126
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x; // 124
```

```
short x = 25; // 0b1 1001
char cx = x; // 25
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 390; // 0b1 1000 0110
char cx = x;
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x;
```

Practice: Truncation

What are the values of `cx` for the passages of code below?

```
short x = 390; // 0b1 1000 0110
char cx = x; // -122
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x; // -15
```

In Sum: Basic Rules of Expanding, Truncating

- Expanding (e.g., `short int` to `int`)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., `unsigned` to `unsigned short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Slide credit: R.E. Bryant and D.R. O'Hallaron

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Number Representations

C Declaration	Intel IA32	X86-64
`int`	4	4
`double`	8	8
`float`	4	4
`char`	1	1
`char *`	4	8
`short`	2	2
`long`	4	8

Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**

- Conventions
  - **Big Endian**: Sun (Oracle SPARC), PPC Mac, Internet
    - Least significant byte has highest address
  - **Little Endian**: x86, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address

Byte Ordering Example

- Big Endian: Sun (Oracle SPARC), PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86, ARM processors running Android, iOS, and Linux
  - Least significant byte has lowest address
- Example:
  - Variable `x` has 4-byte value of `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian

Little Endian

The origin of “endian”

“Gulliver finds out that there is a law, proclaimed by the grandfather of the present ruler, requiring all citizens of Lilliput to break their eggs only at the little ends. Of course, all those citizens who broke their eggs at the big ends were angered by the proclamation. Civil war broke out between the Little-Endians and the Big-Endians, resulting in the Big-Endians taking refuge on a nearby island, the kingdom of Blefuscu.”

– Danny Cohen, On Holy Wars and A Plea For Peace (1980)

Illustration by Chris Beatrice

Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

```
int A = 15213;
```

```
int B = -15213;
```

Two's complement representation

```
long int C = 15213;
```

Aside: ASCII

- ASCII is an encoding from common characters (letters, symbols, etc.) to bit representations (chars).
  - E.g. 'A' is 0x41
- Neat property: all uppercase letters, and all lowercase letters, are sequentially represented!
  - E.g. 'B' is 0x42

More on this next week!

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Now that we understand binary representations, how can we manipulate them at the bit level?

Bitwise Operators

- You're already familiar with many operators in C:
  - **Arithmetic operators:** `+`, `-`, `*`, `/`, `%`
  - **Comparison operators:** `==`, `!=`, `<`, `>`, `<=`, `>=`
  - **Logical Operators**: `&&`, `||`, `!`

- Today, we're introducing a new category of operators: **bitwise operators:**
  - `&`, `|`, `~`, `^`, `<<`, `>>`

And (&)

AND is a binary operator. The AND of 2 bits is 1 if both bits are 1, and 0 otherwise.

`output = a & b;`

a	b	output
0	0	0
0	1	0
1	0	0
1	1	1

`&` with `1` to let a bit through, `&` with `0` to zero out a bit

Or (|)

OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1.

```
output = a | b;
```

a	b	output
0	0	0
0	1	1
1	0	1
1	1	1

`|` with `1` to turn on a bit, `|` with `0` to let a bit go through

Not (~)

NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise.

```
output = ~a;
```

a	output
0	1
1	0

Exclusive Or (^)

Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if *exactly* one of the bits is 1, or 0 otherwise.

```
output = a ^ b;
```

a	b	output
0	0	0
0	1	1
1	0	1
1	1	0

^ with `1` to flip a bit, ^ with `0` to let a bit go through

Operators on Multiple Bits

- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

Note: these are different from the logical operators AND (&&), OR (`||`) and NOT (`!`).

Operators on Multiple Bits

- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

This is different from logical AND (&&). The logical AND returns true if both are nonzero, or false otherwise. With &&, this would be `6 && 12`, which would evaluate to **true** (1).

Operators on Multiple Bits

- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

This is different from logical OR (`||`). The logical OR returns true if either are nonzero, or false otherwise. With `||`, this would be `6 || 12`, which would evaluate to **true** (1).

Operators on Multiple Bits

- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

This is different from logical NOT (`!`). The logical NOT returns true if this is zero, and false otherwise. With `!`, this would be `!12`, which would evaluate to **false** (`0`).

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Bit Vectors and Sets

- We can use bit vectors (ordered collections of bits) to represent finite sets, and perform functions such as union, intersection, and complement.
- **Example:** we can represent current courses taken using a **`char`**.

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	**0**	**1**	**0**	**0**	**0**	**1**	**1**

Bit Vectors and Sets

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

- How do we find the union of two sets of courses taken? Use OR:

```
  00100011
| 01100001
  --------
  01100011
```

Bit Vectors and Sets

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

- How do we find the intersection of two sets of courses taken? Use AND:

```
  00100011
& 01100001
  --------
  00100001
```

Bit Masking

- We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A **bitmask** is a constructed bit pattern that we can use, along with bit operators, to do this.
- **Example:** how do we update our bit vector to indicate we've taken COMP202?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011
| 00001000
  --------
  00101011
```

Bit Masking

```
#define COMP100 0x1      /* 0000 0001 */
#define COMP106 0x2      /* 0000 0010 */
#define COMP132 0x4      /* 0000 0100 */
#define COMP201  0x8     /* 0000 1000 */
#define COMP202  0x10    /* 0001 0000 */
#define COMP291  0x20    /* 0010 0000 */
#define COMP301  0x40    /* 0100 0000 */
#define COMP302  0x80    /* 1000 0000 */

char myClasses = ...;
myClasses = myClasses | COMP201; // Add COMP201
```

Bit Masking

```
#define COMP100 0x1      /* 0000 0001 */
#define COMP106 0x2      /* 0000 0010 */
#define COMP132 0x4      /* 0000 0100 */
#define COMP201  0x8     /* 0000 1000 */
#define COMP202  0x10    /* 0001 0000 */
#define COMP291  0x20    /* 0010 0000 */
#define COMP301  0x40    /* 0100 0000 */
#define COMP302  0x80    /* 1000 0000 */

char myClasses = ...;
myClasses |= COMP201;   // Add COMP201
```

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken COMP132?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011
& 11011111
  --------
  00000011
```

```
char myClasses = ...;
myClasses = myClasses & ~COMP132;     // Remove COMP132
```

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken COMP132?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011
& 11011111
  --------
  00000011
```

```
char myClasses = ...;
myClasses &= ~COMP132;  // Remove COMP132
```

Bit Masking

- **Example:** how do we check if we've taken COMP301?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011
& 00000010
  --------
  00000010
```

```
char myClasses = ...;
if (myClasses & COMP301) {...
    // taken COMP301!
```

Bit Masking

- **Example:** how do we check if we've *not* taken COMP201?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011
& 00010000
  --------
  00000000
```

```
char myClasses = ...;
if (!(myClasses & COMP201)) {...
    // not taken COMP201!
```

Bit Masking

- **Example:** how do we check if we've *not* taken COMP201?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

```
  00100011     00000000
& 00010000   ^ 00010000
  --------     --------
  00000000     00010000
```

```
char myClasses = ...;
if ((myClasses & COMP201) ^ COMP201) {...
    // not taken COMP201!
```

Practice: Bitwise Operations

How can we use bitmasks + bitwise operators to...

`0b00001101`

1. ...turn **on** a particular set of bits? **OR**

```
  0b00001101
  0b00000010
-------------
  0b00001111
```

2. ...turn off a particular set of bits? **AND**

```
  0b00001101
  0b11111011
-------------
  0b00001001
```

3. ...flip a particular set of bits? **XOR**

```
  0b00001101
  0b00000110
-------------
  0b00001011
```

Bitwise Operator Tricks

- `|` with `1` is useful for turning select bits on
- `&` with `0` is useful for turning select bits off
- `|` is useful for taking the union of bits
- `&` is useful for taking the intersection of bits
- `^` is useful for flipping select bits
- `~` is useful for flipping all bits

Bit Masking

- Bit masking is also useful for integer representations as well. For instance, we might want to check the value of the most-significant bit, or just one of the middle bytes.

- **Example:** If I have a 32-bit integer **`j`**, what operation should I perform if I want to get *just the lowest byte* in **`j`**?

```
int j = ...;
int k = j & 0xff;  // mask to get just lowest byte
```

Practice: Bit Masking

- **Practice 1:** write an expression that, given a 32-bit integer `j`, sets its least-significant byte to all 1s, but preserves all other bytes.

```
j | 0xff
```

Practice: Bit Masking

- **Practice 1:** write an expression that, given a 32-bit integer `j`, sets its least-significant byte to all 1s, but preserves all other bytes.

```
j | 0xff
```

- **Practice 2:** write an expression that, given a 32-bit integer `j`, flips ("complements") all but the least-significant byte, and preserves all other bytes.

```
j ^ ~0xff
```

Powers of 2

Without using loops, how can we detect if a binary number is a power of 2? What is special about its binary representation and how can we leverage that?

Demo: Powers of 2

`is_power_of_2`

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Left Shift (<<)

The LEFT SHIFT operator shifts a bit pattern a certain number of positions to the left. New lower order bits are filled in with 0s, and bits shifted off the end are lost.

```
x << k;    // evaluates to x shifted to the left by k bits
x <<= k;   // shifts x to the left by k bits
```

8-bit examples:

```
00110111 << 2 results in 11011100
01100011 << 4 results in 00110000
10010101 << 4 results in 01010000
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Idea: let's follow left-shift and fill with 0s.

```
short x = 2;  // 0000 0000 0000 0010
x >>= 1;      // 0000 0000 0000 0001
printf("%d\n", x); // 1
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Idea: let's follow left-shift and fill with 0s.

```
short x = -2; // 1111 1111 1111 1110
x >>= 1;      // 0111 1111 1111 1111
printf("%d\n", x); // 32767!
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Problem: always filling with zeros means we may change the sign bit.

Solution: let's fill with the sign bit!

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;    // evaluates to x shifted to the right by k bit
x >>= k;   // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Solution: let's fill with the sign bit!

```
short x = 2;  // 0000 0000 0000 0010
x >>= 1;      // 0000 0000 0000 0001
printf("%d\n", x); // 1
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;    // evaluates to x shifted to the right by k bit
x >>= k;   // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Solution: let's fill with the sign bit!

```
short x = -2; // 1111 1111 1111 1110
x >>= 1;      // 1111 1111 1111 1111
printf("%d\n", x); // -1!
```

Right Shift (>>)

There are *two kinds* of right shifts, depending on the value and type you are shifting:

- **Logical Right Shift:** fill new high-order bits with 0s.
- **Arithmetic Right Shift:** fill new high-order bits with the most-significant bit.

Unsigned numbers are right-shifted using **Logical Right Shift**.

Signed numbers are right-shifted using **Arithmetic Right Shift**.

This way, the sign of the number (if applicable) is preserved!

Shift Operation Pitfalls

1. *Technically*, the C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. However, **almost all compilers/machines** use arithmetic, and you can most likely assume this.
2. Operator precedence can be tricky! For example:

   `1<<2 + 3<<4` means **`1 << (2+3) << 4`** because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:

   `(1<<2) + (3<<4)`

Bit Operator Pitfalls

- The default type of a number literal in your code is an **int**.
- Let's say you want a long with the index-32 bit as 1:

```
long num = 1 << 32;
```

- This doesn't work! 1 is by default an **int**, and you can't shift an int by 32 because it only has 32 bits. You must specify that you want 1 to be a **long**.

```
long num = 1L << 32;
```

<u>COMP201 Topic 2</u>: How can a computer represent real numbers in addition to integer numbers?

Learning Goals

Understand the design and compromises of the floating point representation, including:

- Fixed point vs. floating point
- How a floating point number is represented in binary
- Issues with floating point imprecision
- Other potential pitfalls using floating point numbers in programs

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Real Numbers

- We previously discussed representing integer numbers using two's complement.
- However, this system does not represent real numbers such as 3/5 or 0.25.
- How can we design a representation for real numbers?

Real Numbers

Problem: unlike with the integer number line, where there are a finite number of values between two numbers, there are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1

Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001,…

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers*.

Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: $1/6_{10} = 0.16666666\ldots\ldots_{10}$

Base 2: $1/10_{10} = 0.0001100110011001100110011\ldots_{2}$

Therefore, by representing in base 2, *we will not be able to represent all numbers*, even those we can exactly represent in base 10.

Fixed Point

- **Idea:** Like in base 10, let's add binary decimal places to our existing number representation.

5	9	3	4	.	2	1	6
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1	0	1	1	.	0	1	1
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

Lecture Plan

- Casting and Combining Types
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Fixed Point

- **Idea:** Like in base 10, let's add binary decimal places to our existing number representation.

1	0	1	1	.	0	1	1
8s	4s	2s	1s		1/2s	1/4s	1/8s

- **Pros:** arithmetic is easy! And we know exactly how much precision we have.

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

.	0	1	1	0	0	1	1
	1/2s	1/4s	1/8s	...			

1	0	1	1	0	.	1	1
16s	8s	4s	2s	1s		1/2s	1/4s

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

Base 10	Base 2
5.07E30 =	$10\underbrace{____}_{\text{100 zeros}}0.1$
9.86E-32 =	$0.0\underbrace{____}_{\text{100 zeros}}01$

To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible "floating" decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior

Recap

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Next time: *More on how can a computer represent floating point numbers?*

photo by unsplash user @swimstaralex

COMP201

Computer Systems & Programming

Lecture #04 –Floating Point

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap: Bitwise Operators

- You're already familiar with many operators in C:
  - **Arithmetic operators:** +, -, *, /, %
  - **Comparison operators:** ==, !=, <, >, <=, >=
  - **Logical Operators**: &&, ||, !

- **Bitwise operators:**
  - **Logical operators:** &, |, ~, ^,
  - **Bit shift operators:** <<, >>

Recap: Real Numbers

Problem: unlike with the integer number line, where there are a finite number of values between two numbers, there are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1

Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001,…

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers*.

Recap: Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

Base 10 | Base 2

$$5.07E30 = 10\underbrace{____}_{\text{100 zeros}}0.1$$

$$9.86E\text{-}32 = 0.0\underbrace{____}_{\text{100 zeros}}01$$

To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

Plan For Today

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Lecture Plan

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible "floating" decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior

IEEE Floating Point

Let's aim to represent numbers of the following scientific-notation-like format:

$$x * 2^y$$

With this format, 32-bit floats represent numbers in the range ~1.2 x10^{-38} to ~3.4 x10^{38}! Is every number between those representable? **No.**

IEEE Single Precision Floating Point

$$x * 2^y$$

Exponent

Exponent (Binary)	Exponent (Base 10)
11111111	?
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	?

Exponent

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	RESERVED

Exponent

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

Exponent

- The exponent is **not** represented in two's complement.
- Instead, exponents are sequentially represented starting from 000…1 (most negative) to 111…10 (most positive). This makes bit-level comparison fast.
- **Actual value = binary value – 127 ("bias")**

11111110	254 – 127 = 127
11111101	253 – 127 = 126
…	…
00000010	2 – 127 = -125
00000001	1 – 127 = -126

Fraction

$$x * 2^y$$

- We could just encode whatever x is in the fraction field. But there's a trick we can use to make the most out of the bits we have.

An Interesting Observation

In Base 10:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

$0.624 \times 10^5 = 6.24 \times 10^4$

We tend to adjust the exponent until we get down to one place to the left of the decimal point.

In Base 2:

$10.1 \times 2^5 = 1.01 \times 2^6$

$1011.1 \times 2^5 = 1.0111 \times 2^8$

$0.110 \times 2^5 = 1.10 \times 2^4$

Observation: in base 2, this means there is always a 1 to the left of the decimal point!

Fraction

s	exponent (8 bits)	fraction (23 bits)

$$x * 2^y$$

- We can adjust this value to fit the format described previously. Then, x will always be in the format **1.XXXXXXXXX...**
- Therefore, in the fraction portion, we can encode just what is *to the right* of the decimal point! This means we get one more digit for precision.

Value encoded = 1._[FRACTION BINARY DIGITS]_

Practice

Sign	Exponent						Fraction			
0	0	…	0	0	0	1	0	1	0	…

Is this number:

A) Greater than 0?

B) Less than 0?

Practice

Sign	Exponent						Fraction			
0	0	…	0	0	0	1	0	1	0	…

Is this number:

A) **Greater than 0?**

B) **Less than 0?**

Is this number:

A) **Less than -1?**

B) **Between -1 and 1?**

C) **Greater than 1?**

Skipping Numbers

- We said that it's not possible to represent *all* real numbers using a fixed-width representation. What does this look like?

Float Converter

- https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floats and Graphics

- https://www.shadertoy.com/view/4tVyDK

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible ✓
- Flexible "floating" decimal point ✓
- Represent scientific notation numbers, e.g. 1.2×10^6 ?
- Still be able to compare quickly ✓
- Have more predictable over/under-flow behavior ?

Representing Zero

The float representation of zero is all zeros (with any value for the sign bit)

Sign	Exponent	Fraction
`any`	`All zeros`	`All zeros`

- This means there are two representations for zero! ☹

Representing Small Numbers

If the exponent is all zeros, we switch into “denormalized” mode.

Sign	Exponent	Fraction
`any`	`All zeros`	`Any`

- We now treat the exponent as -126, and the fraction as *without* the leading 1.
- This allows us to represent the smallest numbers as precisely as possible.

Representing Exceptional Values

If the exponent is all ones, and the fraction is all zeros, we have +- infinity.

Sign	Exponent	Fraction
`any`	`All ones`	`All zeros`

- The sign bit indicates whether it is positive or negative infinity.
- Floats have built-in handling of over/underflow!
  - Infinity + anything = infinity
  - Negative infinity + negative anything = negative infinity
  - Etc.

Representing Exceptional Values

If the exponent is all ones, and the fraction is nonzero, we have
Not a Number (NaN)

Sign	Exponent						Fraction
any	1	…	…	…	…	1	Any nonzero

- NaN results from computations that produce an invalid mathematical result.
  - Sqrt(negative)
  - Infinity / infinity
  - Infinity + -infinity
  - Etc.

Number Ranges

- 32-bit integer (type **int**):
  - -2,147,483,648 to 2147483647
  - Every integer in that range can be represented
- 64-bit integer (type **long**):
  - –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- 32-bit floating point (type **float**):
  - ~1.2 $x10^{-38}$ to ~3.4 $x10^{38}$
  - Not all numbers in the range can be represented (not even all integers in the range can be represented!)
  - Gaps can get quite large! (larger the exponent, larger the gap between successive fraction values)
- 64-bit floating point (type **double**):
  - ~2.2 $x10^{-308}$ to ~1.8 $x10^{308}$

Precision options

- Single precision: 32 bits

- Double precision: 64 bits

- Extended precision: 80 bits (Intel only)

Visualization: Floating Point Encodings

Lecture Plan

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the **`frac`**

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

$$v = (-1)^s M\, 2^E$$
$$n: E = Exp - Bias$$
$$d: E = 1 - Bias$$

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

- Notice how the distribution gets denser toward zero.

Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3

Special Properties of the IEEE Encoding

- FP Zero Same as Integer Zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider –0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Lecture Plan

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Demo: Float Arithmetic

`float_arithmetic.c`

Floating Point Arithmetic

Is this just overflowing? It turns out it's more subtle.

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints 3.14
```

Let's look at the binary representations for 3.14 and 1e20:

	31	30 … 23	22 … 0
3.14:	0	10000000	10010001111010111000011

	31	30 … 23	22 … 0
1e20:	0	11000001	01011010111100011101100

Floating Point Arithmetic

	31	30 … 23	22 … 0
3.14:	0	10000000	10010001111010111000011

	31	30 … 23	22 … 0
1e20:	0	11000001	01011010111100011101100

To add real numbers, we must align their binary points:

```
                          3.14
+   100000000000000000000.00
    100000000000000000003.14
```

What does this number look like in 32-bit IEEE format?

Floating Point Arithmetic

Step 1: convert from base 10 to binary

What is 10000000000000000000003.14 in binary? Let's find out!

http://web.stanford.edu/class/archive/cs/cs107/cs107.1184/float/convert.html

10101101011110001110101111000101101011000110001000000000000000000011.0010001111010111000010100011…

Floating Point Arithmetic

Step 2: find most significant 1 and take the next 23 digits for the fractional component, rounding if needed.

101011010111100011101011110001011010110001100010000000000000000011.00100011110101110000101000011…

1 01011010111100011101100

Floating Point Arithmetic

Step 3: find how many places we need to shift **left** to put the number in 1.xxx format. This fills in the exponent component.

101011010111100011101011110001011010110001100010000000000000000011.0010001111010111000010100011…

66 shifts -> 66 + 127 = 193

Floating Point Arithmetic

Step 4: if the sign is positive, the sign bit is 0. Otherwise, it's 1.

10101101011110001110101111000101101011000110001000000000000000000011.0010001111010111000010100011…

Sign bit is 0.

Floating Point Arithmetic

The binary representation for 1e20 + 3.14 thus equals the following:

31	30 … 23	22 … 0
0	11000001	01011010111100011101100

This is the **same** as the binary representation for 1e20 that we had before!

We didn't have enough bits to differentiate between 1e20 and 1e20 + 3.14.

Floating Point Arithmetic

Is this just overflowing? It turns out it's more subtle.

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints 3.14
```

Floating point arithmetic is not associative. The order of operations matters!

- The first line loses precision when first adding 3.14 and 1e20, as we have seen.
- The second line first evaluates 1e20 – 1e20 = 0, and then adds 3.14

Demo: Float Equality

`float_equality.c`

Floating Point Arithmetic

Float arithmetic is an issue with most languages, not just C!

- http://geocar.sdf1.org/numbers.html

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible ✓
- Flexible "floating" decimal point ✓
- Represent scientific notation numbers, e.g. 1.2×10^6 ✓
- Still be able to compare quickly ✓
- Have more predictable over/under-flow behavior ✓

Lecture Plan

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Ariane 5: A Bug and A Crash

- On June 4, 1996, Ariane 5 rocket self destructed just after 37 seconds after liftoff
- Cost: $500 million
- **Cause:** An overflow in the conversion from a 64 bit floating point number to a 16 bit signed integer
- A design flaw:
  - 5 times faster than Ariane 4
  - Reused same software specifications from Ariane 4
  - Ariane 4 assumes horizontal velocity would never overflow a 16-bit number

© Fourmy/REA/SABA/Corbis

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither **`d`** nor **`f`** is NaN

- `x == (int)(float) x` — False
- `x == (int)(double) x` — True
- `f == (float)(double) f` — True
- `d == (float) d` — False
- `f == -(-f);` — True
- `2/3 == 2/3.0` — False
- `d < 0.0 ⇒ ((d*2) < 0.0)` — True (OF?)
- `d > f ⇒ -f > -d` — True
- `d * d >= 0.0` — True (OF?)
- `(d+f)-d == f` — False

Floats Summary

- IEEE Floating Point is a carefully-thought-out standard. It's complicated, but engineered for their goals.
- Floats have an extremely wide range, but cannot represent every number in that range.
- Some approximation and rounding may occur! This means you definitely don't want to use floats e.g. for currency.
- Associativity does not hold for numbers far apart in the range
- Equality comparison operations are often unwise.

Additional Reading

What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esotoric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [**Computer Systems Organization**]: General—*instruction set design*; D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*, *error analysis*, *numerical algorithms* (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—*languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics* D.4.1 [**Operating Systems**]: Process Management—*synchronization*

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow

What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg, ACM Computing Surveys, 23(1), 1991

Recap

- Floating Point
- Example and Properties
- Floating Point Arithmetic
- Floating Point in C

Next time: *How can a computer represent and manipulate more complex data like text?*

photo by unsplash user @ creativegangsters

COMP201

Computer Systems & Programming

Lecture #05 –Chars and Strings in C

Aykut Erdem // Koç University // Fall 2021

KOÇ UNIVERSITY

COMP201 Coding Style Guide for C Programming

- Our guide serves as a brief introduction to C coding style.
- Following a formal style is very important to write a clean and easy to read code.
- There are many standards out there!

https://aykuterdem.github.io/classes/comp201/code-style/html/index.html

COMP201 Topic 3: How can a computer represent and manipulate more complex data like text?

Plan for Today

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli and Lisa Yan's Stanford CS107 class

—Swami Iyer's Umass Boston CS110 class

Lecture Plan

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

Char

A **char** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newLine = '\n';
char tab = '\t';
char singleQuote = '\'';
char backSlash = '\\';
```

ASCII

Under the hood, C represents each **char** as an 8-bit *integer* (its "ASCII value").

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

- Uppercase letters are sequentially numbered
- Lowercase letters are sequentially numbered
- Digits are sequentially numbered
- Lowercase letters are 32 more than their uppercase equivalents (bit flip!)

```
char uppercaseA = 'A';          // Actually 65
char lowercaseA = 'a';          // Actually 97
char zeroDigit = '0';          // Actually 48
```

Unicode Transformation Formats

- The International Standards Organization's (ISO) 16-bit Unicode system can represent every character in every known language, with room for more
- Unicode being somewhat wasteful of space for English documents, ISO also defined several "Unicode Transformation Formats" (UTF), the most popular being UTF-8

Emojis

- Emojis are just like characters, and they have a standard, too

Smileys & People

face-positive

№	Code	Browser	Appl	Goog[d]	Twtr.	One	FB	FBM	Sams.	Wind.	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600											—	—	—	grinning face
2	U+1F601														beaming face with smiling eyes
3	U+1F602												—		face with tears of joy
4	U+1F923							—			—	—	—	—	rolling on the floor laughing
5	U+1F603														grinning face with big eyes
6	U+1F604												—	—	grinning face with smiling eyes
7	U+1F605											—		—	grinning face with sweat
8	U+1F606											—		—	grinning squinting face
9	U+1F609														winking face

⋮

- Full Emoji List, v5.0
  https://unicode.org/emoji/charts/full-emoji-list.html

ASCII

We can take advantage of C representing each **`char`** as an *integer:*

```
bool areEqual = 'A' == 'A';          // true
bool earlierLetter = 'f' < 'c';      // false
char uppercaseB = 'A' + 1;
int diff = 'c' - 'a';                // 2
int numLettersInAlphabet = 'z' – 'a' + 1;
// or
int numLettersInAlphabet = 'Z' – 'A' + 1;
```

ASCII

We can take advantage of C representing each **`char`** as an *integer:*

```
// prints out every lowercase character
for (char ch = 'a'; ch <= 'z'; ch++) {
    printf("%c", ch);
}
```

Common `ctype.h` Functions

Function	Description
`isalpha(`***ch***`)`	true if *ch* is `'a'` through `'z'` or `'A'` through `'Z'`
`islower(`***ch***`)`	true if *ch* is `'a'` through `'z'`
`isupper(`***ch***`)`	true if *ch* is `'A'` through `'Z'`
`isspace(`***ch***`)`	true if *ch* is a space, tab, new line, etc.
`isdigit(`***ch***`)`	true if *ch* is `'0'` through `'9'`
`toupper(`***ch***`)`	returns uppercase equivalent of a letter
`tolower(`***ch***`)`	returns lowercase equivalent of a letter

Remember: these **return** a char; they cannot modify an existing char!

More documentation with `man isalpha`, `man tolower`

Common `ctype.h` Functions

```
bool isLetter = isalpha('A');       // true
bool capital = isupper('f');        // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');       // true
```

Lecture Plan

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

C Strings

C has no dedicated variable type for strings. Instead, a string is represented as an **array of characters** with a special ending sentinel value.

`"Hello"`

index	*0*	*1*	*2*	*3*	*4*	*5*
char	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`'\0'`

`'\0'` is the **null-terminating character**; you always need to allocate one extra space in an array for it.

String Length

Strings are **<u>not</u>** objects. They do not embed additional information (e.g., string length). We must calculate this!

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

We can use the provided **`strlen`** function to calculate string length. The null-terminating character does *not* count towards the length.

```
int length = strlen(myStr);        // e.g. 13
```

Caution: `strlen` is `O(N)` because it must scan the entire string! We should save the value if we plan to refer to the length later.

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **`char *`**. C passes the location of the first character rather than a copy of the whole array.

```
int doSomething(char *str) {
    ...
}

char myString[6];
...
doSomething(myString);
```

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **`char *`**. C passes the location of the first character rather than a copy of the whole array.

```
int doSomething(char *str) {
    ...
    str[0] = 'c'; // modifies original string!
    printf("%s\n", str);    // prints cello
}

char myString[6];
...  // e.g. this string is “Hello”
doSomething(myString);
```

We can still use a **`char *`** the same way as a **`char[]`**.

Lecture Plan

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- Practice: Diamonds

Common `string.h` Functions

Function	Description
`strlen(str)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(str1, str2)`, `strncmp(str1, str2, n)`	compares two strings; returns 0 if identical, <0 if ***str1*** comes before ***str2*** in alphabet, >0 if ***str1*** comes after ***str2*** in alphabet. ***strncmp*** stops comparing after at most ***n*** characters.
`strchr(str, ch)` `strrchr(str, ch)`	character search: returns a pointer to the first occurrence of ***ch*** in ***str***, or ***NULL*** if ***ch*** was not found in ***str***. `strrchr` find the last occurrence.
`strstr(haystack, needle)`	string search: returns a pointer to the start of the first occurrence of ***needle*** in ***haystack***, or ***NULL*** if ***needle*** was not found in ***haystack***.
`strcpy(dst, src)`, `strncpy(dst, src, n)`	copies characters in ***src*** to ***dst***, including null-terminating character. Assumes enough space in ***dst***. Strings must not overlap. **strncpy** stops after at most ***n*** chars, and <u>does not</u> add null-terminating char.
`strcat(dst, src)`, `strncat(dst, src, n)`	concatenate ***src*** onto the end of ***dst***. **strncat** stops concatenating after at most ***n*** characters. <u>Always</u> adds a null-terminating character.
`strspn(str, accept)`, `strcspn(str, reject)`	**strspn** returns the length of the initial part of ***str*** which contains <u>only</u> characters in ***accept***. **strcspn** returns the length of the initial part of ***str*** which does <u>not</u> contain any characters in ***reject***.

Common `string.h` Functions

Function	Description
`strlen(str)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(str1, str2)`, `strncmp(str1, str2, n)`	compares two strings; returns 0 if identical, <0 if ***str1*** comes before ***str2*** in alphabet, >0 if ***str1*** comes after ***str2*** in alphabet. ***strncmp*** stops comparing after at most ***n*** characters.
`strchr(str, ch)` `strrchr(str, ch)`	character search: returns a pointer to the first occurrence of ***ch*** in ***str***, or ***NULL*** if ***ch*** was not found in ***str***. `strrchr` find the last occurrence.
`strstr(haystack, n`	he first occurrence of not found in ***haystack***.
`strcpy(dst, src)`, `strncpy(dst, src, n)`	-terminating character. Assumes enough space in ***dst***. Strings must not overlap. **strncpy** stops after at most ***n*** chars, and <u>does not</u> add null-terminating char.
`strcat(dst, src)`, `strncat(dst, src, n)`	concatenate ***src*** onto the end of ***dst***. **strncat** stops concatenating after at most ***n*** characters. <u>Always</u> adds a null-terminating character.
`strspn(str, accept)`, `strcspn(str, reject)`	**strspn** returns the length of the initial part of ***str*** which contains <u>only</u> characters in ***accept***. **strcspn** returns the length of the initial part of ***str*** which does <u>not</u> contain any characters in ***reject***.

Many string functions assume **valid string** input; i.e., ends in a null terminator.

Comparing Strings

We <u>cannot</u> compare C strings using comparison operators like ==, < or >. This compares addresses!

```
// e.g. str1 = 0x7f42, str2 = 0x654d
void doSomething(char *str1, char *str2) {
     if (str1 > str2) { …    // compares 0x7f42 > 0x654d!
```

Instead, use **`strcmp`**.

The string library: `strcmp`

`strcmp(str1, str2)`: compares two strings.

- returns 0 if identical
- <0 if ***`str1`*** comes before ***`str2`*** in alphabet
- >0 if ***`str1`*** comes after ***`str2`*** in alphabet.

```
int compResult = strcmp(str1, str2);
if (compResult == 0) {
      // equal
} else if (compResult < 0) {
      // str1 comes before str2
} else {
      // str1 comes after str2
}
```

Copying Strings

We <u>cannot</u> copy C strings using =. This copies addresses!

```
// e.g. param1 = 0x7f42, param2 = 0x654d
void doSomething(char *param1, char *param2) {
      param1 = param2;    // copies 0x654d.  Points to same string!
      param2[0] = 'H';    // modifies the one original string!
```

Instead, use **`strcpy`**.

The string library: `strcpy`

`strcpy(dst, src)`: copies the contents of **`src`** into the string **`dst`**, <u>including</u> the null terminator.

```
char str1[6];
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
str2[0] = 'c';

printf("%s", str1);        // hello
printf("%s", str2);        // cello
```

Copying Strings – `strcpy`

```
char str1[6];
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
```

	0	1	2	3	4	5
str1	'h'	'e'	'l'	'l'	'o'	'\0'

	0	1	2	3	4	5
str2	?	?	?	?	?	?

Copying Strings – `strcpy`

We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
char str2[6];                    // not enough space!
strcpy(str2, "hello, world!");   // overwrites other memory!
```

Writing past memory bounds is called a “buffer overflow”. It can allow for security vulnerabilities!

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	?	?	?	?	?	?	- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	?	?	?	?	?	- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5
str2	'h'	'e'	?	?	?	?

- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5
str2	'h'	'e'	'l'	?	?	?

- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5
str2	'h'	'e'	'l'	'l'	?	?

- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5
str2	'h'	'e'	'l'	'l'	'o'	?

- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	- other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	' ' - other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	' ' 'w' - other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	' ' 'w' -'o'ther program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5								
str2	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	- other program memory -			

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	' ' 'w' 'o' 'r' 'l' - other program memory -

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5								
str2	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	-	-

other program memory

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	
str2	'h'	'e'	'l'	'l'	'o'	','	' ' 'w' 'o' 'r' 'l' 'd' '!'

other program memory

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5								
str2	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

other program memory

Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5								
str2	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

other program memory

String Copying Exercise

What value should go in the blank at right?

A. `4`

B. `5`

C. `6`

D. `12`

E. `strlen("hello")`

F. Something else

```
char str[______];
strcpy(str, "hello");
```

String Exercise

What is printed out by the following program?

```
int main(int argc, char *argv[]) {
    char str[9];
    strcpy(str, "Hi earth");
    str[2] = '\0';
    printf("str = %s, len = %lu\n",
           str, strlen(str));
    return 0;
}
```

A. `str = Hi, len = 8`
B. `str = Hi, len = 2`
C. `str = Hi earth, len = 8`
D. `str = Hi earth, len = 2`
E. None/other

Copying Strings – `strncpy`

`strncpy(dst, src, n)`: copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **`dst`** will *not be null terminated*!

```
// copying "hello"
char str2[5];
strncpy(str2, "hello, world!", 5);    // doesn't copy '\0'!
```

If there is no null-terminating character, we may not be able to tell where the end of the string is anymore. E.g. `strlen` may continue reading into some other memory in search of `'\0'`!

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	
str2	?	?	?	?	?	- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	?	?	?	?

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	?	?	?

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	?

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	
str2	'h'	'e'	'l'	'l'	'o'	- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	
str2	'h'	'e'	'l'	'l'	'o'	- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	
str2	'h'	'e'	'l'	'l'	'o'	- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	
str2	'h'	'e'	'l'	'l'	'o'	- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – strncpy

```
char str2[5];
strncpy(str2, "hello, world!", 5);
int length = strlen(str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4
str2	'h'	'e'	'l'	'l'	'o'

- other program memory -

Copying Strings – `strncpy`

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Copying Strings – `strncpy`

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	?	?	?	?	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	?	?	?	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	?	?	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	?	?	?	?	?	?	?	?	?	?

Copying Strings – `strncpy`

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

Copying Strings – strncpy

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
str1	'h'	'e'	'l'	'l'	'o'	?	?	?	?	?	?	?	?	?

```
hello?? J???
```

Copying Strings – `strncpy`

If necessary, we can add a null-terminating character ourselves.

```
// copying "hello"
char str2[6];                          // room for string and '\0'
strncpy(str2, "hello, world!", 5);     // doesn’t copy '\0'!
str2[5] = '\0';                        // add null-terminating char
```

Concatenating Strings

We <u>cannot</u> concatenate C strings using +. This adds addresses!

```
// e.g. param1 = 0x7f, param2 = 0x65
void doSomething(char *param1, char *param2) {
    printf("%s", param1 + param2);   // adds 0x7f and 0x65!
```

Instead, use **`strcat`**.

The string library: `str(n)cat`

`strcat(dst, src)`: concatenates the contents of **src** into the string **dst**.

`strncat(dst, src, n)`: same, but concats at most n bytes from **src**.

```
char str1[13];          // enough space for strings + '\0'
strcpy(str1, "hello ");
strcat(str1, "world!");     // removes old '\0', adds new '\0' at end
printf("%s", str1);         // hello world!
```

Both **`strcat`** and **`strncat`** remove the old **`'\0'`** and add a new one at the end.

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'\0'	?	?	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	?	?	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	?	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Substrings and `char *`

You can also create a `char *` variable yourself that points to an address within in an existing string.

```
char myString[3];
myString[0] = 'H';
myString[1] = 'i';
myString[2] = '\0';

char *otherStr = myString;  // points to 'H'
```

Substrings

`char *`s are pointers to characters. We can use them to create substrings of larger strings.

```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
```

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning.

```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
```

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning.

```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
printf("%s\n", str1);          // racecar
printf("%s\n", str2);          // car
```

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1);
printf("%s\n", str2);
```

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning. **NOTE:** the pointer still refers to the same characters!

```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
str2[0] = 'f';
printf("%s %s\n", chars, str1);        // racefar racefar
printf("%s\n", str2);                  // far
```

String copying exercise

```
1  char buf[____];
2  strcpy(buf, "Potatoes");
3  printf("%s\n", buf);
4  char *word = buf + 2;
5  strncpy(word, "mat", 3);
6  printf("%s\n", buf);
```

Line 1: What value should go in the blank?

A. 7
B. 8
C. 9
D. 12
E. `strlen("Potatoes")`

Line 6: What is printed?

A. `matoes`
B. `mattoes`
C. `Pomat`
D. `Pomatoes`
E. Something else
F. Compile error

char * vs. char[]

`char myString[]`

vs

`char *myString`

You can create `char *` pointers to point to any character in an existing string and reassign them since they are just pointer variables. You **cannot** reassign an array.

```
char myString[6];
strcpy(myString, "Hello");
myString = "Another string";                    // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;                  // ok
```

Substrings

To omit characters at the end, make a new string that is a partial copy of the original.

```
// Want just "race"
char str1[8];
strcpy(str1, "racecar");

char str2[5];
strncpy(str2, str1, 4);
str2[4] = '\0';
printf("%s\n", str1);        // racecar
printf("%s\n", str2);        // race
```

Substrings

We can combine pointer arithmetic and copying to make any substrings we'd like.

```
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = '\0';
printf("%s\n", str1);        // racecar
printf("%s\n", str2);        // ace
```

Lecture Plan

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

String Diamond

- Write a function **`diamond`** that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  - For example, `diamond("COMP201")` should print:

```
C
CO
COM
COMP
COMP2
COMP20
COMP201
 OMP201
  MP201
   P201
    201
     01
      1
```

Practice: Diamond

diamond.c

Key takeaways

1. Valid strings are null-terminated.

	0xf0	0xf1	0xf2	0xf3	0xf4	0xf5	*address*
str	'H'	'e'	'l'	'l'	'o'	'\0'	*char*

```
char str[6];
strcpy(str, "Hello");
int length = strlen(str);  // 5
```

Key takeaways from this time

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.

```
char str[6];
strcpy(str, "Hello");
int length = strlen(str);   // 5
char *ptr = str + 1;        // 0xf1
printf("%s\n", ptr);        // ello
```

Key takeaways from this time

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.
3. When you pass a `char[]` as a parameter, it is automatically passed as a `char *` (pointer to its first character)

Why did C bother with this representation?

- C is a powerful, **efficient** language that requires a solid understanding of computer memory.
- We'll hone this understanding over these next two weeks!

Takeaway #3 : man strcpy

```
1  char buf[6];
2  strcpy(buf, "Hello");
3  printf("%s\n", buf);
…  …
```

The address of the first element of buf

```
STRCPY(3)                    Linux Programmer's Manual

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);
```

- Lecture 6: where string constants like "hello" are stored.
- Lecture 11: what const means

Recap

- Characters
- Strings
- Common String Operations
- Practice: Diamonds

Next time: More strings, pointers

photo by unsplash user @ creativegangsters

COMP201

Computer Systems & Programming

Lecture #06 – More Strings, Pointers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Characters
- Strings
- Common String Operations
  - Comparing
  - Copying
  - Concatenating
  - Substrings
- Practice: Diamonds

Plan for Today

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli and Lisa Yan's Stanford CS107 class

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

String operations such as `strlen` use the null-terminating character to find the end of the string.

Side note: use `strlen` to get the length of a string. Don't use `sizeof`!

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.
`strchr(`***`str, ch`***`)` `strrchr(`***`str, ch`***`)`	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.
`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***`needle`*** in ***`haystack`***, or ***`NULL`*** if ***`needle`*** was not found in ***`haystack`***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***`src`*** to ***`dst`***, including null-terminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and <u>does not</u> add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. <u>Always</u> adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***`str`*** which contains <u>only</u> characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does <u>not</u> contain any characters in ***`reject`***.

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char daisy[6];
strcpy(daisy, "Daisy");
char *letterA = strchr(daisy, 'a');
printf("%s\n", daisy);          // Daisy
printf("%s\n", letterA);        // aisy
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `strrchr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy);              // Daisy Dog
printf("%s\n", substr);             // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strspn(daisy, "aDeoi");        // 3
```

"How many places can we go in the first string before I encounter a character <u>not in</u> the second string?"

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters <u>not in</u> the second string.

```
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strcspn(daisy, "driso");      // 2
```

"How many places can we go in the first string before I encounter a character <u>in</u> the second string?"

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. We can still operate on the string the same way as with a `char[]`. (*We'll see why today!*).

```
int doSomething(char *str) {
    char secondChar = str[1];
    ...
}

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...
```

Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *stringArray[5];  // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", stringArray[0]);  // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what **argv** is in **main**! This means we write the parameter type as:

```
void myFunction(char **stringArray) {

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Practice: Password Verification

Write a function **`verifyPassword`** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars,
char *badSubstrings[], int numBadSubstrings);
```

`password` is valid if it contains only letters in `validChars`, and does not contain any substrings in `badSubstrings`.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars,
char *badSubstrings[], int numBadSubstrings);
```

Example:

```
char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789",
          invalidSubstrings, 1);       // true
bool valid2 = verifyPassword("141234", "0123456789",
          invalidSubstrings, 1);       // false
```

Practice: Password Verification

`verify_password.c`

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- One (8 byte) pointer can refer to any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.

Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…

Memory

- Memory is a big array of bytes.
- Each byte has a unique numeric index that is commonly written in hexadecimal.
- A pointer stores one of these memory addresses.

Address	Value
	…
261	'\0'
260	'e'
259	'l'
258	'p'
257	'p'
256	'a'
	…

Looking Closely at C

- All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
- This means whenever we pass something as a parameter, we pass a copy.
- If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the *address* instead of a copy of the *value*.

Pointers

```
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);     // 3!
      ...
}
```

Pointers

A pointer is a variable that stores a memory address.

```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);     // 3!
      ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);     // 3!
      ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
      *intPtr = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);     // 3!
      ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);     // 3!
     ...
}
```

Pointers

<u>A pointer is a variable that stores a memory address.</u>

```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```

Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**. This makes a copy of the data.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data's location.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

Pointers

Without pointers, we would make copies.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);    // 2!
     ...
}
```

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Practice: Printing the value of a pointer

pointer.c

Lecture Plan

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

String Behavior #1: If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

Character Arrays

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can modify what is on the stack.

```
char str[6];
strcpy(str, "apple");
```

String Behavior #2: We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.

Character Arrays

An array variable refers to an entire block of memory. We cannot reassign an existing array to be equal to a new array.

```
char str[6];
strcpy(str, "apple");
char str2[8];
strcpy(str2, "apple 2");

str = str2;   // not allowed!
```

An array's size cannot be changed once we create it; we must create another new array instead.

String Behavior #3: If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.

String Parameters

How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      ...
}

int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      fun_times(local_str);
      return 0;
}
```

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\0'

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

String Parameters

How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      ...
}

int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      fun_times(local_str);
      return 0;
}
```

str	0xa0

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\0'

A. A copy of the array `local_str`

B. A pointer containing an address to the first element in `local_str`

char * Variables

How do you think the local variable **`str`** is being represented?

```
int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      char *str = local_str;
      ...
      return 0;
}
```

A. A copy of the array **`local_str`**
B. A pointer containing an address to the first element in **`local_str`**

char * Variables

How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      char *str = local_str;
      ...
      return 0;
}
```

A. A copy of the array `local_str`

B. A pointer containing an address to the first element in `local_str`

char * Variables

How do you think the local variable **str** is being represented?

```
int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      char *str = local_str + 2;
      ...
      return 0;
}
```

A. A copy of part of the array **local_str**
B. A pointer containing an address to the third element in **local_str**

char * Variables

How do you think the local variable `str` is being represented?

```
int main(int argc, char *argv[]) {
      char local_str[5];
      strcpy(local_str, "rice");
      char *str = local_str + 2;
      ...
      return 0;
}
```

str	0xa2

	0xa0	0xa1	0xa2	0xa3	0xa4
local_str	'r'	'i'	'c'	'e'	'\0'

A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`

String Parameters

All string functions take `char *` parameters – they accept `char[]`, but they are implicitly converted to char * before being passed.

- `strlen(char *str)`
- `strcmp(char *str1, char *str2)`
- …

- char * is still a string in all the core ways a char[] is
  - Access/modify characters using bracket notation
  - Print it out
  - Use string functions
  - But under the hood they are represented differently!
- **Takeaway:** We create strings as `char[]`, pass them around as `char *`

String Behavior #4: If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.

char *

There is another convenient way to create a string if we do not need to modify it later. We can create a `char *` and set it directly equal to a string literal.

```
char *myString = "Hello, world!";
char *empty = "";

myString[0] = 'h';                    // crashes!
printf("%s", myString);               // Hello, world!
```

`char *`

When we declare a char pointer equal to a string literal, the characters are not stored on the stack. Instead, they are stored in a special area of memory called the "data segment". We *cannot modify memory in this segment*.

```
char *str = "hi";
```

The pointer variable (e.g. **str**) refers to the *address of the first character of the string in the data segment.*

This applies only to creating *new* strings with `char *`. This does **<u>not</u>** apply for making a `char *` that points to an existing stack string.

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char myStr[6];
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char *myStr = "Hi";
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char buf[6];
strcpy(buf, "Hi");
char *myStr = buf;
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char *otherStr = "Hi";
char *myStr = otherStr;
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char buf[6];
    strcpy(buf, "Hi");
    myFunc(buf);
    return 0;
}
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

Q: Is there a way to check in code whether a string's characters are modifiable?

A: No. This is something you can only tell by looking at the code itself and how the string was created.

Q: So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?

A: You can't! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that's not your function's fault :-)

String Behavior #5: We can set a **`char *`** equal to another value, because it is a reassign-able pointer.

char *

A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
char *str = "apple";          // e.g. 0xfff0
char *str2 = "apple 2";       // e.g. 0xfe0
str = str2;   // ok!  Both store address 0xfe0
```

Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      ...
}
```

Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;

      // equivalent
      char *ptr = &str[0];

      // confusingly equivalent, avoid
      char *ptr = &str;
      ...
}
```

String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.

Pointer Arithmetic

When we do pointer arithmetic, we are adjusting the pointer by a certain *number of places* (e.g. characters).

```
char *str = "apple";    // e.g. 0xff0
char *str2 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

printf("%s", str);      // apple
printf("%s", str2);     // pple
printf("%s", str3);     // le
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

char *

When we use bracket notation with a pointer, we are performing *pointer arithmetic and dereferencing*:

```
char *str = "apple";    // e.g. 0xff0

// both of these add three places to str,
// and then dereference to get the char
there.
// E.g. get memory at 0xff3.
char thirdLetter = str[3];        // 'l'
char thirdLetter = *(str + 3);    // 'l'
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

Strings as Parameters

When we pass a **`char *`** string as a parameter, C makes a *copy* of the address stored in the **`char *`** and passes it to the function. This means they both refer to the same memory location.

```
void myFunc(char *myStr) {
     ...
}

int main(int argc, char *argv[]) {
     char *str = "apple";
     myFunc(str);
     ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char ***) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char ***) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      // equivalent
      char *strAlt = str;
      myFunc(strAlt);
      ...
```

Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```

Compile error (cannot reassign array)

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```

   Compile error (cannot reassign array)

2. ```
   char *str = "Hello2";
   ```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`
   `strcpy(str, "Hello1");`

   Compile error (cannot reassign array)

2. `char *str = "Hello2";`

   Segmentation fault (string literal)

char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```

char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```
   Compile error (cannot reassign array)

2. ```
   char *str = "Hello2";
   ```
   Segmentation fault (string literal)

3. ```
   char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;
   ```
   Prints eulo3

   ```

# char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```
Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```
Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```
Segmentation fault (string literal)

Extra Slides

1. Pointer arithmetic

```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

int main(int argc, const char *argv[]) {
    char buf[] = "Monday";
    printf("before func: %s\n", buf);
    func(buf);
    printf("after  func: %s\n", buf);
    return 0;
}
```

- Will there be a compile error/segfault?
- If no errors, what is printed?

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

1. Pointer arithmetic

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

What happens if we call `strncpy(buf, str, 5);`?

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

dest	
src	
n	5
i	

What happens if we call `strncpy(buf, str, 5);`?

3. char* vs char[] exercises

Suppose we use a variable str as follows:

```
// initialize as below
A  str = str + 1;
B  str[1] = 'u';
C  printf("%s", str)
```

For each of the following initializations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```

2.
```
char *str = "Hello2";
```

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```

4.
```
char *ptr = "Hello4";
char *str = ptr;
```

3. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```
// initialize as below
A  str = str + 1;
B  str[1] = 'u';
C  printf("%s", str)
```

For each of the following initializations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```

Line A: Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```

Line B: Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```

Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```

Line B: Segmentation fault (string literal)

4. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

What is stored in each variable? (We cover double pointers later in the course)

4. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

Recap

- Searching in Strings
- Practice: Password Verification
- Pointers
- Practice: Printing the value of a pointer
- Strings in Memory

Next time: Arrays and Pointers

0 vs NULL // Twitter @raysato

COMP201

Computer Systems & Programming

Lecture #7 – Arrays and Pointers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Assg1 is due Oct 21
- Assg2 will be out on Oct 21 (due Nov 4)

Recap: Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.
`strchr(`***`str, ch`***`)` `strrchr(`***`str, ch`***`)`	character search: returns a pointer to the first occurrence of ***`ch`*** in ***`str`***, or ***`NULL`*** if ***`ch`*** was not found in ***`str`***. `strrchr` find the last occurrence.
`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***`needle`*** in ***`haystack`***, or ***`NULL`*** if ***`needle`*** was not found in ***`haystack`***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***`src`*** to ***`dst`***, including null-terminating character. Assumes enough space in ***`dst`***. Strings must not overlap. **`strncpy`** stops after at most ***`n`*** chars, and <u>does not</u> add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. <u>Always</u> adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***`str`*** which contains <u>only</u> characters in ***`accept`***. **`strcspn`** returns the length of the initial part of ***`str`*** which does <u>not</u> contain any characters in ***`reject`***.

Recap: Pointers

- A *pointer* is a variable that stores a memory address.
- Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
- One (8 byte) pointer can represent any size memory location!
- Pointers are also essential for allocating memory on the heap, which we will cover later.
- Pointers also let us refer to memory generically, which we will cover later.

Recap: Pointers

```
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

Recap: Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

Recap: Character Arrays

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can modify what is on the stack.

```
char str[6];
strcpy(str, "apple");
```

Recap: Recap: `char *`

When we declare a char pointer equal to a string literal, the characters are not stored on the stack. Instead, they are stored in a special area of memory called the "data segment". We *cannot modify memory in this segment*.

```
char *str = "hi";
```

The pointer variable (e.g. **str**) refers to the *address of the first character of the string in the data segment.*

This applies only to creating *new* strings with `char *`. This does **<u>not</u>** apply for making a `char *` that points to an existing stack string.

Plan for Today

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli and Lisa Yan's Stanford CS107 class

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char myStr[6];
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char *myStr = "Hi";
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char buf[6];
strcpy(buf, "Hi");
char *myStr = buf;
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
char *otherStr = "Hi";
char *myStr = otherStr;
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

For each code snippet below, can we modify the characters in **`myStr`**?

```
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char buf[6];
    strcpy(buf, "Hi");
    myFunc(buf);
    return 0;
}
```

Key Question: where do its characters live? Do they live in memory we own? Or the read-only data segment?

Memory Locations

Q: Is there a way to check in code whether a string's characters are modifiable?

A: No. This is something you can only tell by looking at the code itself and how the string was created.

Q: So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?

A: You can't! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that's not your function's fault :-)

String Behavior #5: We can set a **`char *`** equal to another value, because it is a reassign-able pointer.

char *

A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
char *str = "apple";          // e.g. 0xfff0
char *str2 = "apple 2";       // e.g. 0xfe0
str = str2;   // ok!  Both store address 0xfe0
```

Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      ...
}
```

Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;

      // equivalent
      char *ptr = &str[0];

      // confusingly equivalent, avoid
      char *ptr = &str;
      ...
}
```

String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.

Pointer Arithmetic

When we do pointer arithmetic, we are adjusting the pointer by a certain *number of places* (e.g. characters).

```
char *str = "apple";    // e.g. 0xff0
char *str2 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

printf("%s", str);      // apple
printf("%s", str2);     // pple
printf("%s", str3);     // le
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

char *

When we use bracket notation with a pointer, we are performing *pointer arithmetic and dereferencing*:

```
char *str = "apple";    // e.g. 0xff0

// both of these add three places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff3.
char thirdLetter = str[3];        // 'l'
char thirdLetter = *(str + 3);    // 'l'
```

TEXT SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

Strings as Parameters

When we pass a **`char *`** string as a parameter, C makes a *copy* of the address stored in the **`char *`** and passes it to the function. This means they both refer to the same memory location.

```
void myFunc(char *myStr) {
     ...
}

int main(int argc, char *argv[]) {
     char *str = "apple";
     myFunc(str);
     ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char ***) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char ***) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      // equivalent
      char *strAlt = str;
      myFunc(strAlt);
      ...
```

Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Strings In Memory

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. If we create a new string with new characters as a **`char *`**, we cannot modify its characters because its memory lives in the data segment.
5. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
6. Adding an offset to a C string gives us a substring that many places past the first character.
7. If we change characters in a string parameter, these changes will persist outside of the function.

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```

   Compile error (cannot reassign array)

   ```

# `char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```

   Compile error (cannot reassign array)

2. ```
   char *str = "Hello2";
   ```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];`
   `strcpy(str, "Hello1");`

   Compile error (cannot reassign array)

2. `char *str = "Hello2";`

   Segmentation fault (string literal)

char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```

   Compile error (cannot reassign array)

2. ```
   char *str = "Hello2";
   ```

   Segmentation fault (string literal)

3. ```
   char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;
   ```

   ```

# char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```
   Compile error (cannot reassign array)

2. ```
   char *str = "Hello2";
   ```
   Segmentation fault (string literal)

3. ```
   char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;
   ```
   Prints eulo3

   ```

# char* vs char[] exercises

Suppose we use a variable str as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```
Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```

`char*` vs `char[]` exercises

Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str);
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1.
```
char str[7];
strcpy(str, "Hello1");
```
Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```
Segmentation fault (string literal)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```
Prints eulo3

4.
```
char *ptr = "Hello4";
char *str = ptr;
```
Segmentation fault (string literal)

COMP201 Topic 4: How can we effectively manage all types of memory in our programs?

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(int x) {
     …
}

int main(int argc, char *argv[]) {
     int num = 4;
     myFunction(num);        // passes copy of 4
}
```

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(int *x) {
     …
}

int main(int argc, char *argv[]) {
     int num = 4;
     myFunction(&num);       // passes copy of e.g. 0xffed63
}
```

C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```
void myFunction(char ch) {
     …
}

int main(int argc, char *argv[]) {
     char *myStr = "Hello!";
     myFunction(myStr[1]);        // passes copy of 'e'
}
```

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

```
void myFunction(char ch) {
    printf("%c", ch);
}

int main(int argc, char *argv[]) {
    char *myStr = "Hello!";
    myFunction(myStr[1]);       // prints 'e'
}
```

C Parameters

If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

```
int myFunction(int num1, int num2) {
     return x + y;
}

int main(int argc, char *argv[]) {
     int x = 5;
     int y = 6;
     int sum = myFunction(x, y);      // returns 11
}
```

C Parameters

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

Do I care about modifying *this* instance of my data? If so, I need to pass where that instance lives, as a parameter, so it can be modified.

Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void capitalize(char *ch) {
    // modifies what is at the address stored in ch
}

int main(int argc, char *argv[]) {
    char letter = 'h';
    /* We don’t want to capitalize any instance of 'h'.
     * We want to capitalize *this* instance of 'h'! */
    capitalize(&letter);
    printf("%c", letter);  // want to print 'H';
}
```

Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```
void doubleNum(int *x) {
     // modifies what is at the address stored in x
}

int main(int argc, char *argv[]) {
     int num = 2;
     /* We don’t want to double any instance of 2.
      * We want to double *this* instance of 2! */
     doubleNum(&num);
     printf("%d", num);  // want to print 4;
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
    // *ch gets the character stored at address ch.
    char newChar = toupper(*ch);

    // *ch = goes to address ch and puts newChar there.
    *ch = newChar;
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
     /* go to address ch and put the capitalized version
      * of what is at address ch there. */
      *ch = toupper(*ch);
}
```

Pointers

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```
void capitalize(char *ch) {
    // this capitalizes the address ch! ☹
    char newChar = toupper(ch);

    // this stores newChar in ch as an address! ☹
    ch = newChar;
}
```

`char *`

- A `char *` is technically a pointer to a **<u>single character</u>**.
- We commonly use `char *` as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A `char *` could also just point to a single character (not a string).

String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.

Strings as Parameters

When we pass a `char *` string as a parameter, C makes a *copy* of the address stored in the `char *`, and passes it to the function. This means they both refer to the same memory location.

```
void myFunc(char *myStr) {
     ...
}

int main(int argc, char *argv[]) {
     char *str = "apple";
     myFunc(str);
     ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (as a `char *`) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      ...
}
```

Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (as a `char *`) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      // equivalent
      char *strAlt = str;
      myFunc(strAlt);
      ...
```

Strings as Parameters

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Strings as Parameters

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);    // apply
      ...
}
```

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(__?__) {
    int square = __?__ * __?__;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(__?__);     // should print 9
}
```

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(int x) {
     int square = x * x;
     printf("%d", square);
}

int main(int argc, char *argv[]) {
     int num = 3;
     printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.

Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```
void printSquare(int x) {
    x = x * x;
    printf("%d", x);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.

Exercise 2

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```
void flipCase(__?__) {
      if (isupper(__?__)) {
            __?__ = __?__;
      } else if (islower(__?__)) {
            __?__ = __?__;
      }
}

int main(int argc, char *argv[]) {
      char ch = 'g';
      flipCase(__?__);
      printf("%c", ch);     // want this to print ‘G’
}
```

Exercise 2

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```
void flipCase(char *letter) {
      if (isupper(*letter)) {
            *letter = tolower(*letter);
      } else if (islower(*letter)) {
            *letter = toupper(*letter);
      }
}

int main(int argc, char *argv[]) {
      char ch = 'g';
      flipCase(&ch);
      printf("%c", ch);     // want this to print ‘G’
}
```

We are modifying a specific instance of the letter, so we pass the location of the letter we would like to modify.

Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

Pointers Summary

- **Tip:** setting a function parameter equal to a new value usually doesn't do what you want. Remember that this is setting the function's *own copy* of the parameter equal to some new value.

```
void doubleNum(int x) {
    x = x * x;     // modifies doubleNum's own copy!
}

void advanceStr(char *str) {
    str += 2;      // modifies advanceStr's own copy!
}
```

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function `skipSpaces` that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(__?__) {
     ...
}

int main(int argc, char *argv[]) {
     char *str = "    hello";
     skipSpaces(__?__);
     printf("%s", str);        // should print "hello"
}
```

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function `skipSpaces` that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char **strPtr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(&str);
    printf("%s", str);        // should print "hello"
}
```

We are modifying a specific instance of the string pointer, so we pass the location of the string pointer we would like to modify.

Exercise 3

Sometimes, we would like to modify a string's pointer itself, rather than just the characters it points to. E.g. we want to write a function `skipSpaces` that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```
void skipSpaces(char *strPtr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(str);
    printf("%s", str);        // should print "hello"
}
```

This advances `skipSpace`'s own copy of the string pointer, not the instance in main.

Demo: Skip Spaces

skip_spaces.c

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);        // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```

Pointers to Strings

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);       // hi
    return 0;
}
```

Making Copies

```
void skipSpaces(char *strPtr) {
    int numSpaces = strspn(strPtr, " ");
    strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = "  hi";
    skipSpaces(myStr);
    printf("%s\n", myStr);
    return 0;
}
```

```
myFunc myFunc
  //    hi
```

Skip spaces

```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

A.

B.

C.

Skip spaces

```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Arrays

When you declare an array, contiguous memory is allocated on the stack to store the contents of the entire array.

```
char str[6];
strcpy(str, "apple");
```

The array variable (e.g. **`str`**) is not a pointer; it refers to the entire array contents. In fact, **`sizeof`** returns the size of the entire array!

```
int arrayBytes = sizeof(str);    // 6
```

STACK

	Address	Value
		…
	0x105	'\0'
	0x104	'e'
	0x103	'l'
	0x102	'p'
	0x101	'p'
str	0x100	'a'
		…

Arrays

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.

```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

An array's size cannot be changed once you create it; you must create another new array instead.

Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (a pointer) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      myFunc(str);
      ...
}
```

Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element and* passes it (a pointer) to the function.

```
void myFunc(char *myStr) {
      …
}

int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      // equivalent
      char *arrPtr = str;
      myFunc(arrPtr);
      ...
}
```

Arrays as Parameters

This also means we can no longer get the full size of the array using **sizeof**, because now it is just a pointer.

```
void myFunc(char *myStr) {
      int size = sizeof(myStr); // 8
}

int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      int size = sizeof(str);   // 3
      myFunc(str);
      ...
}
```

`sizeof` returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **`sizeof`** to get its full size.

Arrays and Pointers

You can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      char *ptr = str;
      ...
}
```

Arrays and Pointers

You can also make a pointer equal to an array; it will point to the first element in that array.

```
int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      char *ptr = str;

      // equivalent
      char *ptr = &str[0];

      // equivalent, but avoid
      char *ptr = &str;
      ...
}
```

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

This stores 5 **char *s**, *not* all of the characters for 5 strings!

```
char *str0 = stringArray[0];     // first char *
```

Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```

Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```

What is the value of `argv[2]` in this diagram?

Lecture Plan

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Pointer Arithmetic

When you do pointer arithmetic, you are adjusting the pointer by a certain *number of places* (e.g. characters).

```
char *str = "apple";    // e.g. 0xff0
char *str1 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

printf("%s", str);      // apple
printf("%s", str1);     // pple
printf("%s", str3);     // le
```

DATA SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …                // e.g. 0xff0
int *nums1 = nums + 1;       // e.g. 0xff4
int *nums3 = nums + 3;       // e.g. 0xffc

printf("%d", *nums);         // 52
printf("%d", *nums1);        // 23
printf("%d", *nums3);        // 34
```

STACK

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

Pointer Arithmetic

When you use bracket notation with a pointer, you are actually *performing pointer arithmetic and dereferencing*:

```
char *str = "apple";// e.g. 0xff0

// both of these add two places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff2.
char thirdLetter = str[2];          // 'p'
char thirdLetter = *(str + 2);      // 'p'
```

DATA SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

Pointer Arithmetic

Pointer arithmetic with two pointers does *not* give the byte difference. Instead, it gives the number of places they differ by.

```
// nums points to an int array
int *nums = …                    // e.g. 0xff0
int *nums3 = nums + 3;           // e.g. 0xffc
int diff = nums3 - nums;         // 3
```

STACK

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

String Behavior #6: Adding an offset to a C string gives us a substring that many places past the first character.

Pointer Arithmetic

How does the code know how many bytes it should look at once it visits an address?

```
int x = 2;
int *xPtr = &x;          // e.g. 0xff0

// How does it know to print out just the 4 bytes at xPtr?
printf("%d", *xPtr);    // 2
```

Pointer Arithmetic

How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[6];
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Pointer Arithmetic

- At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.
- For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.

Pointer arithmetic

Array indexing is "syntactic sugar" for pointer arithmetic:

```
ptr + i        ⇔        &ptr[i]
*(ptr + i)     ⇔        ptr[i]
```

⚠️ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)`.

- This means too-large/negative subscripts will compile ☺

```
arr[99]                    arr[-1]
```

- You can use either syntax on either pointer or array.

Extra Slides

1. Pointer arithmetic

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

- Will there be a compile error/segfault?
- If no errors, what is printed?

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

1. Pointer arithmetic

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

What happens if we call `strncpy(buf, str, 5);`?

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

dest	
src	
n	5
i	

What happens if we call `strncpy(buf, str, 5);`?

3. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

What is stored in each variable? (We cover double pointers later in the course)

3. Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref    : %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```

While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

Recap

- Strings in Memory (cont'd.)
- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic

Next Time: dynamically allocated memory

COMP201

Computer Systems & Programming

Lecture #8 – The Stack and The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers

Pointers Practice

* Wars: Episode I (of 2)

Review

In variable **declaration**, * creates a **pointer**.

```
char ch = 'r';
```

ch stores a char

```
char *cptr = &ch;
```

cptr stores an address of a char (**points to** a char)

```
char **strptr = &cptr;
```

strptr stores an address of a char * (**points to** a char *)

* Wars: Episode II (of 2)

Review

In <u>reading values from/storing values</u>, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;

char **strptr = &cptr;
```

Increment value stored in `ch`

* Wars: Episode II (of 2)

Review

In <u>reading values from/storing values</u>, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in `ch`

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in `cptr` (increment char **pointed to**)

```
char **strptr = &cptr;
```

* Wars: Episode II (of 2)

Review

In <u>reading values from/storing values</u>, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in `ch`

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in `cptr` (increment char **pointed to**)

```
char **strptr = &cptr;
*strptr = *strptr + 1;
```

Increment value stored at memory address in `cptr` (increment address **pointed to**)

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

Plan for Today

- The Stack
- The Heap and Dynamic Memory
- `realloc`

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- `realloc`

Memory Layout

- We are going to dive deeper into different areas of memory used by our programs.
- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.
- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The Stack

- The stack behaves like a…well…stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.
- *Interesting fact:* C does not clear out memory when a function's frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!
- A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.
- What are the limitations of the stack?

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Problem: local variables go away when a function finishes. These characters will thus no longer exist, and the address will be for unknown memory!

The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Stacked Against Us

This is a problem! We need a way to have memory that doesn't get cleaned up when a function exits.

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- `realloc`

The Heap

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Us: hey C, is there a way to make this variable in memory that isn't automatically cleaned up?

The Heap

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

C: sure, but since I don't know when to clean it up anymore, it's your responsibility...

The Heap

- The **heap** is a part of memory that you can manage yourself.
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.

malloc

```
void *malloc(size_t size);
```

To allocate memory on the heap, use the **`malloc`** function ("memory allocate") and specify the number of bytes you'd like.

- This function returns a pointer to *the **starting address** of the new memory*. It doesn't know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn't enough memory for this request.

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Exercise: `malloc` multiples

Let's write a function that returns an array of the first **len** multiples of **mult**.

```
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

Line 2: How should we declare `arr`?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(sizeof(int) * len);`
D. `int *arr = malloc(sizeof(int) * (len + 1));`
E. Something else

Exercise: `malloc` multiples

Let's write a function that returns an array of the first **len** multiples of **mult**.

```
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- Use a pointer to store the address returned by malloc.
- Malloc's argument is **the number of bytes** to allocate.

⚠️ **This code is missing an assertion.**

Line 2: How should we declare `arr`?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(sizeof(int) * len);`
D. `int *arr = malloc(sizeof(int) * (len + 1));`
E. Something else

Always `assert` with the heap

Let's write a function that returns an array of the first **len** multiples of **mult**.

```
1 int *array_of_multiples(int mult, int len) {
2     int *arr = malloc(sizeof(int) * len);
3     assert(arr != NULL);
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- If an allocation error occurs (e.g. out of heap memory!), malloc will return NULL. This is an important case to check **for robustness**.
- **assert** will crash the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program.

Other heap allocations: `calloc`

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- `calloc` is more expensive than `malloc` because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

```
char *str = strdup("Hello, world!"); // on heap
str[0] = 'h';
```

Implementing strdup

How can we implement **strdup** using functions we've already seen?

```
char *myStrdup(char *str) {
    char *heapStr = malloc(strlen(str) + 1);
    assert(heapStr != NULL);
    strcpy(heapStr, str);
    return heapStr;
}
```

Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

`free` details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);
…
free(ptr);
```

❌ Memory at this address was already freed!

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);
…
free(ptr + 1);
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");
…
free(str);    // our responsibility to free!
```

Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.
- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!

However, memory leaks rarely (if ever) cause crashes.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks!

More on Valgrind in Lab 3!

`free` Practice

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
printf("%s\n", str);
```

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
    free(num);
}
printf("%s\n", str);
free(str);
```

Demo: Pig Latin

pig_latin.c

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- `realloc`

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

`realloc`

- `realloc` only accepts pointers that were previously returned my `malloc`/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc` —the previous (smaller) one was already reclaimed by `realloc`.

```
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Heap allocator analogy: A hotel

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you...

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes...
- Return to room after checkout?

Demo: Pig Latin Part 2

pig_latin.c

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Compare and contrast the heap memory functions we've learned about.

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check with `assert`
- Memory is contiguous; it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location.
- If you `realloc`/`free` non-heap address

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation;
  declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation; declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful.**
  Can provide more memory on demand!
- **Very flexible.**
  Runtime decisions about how much/when to allocate, can resize easily with realloc
- **Scope under programmer control**
  Can precisely determine lifetime

⚠️ **Lots of opportunity for error**
Low type safety, forget to allocate/free before done, allocate wrong size, etc., Memory leaks (much less critical)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.
- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Recap

- The Stack
- The Heap and Dynamic Memory
- `realloc`

Next time: C Generics – `void *`

COMP201

Computer Systems & Programming

Lecture #9 – `realloc`, Memory Bugs

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Early course feedback form available online on the web.
  - Your feedback is much appreciated!

Recap

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Recap: Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (a pointer) to the function.

```
void myFunc(char *myStr) {
      ...
}

int main(int argc, char *argv[]) {
      char str[3];
      strcpy(str, "hi");
      myFunc(str);
      ...
}
```

Recap: Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array ele*

fu

vo

}

in

```
    char str[3];
    strcpy(str, "hi");
    myFunc(str);
    ...
}
```

myFunc()

`sizeof` returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **`sizeof`** to get its full size.

Recap: Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

This stores 5 **char *s**, *not* all of the characters for 5 strings!

```
char *str0 = stringArray[0];     // first char *
```

Recap: Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …                // e.g. 0xff0
int *nums1 = nums + 1;       // e.g. 0xff4
int *nums3 = nums + 3;       // e.g. 0xffc

printf("%d", *nums);         // 52
printf("%d", *nums1);        // 23
printf("%d", *nums3);        // 34
```

STACK

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

Recap: Pointer Arithmetic

How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[6];
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Recap: Pointer Arithmetic

How does the code know how many bytes it should add when performing

At compile time, C can figure out the sizes of different data types, and the sizes of what they point to. Hence, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type

```
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Recap: Pointer arithmetic

Array indexing is "syntactic sugar" for pointer arithmetic:

```
ptr + i        ⇔        &ptr[i]
*(ptr + i)     ⇔        ptr[i]
```

⚠️ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)`.

- This means too-large/negative subscripts will compile ☺

```
arr[99]                    arr[-1]
```

- You can use either syntax on either pointer or array.

Recap: The Stack

- We are going to dive deeper into different areas of memory used by our programs.
- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.
- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
}

int m
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Problem: local variables go away when a function finishes. These characters will thus no longer exist, and the address will be for unknown memory!

Recap: The Heap

- The **heap** is a part of memory that you can manage yourself.
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.

Recap: `malloc`

```
void *malloc(size_t size);
```

To allocate memory on the heap, use the **`malloc`** function ("memory allocate") and specify the number of bytes you'd like.

- This function returns a pointer to *the **starting address** of the new memory*. It doesn't know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`** means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn't enough memory for this request.

Recap: The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Recap: `calloc`

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- `calloc` is more expensive than `malloc` because it zeros out memory. Use only when necessary!

Recap: `strdup`

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

```
char *str = strdup("Hello, world!"); // on heap
str[0] = 'h';
```

Recap: Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

Recap: Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *str = strdup("Hello!");

…

free(str);     // our responsibility to free!
```

Recap: `free` details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);
…
free(ptr);
```

Memory at this address was already freed!

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);
…
free(ptr + 1);
```

Recap: Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.
- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!

However, memory leaks rarely (if ever) cause crashes.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks!

Plan for Today

- Practice: Pig Latin
- `realloc`
- Memory bugs

Lecture Plan

- Practice: Pig Latin
- `realloc`
- Memory bugs

Demo: Pig Latin

pig_latin.c

Lecture Plan

- Practice: Pig Latin
- `realloc`
- Memory bugs

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **`realloc`** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **`realloc`** simply adds that space to the allocation.
- If there is not enough space, **`realloc`** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

`realloc`

- `realloc` only accepts pointers that were previously returned my `malloc`/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc` —the previous (smaller) one was already reclaimed by `realloc`.

```
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Heap allocator analogy: A hotel

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you...

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes...
- Return to room after checkout?

Demo: Pig Latin Part 2

pig_latin.c

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Compare and contrast the heap memory functions we've learned about.

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check with `assert`
- Memory is contiguous; it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location.
- If you `realloc`/`free` non-heap address

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation;
  declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation; declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful.**
  Can provide more memory on demand!
- **Very flexible.**
  Runtime decisions about how much/when to allocate, can resize easily with realloc
- **Scope under programmer control**
  Can precisely determine lifetime

⚠️ **Lots of opportunity for error**
Low type safety, forget to allocate/free before done, allocate wrong size, etc., Memory leaks (much less critical)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.
- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Lecture Plan

- Practice: Pig Latin
- `realloc`
- Memory bugs

Pointers and Working with Dynamic Memory

Here are some common errors and mistakes that may happen if you're not careful enough:

- storage used after free,
- allocation freed repeatedly,
- insufficient space for a dynamically allocated variable,
- freeing unallocated storage,
- freeing of the stack space,
- memory leakage,
- assignment of incompatible types,
- returning (directly or via an argument) of a pointer to a local variable,
- dereference of wrong type,
- dereference of uninitialized or invalid pointer,
- incorrect use of pointer arithmetic,
- array index out of bounds

Exercise 1

```
void myfunc(int *arr) {
  int *p_arr = (int*) malloc(2*sizeof(int));
  p_arr[0] = 42;
  p_arr[1] = 24;
  arr = p_arr;
}

int main(int argc, char *argv[]) {
  int *arr = NULL;
  myfunc(arr);
  printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
  free(arr);
  return 0;
}
```

Exercise 1

```
void myfunc(int *arr) {
  int *p_arr = (int*) malloc(2*sizeof(int));
  p_arr[0] = 42;
  p_arr[1] = 24;
  arr = p_arr;
}

int main(int argc, char *argv[]) {
  int *arr = NULL;
  myfunc(arr);
  printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
  free(arr);
  return 0;
}
```

1. dereference of uninitialized or invalid pointer: `arr` in `main` is still `NULL`

Exercise 1

```
void myfunc(int *arr) {
  int *p_arr = (int*) malloc(2*sizeof(int));
  p_arr[0] = 42;
  p_arr[1] = 24;
  arr = p_arr;
}

int main(int argc, char *argv[]) {
  int *arr = NULL;
  myfunc(arr);
  printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
  free(arr);
  return 0;
}
```

2. freeing unallocated storage!

Exercise 2

```
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}

int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

Exercise 2

```
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}

int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

1. insufficient space for a dynamically allocated variable: `malloc` should use `sizeof(int*)`

Exercise 2

```
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}

int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

2. dereference of uninitialized or invalid pointer: `array` in `main` is still `NULL`

Exercise 3

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != 0)
    ;

  strcat(ptr+strlen(param1)+1, param2);
  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

Exercise 3

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != 0)
    ;

  strcat(ptr+strlen(param1)+1, param2);
  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

1. Dereference of invalid pointer: `strcat` could not find end of `dest`

Exercise 3

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != 0)
    ;

  strcat(ptr+strlen(param1)+1, param2);
  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

2. memory leakage: `ptr = NULL;` should be `free(ptr);`

Exercise 4

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(a);

  while ((*ptr++ = *param2++) != 0)
    ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

Exercise 4

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(a);

  while ((*ptr++ = *param2++) != 0)
    ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

1. memory leakage: `ptr = NULL;` should be `free(ptr);`

Exercise 4

```
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(a);

  while ((*ptr++ = *param2++)
    ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

2. memory leakage:
   `ptr+=strlen(a);`
   no way to free memory originally pointed by `ptr`

Recap

- Practice: Pig Latin
- `realloc`
- Memory bugs

Next time: C Generics – `void *`

photo by unsplash user @servuspaul

COMP201

Computer Systems & Programming

Lecture #10 – C Generics – `void *`

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

https://forms.gle/3t9zS56g1weh7CUu8

Recap: Heap allocation interface:

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check with `assert`
- Memory is contiguous; it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location.
- If you `realloc`/`free` non-heap address

Recap: The Stack vs The Heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation;
  declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

Recap: The Stack vs The Heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation; declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful.**
  Can provide more memory on demand!
- **Very flexible.**
  Runtime decisions about how much/when to allocate, can resize easily with realloc
- **Scope under programmer control**
  Can precisely determine lifetime

⚠️ **Lots of opportunity for error**
Low type safety, forget to allocate/free before done, allocate wrong size, etc., Memory leaks (much less critical)

COMP201 Topic 5: How can we use our knowledge of memory and data representation to write code that works with any data type?

Learning Goals

- Learn how to write C code that works with any data type.
- Learn about how to use `void *` and avoid potential pitfalls.

Plan for Today

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture plan

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Generics

- We always strive to write code that is as general-purpose as possible.
- Generic code reduces code duplication and means you can make improvements and fix bugs in one place rather than many.
- Generics is used throughout C for functions to sort any array, search any array, free arbitrary memory, and more.
- How can we write generic code in C?

Lecture Plan

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Stack

	Address	Value
		…
x	0xff14	2
y	0xff10	5
		…

main()

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Stack

	Address	Value
		…
main() x	0xff14	5
main() y	0xff10	2
		…

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Stack

	Address	Value
		…
main() x	0xff14	5
main() y	0xff10	2
		…

Swap

You're asked to write a function that swaps two numbers.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Stack

	Address	Value
		…
main() x	0xff14	5
main() y	0xff10	2
		…

“Oh, when I said ’numbers’ I meant `shorts`, not `ints`.”

Swap

```
void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    short x = 2;
    short y = 5;
    swap_short(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swap

```
void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    short x = 2;
    short y = 5;
    swap_short(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

“You know what, I goofed. We’re going to use strings. Could you write something to swap those?”

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

Swap

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

“Awesome! Thanks.”

“Awesome! Thanks. We also have 20 custom `struct` types. Could you write swap for those too?”

“Awesome! Thanks. We also have 20 custom `struct` types. Could you write swap for those too?”

A user-defined structured data type in C (will be covered next week)

Generic Swap

What if we could write *one* function to swap two values of any single type?

```
void swap_int(int *a, int *b) { … }
void swap_float(float *a, float *b) { … }
void swap_size_t(size_t *a, size_t *b) { … }
void swap_double(double *a, double *b) { … }
void swap_string(char **a, char **b) { … }
void swap_mystruct(mystruct *a, mystruct *b) { … }
…
```

Generic Swap

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
```

Generic Swap

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
```

All 3:

- Take pointers to values to swap
- Create temporary storage to store one of the values
- Move data at **b** into where **a** points
- Move data in temporary storage into where **b** points

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

`int temp = *data1ptr;`	4 bytes
`short temp = *data1ptr;`	2 bytes
`char *temp = *data1ptr;`	8 bytes

Problem: each type may need a different size temp!

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

```
*data1Ptr = *data2ptr;
```

4 bytes

```
*data1Ptr = *data2ptr;
```

2 bytes

```
*data1Ptr = *data2ptr;
```

8 bytes

Problem: each type needs to copy a different amount of data!

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

`*data2ptr = temp;`	4 bytes
`*data2ptr = temp;`	2 bytes
`*data2ptr = temp;`	8 bytes

Problem: each type needs to copy a different amount of data!

C knows the size of temp, and knows how many bytes to copy, because of the variable types.

Is there a way to make a version that doesn’t care about the variable types?

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

If we don't know the data type, we don't know how many bytes it is. Let's take that as another parameter.

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

If we don't know the data type, we don't know how many bytes it is. Let's take that as another parameter.

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Let's start by making space to store the temporary value. How can we make **`nbytes`** of temp space?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    void temp; ???
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Let's start by making space to store the temporary value. How can we make **`nbytes`** of temp space?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

temp is **nbytes** of memory, since each **char** is 1 byte!

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Now, how can we copy in what **`data1ptr`** points to into **`temp`**?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Now, how can we copy in what **`data1ptr`** points to into **`temp`**?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

We can't dereference a **`void *`** (or set an array equal to something). C doesn't know what it points to! Therefore, it doesn't know how many bytes there it should be looking at.

memcpy

memcpy is a function that copies a specified amount of bytes at one address to another address.

```
void *memcpy(void *dest, const void *src, size_t n);
```

`const` is a type qualifier which indicates that the data is read only (will be covered next week)

memcpy

memcpy is a function that copies a specified amount of bytes at one address to another address.

```
void *memcpy(void *dest, const void *src, size_t n);
```

It copies the next n bytes that src points to to the location contained in dest. (It also returns **dest**). It does not support regions of memory that overlap.

memcpy must take **pointers** to the bytes to work with to know where they live and where they should be copied to.

```
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x));  // like x = y
```

memmove

memmove is the same as memcpy, but supports overlapping regions of memory. (Unlike its name implies, it still "copies").

```
void *memmove(void *dest, const void *src, size_t n);
```

It copies the next n bytes that src <u>points to</u> to the location contained in dest. (It also returns **dest**).

memmove

When might memmove be useful?

1	2	3	4	5	6	7

4	5	6	7	5	6	7

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

We can't dereference a **`void *`**. C doesn't know what it points to! Therefore, it doesn't know how many bytes there it should be looking at.

Generic Swap

slido

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

How can **memcpy** or **memmove** help us here?

```
void *memcpy(void *dest, const void *src, size_t n);

void *memmove(void *dest, const void *src, size_t n);
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

We can copy the bytes ourselves into temp! This is equivalent to **`temp = *data1ptr`** in non-generic versions, but this works for *any* type of *any* size.

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

How can we copy data2 to the location of data1?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    *data1ptr = *data2ptr; ???
    // copy data in temporary storage to location of data2
}
```

How can we copy data2 to the location of data1?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
}
```

How can we copy data2 to the location of data1?
memcpy!

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
}
```

How can we copy temp's data to the location of data2?

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

How can we copy temp's data to the location of data2? **memcpy**!

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

```
int x = 2;
int y = 5;
swap(&x, &y, sizeof(x));
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

```
short x = 2;
short y = 5;
swap(&x, &y, sizeof(x));
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

```
char *x = "2";
char *y = "5";
swap(&x, &y, sizeof(x));
```

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

```
mystruct x = {…};
mystruct y = {…};
swap(&x, &y, sizeof(x));
```

C Generics

- We can use **`void *`** and **`memcpy`** to handle memory as generic bytes.
- If we are given where the data of importance is, and how big it is, we can handle it!

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes)
{
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

Lecture Plan

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Void * Pitfalls

- **`void *`**s are powerful, but dangerous - C cannot do as much checking!
- E.g. with **`int`**, C would never let you swap *half* of an int. With **`void *s`**, this can happen! (*How? Let's find out!)*

Demo: Void *s Gone Wrong

`swap.c`

Void *Pitfalls

- Void * has more room for error because it manipulates arbitrary bytes without knowing what they represent. This can result in some strange memory Frankensteins!

http://i.ytimg.com/vi/10gPoYjq3EA/hqdefault.jpg

Mid-Lecture Check-In

We can now answer the following questions:

1. What variable type represents a “generic pointer”?
2. What variable type can we use to create a specific number of bytes of space on the stack?
3. How can we copy generic memory from one location to another?
4. What is the difference between **`memcpy`** and **`memmove`**?
5. What are the benefits of generic functions in C? What are the challenges?

Lecture Plan

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers.

```
void swap_ends_int(int *arr, size_t nelems) {
    int tmp = arr[0];
    arr[0] = arr[nelems - 1];
    arr[nelems - 1] = tmp;
}

int main(int argc, char *argv[]) {
    int nums[] = {5, 2, 3, 4, 1};
    size_t nelems = sizeof(nums) / sizeof(nums[0]);
    swap_ends_int(nums, nelems);
    // want nums[0] = 1, nums[4] = 5
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);
    return 0;
}
```

Wait – we just wrote a generic swap function. Let's use that!

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers.

```
void swap_ends_int(int *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

int main(int argc, char *argv[]) {
    int nums[] = {5, 2, 3, 4, 1};
    size_t nelems = sizeof(nums) / sizeof(nums[0]);
    swap_ends_int(nums, nelems);
    // want nums[0] = 1, nums[4] = 5
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);
    return 0;
}
```

Wait – we just wrote a generic swap function. Let's use that!

Swap Ends

Let's write out what some other versions would look like (just in case).

```
void swap_ends_int(int *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_short(short *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_string(char **arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_float(float *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

The code seems to be the same regardless of the type!

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

Is this generic? Does this work?

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

Is this generic? Does this work?

Unfortunately not. First, we no longer know the element size. Second, pointer arithmetic depends on the type of data being pointed to. With a void *, we lose that information!

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

We need to know the element size, so let's add a parameter.

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + nelems - 1, elem_bytes);
}
```

We need to know the element size, so let's add a parameter.

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

`Int?`

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

`Int:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(int)` = 12 bytes

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

`Int:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(int)` = 12 bytes

`Short?`

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

`Int:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(int)` = 12 bytes

`Short:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(short)` = 6 bytes

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

`Int:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(int)` = 12 bytes

`Short:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(short)` = 6 bytes

`Char *:` adds 3 <u>places</u> to `arr`, and 3 * `sizeof(char *)` = 24 bytes

In each case, we need to know the element size to do the arithmetic.

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + nelems - 1, elem_bytes);
}
```

How many bytes past **arr** should we go to get to the last element?

(nelems - 1) * elem_bytes

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

How many bytes past **arr** should we go to get to the last element?

(nelems - 1) * elem_bytes

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

But C still can't do arithmetic with a `void*`. We need to tell it to not worry about it, and just add bytes. **How can we do this?**

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

But C still can't do arithmetic with a void*. We need to tell it to not worry about it, and just add bytes. **How can we do this?**

`char *` pointers already add bytes!

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
short nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
char *strs[] = {"Hi", "Hello", "Howdy"};
size_t nelems = sizeof(strs) / sizeof(strs[0]);
swap_ends(strs, nelems, sizeof(strs[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
mystruct structs[] = …;
size_t nelems = …;
swap_ends(structs, nelems, sizeof(structs[0]));
```

Demo: Void *s Gone Wrong

swap_ends.c

Void * Pitfalls

- **`void *`**s are powerful, but dangerous - C cannot do as much checking!
- E.g. with **`int`**, C would never let you swap *half* of an int. With **`void *`**s, this can happen!

```
int x = 0xffffffff;
int y = 0xeeeeeeee;
swap(&x, &y, sizeof(short));

// now x = 0xffffeeee, y = 0xeeeeffff!
printf("x = 0x%x, y = 0x%x\n", x, y);
```

Recap

- **`void *`** is a variable type that represents a generic pointer "to something".
- We cannot perform pointer arithmetic with or dereference a **`void *`**.
- We can use **`memcpy`** or **`memmove`** to copy data from one memory location to another.
- To do pointer arithmetic with a **`void *`**, we must first cast it to a **`char *`**.
- **`void *`** and generics are powerful but dangerous because of the lack of type checking, so we must be extra careful when working with generic memory.

Recap

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap
- Generic Array Rotation

Next time: Function Pointers

photo by unsplash user @servuspaul

COMP201

Computer Systems & Programming

Lecture #11 – Function Pointers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Overview: Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

Learning Goals

- Learn how to write C code that works with any data type.
- Learn how to pass functions as parameters
- Learn how to write functions that accept functions as parameters

Plan for Today

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Generics So Far

- **`void *`** is a variable type that represents a generic pointer “to something”.
- We cannot perform pointer arithmetic with or dereference a **`void *`**.
- We can use **`memcpy`** or **`memmove`** to copy data from one memory location to another.
- To do pointer arithmetic with a **`void *`**, we must first cast it to a **`char *`**.
- **`void *`** and generics are powerful but dangerous because of the lack of type checking, so we must be extra careful when working with generic memory.

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

We can use **`void *`** to represent a pointer to any data, and **`memcpy/memmove`** to copy arbitrary bytes.

Generic Array Swap

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

We can cast to a **char *** in order to perform manual byte arithmetic with void * pointers.

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
short nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
char *strs[] = {"Hi", "Hello", "Howdy"};
size_t nelems = sizeof(strs) / sizeof(strs[0]);
swap_ends(strs, nelems, sizeof(strs[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```
mystruct structs[] = …;
size_t nelems = …;
swap_ends(structs, nelems, sizeof(structs[0]));
```

Demo: `void *`s Gone Wrong

`swap_ends.c`

Array Rotation

You're asked to provide an implementation for a function called **`rotate`** with the following prototype:

```
void rotate(void *front, void *separator, void *end);
```

The expectation is that **`front`** is the base address of an array, **`end`** is the past-the-end address of the array, and **`separator`** is the address of some element in between. **`rotate`** moves all elements in between **`front`** and **`separator`** to the end of the array, and all elements between **`separator`** and **`end`** move to the front.

Array Rotation

```
int array[7] = {1, 2, 3, 4, 5, 6, 7};
rotate(array, array + 3, array + 7);
```

Array Rotation

A properly implemented **rotate** will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_int_array(array, 10); // intuitive implementation ☺
    rotate(array, array + 5, array + 10);
    print_int_array(array, 10);
    rotate(array, array + 1, array + 10);
    print_int_array(array, 10);
    rotate(array + 4, array + 5, array + 6);
    print_int_array(array, 10);
    return 0;
}
```

```
Output:
linuxpool :~/lect10$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
linuxpool:~/lect10$
```

Array Rotation

A properly implemented **`rotate`** will prompt the following program to generate the provided output.

And here's that properly implemented function!

```
void rotate(void *front, void *separator, void *end) {
    int width = (char *)end - (char *)front;
    int prefix_width = (char *)separator - (char *)front;
    int suffix_width = width - prefix_width;

    char temp[prefix_width];
    memcpy(temp, front, prefix_width);
    memmove(front, separator, suffix_width);
    memcpy((char *)end - prefix_width, temp, prefix_width);
}
```

Demo: Array Rotation

rotate.c

`void *` Pitfalls

- **`void *`**s are powerful, but dangerous - C cannot do as much checking!
- E.g. with **`int`**, C would never let you swap *half* of an int. With **`void *`**s, this can happen!

```
int x = 0xffffffff;
int y = 0xeeeeeeee;
swap(&x, &y, sizeof(short));

// now x = 0xffffeeee, y = 0xeeeeffff!
printf("x = 0x%x, y = 0x%x\n", x, y);
```

memset

memset is a function that sets a specified number of bytes at one address to a certain value.

```
void *memset(void *s, int c, size_t n);
```

It fills **n** bytes starting at memory location **s** with the byte **c**. (It also returns **s**).

```
int counts[5];
memset(counts, 0, 3);     // zero out first 3 bytes at counts
memset(counts + 3, 0xff, 4)    // set 3rd entry's bytes to 1s
```

Why are `void *` pointers useful?

Because each parameter and return type must be a *single type* with a *single size*.

Why Are `void *` Pointers Useful?

- Each parameter and return type must be a *single* type with a *single* size.
- **Problem #1:** for a function parameter to accept multiple data types, it needs to be able to accept data of different sizes.
  - **Key Idea #1:** pointers are all the same size regardless of what they point to. To pass different sizes of data via a single parameter type, make the parameter be a pointer to the data instead.
- **Problem #2:** we still might pass either a `char *`, `int *`, etc. These are the same size, but still different declared types. What should the parameter type be?
  - **Key Idea #2:** A `void *` encompasses all these types – it represents a "pointer to something". A `char *`, `int *`, etc. all implicitly cast to `void *`.
- **Solution:** to pass one of multiple types via a single parameter/return, that parameter/return's type can be **`void *`**, and we can pass a pointer to the data.

Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

4	2	12	-5	56	14

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

- Bubble sort repeatedly goes through the array, swapping any pairs of elem
the a

In general, bubble sort requires up to n - 1 passes to sort an array of length n, though it may end sooner if a pass doesn't swap anything.

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

-5	2	4	12	14	56

- Bubble sort repeatedly goes through the array, swapping any pairs of ele
the

Only two more passes are needed to arrive at the above. The first exchanges the 2 and the -5, and the second leaves everything as is.

Integer Bubble Sort

```
void bubble_sort_int(int *arr, int n) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap_int(&arr[i - 1], &arr[i]);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

How can we make this function generic, to sort an array of *any type*?

Integer Bubble Sort

```
void bubble_sort_int(int *arr, int n) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap_int(&arr[i - 1], &arr[i]);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters and swap generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                swapped = true;
                swap(&arr[i - 1], &arr[i], elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters and swap generic.

Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array. From last lecture, we know how to locate the **last** element:

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

How can we generalize this to get the location of the i-th element?

```
void *ith_elem = (char *)arr + i * elem_bytes;
```

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters and swap generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Wait a minute…this doesn't work! We can't dereference **void** *s OR compare any element with >, since they may not be numbers!

A Generics Conundrum

- We've hit a snag – there is no way to generically compare elements. They could be any type and have complex ways to compare them.
- How can we write code to compare *any two elements of the same type?*
- That's not something that bubble sort can ever know how to do. **BUT** – our caller should know how to do this, because they're supplying the data....let's ask them!

Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

bubble_sort (inner voice): hey, you, person who called us. Do you know how to compare the items at these two addresses?

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Caller: yeah, I know how to compare them. You don't know what data type they are, but I do. I have a function that can do the comparison for you and tell you the result.

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                                        function compare_fn) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

How can we compare these elements? They can pass us this **function as a parameter**. The function's job is to tell us how two elements compare.

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

How can we compare these elements? They can pass us this **function as a parameter**. The function's job is to tell us how two elements compare.

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Return type
(`bool`)

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function pointer name
(`compare_fn`)

Function Pointers

A function pointer is the variable type for passing a function as a parameter. Here is how the parameter's type is declared.

```
bool (*compare_fn)(void *a, void *b)
```

Function parameters
(two `void *`s)

Function Pointers

Here's the general variable type syntax:

```
[return type] (*[name])([parameters])
```

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Function Pointers

```
bool integer_compare(void *ptr1, void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort(nums, nums_count, sizeof(nums[0]), integer_compare);
    ...
}
```

`bubble_sort` is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

Function Pointers

```
bool string_compare(void *ptr1, void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    char *classes[] = {"COMP100", "COMP132", "COMP201", "COMP202"};
    int arr_count = sizeof(classes) / sizeof(classes[0]);
    bubble_sort(classes, arr_count, sizeof(classes[0]), string_compare);
    ...
}
```

`bubble_sort` is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*compare_fn)(void *a, void *b))
```

- Bubble Sort is written as a generic library function to be imported into potentially many programs to be used with many types. It must have a single function signature but work with any type of data.
- Its comparison function type is part of its function signature – the comparison function signature must use one set of types but accept any data of any size. How do we do this?
  - **The function will instead accept pointers to the data via `void *` parameters**
  - This means that the functions must be written to handle parameters which are *pointers to the data* to be compared

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_el
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Function Pointers

This means that functions with generic parameters must always take *pointers to the data they care about.*

We can use the following pattern:

1) Cast the `void *`argument(s) and set typed pointers equal to them.
2) Dereference the typed pointer(s) to access the values.
3) Perform the necessary operation.

(steps 1 and 2 can often be combined into a single step)

Function Pointers

```
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *s
    int *num1ptr = (int *)ptr1;
    int *num2ptr = (int *)ptr2;

    // 2) dereference typed points to access values
    int num1 = *num1ptr;
    int num2 = *num2ptr;

    // 3) perform operation
    return num1 > num2;
}
```

This function is created by the caller specifically to compare integers, knowing their addresses are necessarily disguised as `void *`so that **`bubble_sort`** can work for any array type.

Function Pointers

```
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *s
    int *num1ptr = (int *)ptr1;
    int *num2ptr = (int *)ptr2;

    // 2) dereference typed points to access values
    int num1 = *num1ptr;
    int num2 = *num2ptr;

    // 3) perform operation
    return num1 > num2;
}
```

However, the type of the comparison function that e.g. **bubble_sort** accepts must be generic, since we are writing one **bubble_sort** function to work with any data type.

Function Pointers

```
bool integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 > *(int *)ptr2;
}
```

Comparison Functions

- Function pointers are used often in cases like this to compare two values of the same type. These are called **comparison functions**.
- The standard comparison function in many C functions provides even more information. It should return:
  - < 0 if first value should come before second value
  - > 0 if first value should come after second value
  - 0 if first value and second value are equivalent
- This is the same return value format as **`strcmp`**!

```
int (*compare_fn)(void *a, void *b)
```

Comparison Functions

```
int integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 - *(int *)ptr2;
}
```

Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 int (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem) > 0) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Comparison Functions

- **Exercise:** how can we write a comparison function for bubble sort to sort strings in alphabetical order?
- The common prototype provides even more information. It should return:
  - < 0 if first value should come before second value
  - > 0 if first value should come after second value
  - 0 if first value and second value are equivalent

```
int (*compare_fn)(void *a, void *b)
```

String Comparison Function

```
int string_compare(void *ptr1, void *ptr2) {
      // cast arguments and dereference
      char *str1 = *(char **)ptr1;
      char *str2 = *(char **)ptr2;

      // perform operation
      return strcmp(str1, str2);
}
```

Function Pointer Pitfalls

- If a function takes a function pointer as a parameter, it will accept it if it fits the specified signature.
- *This is dangerous*! E.g. what happens if you pass in a string comparison function when sorting an integer array?

Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
  - A function to compare two elements of a given type
  - A function to print out an element of a given type
  - A function to free memory associated with a given type
  - And more...

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
  - A function to compare two elements of a given type
  - A function to print out an element of a given type
  - A function to free memory associated with a given type
  - And more...

Demo: Generic Printing

`print_array.c`

Common Utility Callback Functions

- Comparison function – compares two elements of a given type.

```
int (*cmp_fn)(void *addr1, void *addr2)
```

- Printing function – prints out an element of a given type

```
void (*print_fn)(void *addr)
```

- There are many more! You can specify any functions you would like passed in when writing your own generic functions.

Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Demo: Count Matches

`count_matches.c`

Count Matches

- Let's write a generic function `count_matches` that can count the number of a certain type of element in a generic array.
- It should take in as parameters information about the generic array, and a function parameter that can take in a pointer to a single array element and tell us if it's a match.

```
int count_matches(void *base, int nelems,
          int elem_size_bytes, bool (*match_fn)(void *));
```

Count Matches

```
int count_matches(void *base, int nelems, int elem_size_bytes,
                  bool (*match_fn)(void *)) {

    int match_count = 0;

    for (int i = 0; i < nelems; i++) {
        void *curr_p = (char *)base + i * elem_size_bytes;
        if (match_fn(curr_p)) {
            match_count++;
        }
    }

    return match_count;
}
```

Function Pointers As Variables

In addition to parameters, you can make normal variables that are functions.

```
int do_something(char *str) {
    …
}

int main(int argc, char *argv[]) {
    …
    int (*func_var)(char *) = do_something;
    …
    func_var("testing");
    return 0;
}
```

Generic C Standard Library Functions

- **`qsort`** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.
- **`bsearch`** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **`lfind`** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **`lsearch`** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can't find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

Generic C Standard Library Functions

- **`scandir`** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.

Summary: Function Pointers

- We can pass functions as parameters to pass logic around in our programs.
- Comparison functions are one common class of functions passed as parameters to generically compare the elements at two addresses.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

Summary: Generics

- We use **`void *`** pointers and memory operations like **`memcpy`** and **`memmove`** to make data operations generic.
- We use **function pointers** to make logic/functionality operations generic.

Recap

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Next time: `const`, Structures

photo by unsplash user @xavi_cabrera

COMP201

Computer Systems & Programming

Lecture #12 –`const`, `structs`

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers

Recap: Generics Overview

- We use **`void *`** pointers and memory operations like **`memcpy`** and **`memmove`** to make data operations generic.
- We use **function pointers** to make logic/functionality operations generic.

Recap: Why Are void * Pointers Useful?

- Each parameter and return type must be a *single* type with a *single* size.
- **Problem #1:** for a function parameter to accept multiple data types, it needs to be able to accept data of different sizes.
  - **Key Idea #1:** pointers are all the same size regardless of what they point to. To pass different sizes of data via a single parameter type, make the parameter be a pointer to the data instead.
- **Problem #2:** we still might pass either a `char *`, `int *`, etc. These are the same size, but still different declared types. What should the parameter type be?
  - **Key Idea #2:** A `void *` encompasses all these types – it represents a "pointer to something". A `char *`, `int *`, etc. all implicitly cast to `void *`.
- **Solution:** to pass one of multiple types via a single parameter/return, that parameter/return's type can be **`void *`**, and we can pass a pointer to the data.

Recap: Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 int (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem) > 0) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Recap: Comparison Functions

- Function pointers are used often in cases like this to compare two values of the same type. These are called **comparison functions**.
- The standard comparison function in many C functions provides even more information. It should return:
  - < 0 if first value should come before second value
  - > 0 if first value should come after second value
  - 0 if first value and second value are equivalent
- This is the same return value format as **`strcmp`**!

```
int (*compare_fn)(void *a, void *b)
```

Recap: `integer_compare`

```
int integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 - *(int *)ptr2;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort(nums, nums_count, sizeof(nums[0]), integer_compare);
    ...
}
```

`bubble_sort` is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

Recap: `string_compare`

```
int string_compare(void *ptr1, void *ptr2) {
      return strcmp(*(char **)ptr1, *(char **)ptr2);
}

int main(int argc, char *argv[]) {
    char *classes[] = {"COMP100", "COMP132", "COMP201", "COMP202"};
    int arr_count = sizeof(classes) / sizeof(classes[0]);
    bubble_sort(classes, arr_count, sizeof(classes[0]), string_compare);
    ...
}
```

`bubble_sort` is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

Recap: Function Pointers

- We can pass functions as parameters to pass logic around in our programs.
- Comparison functions are one common class of functions passed as parameters to generically compare the elements at two addresses.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

Plan for Today

- `const`
- `struct`
- Generic stack

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture Plan

- `const`
- `struct`
- Generic stack

const

- Use **const** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
    …
    if (x == DAYS_IN_WEEK) {
        …
    }
    …
}
```

const

- Use **const** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];
strcpy(str, "Hello");
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';
```

const

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}
```

`memcpy` Revisited

`memcpy` is a function that copies a specified amount of bytes at one address to another address.

```
void *memcpy(void *dest, const void *src, size_t n);
```

It copies the next n bytes that src <u>points to</u> to the location contained in dest. (It also returns **dest**). It does <u>not</u> support regions of memory that overlap.

```
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x));  // like x = y
```

const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
      // compiler warning and error
     char *strToModify = str;
     strToModify[0] = …
}
```

const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify. **Think of const as part of the variable type**.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    const char *strToModify = str;
    strToModify[0] = …
}
```

const

const can be confusing to interpret in some variable types.

```
// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = …

// cannot modify chars pointed to by *strPtr
const char **strPtr = …
```

const

```
1    char buf[6];
2    strcpy(buf, "Hello");
3    const char *str = buf;
4    str[0] = 'M';
5    str = "Mello";
6    buf[0] = 'M';
```

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1    char buf[6];
2    strcpy(buf, "Hello");
3    const char *str = buf;
4    str[0] = 'M';
5    str = "Mello";
6    buf[0] = 'M';
```

Line 1 makes a typical modifiable character array of 6 characters.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1    char buf[6];
2    strcpy(buf, "Hello");
3    const char *str = buf;
4    str[0] = 'M';
5    str = "Mello";
6    buf[0] = 'M';
```

Line 2 copies characters into this modifiable character array.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1   char buf[6];
2   strcpy(buf, "Hello");
3   const char *str = buf;
4   str[0] = 'M';
5   str = "Mello";
6   buf[0] = 'M';
```

Line 3 makes a `const` pointer that points to the first element of `buf`. We cannot use `str` to change the characters it points to because it is `const`.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1   char buf[6];
2   strcpy(buf, "Hello");
3   const char *str = buf;
4   str[0] = 'M';
5   str = "Mello";
6   buf[0] = 'M';
```

Line 4 is not allowed – it attempts to use a const pointer to characters to modify those characters.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1   char buf[6];
2   strcpy(buf, "Hello");
3   const char *str = buf;
4   str[0] = 'M';
5   str = "Mello";
6   buf[0] = 'M';
```

Line 5 is ok – `str`'s type means that while you cannot change the characters at which it points, you can change `str` itself to point somewhere else. `str` is not `const` – its characters are.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

```
1  char buf[6];
2  strcpy(buf, "Hello");
3  const char *str = buf;
4  str[0] = 'M';
5  str = "Mello";
6  buf[0] = 'M';
```

Line 6 is ok – `buf` is a modifiable char array, and we can use it to change its characters. Declaring `str` as `const` doesn't mean that place in memory is not modifiable at all – it just means that you cannot modify it using `str`.

Which lines (if any) above will cause an error due to violating `const`? Remember that `const char *` means that the characters at the location it stores cannot be changed.

Lecture Plan

- `const`
- `struct`
- Generic stack

Structs

A **`struct`** is a way to define a new variable type that is a group of other variables.

```
struct date {           // declaring a struct type
    int month;
    int day;            // members of each date structure
};
…

struct date today;                      // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};  // shorter initializer syntax
```

Structs

Wrap the struct definition in a **`typedef`** to avoid having to include the word **`struct`** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```
void advance_day(date d) {
      d.day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(my_date);
      printf("%d", my_date.day); // 28
      return 0;
}
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
      (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
      d->day++;        // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

C allows you to return structs from functions as well. It returns whatever is contained within the struct.

```
date create_new_years_date() {
      date d = {1, 1};
     return d;          // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
      date my_date = create_new_years_date();
      printf("%d", my_date.day); // 1
      return 0;
}
```

Structs

sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;

int main(int argc, char *argv[]) {
      int size = sizeof(date);   // 8
      return 0;
}
```

Arrays of Structs

You can create arrays of structs just like any other variable type.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…

my_struct array_of_structs[5];
```

Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Lecture Plan

- `const`
- `struct`
- Generic stack

Stacks

- C generics are particularly powerful in helping us create generic data structures.
- Let's see how we might go about making a Stack in C.

Stacks

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack

Stacks

A stack is often implemented using a **linked list** internally.

- "bottom" = tail of linked list
- "top" = head of linked list *(why not the other way around?)*

```
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```

Problem: C is not object-oriented! We can't call methods on variables.

Demo: Int Stack

`int_stack.c`

Int Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

Int Stack Functions

- **`int_stack_create()`**: creates a new stack on the heap and returns a pointer to it
- **`int_stack_push(int_stack *s, int data)`**: pushes data onto the stack
- **`int_stack_pop(int_stack *s)`**: pops and returns topmost stack element

int_stack_create

```
int_stack *int_stack_create() {
    int_stack *s = malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

From previous slide:

```
typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

int_stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

int_stack_pop

```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

From previous slide:

```
typedef struct int_stack {
    int nelems;
    node *top;
} int_stack;
```

Pushing Elements to Int Stack

```
int_stack *intstack = int_stack_create();
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    int_stack_push(intstack, i);
}
```

Popping Elements from Int Stack

```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}
```

What modifications are necessary to make a generic stack?

Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

How might we modify the Stack data representation itself to be generic?

Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

Problem: each node can no longer store the data itself, because it could be any size!

Generic Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    void *data;
} int_node;


typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Solution: each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size in the Stack struct.

Stack Functions

- **`int_stack_create()`**: creates a new stack on the heap and returns a pointer to it
- **`int_stack_push(int_stack *s, int data)`**: pushes data onto the stack
- **`int_stack_pop(int_stack *s)`**: pops and returns topmost stack element

int_stack_create

```
int_stack *int_stack_create() {
    int_stack *s = malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Generic `stack_create`

```
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

int_stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

Generic `stack_push`

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Problem 1: we can no longer pass the data itself as a parameter, because it could be any size!

Generic `stack_push`

```
void int_stack_push(int_stack *s, const void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution 1: pass a pointer to the data as a parameter instead.

Generic `stack_push`

```
void int_stack_push(int_stack *s, const void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Problem 2: we cannot copy the existing data pointer into **new_node**. The data structure must manage its own copy that exists for its entire lifetime. The provided copy may go away!

Generic stack_push

```
int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
    // now stack stores pointer to invalid memory for 7!
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```

Generic stack_push

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution 2: make a heap-allocated copy of the data that the node points to.

int_stack_pop

```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

Generic stack_pop

```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

Problem: we can no longer return the data itself, because it could be any size!

Generic stack_pop

```
void *int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    void *value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

While it's possible to return the heap address of the element, this means the client would be responsible for freeing it. Ideally, the data structure should manage its own memory here.

Generic stack_pop

```
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
}
```

Solution: have the caller pass a memory location as a parameter and copy the data to that location.

Using Generic Stack

```
int_stack *intstack = int_stack_create();
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    int_stack_push(intstack, i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
int_stack *intstack = int_stack_create();
int_stack_push(intstack, 7);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));
int num = 7;
stack_push(intstack, &num);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

Using Generic Stack

```
// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

Demo: Generic Stack

generic_stack.c

Recap

- `const`
- `struct`
- Generic stack

Next Time: Compiling C programs

photo by unsplash user @xavi_cabrera

COMP201

Computer Systems & Programming

Lecture #13 – Compiling C Programs

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Assignment 3 is out today! (due November 25, 23:59)
  - Mustafa will announce an extra office hour to answer your questions

Recap

- `const`
- `struct`
- Generic stack

Recap: `const`

- Use **`const`** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
    …
    if (x == DAYS_IN_WEEK) {
        …
    }
    …
}
```

Recap: `const`

- Use **`const`** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];
strcpy(str, "Hello");
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';
```

Recap: `const`

Sometimes we use **`const`** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}
```

Recap: Structs

A **`struct`** is a way to define a new variable type that is a group of other variables.

```
typedef struct date {       // declaring a struct type
    int month;
    int day;            // members of each date structure
} date;
…

date today;                         // construct structure instances
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};  // shorter initializer syntax
```

Recap: Stacks

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack

Int vs. Generic Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;
```

```
typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

```
typedef struct int_node {
    struct int_node *next;
    void *data;
} int_node;
```

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Int vs. Generic `stack_create`

```
int_stack *int_stack_create() {
    int_stack *s =
        malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

```
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Int vs. Generic `stack_push`

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node =
              malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data,data,s->elem_size_bytes);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}


int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

Int vs. Generic stack_pop

```
int int_stack_pop(int_stack *s) {
  if (s->nelems == 0) {
     error(1,0,"Cannot pop from empty stack");
  }
  int_node *n = s->top;
  int value = n->data;

  s->top = n->next;

  free(n);
  s->nelems--;

  return value;
}
```

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

```
void stack_pop(stack *s, void *addr) {
  if (s->nelems == 0)
      error(1,0,"Cannot pop from empty stack");
  node *n = s->top;
  memcpy(addr, n->data, s->elem_size_bytes);
  s->top = n->next;

  free(n->data);
  free(n);
  s->nelems--;
}

int main() {
  stack *intstack = stack_create(sizeof(int));
  for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
  }
  // Pop off all elements
  int popped_int;
  while (intstack->nelems > 0) {
     int_stack_pop(intstack, &popped_int);
     printf("%d\n", popped_int);
  }
}
```

Plan for Today

- What really happens in GCC?
- Make and Makefiles

xkcd.com/303/

Disclaimer: Slides for this lecture were borrowed from
—Gabbi Fisher and Chris Chute's Stanford CS107 class
—Jae Woo Lee's Columbia COMS W3157 class

Lecture Plan

- What really happens in Gnu Compiler Collection (`gcc`)?
  - The Preprocessor
  - The Compiler
  - The Assembler
  - The Linker
- Make and Makefiles

Compiling a C program with GCC

```
gcc -g -O0 hello.c –o hello
```

The GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC)

The Preprocessor

```
#define
```

```
#include
```

The Preprocessor – Object Macros

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);
```

The `#define` directive can be used to set up symbolic replacements in the source.

The Preprocessor – Object Macros

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);

=> foo = (char *) malloc (1024);
```

The Preprocessor – Function Macros

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define twice(X) (2*(X))

y = min(1,2);


y = twice(1+1);
```

The Preprocessor – Function Macros

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define twice(X) (2*(X))

y = min(1,2);
=> y = ((1) < (2) ? (1) : (2));

y = twice(1+1);
=> y = (2*(1+1));
```

The Preprocessor – Imports

```
#include
```

The Preprocessor – Imports

header.h

```
char *test(void)
```

program.c

```
#include "header.h"

int x;

int main(int argc, char *argv[]) {
  puts(test());
}
```

The `#include` directive just pastes in the text from the given file.

The Preprocessor – Imports

header.h

```
char *test(void)
```

program.c

```
char *test(void);

int x;

int main(int argc, char *argv[]) {
  puts(test());
}
```

The Preprocessor – Demo

```
gcc -E -o hello.i hello.c
```

Preprocess `hello.c`, store output in `hello.i`

The GNU Compiler Collection (GCC)

The Compiler

- They're too complicated to explain in 5 minutes.

¯_(ツ)_/¯

- **It's important to know that they parse source code and compile it into assembly code. You will learn more about assembly in the second part the course.**

The Compiler – Demo

```
gcc -S hello.i
```

Compile preprocessed `.i` code into assembly instructions

The GNU Compiler Collection (GCC)

The Assembler – Demo

```
as -o hello.o hello.s
```

Assemble object code from `hello.s`

The Assembler – ELF

a rare depiction of an Elf made by Tolkien

ELF: the Executable and Linkable Format

The Assembler – ELF

ELF: the Executable and Linkable Format

Cross-platform, used across multiple operating systems to represent components (object code) of a program. This comes in handy for linking and execution across different computers.

The Assembler – ELF

ELF: the Executable and Linkable Format

```
readelf -e hello.o
```

Actually read `hello.o`!
"`-e`" flag is for printing headers out only

The Assembler – ELF

Section	Contents	Code Example
`.text`	Executable code (x86 assembly)	`mov -0x8(%rbp),%rax`
`.data`	Any global or static vars that have a pre-defined value and can be modified	`int val = 3` (as global var)
`.rodata`	Variables that are only read (never written)	`const int a = 0;`
`.bss`	All uninitialized data; global variables and static variables initialized to zero or or not explicitly `static int i;` initialized in source code	`static int i;`
`.comment`	Comments about the generated ELF (details such as compiler version and execution platform)	

The Assembler – ELF

The Assembler – ELF

The Assembler

```
nm hello.o
```

Dump the variables and functions in hello and see what sections they belong to!

The Assembler – ELF

The GNU Compiler Collection (GCC)

The Linker – Shared vs. Static Libraries

Static Linking

1. When your program uses static linking, the machine code of external functions used in your program is copied into the executable.
2. A static library has file extension of "`.a`" (archive file) in Unix.

Dynamic Linking

1. When your program is dynamically linked, only an offset table is created in the executable. The operating system loads the machine code needed for external functions during execution —a process known as dynamic linking.
2. A shared library has file extension of "`.so`" (shared objects) in Unix.

The Linker

```
ld --dynamic-linker /lib64/ld-linux-x86-64.so.2 hello.o
   -o  hello -lc --entry main
```

1. **`--dynamic-linker`** is used to specify the linker we must use to load stdlib.
2. **`-lc`** tells the linker to link to the standard C library.
3. **`--entry main`** specifies the entry point of the program (the method “main”).

Note: You may not get this command working, because it will be slightly different on different Linux distributions

Finally…

```
./hello
```

(Run your executable!)

The Executable

```
nm hello
```

Let’s prove to ourselves linking did something…

The Assembler – ELF

Finally… (Really!)

```
./hello
```

(Run your executable!)

Linking Multiple Files and Library

```
gcc -c myfile1.c
gcc -c myfile2.c
gcc -g myfile1.o myfile2.o -lm -o myprogram
```

Using Multiple Functions

program.c

```
int add(int x, int y);

int main(int argc, char **argv)
{
    int sum;
    sum = add(1, 2);
    printf("%d\n", sum);
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

- function declaration (also called a prototype)
- a function must have been seen before it's called
- enables compiler to do type-checking

Using Multiple Files

myadd.h (called a header file)

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

myadd.c

```
#include "myadd.h"
int add(int x, int y)
{
    return x + y;
}
```

main.c

```
#include "myadd.h"

int main(int argc, char **argv)
{
    int a = 1;
    int b = 2;

    c = add(a,b);

    printf("%d + %d = %d", a, b, c);
}
```

Lecture Plan

- What really happens in GCC?
- Make and Makefiles
  - Overview of Make
  - Makefiles from scratch
  - Template for your Makefiles

What is Make?

Main Idea

- You write the “recipe”
- Make builds target

What is Make?

Main Idea

- You write the “recipe”
- Make builds target

Definition

- “GNU Make is a tool which *controls the generation of executables*... from the program's source files.”

  - GNU Make Docs

What is Make?

Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

What is Make?

Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

Makefile Demo

What is Make?

Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

Makefile Demo

```
simple: simple.c
    gcc -o simple simple.c
```

So is Make just a shorter GCC?

No!

- More general
- Any target, any shell command

So is Make just a shorter GCC?

No!

- More general
- Any target, any shell command

Makefile Demo

So is Make just a shorter GCC?

No!

- More general
- Any target, any shell command

Makefile Demo

```
clean:
    rm -rf simple
```

Usage:

```
make clean
```

So is Make just a shorter GCC?

Advantages of Make

- *General:* Not just for compiling C source files
- *Fast:* Only rebuilds what's necessary
- *Shareable:* End users just call "make"

Makefiles

Makefile

- *Makefile:* A list of *rules*.
- *Rule:* Tells Make the ***commands*** to build a ***target*** from 0 or more ***dependencies***

```
target: dependencies...
    commands
    …
```

Makefiles

Makefile

- *Makefile:* A list of *rules*.
- *Rule:* Tells Make the ***commands*** to build a ***target*** from 0 or more ***dependencies***

```
target: dependencies...
    commands
    …
```

Must indent with '`\t`', not spaces

Makefiles

Makefile = List of Rules

- *Rule:* Tells Make how to get to a ***target*** from ***source files***

```
target: dependencies...
    commands
    …
```

"If dependencies have changed or don't exist, rebuild them…
Then execute these commands."

Realistic Example

- Like Zip
- Traverses FS tree, builds a list of files
- Don’t know length ahead of time? Need growable data structure

Realistic Example

File Archiver

- Target file: `Far` (an executable)
- Source files: `Far.c Far.h vector.c vector.h`

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

Makefile Demo

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

Makefile Demo

```
CC=gcc
CFLAGS=-g -std=c99 -pedantic –Wall

all: Far

Far: Far.o vector.o
     ${CC} ${CFLAGS} $^ -o $@
Far.o: Far.c Far.h vector.h
     ${CC} ${CFLAGS} -c Far.c
vector.o: vector.c vector.h
     ${CC} ${CFLAGS} -c vector.c
clean:
     ${RM} Far.o vector.o Far
```

$@: The file name of the target of the rule
$^ : The names of all the prerequisites, with spaces between them

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

Good Test Problem!

Suppose I update `Far.c`,
Then call `make Far`.

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

Good Test Problem!

Suppose I update `Far.c`,
Then call `make Far`.

Which commands does Make run?

What is Make?

Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

Good Test Problem!

Suppose I update `Far.c`,
Then call `make Far`.

Which commands does Make run?

Answer:

gcc -g -std=c99 -pedantic -Wall -c Far.c
gcc -g -std=c99 -pedantic -Wall Far.o vector.o -o Far

Takeaways

Takeaways from File Archiver Example

- Recursive rules
- Bigger projects practically *need* Make (or another build system)
- Makefile variables (*e.g.,* `CC` and `CFLAGS`)
- Target need not be a file! (*e.g.,* `clean`)

Generic Makefile

Reusable Makefile

- Any simple project
- Main program and its header
- Can be easily extended to include libraries
- Feel free to copy-paste

Generic Makefile

```
# A simple makefile for building a program composed of C source files.
#
PROGRAMS = hello

all:: $(PROGRAMS)

# It is likely that default C compiler is already gcc, but explicitly
# set, just to be sure
CC = gcc

# The CFLAGS variable sets compile flags for gcc:
#  -g          compile with debug information
#  -Wall       give verbose compiler warnings
#  -O0         do not optimize generated code
#  -std=gnu99  use the GNU99 standard language definition
CFLAGS = -g -Wall -O0 -std=gnu99

# The LDFLAGS variable sets flags for linker
#  -lm   says to link in libm (the math library)
LDFLAGS = -lm

$(PROGRAMS): %:%.c
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

.PHONY: clean all

clean::
        rm -f $(PROGRAMS) *.o
```

Example – Source Files

myadd.h (called a header file)

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

myadd.c

```
#include "myadd.h"
int add(int x, int y)
{
    return x + y;
}
```

main.c

```
#include "myadd.h"

int main(int argc, char **argv)
{
    int a = 1;
    int b = 2;

    c = add(a,b);

    printf("%d + %d = %d", a, b, c);
}
```

Example – Makefile

```
# This Makefile should be used as a template for future Makefiles.
# It’s heavily commented, so hopefully you can understand what each
# line does.

# We’ll use gcc for C compilation and g++ for C++ compilation
CC  = gcc
CXX = g++

# Let’s leave a place holder for additional include directories
INCLUDES =

# Compilation options:
# -g for debugging info and -Wall enables all warnings
CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

# Linking options:
# -g for debugging info
LDFLAGS = -g

# List the libraries you need to link with in LDLIBS
# For example, use "-lm" for the math library
LDLIBS =

# The 1st target gets built when you type "make".
# It’s usually your executable.  ("main" in this case.)
#
# Note that we did not specify the linking rule.
# Instead, we rely on one of make’s implicit rules:
#
#     $(CC) $(LDFLAGS) <all-dependent-.o-files> $(LDLIBS)
#
# Also note that make assumes that main depends on main.o,
# so we can omit it if we want to.
```

```
main: main.o myadd.o

# main.o depends not only on main.c, but also on myadd.h because
# main.c includes myadd.h.  main.o will get recompiled if either
# main.c or myadd.h get modified.
#
# make already knows main.o depends on main.c, so we can omit main.c
# in the dependency list if we want to.
#
# make uses the following implicit rule to compile a .c file into a .o
# file:
#
#     $(CC) -c $(CFLAGS) <the-.c-file>
#
main.o: main.c myadd.h

# And myadd.o depends on myadd.c and myadd.h.
myadd.o: myadd.c myadd.h

# Always provide the "clean" target that removes intermediate files.
# What you remove depend on your choice of coding tools
# (different editors generate different backup files for example).
#
# And the "clean" target is not a file name, so we tell make that
# it’s a "phony" target.
.PHONY: clean
clean:
        rm -f *.o a.out core main

# "all" target is useful if your Makefile builds multiple programs.
# Here we’ll have it first do "clean", and rebuild the main target.
.PHONY: all
all: clean main
```

Make Takeaways

In The Wild

- Will see very complex makefiles — Don't be intimidated
- Will see other build systems (*e.g.,* CMake) — Same idea as Make
- Will see Make for other languages — Same source -> executable mapping

References

- https://www.gnu.org/software/make/
- https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
  Good Makefile examples/templates.

Recap

- What really happens in GCC?
  - The Preprocessor
  - The Compiler
  - The Assembler
  - The Linker
- Make and Makefiles
  - Overview of Make
  - Makefiles from scratch
  - Template for your Makefiles

xkcd.com/303/

Next Time: Assembly language

photo: Intel CPU Wafers

COMP201

Computer Systems & Programming

Lecture #14 – Introduction to x86-64 Assembly

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

1. **Bits and Bytes -** *How can a computer represent numbers?*
2. **Chars and C-Strings -** *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap –** *How can we effectively manage all types of memory in our programs?*
4. **Generics -** *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. **Working with Multiple Files** – *What really happens in GCC? How to write your own Makefiles?*

Course Overview

1. **Bits and Bytes** - *How can a computer represent numbers?*
2. **Chars and C-Strings** - *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap** – *How can we effectively manage all types of memory in our programs?*
4. **Generics** - *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. **Working with Multiple Files** – *What really happens in GCC? How to write your own Makefiles?*

---

6. **Assembly** - *How does a computer interpret and execute C programs?*
7. **The Memory Hierarchy** – *How to improve the performance of application programs by improving their temporal and spatial locality?*
8. **Code Optimization** – *How write C code so that a compiler can then generate efficient machine code?*
9. **Linking** – *How static and dynamic linking work?*
10. **Heap Allocators** - *How do core memory-allocation operations like malloc and free work?*

COMP201 Topic 6: How does a computer interpret and execute C programs?

Learning Assembly

Learning Goals

- Learn what assembly language is and why it is important
- Become familiar with the format of human-readable assembly and x86
- Learn the **`mov`** instruction and how data moves around at the assembly level

Plan for Today

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The `mov` instruction

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **`mov`** instruction

Bits all the way down

Data representation so far

- Integer (`unsigned int`, 2's complement signed `int`)
- Floating Points (IEEE single (`float`) and double (`double`) precision
- `char` (ASCII)
- Address (unsigned long)
- Aggregates (arrays, `structs`)

The code itself is binary too!

- Instructions (machine encoding)

GCC

- **GCC** is the compiler that converts your human-readable code into machine-readable instructions.
- C, and other languages, are high-level abstractions we use to write code efficiently. But computers don't really understand things like data structures, variable types, etc. Compilers are the translator!
- Pure machine code is 1s and 0s – everything is bits, even your programs! But we can read it in a human-readable form called **assembly**. (Engineers used to write code in assembly before C).
- There may be multiple assembly instructions needed to encode a single C instruction.
- We're going to go behind the curtain to see what the assembly code for our programs looks like.

Lecture Plan

- Overview: GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The `mov` instruction

Demo: Looking at an Executable (`objdump -d`)

Our First Assembly

```
int sum_array(int arr[], int nelems) {
   int sum = 0;
   for (int i = 0; i < nelems; i++) {
      sum += arr[i];
   }
   return sum;
}
```

What does this look like in assembly?

Our First Assembly

```
int sum_array(int arr[], int nelems) {
   int sum = 0;
   for (int i = 0; i < nelems; i++) {
      sum += arr[i];
   }
   return sum;
}
```

```
make
objdump -d sum
```

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00          mov    $0x0,%edx
  4005bb:    b8 00 00 00 00          mov    $0x0,%eax
  4005c0:    eb 0b                   jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 d2                movslq %edx,%rcx
  4005c5:    8b 04 8f                mov    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01                add    $0x1,%edx
  4005cb:    39 f2                   cmp    %esi,%edx
  4005cd:    7c f3                   jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                   repz retq
```

This is the name of the function (same as C) and the memory address where the code for this function starts.

Our First Assembly

```
0000000004005b6 <sum_array>:
  4005b6:   ba 00 00 00 00      mov    $0x0,%edx
  4005bb:   b8 00 00 00 00      mov    $0x0,%eax
  4005c0:   eb 09               jmp    4005cb <sum_array+0x15>
  4005c2:   4
  4005c5:   0                                  4),%eax
  4005c8:   8
  4005cb:   39 f2               cmp    %esi,%edx
  4005cd:   7c f3               jl     4005c2 <sum_array+0xc>
  4005cf:   f3 c3               repz retq
```

These are the memory addresses where each of the instructions live. Sequential instructions are sequential in memory.

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00
  4005bb:    b8 00 00 00 00
  4005c0:    eb 09
  4005cb:    39 f2
  4005cd:    7c f3
  4005cf:    f3 c3
```

This is the assembly code: "human-readable" versions of each machine code instruction.

```
mov     $0x0,%edx
mov     $0x0,%eax
jmp     4005cb <sum_array+0x15>
movslq  %edx,%rcx
add     (%rdi,%rcx,4),%eax
add     $0x1,%edx
cmp     %esi,%edx
jl      4005c2 <sum_array+0xc>
repz retq
```

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:     ba 00 00 00 00      mov    $0x0,%edx
  4005bb:     b8 00 00 00 00
  4005c0:     eb 09
  4005c2:     48 63 ca
  4005c5:     03 04 8f
  4005c8:     83 c2 01
  4005cb:     39 f2
  4005cd:     7c f3
  4005cf:     f3 c3              repz retq
```

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

Each instruction has an operation name (“opcode”).

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3
```

Each instruction can also have arguments (“operands”).

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

`$[number]` means a constant value, or "immediate" (e.g. 1 here).

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

%[name] means a register, a storage location on the CPU (e.g. edx here).

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **`mov`** instruction

Assembly Abstraction

- C abstracts away the low-level details of machine code. It lets us work using variables, variable types, and other higher-level abstractions.
- C and other languages let us write code that works on most machines.
- Assembly code is just bytes! No variable types, no type checking, etc.
- Assembly/machine code is processor-specific.
- What is the level of abstraction for assembly code?

Registers

Registers

What is a register?

A register is a fast read/write memory slot right on the CPU that can hold variable values.

Registers are **not** located in memory.

Registers

- A **register** is a 64-bit space inside the processor.
- There are 16 registers available, each with a unique name.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold parameters and return values for functions.
- Registers are extremely *fast* memory!
- Processor instructions consist mostly of moving data into/out of registers and performing arithmetic on them. This is the level of logic your program must be in to execute!

Machine-Level Code

Assembly instructions manipulate these registers. For example:

- One instruction adds two numbers in registers
- One instruction transfers data from a register to memory
- One instruction transfers data from memory to a register

Computer architecture

registers accessed by name
ALU is main workhorse of CPU

memory needed for program execution (stack, heap, etc.) accessed by address

disk/server stores program when not executing

GCC And Assembly

- GCC compiles your program – it lays out memory on the stack and heap and generates assembly instructions to access and do calculations on those memory locations.
- Here's what the "assembly-level abstraction" of C code might look like:

C	Assembly Abstraction
`int sum = x + y;`	*1) Copy x into register 1* *2) Copy y into register 2* *3) Add register 2 to register 1* *4) Write register 1 to memory for sum*

Assembly

- We are going to learn the **x86-64** instruction set architecture. This instruction set is used by Intel and AMD processors.
- There are many other instruction sets: ARM, MIPS, etc.

Instruction set architecture (ISA)

A contract between program/compiler and hardware:

- Defines operations that the processor (CPU) can execute
- Data read/write/transfer operations
- Control mechanisms

Intel originally designed their instruction set back in 1978.

- Legacy support is a huge issue for x86-64
- Originally 16-bit processor, then 32 bit, now 64 bit.
  These design choices dictated the register sizes
  (and even register/instruction names).

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **`mov`** instruction

mov

The **mov** instruction copies bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov      src,dst
```

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only **src***) **$0x104**
- Register **%rbx**
- Memory Location (*at most one of **src, dst***) Direct address **0x6005c0**

Operand Forms: Immediate

```
mov     $0x104,_____
```

Copy the value `0x104` into some destination.

Operand Forms: Registers

Copy the value in register ***%rbx*** *into some destination.*

```
mov    %rbx,_____
```

```
mov    _____,%rbx
```

Copy the value from some source into register ***%rbx****.*

Operand Forms: Absolute Addresses

*Copy the value at address **`0x104`** into some destination.*

```
mov     0x104,_______
```

```
mov     _______,0x104
```

*Copy the value from some source into the memory at address **`0x104`**.*

Practice #1: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value `5` is stored at address `0x42`, and the value `8` is stored in `%rbx`.

`1. mov  $0x42,%rax`	Move `0x42` into `%rax`
`2. mov  0x42,%rax`	Move `5` into `%rax`
`3. mov  %rbx,0x55`	Move `8` to address `0x55`

Operand Forms: Indirect

Copy the value at the address stored in register ***%rbx*** *into some destination.*

```
mov     (%rbx),_______
```

```
mov     _______,(%rbx)
```

Copy the value from some source into the memory at the address stored in register ***%rbx****.*

Operand Forms: Base + Displacement

*Copy the value at the address (**0x10 plus** what is stored in register **%rax**) into some destination.*

```
mov    0x10(%rax),_______________

mov    _______________,0x10(%rax)
```

*Copy the value from some source into the memory at the address (**0x10 plus** what is stored in register **%rax**).*

Operand Forms: Indexed

Copy the value at the address which is (the sum of the values in registers ***%rax*** *and* ***%rdx****) into some destination.*

```
mov     (%rax,%rdx),_____
```

```
mov     _____,(%rax,%rdx)
```

Copy the value from some source into the memory at the address which is (the sum of the values in registers ***%rax*** *and* ***%rdx****).*

Operand Forms: Indexed

*Copy the value at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx) into some destination.*

```
mov     0x10(%rax,%rdx),_______
```

```
mov     ________,0x10(%rax,%rdx)
```

*Copy the value from some source into the memory at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx).*

Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume

the value `0x11` is stored at address `0x10C`,
the value `0xAB` is stored at address `0x104`,
`0x100` is stored in register `%rax` and `0x3` is stored in `%rdx`.

1. `mov $0x42,(%rax)`	Move `0x42` to memory address `0x100`
2. `mov 4(%rax),%rcx`	Move `0xAB` into `%rcx`
3. `mov 9(%rax,%rdx),%rcx`	Move `0x11` into `%rcx`

`Imm(`r_b`, `r_i`)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]`**

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**4 times** the value in register **%rdx**) into some destination.*

```
mov    (,%rdx,4),_______
```

The *scaling factor* (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

```
mov    _______,(,%rdx,4)
```

*Copy the value from some source into the memory at the address which is (**4 times** the value in register **%rdx**).*

Operand Forms: Scaled Indexed

Copy the value at the address which is (4 times the value in register ***%rdx****, **plus 0x4**), into some destination.*

```
mov     0x4(,%rdx,4),_______
```

```
mov     _______,0x4(,%rdx,4)
```

Copy the value from some source into the memory at the address which is (4 times the value in register ***%rdx****, **plus 0x4).***

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**the value in register %rax** plus 2 times the value in register %rdx) into some destination.*

```
mov     (%rax,%rdx,2),_________
```

```
mov     _________,(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**the value in register %rax** plus 2 times the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**0x4 plus** the value in register **%rax** plus 2 times the value in register **%rdx**) into some destination.*

```
mov     0x4(%rax,%rdx,2),_____
```

```
mov     _______,0x4(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**0x4 plus** the value in register **%rax** plus 2 times the value in register **%rdx**).*

Most General Operand Form

$\mathtt{Imm(r_b,r_i,s)}$

is equivalent to...

$\mathtt{Imm + R[r_b] + R[r_i]*s}$

Most General Operand Form

`Imm(`r_b`, `r_i`, s)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]*s`**

Displacement: pos/neg constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Scale must be 1,2,4, or 8 (if missing, = 1)

Memory Location Syntax

Syntax	Meaning
`0x104`	Address `0x104` (no `$`)
`(%rax)`	What's in `%rax`
`4(%rax)`	What's in `%rax`, plus `4`
`(%rax, %rdx)`	Sum of what's in `%rax` and `%rdx`
`4(%rax, %rdx)`	Sum of values in `%rax` and `%rdx`, plus 4
`(, %rcx, 4)`	What's in `%rcx`, times 4 (multiplier can be 1, 2, 4, 8)
`(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`
`8(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`, plus 8

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$\mathrm{Imm}$	Imm	Immediate
Register	$\mathrm{r_a}$	$\mathrm{R[r_a]}$	Register
Memory	Imm	$\mathrm{M[Imm]}$	Absolute
Memory	$(\mathrm{r_a})$	$\mathrm{M[R[r_a]]}$	Indirect
Memory	$\mathrm{Imm(r_b)}$	$\mathrm{M[Imm + R[r_b]]}$	Base + displacement
Memory	$(\mathrm{r_b}, \mathrm{r_i})$	$\mathrm{M[R[r_b] + R[r_i]]}$	Indexed
Memory	$\mathrm{Imm(r_b}, \mathrm{r_i})$	$\mathrm{M[Imm + R[r_b] + R[r_i]]}$	Indexed
Memory	$(, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[R[r_i] \cdot s]}$	Scaled indexed
Memory	$\mathrm{Imm}(, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[Imm + R[r_i] \cdot s]}$	Scaled indexed
Memory	$(\mathrm{r_b}, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[R[r_b] + R[r_i] \cdot s]}$	Scaled indexed
Memory	$\mathrm{Imm}(\mathrm{r_b}, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[Imm + R[r_b] + R[r_i] \cdot s]}$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either. 1, 2, 4, or 8."

Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume

the value `0x1` is stored in register `%rcx`,
the value `0x100` is stored in register `%rax`,
the value `0x3` is stored in register `%rdx`, and
the value `0x11` is stored at address `0x10C`.

1. `mov $0x42,0xfc(,%rcx,4)`

Move `0x42` to memory address `0x100`

2. `mov (%rax,%rdx,4),%rbx`

Move `0x11` into `%rbx`

`Imm(`r_b`, `r_i`, s)` *is equivalent to address* **`Imm + R[`r_b`] + R[`r_i`]*s`**

Displacement Base Index Scale (1,2,4,8)

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're 1/4th of the way to understanding assembly!
What looks understandable right now?

Some notes:

- Registers store addresses and values
- `mov src, dst` ***copies*** value into dst
- `sizeof(int)` is 4
- Instructions executed sequentially

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
                                   cmp    %esi,%edx
                                   jl     4005c2 <sum_array+0xc>
                                   repz retq
```

We'll come back to this example in future lectures!

Additional Reading

arstechnica.com

ars TECHNICA | BIZ & IT | TECH | SCIENCE | POLICY | CARS | GAMING & CULTURE | STORE | FORUMS | SUBSCRIBE | SIGN IN

TECH —

An Introduction to 64-bit Computing and x86-64

When I first heard that AMD had plans to extend the x86 ISA to 64 bits, I ...

JON STOKES - 3/12/2002, 1:36 AM

110

When I first heard that AMD had plans to extend the x86 ISA to 64 bits, I thought it was a terrible idea. Though x86 is the world's most successful ISA, it's also the world's most widely disparaged. Programmers, analysts, architecture buffs, and enthusiasts often see x86 as a leaden albatross around the neck of the entire computing industry, and like the Mariner's albatross we were all hoping it would just fall off at some point and slip quietly into the deep. But in spite of such hopes, I really knew better. In fact, I've argued elsewhere that x86 isn't going away anytime soon, and it no longer makes any real sense to gripe about its quirks from a performance perspective. I won't recap that argument here, but I can sum it up briefly.

Most of us would probably assent to the following statement: "there's a huge global market for mainstream business and consumer software, and the overwhelming majority of that software just so happens to use the x86 ISA." This statement is true, as far as it goes, but framing x86's role in the software industry this way misses an important point. In my article "The Future of x86 and the Concept of the ISA," I argue that a statement like the following would provide a more relevant assessment of the true state of the software industry: "There's a huge global market for mainstream business and consumer *x86 software*, and several smaller markets for software written to other ISAs." All discussions of the desktop prospects of widely ported operating systems (i.e. Linux) or of the possible effects of greater open source market penetration aside, this statement should still ring true to anyone who's acquainted with the present realities of the installed base of IT and consumer software.

If we think realistically about most of the world's commercial software not as "software" in the abstract but as x86 binary code, then it becomes apparent that improvements to the x86 ISA

ARS VIDEO

30 People Play Mario Kart 8 From Newbies to Pros

They usually don't have very many railings.

https://arstechnica.com/gadgets/2002/03/an-introduction-to-64-bit-computing-and-x86-64/

Additional Reading

Not Secure — pbm.com

The story of Mel

Source: usenet: utastro!nather, May 21, 1983.

A recent article devoted to the *macho* side of programming made the bald and unvarnished statement:

Real Programmers write in Fortran.

Maybe they do now, in this decadent era of Lite beer, hand calculators and "user-friendly" software but back in the Good Old Days, when the term "software" sounded funny and Real Computers were made out of drums and vacuum tubes, Real Programmers wrote in machine code. Not Fortran. Not RATFOR. Not, even, assembly language. Machine Code. Raw, unadorned, inscrutable hexadecimal numbers. Directly.

Lest a whole new generation of programmers grow up in ignorance of this glorious past, I feel duty-bound to describe, as best I can through the generation gap, how a Real Programmer wrote code. I'll call him Mel, because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp., a now-defunct subsidiary of the typewriter company. The firm manufactured the LGP-30, a small, cheap (by the standards of the day) drum-memory computer, and had just started to manufacture the RPC-4000, a much-improved, bigger, better, faster -- drum-memory computer. Cores cost too much, and weren't here to stay, anyway. (That's why you haven't heard of the company, or the computer.)

I had been hired to write a Fortran compiler for this new marvel and Mel was my guide to its wonders. Mel didn't approve of compilers.

"If a program can't rewrite its own code," he asked, "what good is it?"

Mel had written, in hexadecimal, the most popular computer program the company owned. It ran on the LGP-30 and played blackjack with potential customers at computer shows. Its effect was always dramatic. The LGP-30 booth was packed at every show, and the IBM salesmen stood around talking to each other. Whether or not this actually sold computers was a question we never discussed.

Mel's job was to re-write the blackjack program for the RPC-4000. (Port? What does that mean?) The new computer had a one-plus-one addressing scheme, in which each machine instruction, in addition to the operation code and the address of the needed operand, had a second address that indicated where, on the revolving drum, the next instruction was located. In modern parlance, every single instruction was followed by a GO TO! Put *that* in Pascal's pipe and smoke it.

Mel loved the RPC-4000 because he could optimize his code: that is, locate instructions on the drum so that just as one finished its job, the next would be just arriving at the "read head" and available for immediate execution. There was a program to do that job, an "optimizing assembler", but Mel refused to use it.

"You never know where it's going to put things", he explained, "so you'd have to use separate constants".

It was a long time before I understood that remark. Since Mel knew the numerical value of every operation code, and assigned his own drum addresses, every instruction he wrote could also be considered a numerical constant. He could pick up an earlier "add" instruction, say, and multiply by it, if it had the right numeric value. His code was not easy for someone else to modify.

http://www.pbm.com/~lindahl/mel.html

Extra Practice

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = ____???___;
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = arr[3];
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
___???___ = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
*ptr = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
___???___ = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
str[2] = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

Recap

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The `mov` instruction

Next time: diving deeper into assembly

photo: Intel CPU Wafers

COMP201

Computer Systems & Programming

Lecture #15 – Arithmetic and Logic Operations

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

COMP201 Topic 6: How does a computer interpret and execute C programs?

Learning Assembly

Learning Goals

- Learn how to perform arithmetic and logical operations in assembly
- Begin to learn how to read assembly and understand the C code that generated it

Plan for Today

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Helpful Assembly Resources

- **Course textbook**
  Reminder: see relevant readings for each lecture on the Schedule section:
  https://aykuterdem.github.io/classes/comp201/index.html#div_schedule

- **Other resources**
  See the guides on the resources section of the course website:
  https://aykuterdem.github.io/classes/comp201/index.html#div_resources
  - **Stanford CS107 Assembly Reference Sheet**
  - **Stanford CS107 Guide to x86-64**
  - **CMU 15-213 x86-64 Machine-Level Programming**

Lecture Plan

- Recap: `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

mov

The **`mov`** instruction copies bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov      src,dst
```

The **`src`** and **`dst`** can each be one of:

- Immediate (constant value, like a number) (*only* ***src****)*
- Register
- Memory Location
  (*at most one of* ***src, dst****)*

Memory Location Syntax

Syntax	Meaning
`0x104`	Address `0x104` (no $)
`(%rax)`	What's in `%rax`
`4(%rax)`	What's in `%rax`, plus 4
`(%rax, %rdx)`	Sum of what's in `%rax` and `%rdx`
`4(%rax, %rdx)`	Sum of values in `%rax` and `%rdx`, plus 4
`(, %rcx, 4)`	What's in `%rcx`, times 4 (multiplier can be 1, 2, 4, 8)
`(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`
`8(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`, plus 8

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$\mathrm{Imm}$	Imm	Immediate
Register	$\mathrm{r_a}$	$\mathrm{R[r_a]}$	Register
Memory	Imm	$\mathrm{M[Imm]}$	Absolute
Memory	$(\mathrm{r_a})$	$\mathrm{M[R[r_a]]}$	Indirect
Memory	$\mathrm{Imm(r_b)}$	$\mathrm{M[Imm + R[r_b]]}$	Base + displacement
Memory	$(\mathrm{r_b}, \mathrm{r_i})$	$\mathrm{M[R[r_b] + R[r_i]]}$	Indexed
Memory	$\mathrm{Imm(r_b}, \mathrm{r_i})$	$\mathrm{M[Imm + R[r_b] + R[r_i]]}$	Indexed
Memory	$(, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[R[r_i] \cdot s]}$	Scaled indexed
Memory	$\mathrm{Imm}(, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[Imm + R[r_i] \cdot s]}$	Scaled indexed
Memory	$(\mathrm{r_b}, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[R[r_b] + R[r_i] \cdot s]}$	Scaled indexed
Memory	$\mathrm{Imm}(\mathrm{r_b}, \mathrm{r_i}, \mathrm{s})$	$\mathrm{M[Imm + R[r_b] + R[r_i] \cdot s]}$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either. 1, 2, 4, or 8."

Lecture Plan

- Recap: `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

Assembly instructions can have suffixes to refer to these sizes:

- `b` means **byte**
- `w` means **word**
- `l` means **double word**
- `q` means **quad word**

Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

C Type	Suffix	Byte	Intel Data Type
`char`	`b`	1	Byte
`short`	`w`	2	Word
`int`	`l`	4	Double word
`long`	`q`	8	Quad word
`char *`	`q`	8	Quad word
`float`	`s`	4	Single precision
`double`	`l`	8	Double precision

Register Sizes

Bit: 63	31	15	7 0
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil

Register Sizes

Bit: 63	31	15	7	0
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	

Register Sizes

Bit: 63	31	15	7	0
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **`%rax`** stores the return value
- **`%rdi`** stores the first parameter to a function
- **`%rsi`** stores the second parameter to a function
- **`%rdx`** stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- **`%rsp`** stores the address of the current top of the stack

See **Stanford CS107 x86-64 Reference Sheet** on Resources page of the course website!
https://aykuterdem.github.io/classes/comp201/index.html#div_resources

`mov` Variants

- **`mov`** can take an optional suffix (`b`,`w`,`l`,`q`) that specifies the size of data to move: `movb, movw, movl, movq`
- **`mov`** only updates the specific register bytes or memory locations indicated.
  - **Exception: `movl`** writing to a register will also set high order 4 bytes to 0.

Practice #1: `mov` And Data Sizes

For each of the following `mov` instructions, determine the appropriate suffix based on the operands (e.g. `movb`, `movw`, `movl` or `movq`).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

```
movl %eax, (%rsp)
movw (%rax), %dx
movb $0xff, %bl
movb (%rsp,%rdx,4),%dl
movq (%rdx), %rax
movw %dx, (%rax)
```

mov

- The **`movabsq`** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **`movq`** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

```
movabsq $0x0011223344556677, %rax
```

Practice #2: `mov` And Data Sizes

For each of the following `mov` instructions, determine how data movement instructions modify the upper bytes of a destination register.

```
1. movabs $0x0011223344556677, %rax   %rax = 0011223344556677
2. movb $-1, %al                      %rax = 0011223344556FF
3. movw $-1, %ax                      %rax = 0011223344 5FFFF
4. movl $-1, %eax                     %rax = 00000000FFFFFFFF
5. movq $-1, %rax                     %rax = FFFFFFFFFFFFFFFF
```

`movz` and `movs`

- There are two **`mov`** instructions that can be used to copy a smaller source to a larger destination: **`movz`** and **`movs`**.
- **`movz`** fills the remaining bytes with zeros
- **`movs`** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

movz and movs

```
MOVZ S,R          R ← ZeroExtend(S)
```

Instruction	Description
movzbw	Move zero-extended byte to word
movzbl	Move zero-extended byte to double word
movzwl	Move zero-extended word to double word
movzbq	Move zero-extended byte to quad word
movzwq	Move zero-extended word to quad word

movz and movs

```
MOVS S,R          R ← SignExtend(S)
```

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax %rax ← SignExtend(%eax)

Lecture Plan

- Recap: `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

`lea`

The `lea` instruction copies an “effective address” from one place to another.

```
lea     src,dst
```

Unlike **`mov`**, which copies data at the address src to the destination, **`lea`** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **`mov`**. The difference is how it handles the `src`.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.
`(%rax, %rcx, 4), %rdx`	Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`.	Copy (`%rax` + 4 * `%rcx`) into `%rdx`.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
6(%rax), %rdx	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
(%rax, %rcx), %rdx	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
(%rax, %rcx, 4), %rdx	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
7(%rax, %rax, 8), %rdx	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike **mov**, which copies data <u>at</u> the address src to the destination, **lea** copies the value of src itself to the destination.

Lecture Plan

- Recap: `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **`imul`**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **`%rax`**, and splits the product across **2** registers. It puts the high-order 64 bits in **`%rdx`** and the low-order 64 bits in **`%rax`**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- <u>Terminology:</u> **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- The quotient is stored in **`%rax`**, and the remainder in **`%rdx`**.

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient + remainder**
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **`cqto`** instruction sign-extends the 64-bit value in **`%rax`** into **`%rdx`** to fill both registers with the dividend, as the division instruction expects.

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **`%cl`** (and only that register!)

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D << k$	Left shift
`shl k, D`	$D \leftarrow D << k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D << k$	Left shift
`shl k, D`	$D \leftarrow D << k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

- When using **`%cl`**, the width of what you are shifting determines what portion of **`%cl`** is used.
- For **`w`** bits of data, it looks at the low-order **`log2(w)`** bits of **`%cl`** to know how much to shift.
  - If **`%cl`** = 0xff (0b11111111), then: **`shlb`** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **`shlw`** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Lecture Plan

- Recap: `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Assembly Exploration

- Let's pull these commands together and see how some C code might be translated to assembly.
- Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let's check it out!
- https://godbolt.org/z/NLYhVf

Code Reference: `add_to_first`

```
// Returns the sum of x and the first
// element in arr
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}


----------

add_to_first:
  movl %edi, %eax
  addl (%rsi), %eax
  ret
```

63	31	15	7 0	
%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

Code Reference: `full_divide`

```
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

-------

full_divide:
  movq %rdx, %rcx
  movq %rdi, %rax
  cqto
  idivq %rsi
  movq %rdx, (%rcx)
  ret
```

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

Assembly Exercise 1

slido

```
00000000004005ac <sum_example1>:
     4005bd:   8b 45 e8        mov  %esi,%eax
     4005c3:   01 d0           add  %edi,%eax
     4005cc:   c3              retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

Assembly Exercise 2

slido

```
0000000000400578 <sum_example2>:
     400578:   8b 47 0c        mov  0xc(%rdi),%eax
     40057b:   03 07           add  (%rdi),%eax
     40057d:   2b 47 18        sub  0x18(%rdi),%eax
     400580:   c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

%eax

Assembly Exercise 3

slido

```
0000000000400578 <sum_example2>:
     400578:   8b 47 0c        mov  0xc(%rdi),%eax
     40057b:   03 07           add  (%rdi),%eax
     40057d:   2b 47 18        sub  0x18(%rdi),%eax
     400580:   c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

0x18

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're 1/2 of the way to understanding assembly!
What looks understandable right now?

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

A Note About Operand Forms

- Many instructions share the same address operand forms that **`mov`** uses.
  - Eg. `7(%rax, %rcx, 2)`.
- These forms work the same way for other instructions, e.g. `sub`:
  - `sub 8(%rax,%rdx),%rcx` -> Go to 8 + `%rax` + `%rdx`, subtract what's there from `%rcx`
- The exception is **`lea`**:
  - It interprets this form as just the calculation, *not the dereferencing*
  - `lea 8(%rax,%rdx),%rcx` -> Calculate 8 + `%rax` + `%rdx`, put it in `%rcx`

Extra Practice

https://godbolt.org/z/QQj77g

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------

add_to_ith:
  movslq %edx, %rdx
  movl %edi, %eax
  addl (%rsi,%rdx,4), %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to_ith:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to_ith:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------

elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------

func:
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  movq %rdi, %rax
  cqto
  idivq %rcx
  movq %rdx, %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```

Recap

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Next Time: control flow in assembly (while loops, if statements, and more)

photo by unsplash user @carrier_lost

COMP201

Computer Systems & Programming

Lecture #16 – x86-64 Condition Codes & Control Flow

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- The `lea` Instruction
- Logical and Arithmetic Operations

Recap: `lea`

The **`lea`** instruction copies an "effective address" from one place to another.

```
lea      src,dst
```

Unlike **`mov`**, which copies data at the address src to the destination, **`lea`** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **`mov`**. The difference is how it handles the **`src`**.

Recap: Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Recap: Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations! Read it as, e.g., "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Recap: Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **`imul`**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **`%rax`**, and splits the product across **2** registers. It puts the high-order 64 bits in **`%rdx`** and the low-order 64 bits in **`%rax`**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Recap: Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- <u>Terminology:</u> **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- The quotient is stored in **`%rax`**, and the remainder in **`%rdx`**.

Recap: Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient + remainder**
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **`cqto`** instruction sign-extends the 64-bit value in **`%rax`** into **`%rdx`** to fill both registers with the dividend, as the division instruction expects.

Recap: Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **`%cl`** (and only that register!)

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D << k$	Left shift
`shl k, D`	$D \leftarrow D << k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Recap: Shift Amount

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D << k$	Left shift
`shl k, D`	$D \leftarrow D << k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

- When using **`%cl`**, the width of what you are shifting determines what portion of **`%cl`** is used.
- For **`w`** bits of data, it looks at the low-order **`log2(w)`** bits of **`%cl`** to know how much to shift.
  - If **`%cl`** = 0xff (0b11111111), then: **`shlb`** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **`shlw`** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Recap: A Note About Operand Forms

- Many instructions share the same address operand forms that **`mov`** uses.
  - Eg. `7(%rax, %rcx, 2)`.
- These forms work the same way for other instructions, e.g. **`sub`**:
  - `sub 8(%rax,%rdx),%rcx` ➔ Go to 8 + `%rax` + `%rdx`, subtract what's there from `%rcx`

- The exception is **`lea`**:
  - It interprets this form as just the calculation, *not the dereferencing*
  - `lea 8(%rax,%rdx),%rcx` ➔ Calculate 8 + `%rax` + `%rdx`, put it in `%rcx`

Plan for Today

- Assembly Execution and `%rip`
- Control Flow Mechanics

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Learning Assembly

Lecture Plan

- Assembly Execution and `%rip`
- Control Flow Mechanics

Learning Goals

- Learn about how assembly stores comparison and operation results in condition codes
- Understand how assembly implements loops and control flow

Executing Instructions

What does it mean for a program to execute?

Executing Instructions

So far:

- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Today:

- **Who controls the instructions?**
  How do we know what to do now or next?

Answer:

- The **program counter** (PC), `%rip`.

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **`%rax`** stores the return value
- **`%rdi`** stores the first parameter to a function
- **`%rsi`** stores the second parameter to a function
- **`%rdx`** stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- **`%rsp`** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **`%rax`** stores the return value
- **`%rdi`** stores the first parameter to a function
- **`%rsi`** stores the second parameter to a function
- **`%rdx`** stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- **`%rsp`** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Instructions Are Just Bytes!

Memory bus
Main memory
“hello, world\r
hello code

Instructions Are Just Bytes!

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                          push   %rbp
4004ee: 48 89 e5                    mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                 addl   $0x1,-0x4(%rbp)
4004fc: eb fa                       jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004ed`

`%rip`

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004ee`

`%rip`

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004f1`

`%rip`

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004f8`

`%rip`

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004fc`

`%rip`

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

Special hardware sets the program counter to the next instruction:

`%rip` += size of bytes of current instruction

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Going In Circles

- How can we use this representation of execution to represent e.g. a **loop**?
- **Key Idea:** we can "interfere" with **`%rip`** and set it back to an earlier instruction!

Jump!

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

0x4004fc

%rip

Jump!

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The **`jmp`** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

`0x4004fc`

`%rip`

Jump!

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

This assembly represents an infinite loop in C!

```
while (true) {…}
```

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

jmp

The **jmp** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

jmp Label (Direct Jump)

jmp *Operand (Indirect Jump)

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be one of the usual operand forms (indirect jump):

```
jmp *%rax     # jump to instruction at address in %rax
```

“Interfering” with `%rip`

1. How do we repeat instructions in a loop?

`jmp [target]`

- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a **conditional jump**?

Lecture Plan

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.
- How is this represented in assembly?

Control

```
if (x > y) {
    // a
} else {
    // b
}
```

In Assembly:

1. **Calculate the condition result**
2. **Based on the result, go to a or b**

Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. cmp S1, S2 // compare two values

2. je [target] *or* **jne [target]** *or* **jl [target]** *or* … // conditionally
// jump

"jump if equal"

"jump if not equal"

"jump if less than"

Conditional Jumps

There are also variants of **`jmp`** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`je` *`Label`*	`jz`	Equal / zero
`jne` *`Label`*	`jnz`	Not equal / not zero
`js` *`Label`*		Negative
`jns` *`Label`*		Nonnegative
`jg` *`Label`*	`jnle`	Greater (signed >)
`jge` *`Label`*	`jnl`	Greater or equal (signed >=)
`jl` *`Label`*	`jnge`	Less (signed <)
`jle` *`Label`*	`jng`	Less or equal (signed <=)
`ja` *`Label`*	`jnbe`	Above (unsigned >)
`jae` *`Label`*	`jnb`	Above or equal (unsigned >=)
`jb` *`Label`*	`jnae`	Below (unsigned <)
`jbe` *`Label`*	`jna`	Below or equal (unsigned <=)

Control

Read **`cmp S1,S2`** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]

// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]

// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Control

Read **`cmp S1,S2`** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
%edi
get]
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?

Control

- The CPU has special registers called ***condition codes*** that are like "global variables". They *automatically* keep track of information about the most recent arithmetic or logical operation.
  - **`cmp`** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded zero.
- **SF:** Sign flag. The most recent operation yielded a negative value.
- **OF:** Overflow flag. The most recent operation caused a two's-complement overflow-either negative or positive.

Condition Codes

Alongside normal registers, the CPU also has <u>single-bit</u> *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Example: if we calculate t = a + b, condition codes are set according to:

- **CF:** Carry flag (Unsigned Overflow). `(unsigned) t < (unsigned) a`
- **ZF:** Zero flag (Zero). `(t == 0)`
- **SF:** Sign flag (Negative). `(t < 0)`
- **OF:** Overflow flag (Signed Overflow). `(a<0 == b<0) && (t<0 != a<0)`

Setting Condition Codes

The **`cmp`** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 - S1
```

Instruction	Description
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Control

Read **`cmp S1,S2`** as "compare S2 to S1". It calculates S2 – S1 and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi – 2
cmp $2, %edi
jg [target]

// Jump if %edi != 3
// calculates %edi – 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi == 4
// calculates %edi – 4
cmp $4, %edi
je [target]

// Jump if %edi <= 1
// calculates %edi – 1
cmp $1, %edi
jle [target]
```

Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

Instruction	Synonym	Set Condition
`je` *`Label`*	`jz`	Equal / zero (ZF = 1)
`jne` *`Label`*	`jnz`	Not equal / not zero (ZF = 0)
`js` *`Label`*		Negative (SF = 1)
`jns` *`Label`*		Nonnegative (SF = 0)
`jg` *`Label`*	`jnle`	Greater (signed >) (ZF = 0 and SF = OF)
`jge` *`Label`*	`jnl`	Greater or equal (signed >=) (SF = OF)
`jl` *`Label`*	`jnge`	Less (signed <) (SF != OF)
`jle` *`Label`*	`jng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`ja` *`Label`*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *`Label`*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb` *`Label`*	`jnae`	Below (unsigned <) (CF = 1)
`jbe` *`Label`*	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Setting Condition Codes

The **`test`** instruction is like **`cmp`**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

Condition Codes

- Previously-discussed arithmetic and logical instructions update these flags. **`lea`** does not (it was intended only for address computations).
- Logical operations (**`xor`**, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.
- For more complicated reasons, **`inc`** and **`dec`** set the overflow and zero flags, but leave the carry flag unchanged.

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics

Next time: Conditional branches

photo by unsplash user @carrier_lost

COMP201

Computer Systems & Programming

Lecture #17 – More Control Flow

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions

Recap: Executing Instructions

So far:

- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Today:

- **Who controls the instructions?**
  How do we know what to do now or next?

Answer:

- The **program counter** (PC), `%rip`.

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Recap: Instructions Are Just Bytes!

Recap: Instructions Are Just Bytes!

Recap: %rip

```
00000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The **program counter** (PC), known as `%rip` in x86-64, stores the address in memory of the ***next instruction*** to be executed.

`0x4004ed`

`%rip`

Recap: `jmp`

The **`jmp`** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

```
jmp Label        (Direct Jump)
jmp *Operand     (Indirect Jump)
```

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be one of the usual operand forms (indirect jump):

```
jmp *%rax     # jump to instruction at address in %rax
```

Recap: Conditional Jumps

There are also variants of **`jmp`** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`je` *`Label`*	`jz`	Equal / zero
`jne` *`Label`*	`jnz`	Not equal / not zero
`js` *`Label`*		Negative
`jns` *`Label`*		Nonnegative
`jg` *`Label`*	`jnle`	Greater (signed >)
`jge` *`Label`*	`jnl`	Greater or equal (signed >=)
`jl` *`Label`*	`jnge`	Less (signed <)
`jle` *`Label`*	`jng`	Less or equal (signed <=)
`ja` *`Label`*	`jnbe`	Above (unsigned >)
`jae` *`Label`*	`jnb`	Above or equal (unsigned >=)
`jb` *`Label`*	`jnae`	Below (unsigned <)
`jbe` *`Label`*	`jna`	Below or equal (unsigned <=)

Recap: Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded zero.
- **SF:** Sign flag. The most recent operation yielded a negative value.
- **OF:** Overflow flag. The most recent operation caused a two's-complement overflow-either negative or positive.

Recap: Setting Condition Codes

The **`cmp`** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 - S1
```

Instruction	Description
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Recap: Setting Condition Codes

The **`test`** instruction is like **`cmp`**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

Exercise 1: Conditional jump

```
je target
```

jump if ZF is 1

Let `%edi` store 0x10. Will we jump in the following cases? `%edi` `0x10`

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

Exercise 1: Conditional jump

```
je target
```

jump if ZF is 1

Let `%edi` store 0x10. Will we jump in the following cases? `%edi` `0x10`

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don't jump

Exercise 2: Conditional jump

slido

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi: 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

Exercise 2: Conditional jump

slido

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi: 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   A. `4004e1`
   B. `0x2`
   C. `0xa`
   D. `0xc`
   E. Other

Exercise 2: Conditional jump

slido

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi: 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   A. `4004e1`
   B. `0x2`
   C. `0xa`
   D. `0xc`
   E. Other

Plan for Today

- If statements
- Loops
- Other Instructions That Depend On Condition Codes

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture Plan

- If statements
- Loops
- Other Instructions That Depend On Condition Codes

Practice: Fill In The Blank

```
int if_then(int param1) {
  if ( __________ ) {
      __________ ;
  }

  return __________ ;
}
```

```
00000000004004d6 <if_then>:
  4004d6:   cmp  $0x6,%edi
  4004d9:   jne  4004de
  4004db:   add  $0x1,%edi
  4004de:   lea  (%rdi,%rdi,1),%eax
  4004e1:   retq
```

Practice: Fill In The Blank

```
int if_then(int param1) {
  if (param1 == 6 ) {
        param1++;
  }

  return  param1 * 2;
}
```

```
00000000004004d6 <if_then>:
  4004d6:    cmp   $0x6,%edi
  4004d9:    jne   4004de
  4004db:    add   $0x1,%edi
  4004de:    lea   (%rdi,%rdi,1),%eax
  4004e1:    retq
```

Practice: Fill In The Blank

If-Else In C

```
if (__________) {
    __________;
} else {
    __________;
}

__________;
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill In The Blank

slido

If-Else In C

```
if ( ________ ) {
    ________;
} else {
    _______;
}
____;
```

```
400552 <+0>:  cmp  $0x3,%edi
400555 <+3>:  jle  0x40055e <if_else+12>
400557 <+5>:  mov  $0xa,%eax
40055c <+10>: jmp  0x400563 <if_else+17>
40055e <+12>: mov  $0x0,%eax
400563 <+17>: add  $0x1,%eax
```

If-Else In Assembly pseudocode

Test
Jump to else-body if test **fails**
If-body
Jump to past else-body
Else-body
Past else body

Practice: Fill In The Blank

slido

If-Else In C

```
if (  arg > 3   ) {
    ret = 10;
} else {
    ret = 0;
}
ret++;
```

```
400552 <+0>:  cmp  $0x3,%edi
400555 <+3>:  jle  0x40055e <if_else+12>
400557 <+5>:  mov  $0xa,%eax
40055c <+10>: jmp  0x400563 <if_else+17>
40055e <+12>: mov  $0x0,%eax
400563 <+17>: add  $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

Lecture Plan

- If statements (cont'd.)
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Set `%eax` (`i`) to 0.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Jump to another instruction.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Compare `%eax` (`i`) to 0x63 (99) by calculating `%eax` – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

`jle` means "jump if less than or equal". This jumps if **`%eax`** <= 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Add 1 to `%eax` (`i`).

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Compare `%eax` (`i`) to 0x63 (99) by calculating `%eax` – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

`jle` means "jump if less than or equal". This jumps if **`%eax`** <= 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

We continue in this pattern until we do not make this conditional jump. When will that be?

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

We will stop looping when this comparison says that `%eax` – 0x63 > 0!

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Then, we return from the function.

Common While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Jump to test
Body
Test
Jump to body if success
```

From Previous Slide:

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Lecture Plan

- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Common While Loop Construction

C For loop

```
for (init; test; update) {
      body
}
```

C Equivalent While Loop

```
init
while(test) {
     body
     update
}
```

Assembly pseudocode

```
Init
Jump to test
Body
Update
Test
Jump to body if success
```

`for` loops and `while` loops are treated (essentially) the same when compiled down to assembly.

Back to Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```
0000000004005b6 <sum_array>:
  4005b6:          mov    $0x0,%edx
  4005bb<+5>:      mov    $0x0,%eax
  4005c0<+10>:     jmp    4005cb <sum_array+21>
  4005c2<+12>:     movslq %edx,%rcx
  4005c5<+15>:     add    (%rdi,%rcx,4),%eax
  4005c8<+18>:     add    $0x1,%edx
  4005cb<+21>:     cmp    %esi,%edx
  4005cd<+23>:     jl     4005c2 <sum_array+12>
  4005cf<+25>:     repz retq
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jl`** instructions doing?
   (**`jl`**: jump less; signed <)

Lecture Plan

- If Statements
- Loops
- Other Instructions That Depend On Condition Codes

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **`jmp`** instructions conditionally jump to a different next instruction
- **`set`** instructions conditionally set a byte to 0 or 1
- new versions of **`mov`** instructions conditionally move data

`set`: Read condition codes

`set` instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte register (e.g., `%al`) or single-byte memory location
- Does not perturb other bytes of register
- Typically followed by `movzbl` to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

`cmov`: Conditional move

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition `x` holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp    %edi,%esi
mov    %edi, %eax
cmovge %esi, %eax
retq
```

Ternary Operator

The ternary operator is a shorthand for using if/else to evaluate to a value.

```
condition ? expressionIfTrue : expressionIfFalse
```

```
int x;
if (argc > 1) {
    x = 50;
} else {
    x = 0;
}

// equivalent to
int x = argc > 1 ? 50 : 0;
```

cmov: Conditional move

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		Nonnegative (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Practice: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

(See Sec. 2.3.7)

Put x + 3 into %eax (add appropriate bias, 2^2-1)

To see whether x is negative, zero, or positive

If x is positive, put x into %eax

Divide %eax by 4

Extra Practice

Practice: Fill In The Blank

Note: .L2/.L3 are "labels" that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = _______;
    while (_________) {
      result = ____________;
      a = _________;
    }
    return result;
}
```

Common while loop construction:

Jump to test
Body
Test
Jump to body if success

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice: Fill In The Blank

Note: .L2/.L3 are "labels" that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = ___1___;
    while (__a < b__) {
      result = __result*(a+b)__;
      a = __a + 1__;
    }
    return result;
}
```

Common while loop construction:

Jump to test
Body
Test
Jump to body if success

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

Practice: "Escape Room"

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If statements
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Next time: Function calls in assembly

photo by CGP Grey

COMP201

Computer Systems & Programming

Lecture #18 – x86-64 Procedures

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- No labs this week!
- Midterm exam will be held on December 5, 2021 (Sunday) at 3pm at Sevgi Gönül Auditorium
  - More info on Blackboard

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If statements
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Practice: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

Practice: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Learning Assembly

Learning Goals

- Learn how assembly calls functions and manages stack frames.
- Learn the rules of register use when calling functions.

Plan for Today

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

`%rip`

- **`%rip`** is a special register that points to the next instruction to execute.
- **Let's dive deeper into how `%rip` works, and how jumps modify it.**

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

These are 0-based offsets in bytes for each instruction relative to the start of this function.

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

These are bytes for the machine code instructions. Instructions are variable length.

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xeb means **jmp**.

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0x03 is the number of instruction bytes to jump relative to **%rip**.

With no jump, **%rip** would advance to the next line. This **jmp** says to <u>then</u> go **3** bytes further!

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0x03 is the number of instruction bytes to jump relative to **%rip**.

With no jump, **%rip** would advance to the next line. This **jmp** says to <u>then</u> go **3** bytes further!

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0x73 means **jle**.

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xf8 is the number of instruction bytes to jump relative to `%rip`. This is -8 (in two's complement!).

With no jump, `%rip` would advance to the next line. This **jmp** says to <u>then</u> go **8** bytes back!

%rip

```
0000000000400570 <loop>:
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xf8 is the number of instruction bytes to jump relative to `%rip`. This is -8 (in two's complement!).

With no jump, `%rip` would advance to the next line. This **`jmp`** says to <u>then</u> go **8** bytes back!

Summary: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.
- `%rip` is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **`jmp`** instructions work by adjusting `%rip` by a specified amount.

Question Break

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

How do we call functions in assembly?

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

Terminology: **caller** function calls the **callee** function.

Lecture Plan

- Revisiting %rip
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

push

- The **`push`** instruction pushes the data at the specified source onto the top of the stack, adjusting **`%rsp`** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	**R[%rsp] ⟵ R[%rsp] - 8;** M[R[%rsp]] ⟵ S

push

- The **`push`** instruction pushes the data at the specified source onto the top of the stack, adjusting **`%rsp`** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` **`M[R[%rsp]] ⟵ S`**

push

- The **`push`** instruction pushes the data at the specified source onto the top of the stack, adjusting **`%rsp`** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

- This behavior is equivalent to the following, but `pushq` is a shorter instruction:

```
 subq $8, %rsp
movq  S, (%rsp)
```

- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for future data. More on this later!

pop

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **`%rsp`** accordingly.

Instruction	Effect
`popq D`	`D ⟵ M[R[%rsp]]` `R[%rsp] ⟵ R[%rsp] + 8;`

- **Note:** this *does not* remove/clear out the data! It just increments `%rsp` to indicate the next push can overwrite that location.

pop

- The **`pop`** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **`%rsp`** accordingly.

Instruction	Effect
`popq D`	`D ⟵ M[R[%rsp]]` `R[%rsp] ⟵ R[%rsp] + 8;`

- This behavior is equivalent to the following, but `popq` is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

Initially	
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Question Break

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must <u>remember</u> the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

Example: Remembering Where We Left Off

```
void multstore
 (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                 # Save %rbx
  400541: mov    %rdx,%rbx            # Save dest
  400544: callq  400550 <mult2>       # mult2(x,y)
  400549: mov    %rax,(%rbx)          # Save at dest
  40054c: pop    %rbx                 # Restore %rbx
  40054d: retq                        # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax           # a
  400553:  imul   %rsi,%rax           # a * b
  400557:  retq                       # Return
```

Example: Remembering Where We Left Off

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Example: Remembering Where We Left Off

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Example: Remembering Where We Left Off

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Example: Remembering Where We Left Off

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Call And Return

The **`call`** instruction pushes the address of the instruction immediately following the **`call`** instruction onto the stack and sets `%rip` to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

The **`ret`** instruction pops this instruction address from the stack and stores it in `%rip`.

```
ret
```

The stored `%rip` value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

What's left? Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Parameters and Return

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return value, it looks in `%rax` after the callee completes.

Example 1: Parameters and Return

```
void multstore
 (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  • • •
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax         # a
  400553:  imul   %rsi,%rax         # a * b
  # s in %rax
  400557:  retq                     # Return
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

main()

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400574 <+37>:   pushq   $0x3
0x400576 <+39>:   mov     $0x2,%r9d
0x40057c <+45>:   mov     $0x1,%r8d
0x400582 <+51>:   lea     0x10(%rsp),%rcx
0x400587 <+56>:   lea     0x14(%rsp),%rdx
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400582 <+51>:   lea    0x10(%rsp),%rcx
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
0x40059b <+76>:   add    $0x10,%rsp
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
0x40059b <+76>:   add    $0x10,%rsp
…
```

Example 2: Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x400546 <func>
0x40059b <+76>:   add    $0x10,%rsp
…
```

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Local Storage

```
long caller() {
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    ...
}
```

```
caller:
    subq $0x10, %rsp        // 16 bytes for stack frame
    movq $0x216, (%rsp)     // store 534 in arg1
    movq $0x421, 8(%rsp)    // store 1057 in arg2
    leaq 8(%rsp), %rsi      // compute &arg2 as second arg
    movq %rsp, %rdi         // compute &arg1 as first arg
    call swap_add           // call swap_add(&arg1, &arg2)
```

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Register Restrictions

- When procedure **yoo** calls **who**:
  - **yoo** is the **caller**
  - **who** is the **callee**
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

  - Contents of register **%rdx** overwritten by **who**
  - This could be trouble → something should be done!
    - Need some coordination

Register Restrictions

There is only one copy of registers for all programs and functions.

- **Problem:** what if ***funcA*** is building up a value in register `%r10`, and calls ***funcB*** in the middle, which also has instructions that modify `%r10`? ***funcA***'s value will be overwritten!
- **Solution:** make some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).

Register Restrictions

Caller-Owned (Callee Saved)

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned (Caller Saved)

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Caller-Owned Registers

`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

If `function1` wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers

```
function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
```

Callee-Owned Registers

`main` can use callee-owned registers but calling `function1` may permanently modify their values.

If `function1` wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers

```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

A Day In the Life of `function1`

Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Example: Recursion

- Let's look at an example of recursion at the assembly level.
- We'll use everything we've learned about registers, the stack, function calls, parameters, and assembly instructions!

https://godbolt.org/z/f43dz1

`factorial.c and factorial`

Our First Assembly

```
int sum_array(int arr[], int nelems) {
   int sum = 0;
   for (int i = 0; i < nelems; i++) {
      sum += arr[i];
   }
   return sum;
}
```

We're done with all our assembly lectures! Now we can fully understand what's going on in the assembly below, including how someone would call `sum_array` in assembly and what the **`ret`** instruction does.

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in **`%rax`**
- Pointers are addresses of values
  - On stack or global

Recap

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example

Next time: data stack frames

photo by CGP Grey

COMP201

Computer Systems & Programming

Lecture #19 – Data and Stack Frames

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

Recap: Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Recap: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.
- `%rip` is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **`jmp`** instructions work by adjusting `%rip` by a specified amount.

Recap: `%rsp`

- **`%rsp`** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).

Key idea: `%rsp` must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

Recap: push and pop

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

- The **`push`** instruction pushes the data at the specified source onto the top of the stack, adjusting **`%rsp`** accordingly.

Instruction	Effect
`popq D`	`D ⟵ M[R[%rsp]]` `R[%rsp] ⟵ R[%rsp] + 8;`

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **`%rsp`** accordingly.
- **Note:** this *does not* remove/clear out the data! It just increments `%rsp` to indicate the next push can overwrite that location.

Recap: Call And Return

The **`call`** instruction pushes the address of the instruction immediately following the **`call`** instruction onto the stack and sets **`%rip`** to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

The **`ret`** instruction pops this instruction address from the stack and stores it in **`%rip`**.

```
ret
```

The stored **`%rip`** value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

Recap: Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Recap: Register Restrictions

Caller-Owned (Callee Saved)

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned (Caller Saved)

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

63	31	15	7 0	
%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Recap: Caller-Owned Registers

```
function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
```

Recap: Callee-Owned Registers

```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

Recap: x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in **`%rax`**
- Pointers are addresses of values
  - On stack or global

Plan for Today

- Arrays
- Structures
- Floating Point

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

Lecture Plan

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
- Floating Point

Array Allocation

Basic Principle

T **A[***L***];**

- Array of data type *T* and length *L*
- Contiguously allocated region of *L****sizeof(***T***)** bytes in memory

```
char string[12];
```

```
int val[5];
```

```
double a[3];
```

```
char *p[3];
```

Array Access

- **Basic Principle**

  T **`A[`***L***`];`**

  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	x + 4
`&val[2]`	`int *`	x + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	x + 4 i

Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = {1,5,2,1,3};
zip_dig mit = {0,2,1,3,9};
zip_dig ku =  {3,4,4,5,0};
```

- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

```
zip_dig ku;
```

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Array Loop Example

```
void zincr(zip_dig z) {
  size_t i;
  for (i=0; i<ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Multidimensional (Nested) Arrays

Declaration

T `A[`*R*`][`*C*`];`

- 2D array of data type *T*
- *R* rows, *C* columns
- Type *T* element requires *K* bytes

Array Size

- *R* * *C* * *K* bytes

Arrangement

- Row-Major Ordering

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

Nested Array Row Access

Row Vectors

- **`A[i]`** is array of *C* elements
- Each element of type *T* requires *K* bytes
- Starting address **`A +`** *i * (C * K)*

```
int A[R][C];
```

Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax   # 5 * index
leaq pgh(,%rax,4),%rax    # pgh + (20 * index)
```

Row Vector

- `pgh[index]` is array of 5 `int`'s
- Starting address `pgh+20*index`

Machine Code

- Computes and returns address
- Compute as `pgh+4*(index+4*index)`

Nested Array Element Access

Array Elements

- `A[i][j]` is element of type *T*, which requires *K* bytes
- Address `A` $+ i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```

Nested Array Element Access Code

```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax   # 5*index
addl   %rax, %rsi            # 5*index+dig
movl   pgh(,%rsi,4), %eax    # M[pgh + 4*(5*index+dig)]
```

Array Elements

- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`
  `= pgh + 4*(5*index + dig)`

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ku  = { 3, 4, 4, 5, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ku};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s

Element Access in Multi-Level Array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```

```
salq    $2, %rsi            # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax        # return *p
ret
```

Computation

- Element access `Mem[Mem[univ+8*index]+4*digit]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

Array Element Accesses

Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```

- Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`

`Mem[Mem[univ+8*index]+4*digit]`

N × N Matrix Code

Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j) {
  return a[i][j];
}
```

Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j) {
  return a[IDX(n,i,j)];
}
```

Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

16 × 16 Matrix Access

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
# a in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi              # 64*i
addq    %rsi, %rdi            # a + 64*i
movl    (%rdi,%rdx,4), %eax   # M[a + 64*i + 4*j]
ret
```

Array Elements

- Address `A` $+ i * (C * K) + j * K$
- $C = 16, K = 4$

n × n Matrix Access

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j) {
  return a[i][j];
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx
imulq   %rdx, %rdi            # n*i
leaq    (%rsi,%rdi,4), %rax   # a + 4*n*i
movl    (%rax,%rcx,4), %eax   # a + 4*n*i + 4*j
ret
```

Array Elements

- Address `A` $+ i * (C * K) + j * K$
- $C = 16$, $K = 4$
- Must perform integer multiplication

Practice 1: Reverse Engineering

```
#define M ??
#define N ??

long P[M][N];
long Q[N][M];
long sum_elem(long i, long j)
{
    return P[i][j] + Q[j][i];
}
```

```
# long sum_elem(long i, long j)
# in %rdi, j in %rsi
sum_element:
  leaq  0(,%rdi,8), %rdx      Compute 8*i
  subq  %rdi, %rdx            Compute 7*i
  addq  %rsi, %rdx            Compute 7*i+j
  leaq  (%rsi,%rsi,4), %rax   Compute 5*j
  addq  %rax, %rdi            Compute i+5*j
  movq  Q(,%rdi,8), %rax      Retrieve M[Q+8*(5*j+i)]
  add   P(,%rdx,8), %rax      Add M[P+8*(7*i+j)]
  ret
```

What is the value of M and N?

M = 5 and N = 7

slido

Lecture Plan

- Arrays
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Structure Representation

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as **`r + 4*idx`**

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Following Linked List

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};

void set_val (struct rec *r, int val) {
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

Register	Value
%rdi	r
%esi	val

```
.L11:                          # loop:
  movslq  16(%rdi), %rax       #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)  #   M[r+4*i] = val
  movq    20(%rdi), %rdi       #   r = M[r+20]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

Practice 2: Reverse Engineering

Fill in the blanks by inspecting the assembly code generated by gcc.

```
struct test {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct test *next;
};

void st_init(struct test *st) {
    st->s.y  = st->s.x   ;
    st->p    = &(st->s.y);
    st->next = st        ;
}
```

```
# void st_init(struct test *st)
# st in %rdi
st_init:
    movl 8(%rdi), %eax      Get st->s.x
    movl %eax, 10(%rdi)     Save in st->s.y
    leaq 10(%rdi), %rax     Compute &(st->s.y)
    movq %rax, (%rdi)       Store in st->p
    movq %rdi, 12(%rdi)     Store st in st->next
    ret
```

Structures & Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Unaligned Data

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K

Alignment Principles

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

- 1 byte: `char`, ...
  - no restrictions on address
- 2 bytes: `short`, ...
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, ...
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long, char *`, ...
  - lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

Within structure:

- Must satisfy each element's alignment requirement

Overall structure placement

- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

Example:

- K = 8, due to **`double`** element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

Arrays of Structures

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- Compute array offset 12*idx
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8` (resolved during linking)

```
short get_j(int idx) {
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)

Practice 3: Alignment

slido

Determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64.

```
struct mystruct {
    int *a;
    float b;
    char c;
    short d;
    float e;
    double f;
    int g;
    char *h;
};
```

Field	*a	b	c	d	e	f	g	*h	Total	Alignment
Size	8	4	1	2	4	8	4	8	48	8
Offset	0	8	12	14	16	24	32	36		

4 bytes padded to satisfy alignment requirement

Rearranged structure with minimum wasted space:

Field	*a	f	h	b	e	g	d	c	Total	Alignment
Size	8	8	8	4	4	4	2	1	40	8
Offset	0	8	16	24	28	32	36	38		

1 bytes padded to satisfy alignment requirement

Lecture Plan

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
- Floating Point

Background

- History
  - x87 FP
    - Legacy, very ugly
  - Streaming SIMD Extensions (SSE) FP
    - SIMD: single instruction, multiple data
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

Programming with SSE3

XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers

- 8 16-bit integers

- 4 32-bit integers

- 4 single-precision floats

- 2 double-precision floats

- 1 single-precision float

- 1 double-precision float

Scalar & SIMD Operations

- Scalar Operations: Single Precision

- SIMD Operations: Single Precision

- Scalar Operations: Double Precision

FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, …
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss   %xmm1, %xmm0
ret
```

```
double dadd(double x, double y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addsd   %xmm1, %xmm0
ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1   # Copy v
movsd   (%rdi), %xmm0  # x = *p
addsd   %xmm0, %xmm1   # t = x + v
movsd   %xmm1, (%rdi)  # *p = t
ret
```

Other Aspects of FP Code

- Lots of instructions
  - Different operations, different formats, …
- Floating-point comparisons
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF
- Using constant values
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory

Recap

- Arrays
- Structures
- Floating Point

That's it for assembly!

Next time: security vulnerabilities, memory hierarchy

photo by unsplash.com user @tabithabrooke

COMP201

Computer Systems & Programming

Lecture #20 – Security Vulnerabilities

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Assignment 4 will be out tomorrow!
  - Don't miss Mert's extra office hour (time and date will be announced soon).

Recap

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Recap: Structures & Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Unaligned Data

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K

Recap: Alignment Principles

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Recap: Specific Cases of Alignment (x86-64)

- 1 byte: `char`, ...
  - no restrictions on address
- 2 bytes: `short`, ...
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, ...
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long, char *`, ...
  - lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Recap: Satisfying Alignment with Structures

Within structure:

- Must satisfy each element's alignment requirement

Overall structure placement

- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

Example:

- K = 8, due to **`double`** element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Recap: Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

Recap: Arrays of Structures

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

Recap: Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- Compute array offset `12*idx`
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8` (resolved during linking)

```
short get_j(int idx) {
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Recap: Programming with SSE3

XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers

- 8 16-bit integers

- 4 32-bit integers

- 4 single-precision floats

- 2 double-precision floats

- 1 single-precision float

- 1 double-precision float

Recap: Scalar & SIMD Operations

- Scalar Operations: Single Precision

- SIMD Operations: Single Precision

- Scalar Operations: Double Precision

Plan for Today

- Memory Layout
- Buffer Overflow

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

—Ruth Anderson's UW CSE 351 class

Lecture Plan

- Memory Layout
- Buffer Overflow

x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- Data
  - Statically allocated data
  - E.g., global vars, static vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

- Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590

Reminder: x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Arguments (if > 6 args) for this call
- Current/ Callee Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Caller-saved pushed before setting up arguments for a function call
  - Callee-saved pushed before using long-term registers
  - Local variables
    (if can't be kept in registers)
  - "Argument build" area
    (Need to call a function with >6 arguments? Put them here)

Lecture Plan

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault
```

Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault
```

Explanation:

Buffer Overflow in a Nutshell

- C does not check array bounds
  - Many Unix/Linux/C functions don't check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

- "Buffer Overflow" = Writing past the end of an array

- Characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows "backwards" in memory
  - Data and instructions both stored in the same memory

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Demo: Bufdemo

`bufdemo.c`

https://godbolt.org/z/vE5zWrzfr

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

btw, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:
0123456789012345678901 23
0123456789012345678901 23
```

```
unix>./bufdemo-nsp
Type a string:
0123456789012345678901234
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
 4006cf: 48 83 ec 18                sub    $0x18,%rsp
 4006d3: 48 89 e7                   mov    %rsp,%rdi
 4006d6: e8 a5 ff ff ff             callq  400680 <gets>
 4006db: 48 89 e7                   mov    %rsp,%rdi
 4006de: e8 3d fe ff ff             callq  400520 <puts@plt>
 4006e3: 48 83 c4 18                add    $0x18,%rsp
 4006e7: c3                         retq
```

call_echo:

```
 4006e8: 48 83 ec 08                sub    $0x8,%rsp
 4006ec: b8 00 00 00 00             mov    $0x0,%eax
 4006f1: e8 d9 ff ff ff             callq  4006cf <echo>
 4006f6: 48 83 c4 08                add    $0x8,%rsp
 4006fa: c3                         retq
```

Buffer Overflow Stack

Before call to gets

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp,
%rdi
  call  gets
  . . .
```

```
call_echo:

  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp,
%rdi
  call  gets
  . . .
```

```
call_echo:

  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:
0123456789012345678901 2
0123456789012345678901 2
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp,
%rdi
  call  gets
  . . .
```

```
call_echo:

  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nspunix>./bufdemo-nsp
Type a string:
01234567890123456789012 34
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp,
%rdi
  call  gets
  . . .
```

```
call_echo:

  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:
0123456789012345678901 23
0123456789012345678901 23
```

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Stack Example #3 Explained

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
register_tm_clones:
    . . .
    400600: mov    %rsp,%rbp
    400603: mov    %rax,%rdx
    400606: shr    $0x3f,%rdx
    40060a: add    %rdx,%rax
    40060d: sar    %rax
    400610: jne    400614
    400612: pop    %rbp
    400613: retq
```

"Returns" to unrelated code

Lots of things happen, without modifying critical state

Eventually executes `retq` back to main

Why is buffer overflow a big deal?

- Buffer overflows on the stack can overwrite "interesting" data
  - Attackers just choose the right inputs

- Simplest form (sometimes called "stack smashing")
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure

- Why is this a big deal?
  - It was the #1 *technical* cause of security vulnerabilities
    - #1 *overall* cause is social engineering / user ignorance

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack after call to `gets()`

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

Malicious Use of Buffer Overflow: Code Injection Attacks

```
void foo(){
  bar();
A:...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code

Question

- `smash_me` is vulnerable to stack smashing!
- What is the minimum number of characters that `gets` must read in order for us to change the return address to a stack address?
  - For example: (`0x00 00 7f ff CA FE F0 0D`)

    Always 0's

Previous stack frame			
00	00	00	00
00	40	05	d1
. . .			
			[0]

```
smash_me:
  subq   $0x40, %rsp
   ...
  leaq   16(%rsp), %rdi
  call   gets
   ...
```

A. 27
B. 30
C. 51
D. 54
E. We're lost...

Question

- `smash_me` is vulnerable to stack smashing!
- What is the minimum number of characters that `gets` must read in order for us to change the return address to a stack address?
  - For example: (`0x00 00 7f ff CA FE F0 0D`)

    Always 0's

Previous stack frame			
00	00	00	00
00	40	05	d1
. . .			
			[0]

```
smash_me:
  subq   $0x40, %rsp
   ...
  leaq   16(%rsp), %rdi
  call   gets
   ...
```

A. 27
B. 30
C. 51
D. 54 = 64 – 16 + 6
E. We're lost...

Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹️
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - Hacking embedding devices (e.g. cars, smart homes, planes)
- You will learn some of the tricks in Assignment 5
  - Hopefully to convince you to never leave such holes in your programs!!

Example: the original Internet worm (1988)

- Exploited a few vulnerabilities to spread
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@linuxpool.ku.edu.tr`
  - Worm attacked fingerd server by sending phony argument:
    - `finger “exploit-code  padding  new-return-address”`
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 🙂)
    - see June 1989 article in Comm. of the ACM
  - the young author of the worm was prosecuted…
  - and CERT was formed… homed at CMU

Example 2: IM War

July 1999:

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers

IM War (cont.)

August 1999:

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes
  - At least 13 such skirmishes
- What was really happening?
  - AOL had discovered a buffer overflow bug in their own AIM clients
  - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
  - When Microsoft changed code to match signature, AOL changed signature location

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

Aside: Worms and Viruses

- **Worm:** A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers

- **Virus:** Code that
  - Adds itself to other programs
  - Does not run independently

- Both are (usually) designed to spread among computers and to wreak havoc

Hacking Cars

- UW CSE research from 2010 demonstrated wirelessly hacking a car using buffer overflow
- Overwrote the onboard control system's code
  - Disable brakes
  - Unlock doors
  - Turn engine on/off

https://www.youtube.com/watch?v=hB6eY73sLV

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

For example, use library routines that limit string lengths

- `fgets` instead of `gets`
- `strncpy` instead of `strcpy`
- Don't use `scanf` with `%s` conversion specification
  - Use `fgets` to read the string
  - Or use `%ns` where `n` is a suitable integer

2. System-Level Protections can help

Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- E.g.: 5 executions of memory allocation code

local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

- Stack repositioned each time program executes

2. System-Level Protections can help

Nonexecutable code segments

- In traditional x86, can mark region of memory as either "read-only" or "writeable"
  - Can execute anything readable
- X86-64 added explicit "execute" permission
- Stack marked as non-executable

Any attempt to execute this code will fail

3. Stack Canaries can help

Idea:

- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

GCC Implementation

- `-fstack-protector`
- Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:
0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:
01234567
*** stack smashing detected ***
```

Protected Buffer Disassembly

```
echo:
  40072f: sub    $0x18,%rsp
  400733: mov    %fs:0x28,%rax
  40073c: mov    %rax,0x8(%rsp)
  400741: xor    %eax,%eax
  400743: mov    %rsp,%rdi
  400746: callq  4006e0 <gets>
  40074b: mov    %rsp,%rdi
  40074e: callq  400570 <puts@plt>
  400753: mov    0x8(%rsp),%rax
  400758: xor    %fs:0x28,%rax
  400761: je     400768 <echo+0x39>
  400763: callq  400580 <__stack_chk_fail@plt>
  400768: add    $0x18,%rsp
  40076c: retq
```

Setting Up Canary

Before call to gets

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: `0123456`

```
echo:
    . . .
    movq    8(%rsp), %rax      # Retrieve from stack
    xorq    %fs:40, %rax       # Compare to canary
    je      .L6                # If same, OK
    call    __stack_chk_fail   # FAIL
.L6:        . . .
```

Return-Oriented Programming Attacks

- Challenge (for hackers)
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- Alternative Strategy
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*
- Construct program from gadgets
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable

Gadget Example #1

```
long ab_plus_c
  (long a, long b, long c) {
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

`rax ← rdi + rdx`

Gadget address = `0x4004d4`

- Use tail end of existing functions

Gadget Example #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

Encodes `movq %rax, %rdi`

```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

`rdi ← rax`

Gadget address = `0x4004dc`

- Repurpose byte codes

ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

Recap

- Memory Layout
- Buffer Overflow

Next time: memory hierarchy

photo by unsplash.com user @martijnbaudoin

COMP201

Computer Systems & Programming

Lecture #20 – The Memory Hierarchy

Aykut Erdem // Koç University // Fall 2021

Recap

- Floating Point
- Memory Layout
- Buffer Overflow

Recap: x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- Data
  - Statically allocated data
  - E.g., global vars, static vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

Recap: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault
```

Result is system specific

Recap: Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹️
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
- You will learn some of the tricks in Assignment 5
  - Hopefully to convince you to never leave such holes in your programs!!

<u>COMP201 Topic 7</u>: How does the memory system is organized as a hierarchy of different storage devices with unique capacities?

Plan for Today

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

—Porter Jones' UW CSE 351 class

Lecture Plan

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy

Writing & Reading Memory

- **Write**
  - Transfer data from CPU to memory
    `movq %rax, 8(%rsp)`
  - “Store” operation

- **Read**
  - Transfer data from memory to CPU
    `movq 8(%rsp), %rax`
  - “Load” operation

Traditional Bus Structure Connecting CPU and Memory

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

Memory Read Transaction (1)

- CPU places address `A` on the memory bus.

Memory Read Transaction (2)

- Main memory reads `A` from the memory bus, retrieves word `x`, and places it on the bus.

Memory Read Transaction (3)

- CPU read word `x` from the bus and copies it into register `%rax`.

Memory Write Transaction (1)

- CPU places address `A` on bus. Main memory reads it and waits for the corresponding data word to arrive.

Memory Write Transaction (2)

- CPU places data word y on the bus.

Memory Write Transaction (3)

- Main memory reads data word `y` from the bus and stores it at address `A`.

Store operation: `movq %rax, A`

Lecture Plan

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy

Random-Access Memory (RAM)

- Key features
  - **RAM** is traditionally packaged as a chip.
  - Basic storage unit is normally a **cell** (one bit per cell).
  - Multiple RAM chips form a memory.

- RAM comes in two varieties:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)

DRAM

SRAM

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Need EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100X	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

EDC: Error detection and correction

Trends

- SRAM scales with semiconductor technology
  - Reaching its limits
- DRAM scaling limited by need for minimum capacitance
  - Aspect ratio limits how deep can make capacitor
  - Also reaching its limits

Enhanced DRAMs

- Operation of DRAM cell has not changed since its invention
  - Commercialized by Intel in 1970.
- DRAM cores with better interface logic and faster I/O:
  - Synchronous DRAM (**SDRAM**)
    - Uses a conventional clock signal instead of asynchronous control
  - Double data-rate synchronous DRAM (**DDR SDRAM**)
    - Double edge clocking sends two bits per cycle per pin
    - Different types distinguished by size of small prefetch buffer:
      - **DDR** (2 bits), **DDR2** (4 bits), **DDR3** (8 bits), **DDR4** (16 bits)
    - By 2010, standard for most server and desktop systems
    - Intel Core i7 supports DDR3 and DDR4 SDRAM

Storage Technologies

- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array

- Store as persistent charge
- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Magnetic Disks

- Store on magnetic medium
- Electromechanical access

Storage Technologies

- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array

- Store as persistent charge
- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Magnetic Disks

- Store on magnetic medium
- Electromechanical access

Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
  - Read-only memory (**ROM**): programmed during production
  - Electrically eraseable PROM (**EEPROM**): electronic erase capability
  - Flash memory: EEPROMs with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
  - 3D XPoint (Intel Optane) & emerging NVMs
    - New materials

- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
  - Disk caches

Storage Technologies

- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array

- Store as persistent charge
- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Magnetic Disks

- Store on magnetic medium
- Electromechanical access

What's Inside A Disk Drive?

Image courtesy of Seagate Technology

Disk Geometry

- Disks consist of **platters**, each with two **surfaces.**
- Each surface consists of concentric rings called **tracks.**
- Each track consists of **sectors** separated by **gaps.**

Disk Geometry (Muliple-Platter View)

- Aligned tracks form a cylinder.

Disk Capacity

- **Capacity**: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB) or terabytes (TB), where 1 GB = 10^9 bytes and 1 GB = 10^{12} bytes.

- Capacity is determined by these technology factors:
  - **Recording density** (bits/in): number of bits that can be squeezed into a 1-inch segment of a track.
  - **Track density** (tracks/in): number of tracks that can be squeezed into a 1-inch radial segment.
  - **Areal density** (bits/in2): product of recording and track density.

Disk Operation (Single-Platter View)

The disk surface spins at a fixed rotational rate

The read/write **head** is attached to the end of the **arm** and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)

Recording zones

- Modern disks partition tracks into disjoint subsets called **recording zones**
  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
  - So we use average number of sectors/track when computing capacity.

Computing Disk Capacity

Capacity = (# bytes/sector) × (avg. # sectors/track) × (# tracks/surface) × (# surfaces/platter) × (# platters/disk)

Example:

- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 × 300 × 20000 × 2 × 5
= 30,720,000,000
= 30.72 GB

Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized **logical blocks** (0, 1, 2, …)
- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface, track, sector) triples.
- Allows controller to set aside spare cylinders for each zone.
  - Accounts for the difference in “formatted capacity” and “maximum capacity”.

Disk Structure - top view of single platter

Surface organized into tracks

Tracks divided into sectors

Disk Access

Head in position above a track

Disk Access

Rotation is counter-clockwise

Disk Access – Read

About to read blue sector

Disk Access – Read

After BLUE read

After reading blue sector

Disk Access – Read

After BLUE read

Red request scheduled next

Disk Access – Seek

Seek to red's track

Disk Access – Rotational Latency

After BLUE read Seek for RED Rotational latency

Wait for red sector to rotate around

Disk Access – Read

After BLUE read | Seek for RED | Rotational latency | After RED read

Complete read of red

Disk Access – Service Time Components

Disk Access Time

- **Average time to access some target sector approximated by:**
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- **Seek time (Tavg seek)**
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms
- **Rotational latency (Tavg rotation)**
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 × 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs
- **Transfer time (Tavg transfer)**
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM × 1/(avg # sectors/track) × 60 secs/1 min.

Disk Access Time Example

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 × (60 secs/7200 RPM) × 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM × 1/400 secs/track × 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- Important points:
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower than DRAM.

I/O Bus

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

Reading a Disk Sector (1)

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special "interrupt" pin on the CPU)

Solid State Disks (SSDs)

- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

SSD Performance Characteristics

- Benchmark of Samsung 940 EVO Plus

Sequential read tput	2,126 MB/s	Sequential write tput	1,880 MB/s
Random read tput	140 MB/s	Random write tput	59 MB/s

https://ssd.userbenchmark.com/SpeedTest/711305/Samsung-SSD-970-EVO-Plus-250GB

- Sequential access faster than random access
  - Common theme in the memory hierarchy
- Random writes are somewhat slower
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - Flash translation layer allows accumulating series of small writes before doing block write.

SSD Tradeoffs vs Rotating Disks

- **Advantages**
  - No moving parts → faster, less power, more rugged
- **Disadvantages**
  - Have the potential to wear out
    - Mitigated by “wear leveling logic” in flash translation layer
    - E.g. Samsung 940 EVO Plus guarantees 600 writes/byte of writes before they wear out
    - Controller migrates data to minimize wear level
  - In 2019, about 4 times more expensive per byte
    - And, relative cost will keep dropping
- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

The CPU-Memory Gap

- The gap widens between DRAM, disk, and CPU speeds.

Locality to the Rescue!

- The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

Lecture Plan

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy

Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). Spatial locality
  - Reference variable sum each iteration. Temporal locality
- Instruction references
  - Reference instructions in sequence. Spatial locality
  - Cycle through loop repeatedly. Temporal locality

Lecture Plan

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.
  - For each level k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1

Example Memory Hierarchy

Example Memory Hierarchy

Example Memory Hierarchy

Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

<u>Big Idea:</u> The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Intel Core i7 Cache Hierarchy

L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

General Cache Concepts

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: **Hit!**

General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: **Miss!**

Block b is fetched from memory

Block b is stored in cache

- **Placement policy:** determines where b goes
- **Replacement policy**: determines which block gets evicted (victim)

Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Summary

- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called **locality**.
- Memory hierarchies based on **caching** close the gap by exploiting locality.
- Flash memory progress outpacing all other memory and storage technologies (DRAM, SRAM, magnetic disk)
  - Able to stack cells in three dimensions

Recap

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

Next: Cache memories

photo by unsplash.com user @martijnbaudoin

COMP201

Computer Systems & Programming

Lecture #22 – Cache Memories

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

Recap: Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

Recap: I/O Bus

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

Recap: Disk Access Time

- **Average time to access some target sector approximated by:**
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- **Seek time (Tavg seek)**
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms
- **Rotational latency (Tavg rotation)**
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 × 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs
- **Transfer time (Tavg transfer)**
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM × 1/(avg # sectors/track) × 60 secs/1 min.

Recap: Disk Access Time Example

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 × (60 secs/7200 RPM) × 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM × 1/400 secs/track × 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- Important points:
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower then DRAM.

Recap: Solid State Disks (SSDs)

- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

Recap: SSD Tradeoffs vs Rotating Disks

- **Advantages**
  - No moving parts → faster, less power, more rugged
- **Disadvantages**
  - Have the potential to wear out
    - Mitigated by "wear leveling logic" in flash translation layer
    - E.g. Samsung 940 EVO Plus guarantees 600 writes/byte of writes before they wear out
    - Controller migrates data to minimize wear level
  - In 2019, about 4 times more expensive per byte
    - And, relative cost will keep dropping
- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

Recap: The CPU-Memory Gap

- The gap widens between DRAM, disk, and CPU speeds.

Until 2003, DRAM and disk access times were decreasing more slowly than the cycle time of a processor.

Today, with the introduction of multiple cores, this performance gap is now more and more a function of throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.

Recap: Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

Recap: Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). Spatial locality
  - Reference variable sum each iteration. Temporal locality
- Instruction references
  - Reference instructions in sequence. Spatial locality
  - Cycle through loop repeatedly. Temporal locality

Recap: Memory Hierarchy

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers

Recap: Memory Hierarchy

Recap: Memory Hierarchy

Recap: Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Plan for Today

- Cache basics
- Principle of locality
- Cache organization

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

—Porter Jones' UW CSE 351 class

How does execution time grow with SIZE?

```
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    sum += array[j];
  }
}
```

Plot:

Actual Data

Processor-Memory Gap

1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Core 2 Duo:
Can process at least
256 Bytes/cycle

cycle: single machine step (fixed-time)

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)

Problem: lots of waiting on memory

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)

cycle: single machine step (fixed-time)

Solution: caches

Lecture Plan

- Cache basics
- Principle of locality
- Cache organization

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:

General Cache Concepts

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: **Hit!**

General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: **Miss!**

Block b is fetched from memory

Block b is stored in cache

- **Placement policy:** determines where b goes
- **Replacement policy**: determines which block gets evicted (victim)

Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Lecture Plan

- Cache basics
- Principle of locality
- Cache organization

Why Caches Work

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example 1

- Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example 1

- Does this function have good locality with respect to array a?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; j < M; j++)
        for (j = 0; i < N; i++)
            sum += a[i][j];
    return sum;
}
```

M = 3,
N=4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = ?

1) a[0][0]
2) a[0][1]
3) a[0][2]
4) a[0][3]
5) a[1][0]
6) a[1][1]
7) a[1][2]
8) a[1][3]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

Layout in Memory

Note: 76 is just one possible starting address of array a

Locality Example 2

- Does this function have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

M = 3,
N=4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Layout in Memory

Note: 76 is just one possible starting address of array a

Locality Example 2

- Does this function have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

M = 3,
N=4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = ?

1) a[0][0]
2) a[1][0]
3) a[2][0]
4) a[0][1]
5) a[1][1]
6) a[2][1]
7) a[0][2]
8) a[1][2]
9) a[2][2]
10) a[0][3]
11) a[1][3]
12) a[2][3]

Layout in Memory

a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

76 92 108

Note: 76 is just one possible starting address of array a

Locality Example 3

```
int sum_array_3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?

Layout in Memory (M = ?, N = 3, L = 4)

Locality Example 3

- Can you permute the loops so that the function scans the 3-d array a with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Layout in Memory (M = ?, N = 3, L = 4)

Cache Performance Metrics

- Huge difference between a cache hit and a cache miss
  - Could be 100x speed difference between accessing cache and main memory (measured in *clock cycles*)
- Miss Rate (MR)
  - Fraction of memory references not found in cache (misses / accesses) = 1 - Hit Rate
- Hit Time (HT)
  - Time to deliver a block in the cache to the processor
    - Includes time to determine whether the block is in the cache
- Miss Penalty (MP)
  - Additional time required because of a miss

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

- **This is why "miss rate" is used instead of "hit rate"**

Can we have more than one cache?

- Why would we want to do that?
  - Avoid going to memory!
- Typical performance numbers:
  - Miss Rate
    - L1 MR = 3-10%
    - L2 MR = Quite small (*e.g.* < 1%), depending on parameters, etc.
  - Hit Time
    - L1 HT = 4 clock cycles
    - L2 HT = 10 clock cycles
  - Miss Penalty
    - P = 50-200 cycles for missing in L2 & going to main memory
    - Trend: increasing!

Summary

- Memory Hierarchy
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics
- Cache Performance
  - Ideal case: found in cache (hit)
  - Bad case: not found in cache (miss), search in next level
  - Average Memory Access Time (AMAT) = HT + MR × MP
    - Hurt by Miss Rate and Miss Penalty

Lecture Plan

- Cache basics
- Principle of locality
- Cache organization

Cache Organization

- **Block Size (B):** unit of transfer between cache and main memory
  - Given in bytes and always a power of 2 (*e.g.* 64 bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

Cache Organization

- **Block Size (B):** unit of transfer between cache and main memory
  - Given in bytes and always a power of 2 (*e.g.* 64 bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!
- Offset field
  - Low-order $\log_2(B) = b$ bits of address tell you which byte within a block
    - (address) mod 2^n = n lowest bits of address
  - (address) modulo (# of bytes in a block)

	m - b bits	b bits
m-bit address: (refers to byte in memory)	Block Number	Block Offset

Question

slido

- If we have 6-bit addresses and block size B = 4 bytes, which block and byte does 0x15 refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	5	1
D.	5	5

E. We're lost…

Question

slido

- If we have 6-bit addresses and block size B = 4 bytes, which block and byte does 0x15 refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	**5**	**1**
D.	5	5
E.	We're lost…	

0x 1 5

Address: 0b 0 1 0 1 0 1

Offset width = $\log_2(B) = \log_2(4) = 2$ bits

Cache Organization

- **Cache Size (C):** amount of *data* the cache can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes (C) or number of blocks (C/B)
  - Example: C = 32 KiB = 512 blocks if using 64-byte blocks

- Where should data go in the cache?
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**

- What is a data structure that provides fast lookup?
  - Hash table!

Review: Hash Tables for Fast Lookup

Insert:

5
27
34
102
119

Apply hash function to map data to “buckets”

hash(27) % N (10) = 7

Place Data in Cache by Hashing Address

- Map to *cache index* from block number
  - Use next $\log_2(C/B)$ = **s** bits in the address (after offset bits)
    - *C/B* is the number of sets here
  - (block number) mod (# blocks in cache)

Place Data in Cache by Hashing Address

- Map to *cache index* from block number
  - Let's adjacent blocks fit in cache simultaneously!
    - Consecutive blocks go in consecutive cache indices

Practice Question

- 6-bit addresses, block size B = 4 bytes, and our cache holds S = 4 blocks.
- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?

Place Data in Cache by Hashing Address

- Collision!
  - This might confuse the cache later when we access the data
  - Solution?

Tags Differentiate Blocks in Same Index

- Tag = rest of address bits
  - **t** bits = **m** – **s** – **b**
  - Check this during a cache lookup

Checking for a Requested Address

- CPU sends address request for chunk of data
  - Address and requested data are not the same thing!
    - Analogy: your friend ≠ their phone number

- TIO address breakdown:

  - Index field tells you where to look in cache
  - Tag field lets you check that data is the block you want
  - Offset field selects specified start byte within block

    - **Note:** **t** and **s** sizes will change based on hash function

Checking for a Requested Address Example

- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is C/B = 8 sets).
  - Offset bits (b) = $\log_2(B)$ = 2 bits
  - Index bits (s) = $\log_2$(number of sets) = 3 bits
  - Tag bits (t) = Rest of the bits in the address = 8 – 2 – 3 = 3 bits

- What are the fields for address 0xBA?
  - Tag bits (unique id for block): 101
  - Index bits (cache set block maps to): 110
  - Offset bits (byte offset within block): 10

5 6 2

Checking for a Requested Address Example

- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is C/B = 8 sets).
  - Offset bits (b) = $\log_2(B)$ = 2 bits
  - Index bits (s) = $\log_2$(number of sets) = 3 bits
  - Tag bits (t) = Rest of the bits in the address = 8 – 2 – 3 = 3 bits

- What are the fields for address 0xBA?
  - Tag bits (unique id for block): 0x5
  - Index bits (cache set block maps to): 0x6
  - Offset bits (byte offset within block): 0x2

101 110 10

5 6 2

Cache Puzzle

slido

- Based on the following behavior, which of the following block sizes is NOT possible for our cache?
  - Cache starts *empty*, also known as a **cold cache**
  - Access (addr: hit/miss) stream:
    - (14: miss), (15: hit), (16: miss)

A. 4 bytes

B. 8 bytes

C. 16 bytes

D. 32 bytes

E. We're lost…

Cache Puzzle

slido

- Based on the following behavior, which of the following block size is NOT possible for our cache?
  - Cache starts *empty*, also known as a **cold cache**
  - Access (addr: hit/miss) stream:
    - (14: miss), (15: hit), (16: miss)

hit: block is already in cache!
miss: block is not in cache, pulls block from memory and puts it in cache

A. 4 bytes

B. 8 bytes

C. 16 bytes

D. 32 bytes

E. We're lost…

1 Pulls block /w /14 into cache

2 15 is in the same block at 14

3 16 is not in block w/ 14 and 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Mem																	
K = 4															X	✓	X
K = 8															X	✓	X
K = 16															X	✓	X
K = 32															X	✓	✓

Direct-Mapped Cache Problem

- What happens if we access the following addresses?
  - 8, 24, 8, 24, 8, …?
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?

Associativity

- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**

Cache Organization

- **Associativity (E)**: # of ways for each set
  - Such a cache is called an "*E-way set associative cache*"
  - We now index into cache *sets*, of which there are S = C/B/E
  - Use lowest $\log_2(C/B/E)$ = **s** bits of block address
    - Direct-mapped: E = 1, so **s** = $\log_2(C/K)$ as we saw previously
    - Fully associative: E = C/K, so **s** = 0 bits

Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

$t = m - s - b$ $s = \log2(C/B/E)$ $b = \log_2(B)$

m-bit address:	Tag(t)	Index (s)	Offset (b)

s = ?
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

s = ?
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

s = ?
4-way set associative

Set	Tag	Data
0		
1		

Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

E = 4
E = 2
E = 1

$t = m - s - b$ $s = \log2(C/B/E)$ $b = \log_2(B)$

m-bit address:	Tag(t)	Index (s)	Offset (b)

$s = \log_2(8)=3$ bits
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		✓
4		
5		
6		
7		

$s = \log_2(8/2)=2$ bits
2-way set associative

Set	Tag	Data
0		
1		
2		
3		✓
		✓

$s = \log_2(8/4)=1$ bit
4-way set associative

Set	Tag	Data
0		
1		✓
		✓
		✓
		✓

Block Placement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to **least recently used (LRU)** (hardware usually implements "*not most recently used*")

Question

slido

- We have a cache of size 2 KiB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?

  A. 2

  B. 4

  C. 8

  D. 16

  E. We're lost…

- If addresses are 16 bits wide, how wide is the Tag field?

Question

slido

(C = $2*2^{10}$ bytes) (B = 2^7 bytes)

- We have a cache of size 2 KB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?

  (S = 2)

  A. 2

  B. 4

  C. 8

  D. 16

  E. We're lost…

num blocks = C / K = $2^{11}/2^7 = 2^4$ = 16 blocks

blocks per set = E = 16 /2 = 8

- If addresses are 16 bits wide, how wide is the Tag field?

Recap

- Cache basics
- Principle of locality
- Cache organization

Next time: More cache memories

photo by unsplash.com user @martijnbaudoin

COMP201

Computer Systems & Programming

Lecture #23 – More Cache Memories

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Cache basics
- Principle of locality
- Cache memory organization and operation

Recap: Cache Parameters

- **Block size (B):** basic unit of transfer between memory and the cache, given in bytes (e.g. 64 B).
- **Cache size (C):** Total amount of data that can be stored in the cache, given in bytes (e.g. 32 KiB).
  - Must be multiple of block size
  - Number of blocks in cache is calculated by C/K
- **Associativity (E):** Number of ways blocks can be stored in a cache set, or how many blocks in each set
- **Number of sets (S):** Number of unique sets that blocks can be placed into in a cache (calculated as C/K/E).

Recap: TIO address breakdown

- TIO address breakdown:

  - **Index (s)** field tells you where to look in cache
    - Number of bits is determined by number of sets ($\log_2(C/B/E)$)
    - Need enough bits to reference every set in the cache
  - **Tag (t)** field lets you check that data is the block you want
    - Rest of the bits not used for index and offset (**m – s – b**)
  - **Offset (b)** field selects specified start byte within block
    - Number of bits is determined by block size ($\log_2(B)$)
    - Need enough bits to reference every byte in a block

Plan for Today

- Cache memory organization and operation
- Writing cache-friendly code

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

—Porter Jones' UW CSE 351 class

Lecture Plan

- Cache memory organization and operation
- Performance impact of caches

Direct-Mapped Cache Problem

- What happens if we access the following addresses?
  - 8, 24, 8, 24, 8, …?
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?

Associativity

- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**

Cache Organization

- **Associativity (E)**: # of ways for each set
  - Such a cache is called an "*E-way set associative cache*"
  - We now index into cache *sets*, of which there are S = C/B/E
  - Use lowest $\log_2(C/B/E)$ = **s** bits of block address
    - Direct-mapped: E = 1, so **s** = $\log_2(C/K)$ as we saw previously
    - Fully associative: E = C/K, so **s** = 0 bits

Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

$t = m - s - b$ $s = \log2(C/B/E)$ $b = \log_2(B)$

m-bit address:	Tag(t)	Index (s)	Offset (b)

$s = ?$
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

$s = ?$
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

$s = ?$
4-way set associative

Set	Tag	Data
0		
1		

Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

E = 4
E = 2
E = 1

$t = m - s - b$ $s = \log2(C/B/E)$ $b = \log_2(B)$

m-bit address:	Tag(t)	Index (s)	Offset (b)

$s = \log_2(8)=3$ bits
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		✓
4		
5		
6		
7		

$s = \log_2(8/2)=2$ bits
2-way set associative

Set	Tag	Data
0		
1		
2		
3		✓
		✓

$s = \log_2(8/4)=1$ bit
4-way set associative

Set	Tag	Data
0		
1		✓
		✓
		✓
		✓

Block Placement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to **least recently used (LRU)** (hardware usually implements "*not most recently used*")

Question

slido

- We have a cache of size 2 KiB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?

  A. 2

  B. 4

  C. 8

  D. 16

  E. We're lost…

- If addresses are 16 bits wide, how wide is the Tag field?

Question

slido

($C = 2*2^{10}$ bytes) ($B = 2^7$ bytes)

- We have a cache of size 2 KiB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?

  ($S = 2$)

  A. 2

  B. 4

  C. 8

  D. 16

  E. We're lost…

num blocks $= C / K = 2^{11}/2^7 = 2^4 = 16$ blocks

blocks per set $= E = 16 / 2 = 8$

- If addresses are 16 bits wide, how wide is the Tag field?

General Cache Organization (S, E, B)

Cache size:
C = S × E x B data bytes

Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block, S=4 sets, E=1 Blocks/set

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	miss
1	$[0\underline{00}1_2]$,	hit
7	$[0\underline{11}1_2]$,	miss
8	$[1\underline{00}0_2]$,	miss
0	$[0\underline{00}0_2]$	miss

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[$00\underline{0}0_2$],	miss
1	[$00\underline{0}1_2$],	hit
7	[$01\underline{1}1_2$],	miss
8	[$10\underline{0}0_2$],	miss
0	[$00\underline{0}0_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Write-back, Write Allocate Example

Note: While unrealistic, this example assumes that all requests have offset 0 and are for a block's worth of data.

Write-back, Write Allocate Example

Not valid x86, just using block num instead of full byte address to keep the example simple

1) `mov $0xFACE, (F)`

Write Miss!

Cache:

Step 1: Bring F into cache

Memory:

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`

Write Miss

Cache:

Valid	Dirty	Tag	Block Contents
1	0	F	0xCAFE

Memory:

Block Num	
	⋮
F	0xCAFE
	⋮
G	0xBEEF
	⋮

Step 1: Bring F into cache

Step 2: Write 0xFACE to cache only and set the dirty bit

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`

Write Miss

Cache:

Valid	Dirty	Tag	Block Contents
1	1	F	0xFACE

Memory:

Block Num	
	⋮
F	0xCAFE
	⋮
G	0xBEEF
	⋮

Step 1: Bring `F` into cache

Step 2: Write `0xFACE` to cache only and set the dirty bit

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
Write Miss

(2) `mov $0xFEED, (F)`
Write Hit!

Cache:

Step: Write `0xFEED` to cache only (and set the dirty bit)

Memory:

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
Write Miss

(2) `mov $0xFEED, (F)`
Write Hit!

Cache:

Memory:

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
Write Miss

(2) `mov $0xFEED, (F)`
Write Hit!

(3) `mov (G), %ax`
Read Miss!

Cache:

Step 1: Write F back to memory since it is dirty

Memory:

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
Write Miss

(2) `mov $0xFEED, (F)`
Write Hit!

(3) `mov (G), %ax`
Read Miss!

Cache:

Memory:

Step 1: Write `F` back to memory since it is dirty

Step 2: Bring `G` into the cache so that we can copy it into `%ax`

Cache Simulator

https://courses.cs.washington.edu/courses/cse351/cachesim

Polling Question

- Which of the following cache statements is FALSE?
  A. We can reduce compulsory misses by decreasing our block size
  B. We can reduce conflict misses by increasing associativity
  C. A write-back cache will save time for code with good temporal locality on writes
  D. A write-through cache will always match data with the memory hierarchy level below it
  E. We're lost...

Polling Question

- Which of the following cache statements is FALSE?
  A. We can reduce compulsory misses by decreasing our block size

     smaller block size pulls fewer bytes into cache on a miss

  B. We can reduce conflict misses by increasing associativity

     more options to place blocks before evictions occur

  C. A write-back cache will save time for code with good temporal locality on writes

     yes, its main goal is data consistency

  D. A write-through cache will always match data with the memory hierarchy level below it

     frequently-used blocks rarely get evicted, so fewer write-backs

  E. We're lost...

Intel Core i7 Cache Hierarchy

L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Lecture Plan

- Cache memory organization and operation
- Performance impact of caches

Lecture Plan

- Cache basics
- Principle of locality
- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain:** Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array “data” with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

For each elems and stride:

1. Call test() once to warm up the caches.

2. Call test() again and measure the read throughput(MB/s)

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Lecture Plan

- Cache basics
- Principle of locality
- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

$$c_{ij} = \sum_{k=1}^{n} a_{ij} \times b_{kj}$$

Matrix Multiplication Example

- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` held in register

`matmult/mm.c`

Miss Rate Analysis for Matrix Multiply

- Assume
  - Block size = 32B (<u>big enough for four doubles</u>)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- **Analysis Method:**
  - Look at access pattern of inner loop

Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations

- Stepping through columns in one row:

```
for (i = 0; i < N; i++)
  sum += a[0][i];
```

  - accesses successive elements
  - if block size (B) > sizeof(aij) bytes, exploit spatial locality

    miss rate = sizeof(aij) / B

- Stepping through rows in one column:

```
for (i = 0; i < n; i++)
  sum += a[i][0];
```

  - accesses distant elements
  - no spatial locality!

    miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Core i7 Matrix Multiply Performance

Lecture Plan

- Cache basics
- Principle of locality
- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
             for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

Cache Miss Analysis

- Assume
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 + n = 9n/8 misses

  - Afterwards in cache: (schematic)

Cache Miss Analysis

- Assume
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses

- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

Block size B x B

Cache Miss Analysis

- Assume
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)

  - Afterwards in cache (schematic)

Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **Second (block) iteration**:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

- **No blocking:** $(9/8) * n^3$
- **Blocking:** $1/(4B) * n^3$

- **Suggest largest possible block size B, but limit $3B^2 < C$!**

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Naïve vs. Blocked Matrix Multiplication

Naïve Multiplication

Blocked Multiplication

≈ 1,020,000 cache misses

≈ 90,000 cache misses

Video credit: Vasily Volkov

Recap

- Cache memories can have significant performance impact
- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Next time: Debugging and Design

The Alchemist by Mattheus von Helmont

COMP201

Computer Systems & Programming

Lecture #24 – Debugging, Design and Optimization

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Recap: The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Recap: Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Recap: Matrix Multiplication

Recap: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
             for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

- **No blocking:** $(9/8) * n^3$

Recap: Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

- **No blocking:** $(9/8) * n^3$
- **Blocking:** $1/(4B) * n^3$

Block size B x B

Plan for Today

- Debugging
- Design
- Optimization

Disclaimer: Slides for this lecture were borrowed from

—Michael Hilton and Brian Railing's CMU 15-213 class

—Nick Troccoli's Stanford CS107 class

Learning Goals

- Describe the steps to debug complex code failures
- Identify ways to manage the complexity when programming
- State guidelines for communicating the intention of the code
- Understand how we can optimize our code to improve efficiency and speed
- Learn about the optimizations GCC can perform

Lecture Plan

- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- Design
- Optimization

https://www.reddit.com/r/ProgrammerHumor/comments/1r0cw7/the_5_stages_of_debugging/

Defects and Infections

1. The programmer creates a defect
2. The defect causes an infection
3. The infection propagates
4. The infection causes a failure

Curse of Debugging

- Not every defect causes a failure!

- "Testing can only show the presence of errors – not their absence."

  – Edsger W. Dijkstra, 1972

Defects to Failures

- Code with defects will introduce erroneous or “infected” state
  - Correct code may propagate this state
  - Eventually an erroneous state is observed
- Some executions will not trigger the defect
  - Others will not propagate “infected” state
- Debugging sifts through the code to find the defect

Explicit Debugging

- Stating the problem
  - Describe the problem aloud or in writing
    - A.k.a. “Rubber duck” or “teddy bear” method
  - Often a comprehensive problem description is sufficient to solve the failure

Explicit Debugging

- Stating the p
  - Describe tl
    - A.k.a. "Ru
  - Often a co

:he failure

Rubber Duck Debugging
Debugging software with a rubber ducky

About Talk to a Duck

Rubber Duck Debugging

The rubber duck debugging method is as follows:

1. Beg, borrow, steal, buy, fabricate or otherwise obtain a rubber duck (bathtub variety).
2. Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.
3. Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.
4. At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

Note: In a pinch a coworker might be able to substitute for the duck, however, it is often preferred to confide mistakes to the duck instead of your coworker.

Original Credit: ~Andy from lists.ethernal.org

Scientific Debugging

- Before debugging, you need to construct a hypothesis as to the defect
  - Propose a possible defect and why it explains the failure conditions
- Ockham's Razor – given several hypotheses, pick the simplest / closest to current work

Scientific Debugging

- Make predictions based on your hypothesis
  - What do you expect to happen under new conditions
  - What data could confirm or refute your hypothesis
- How can I collect that data?
  - What experiments?
  - What collection mechanism?
- Does the data refute the hypothesis?
  - Refine the hypothesis based on the new inputs

Scientific Debugging

- A set of experiments has confirmed the hypothesis
  - This is the diagnosis of the defect
- Develop a fix for the defect
- Run experiments to confirm the fix
  - Otherwise, how do you know that it is fixed?

Code with a Bug

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
            i, fib(i));
```

```
$ gcc -o fib fib.c
$ ./fib
fib(9)=55
fib(8)=34
...
fib(2)=2 fib(1)=134513905
```

A defect has caused a failure.

Constructing a Hypothesis

- Specification defined the first Fibonacci number as 1
  - We have observed working runs (e.g., `fib(2)`)
  - We have observed a failing run
  - We then read the code

- `fib(1)` failed // Hypothesis

Code	Hypothesis
`for (i = 9; …)`	Result depends on order of calls
`while (n > 1) {`	Loop check is incorrect
`int f;`	`f` is uninitialized

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
                 i, fib(i));
```

Prediction

- Propose a new condition or conditions
  - What will logically happen if your hypothesis is correct?
  - What data can be

- `fib(1)` failed // Hypothesis
  - // Result depends on order of calls
    - If `fib(1)` is called first, it will return correctly.
  - // Loop check is incorrect
    - Change to `n >= 1` and run again.
  - // `f` is uninitialized
    - Change to int `f = 1;`

Experiment

- Identical to the conditions of a prior run
  - Except with one condition changed
- Conditions
  - Program input, using a debugger, altering the code

- `fib(1)` failed // Hypothesis
  - If `fib(1)` is called first, it will return correctly.
    - Fails.
  - Change to `n >= 1`
    - `fib(1)=2`
    - `fib(0)=...`
  - Change to `int f = 1;`
    - Works. Sometimes a prediction can be a fix.

Observation

- What is the observed result?
  - Factual observation, such as “Calling `fib(1)` will return 1.”
  - The conclusion will interpret the observation(s)

- Don’t interfere.
  - `printf()` can interfere
  - Like quantum physics, sometimes observations are part of the experiment

- Proceed systematically.
  - Update the conditions incrementally so each observation relates to a specific change

Debugging Tools

Observing program state can require a variety of tools

- Debugger (e.g., `gdb`)
  - What state is in local / global variables (if known)
  - What path through the program was taken

- `valgrind`
  - Does execution depend on uninitialized variables
  - Are memory accesses ever out-of-bounds

Diagnosis

- A scientific hypothesis that explains current observations and makes future predictions becomes a theory
  - We'll call this a diagnosis

- Use the diagnosis to develop a fix for the defect
  - Avoid post hoc, ergo propter hoc fallacy
  - Or correlation does not imply causation

- Understand why the defect and fix relate

Fix and Confirm

- Confirm that the fix resolves the failure

- If you fix multiple perceived defects, which fix was for the failure?
  - Be systematic

David Amador
@DJ_Link

when a "simple" bug is found and we start looking at the code

12:22 PM · Jul 3, 2017

2,181 Retweets **3,400** Likes

Hal replacing a light bulb - Malcolm in the Middle, S03E06

Learn

- Common failures and insights
  - Why did the code fail?
  - What are my common defects?

- Assertions and invariants
  - Add checks for expected behavior
  - Extend checks to detect the fixed failure

- Testing
  - Every successful set of conditions is added to the test suite

Quick and Dirty

- Not every problem needs scientific debugging
  - Set a time limit: (for example)
    - 0 – 10 minutes – Informal Debugging
    - 10 – 60 minutes – Scientific Debugging
    - > 60 minutes – Take a break / Ask for help

Code Smells

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Lecture Plan

- Debugging
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments
- Optimization

Design

- A good design needs to achieve many things:
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

Design

Good Design does:

Complexity Management

&

Communication

Complexity

- There are well known limits to how much complexity a human can manage easily.

VOL. 63, NO. 2 MARCH, 1956

THE PSYCHOLOGICAL REVIEW

THE MAGICAL NUMBER SEVEN, PLUS OR MINUS TWO: SOME LIMITS ON OUR CAPACITY FOR PROCESSING INFORMATION [1]

GEORGE A. MILLER

Harvard University

Complexity Management

- However, patterns can be very helpful…

COGNITIVE PSYCHOLOGY 4, 55–81 (1973)

Perception in Chess[1]

WILLIAM G. CHASE AND HERBERT A. SIMON
Carnegie–Mellon University

This paper develops a technique for isolating and studying the perceptual structures that chess players perceive. Three chess players of varying strength — from master to novice — were confronted with two tasks: (1) A perception task, where the player reproduces a chess position in plain view, and (2) de Groot's (1965) short-term recall task, where the player reproduces a chess position after viewing it for 5 sec. The successive glances at the position in the perceptual task and long pauses in the memory task were used to segment the structures in the reconstruction protocol. The size and nature of these structures were then analyzed as a function of chess skill.

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- DRY
- Abstraction
- Information Hiding
- …

Managing Complexity

- Given the many ways to manage complexity
  - Design code to be testable
  - Try to reuse testable chunks

Complexity Example

- Split a cache access into three+ testable components
  - State all of the steps that a cache access requires
  - Which steps depend on the operation being a load or a store?

Complexity Example

- Split a cache access into three+ testable components
  - State all of the steps that a cache access requires

    Convert address into tag, set index, block offset

    Look up the set using the set index

    Check if the tag matches any line in the set

    If so, hit

    If not a match, miss, then

    Find the LRU block

    Evict the LRU block

    Read in the new line from memory

    Update LRU

    Update dirty if the access was a store

  - Which steps depend on the operation being a load or a store?

Designs need to be testable

- Testable design
  - Testing versus Contracts
  - These are complementary techniques

- Testing and Contracts are
  - Acts of design more than verification
  - Acts of documentation

Testing Example

- For your cache simulator, you can write your own traces
  - Write a trace to test for a cache hit

    L 50, 1
    L 50, 1

  - Write a trace to test dirty bytes in cache

    S 100, 1

Communication

When writing code, the author is communicating with:

- The machine
- Other developers of the system
- Code reviewers
- Their future self

Communication

There are many techniques that have been developed around code communication:

- Tests
- Naming
- Comments
- Commit Messages
- Code Review
- Design Patterns
- …

Naming

Avoid deliberately meaningless names:

Naming is understanding

“If you don’t know what a thing should be called, you cannot know what it is.

If you don’t know what it is, you cannot sit down and write the code.” - Sam Gardiner

Better naming practices

1. Start with meaning and intention
2. Use words with precise meanings (avoid “data”, “info”, “perform”)
3. Prefer fewer words in names
4. Avoid abbreviations in names
5. Use code review to improve names
6. Read the code out loud to check that it sounds okay
7. Actually rename things

Naming guidelines – Use dictionary words

- Only use dictionary words and abbreviations that appear in a dictionary.
  - For example: FileCpy -> FileCopy
  - Avoid vague abbreviations such as acc, mod, auth, etc..

Avoid using single-letter names

- Single letters are unsearchable
  - Give no hints as to the variable's usage

- Exceptions are loop counters
  - Especially if you know why `i`, `j`, etc were originally used

Limit name character length

“Good naming limits individual name length, and reduces the need for specialized vocabulary” – Philip Relf

Limit name word count

- Keep names to a four words maximum
- Limit names to the number of words that people can read at a glance.

- Which of each pair do you prefer?

```
a1) arraysOfSetsOfLinesOfBlocks
a2) cache

b1) evictedData
b2) evictedDataBytes
```

Describe Meaning

- Use descriptive names.
- Avoid names with no meaning: `a`, `foo`, `blah`, `tmp`, etc

- There are reasonable exceptions:

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Use a large vocabulary

- Be more specific when possible:
  - `Person -> Employee`

- What is size in this `binaryTree`?

```
struct binaryTree {
  int size;
  …
};
```

Use problem domain terms

- Use the correct term in the problem domain's language.
  - Hint: as a student, consider the terms in the assignment

- In Assignment 5, consider the following:

```
username

secret_pin
```

Use opposites precisely

- Consistently use opposites in standard pairs
  - `first/end` -> `first/last`

Comments

Don’t Comments

- Don’t say what the code does
  - because the code already says that
- Don’t explain awkward logic
  - improve the code to make it clear
- Don’t add too many comments
  - it’s messy, and they get out of date

Awkward Code

Imagine someone (TA, employer, etc) has to read your code

- Would you rather rewrite or comment the following?

```
(*(void **)((*(void **)(bp)) + DSIZE)) = (*(void **)(bp + DSIZE));
```

- How about?

```
bp->prev->next = bp->next;
```

- Both lines update program state in the same way.

Do Comments

- Answer the question: why the code exists

- When should I use this code?
- When shouldn’t I use it?
- What are the alternatives to this code?

Why does this exist?

- Explain why a magic number is what it is.

```
// Each address is 64-bit, which is 16 + 1 hex characters
const int MAX_ADDRESS_LENGTH = 17;
```

- When should this code be used? Is there an alternative?

```
unsigned power2(unsigned base, unsigned expo){
    unsigned i;
    unsigned result = 1;
    for(i=0;i<expo;i++){
        result+=result;
    }
    return result;
}
```

How to write good comments

1. Write short comments of what the code will do.
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

```
// Initialize locals
// Pick a pivot value
// Reorder array around the pivot
// Recurse
```

How to write good comments

1. Write short comments of what the code will do.
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

2. Write that code.

3. Revise comments / code
    1. If the code or comments are awkward or complex
    2. Join / Split comments as needed

4. Maintain code and comments

Commit Messages

- Committing code to a source repository is a vital part of development
  - Protects against system failures and typos:
    - `cat foo.c` versus `cat > foo.c`
  - The commit messages are your record of your work
    - Communicating to your future self
    - Describe in one line what you did

  "Parses command line arguments"

  "fix bug in unique tests, race condition not solved"

  "seg list finished, performance is ..."

Debugging and Design

- Programs have defects
  - Be systematic about finding them
- Programs are more complex than humans can manage
  - Write code to be manageable
- Programming is not solitary, even if you are communicating with a grader or a future self
  - Be understandable in your communication

Lecture Plan

- Debugging
- Design
- Optimization
  - What is optimization?
  - GCC Optimization
  - Limitations of GCC Optimization
  - Caching revisited

Optimization

- Optimization is the task of making your program faster or more efficient with space or time. Later you will learn about explorations of efficiency with Big-O notation!
- *Targeted, intentional* optimizations to alleviate bottlenecks can result in big gains. But it's important to only work to optimize where necessary.

Optimization

Most of what <u>you</u> need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things thing a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
3) **Let `gcc` do its magic from there**
4) Optimize explicitly as a last resort

GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc -O0  // mostly just literal translation of C`
  - `gcc -O2  // enable nearly all reasonable optimizations`
  - (we use `-Og`, like `-O0` but with less needless use of the stack)

- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3  //more aggressive than -O2, trade size for speed`
  - `-Os  //optimize for size`
  - `-Ofast  //disregard standards compliance (!!)`

- **Exhaustive list of gcc optimization-related flags:**
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Example: Matrix Multiplication

Here's a standard matrix multiply, a triply-nested for loop:

```
void mmm(double a[][DIM], double b[][DIM], double c[][DIM], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

```
./mult          // -O0 (no optimization)
matrix multiply  25^2: cycles   0.43M
matrix multiply  50^2: cycles   3.02M
matrix multiply 100^2: cycles  24.82M
```

```
./mult_opt     // -O2 (with optimization)
matrix multiply  25^2: cycles   0.13M (opt)
matrix multiply  50^2: cycles   0.66M (opt)
matrix multiply 100^2: cycles   5.55M (opt)
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- ~~The Force~~

(may be not 🤓)

GCC Optimizations

Optimizations may target one or more of:

- Static instruction count
- Dynamic instruction count
- Cycle count / execution time

GCC Optimizations

- **<u>Constant Folding</u>**
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Constant Folding

Constant Folding pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

What is the consequence of this for you as a programmer?
What should you do differently or the same knowing that compilers can do this for you?

Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

Constant Folding: Before (-O0)

```
0000000000400626 <fold>:
  400626:        55                          push   %rbp
  400627:        53                          push   %rbx
  400628:        48 83 ec 08                 sub    $0x8,%rsp
  40062c:        89 fd                       mov    %edi,%ebp
  40062e:        f2 0f 10 05 da 00 00        movsd  0xda(%rip),%xmm0
  400635:        00
  400636:        e8 d5 fe ff ff              callq  400510 <sqrt@plt>
  40063b:        f2 0f 2c c8                 cvttsd2si %xmm0,%ecx
  40063f:        69 ed 07 01 00 00           imul   $0x107,%ebp,%ebp
  400645:        b8 15 00 00 00              mov    $0x15,%eax
  40064a:        99                          cltd
  40064b:        f7 f9                       idiv   %ecx
  40064d:        8d 98 07 01 00 00           lea    0x107(%rax),%ebx
  400653:        bf 04 07 40 00              mov    $0x400704,%edi
  400658:        e8 93 fe ff ff              callq  4004f0 <strlen@plt>
  40065d:        48 69 c0 23 05 00 00        imul   $0x523,%rax,%rax
  400664:        48 63 db                    movslq %ebx,%rbx
  400667:        48 8d 44 18 c9              lea    -0x37(%rax,%rbx,1),%rax
  40066c:        48 c1 e8 02                 shr    $0x2,%rax
  400670:        01 e8                       add    %ebp,%eax
  400672:        48 83 c4 08                 add    $0x8,%rsp
  400676:        5b                          pop    %rbx
  400677:        5d                          pop    %rbp
  400678:        c3                          retq
```

Constant Folding: After (-O2)

```
00000000004004f0 <fold>:
  4004f0:        69 c7 07 01 00 00         imul   $0x107,%edi,%eax
  4004f6:        05 a5 06 00 00            add    $0x6a5,%eax
  4004fb:        c3                        retq
  4004fc:        0f 1f 40 00               nopl   0x0(%rax)
```

GCC Optimizations

- Constant Folding
- **<u>Common Sub-expression Elimination</u>**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

This optimization is done even at –O0!

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

```
00000000004004f0 <subexp>:
  4004f0:  81 c6 07 01 00 00     add    $0x107,%esi
  4004f6:  0f af fe              imul   %esi,%edi
  4004f9:  8d 04 77              lea    (%rdi,%rsi,2),%eax
  4004fc:  0f af c6              imul   %esi,%eax
  4004ff:  c3                    retq
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **<u>Dead Code</u>**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Dead Code

Dead code elimination removes code that doesn't serve a purpose:

```
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```

Dead Code: Before (-O0)

```
00000000004004d6 <dead_code>:
  4004d6:    b8 00 00 00 00          mov    $0x0,%eax
  4004db:    eb 03                   jmp    4004e0 <dead_code+0xa>
  4004dd:    83 c0 01                add    $0x1,%eax
  4004e0:    3d e7 03 00 00          cmp    $0x3e7,%eax
  4004e5:    7e f6                   jle    4004dd <dead_code+0x7>
  4004e7:    39 f7                   cmp    %esi,%edi
  4004e9:    75 05                   jne    4004f0 <dead_code+0x1a>
  4004eb:    8d 47 01                lea    0x1(%rdi),%eax
  4004ee:    eb 03                   jmp    4004f3 <dead_code+0x1d>
  4004f0:    8d 47 01                lea    0x1(%rdi),%eax
  4004f3:    f3 c3                   repz retq
```

Dead Code: After (-O2)

```
0000000004004f0 <dead_code>:
  4004f0: 8d 47 01                lea    0x1(%rdi),%eax
  4004f3: c3                      retq
  4004f4: 66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4004fb: 00 00 00
  4004fe: 66 90                   xchg   %ax,%ax
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- **Strength Reduction**
- Code Motion
- Tail Recursion
- Loop Unrolling

Strength Reduction

Strength reduction changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```
int a = param2 * 32;
int b = a * 7;
int c = b / 3;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}
return c + d;
```

Strength Reduction: After (-O3)

```
unsigned udiv19(unsigned arg) {
return arg / 19;
}
```

```
udiv19(unsigned int):
    mov   eax, edi
    mov   edx, 2938661835
    imul   rax, rdx
    shr   rax, 32
    sub   edi, eax
    shr   edi
    add   eax, edi
    shr   eax, 4
    ret
```

https://godbolt.org/z/Wq8ra3

What really happens here?

$$a \cdot \frac{1}{19} \approx \frac{a \cdot \frac{2938661835}{2^{32}} + \frac{a - a \cdot \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \cdot \frac{1}{19} \approx \left(a \cdot 2938661835 \cdot 2^{-32} + \left(a - a \cdot 2938661835 \cdot 2^{-32}\right) \cdot 2^{-1}\right) \cdot 2^{-4}$$

$$a \cdot \frac{1}{19} \approx a \cdot \frac{7233629131}{137438953472}$$

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **<u>Code Motion</u>**
- Tail Recursion
- Loop Unrolling

Code Motion

Code motion moves code outside of a loop if possible.

```
for (int i = 0; i < n; i++) {
  sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration.

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- **Tail Recursion**
- Loop Unrolling

Tail Recursion

Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```
long factorial(int n) {
  if (n <= 1) {
      return 1;
  }
  else return n * factorial(n - 1);
}
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- **Loop Unrolling**

Loop Unrolling

Loop Unrolling: Do **n** loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i <= n - 4; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
} // after the loop handle any leftovers
```

Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? **`strlen` called for every character**
What can GCC do? **code motion – pull `strlen` out of loop**

Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
void lower1(char *s) {
    for (size_t i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
```

What is the bottleneck?
What can GCC do?

`strlen` called for every character

nothing! `s` is changing, so GCC doesn't know if length is constant across iterations. But <u>we</u> know its length doesn't change.

Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from *really fast* (registers) to *really slow* (disk).
- As data is more frequently used, it ends up in faster and faster memory.

Caching

All caching depends on locality.

Temporal locality

- Repeat access to the same data tends to be co-located in TIME
- Intuitively: things I have used recently, I am likely to use again soon

Spatial locality

- Related data tends to be co-located in SPACE
- Intuitively: data that is near a used item is more likely to also be accessed

Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using `callgrind`
    - Optimize using `-O2`
    - And more...

Compiler Optimizations

Why not always just compile with **–O2***?*

- Difficult to debug optimized executables – only optimize when complete
- Optimizations may not *always* improve your program. The compiler does its best, but may not work, or slow things down, etc. Experiment to see what works best!

Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!

Recap

- Debugging
- Design
- Optimization

Next time: Linking

The Alchemist by Mattheus von Helmont

COMP201

Computer Systems & Programming

Lecture #24 – Debugging and Design

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- The unofficial end-term course feedback form is available: https://forms.gle/aUbJBK63NnBeJ8o58
- The official course feedback form is also available (will be active from Dec 20 to Jan 7)

Recap

- Writing cache-friendly code
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
- Optimization
  - What is optimization?
  - GCC Optimization
  - Limitations of GCC Optimization
  - Caching revisited

Recap: Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

Recap: Matrix Multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ij} \times b_{kj}$$

Recap: Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Recap: Core i7 Matrix Multiply Performance

Recap: Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

Block size B x B

Recap: Naïve vs. Blocked Matrix Multiplication

Naïve Multiplication

Blocked Multiplication

≈ 1,020,000 cache misses

≈ 90,000 cache misses

Video credit: Vasily Volkov

Recap: Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using `callgrind`
    - Optimize using `-O2`
    - And more...

Recap: GCC Optimizations

Optimizations may target one or more of:

- Static instruction count
- Dynamic instruction count
- Cycle count / execution time

Recap: GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force

Plan for Today

- Design
- Optimization

Disclaimer: Slides for this lecture were borrowed from

—Michael Hilton and Brian Railing's CMU 15-213 class

Learning Goals

- Describe the steps to debug complex code failures
- Identify ways to manage the complexity when programming
- State guidelines for communicating the intention of the code

Lecture Plan

- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- Design

https://www.reddit.com/r/ProgrammerHumor/comments/1r0cw7/the_5_stages_of_debugging/

Defects and Infections

1. The programmer creates a defect
2. The defect causes an infection
3. The infection propagates
4. The infection causes a failure

Curse of Debugging

- Not every defect causes a failure!

- "Testing can only show the presence of errors – not their absence."

  – Edsger W. Dijkstra, 1972

Defects to Failures

- Code with defects will introduce erroneous or “infected” state
  - Correct code may propagate this state
  - Eventually an erroneous state is observed
- Some executions will not trigger the defect
  - Others will not propagate “infected” state
- Debugging sifts through the code to find the defect

Explicit Debugging

- Stating the problem
  - Describe the problem aloud or in writing
    - A.k.a. “Rubber duck” or “teddy bear” method
  - Often a comprehensive problem description is sufficient to solve the failure

Explicit Debugging

- Stating the p
  - Describe th
    - A.k.a. "Ru
  - Often a co the failure

Rubber Duck Debugging
Debugging software with a rubber ducky

About Talk to a Duck

Rubber Duck Debugging

The rubber duck debugging method is as follows:

1. Beg, borrow, steal, buy, fabricate or otherwise obtain a rubber duck (bathtub variety).
2. Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.
3. Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.
4. At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

Note: In a pinch a coworker might be able to substitute for the duck, however, it is often preferred to confide mistakes to the duck instead of your coworker.

Original Credit: ~Andy from lists.ethernal.org

Scientific Debugging

- Before debugging, you need to construct a hypothesis as to the defect
  - Propose a possible defect and why it explains the failure conditions
- Ockham's Razor – given several hypotheses, pick the simplest / closest to current work

Scientific Debugging

- Make predictions based on your hypothesis
  - What do you expect to happen under new conditions
  - What data could confirm or refute your hypothesis
- How can I collect that data?
  - What experiments?
  - What collection mechanism?
- Does the data refute the hypothesis?
  - Refine the hypothesis based on the new inputs

Scientific Debugging

- A set of experiments has confirmed the hypothesis
  - This is the diagnosis of the defect
- Develop a fix for the defect
- Run experiments to confirm the fix
  - Otherwise, how do you know that it is fixed?

Code with a Bug

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
            i, fib(i));
```

```
$ gcc -o fib fib.c
$ ./fib
fib(9)=55
fib(8)=34
...
fib(2)=2 fib(1)=134513905
```

A defect has caused a failure.

Constructing a Hypothesis

- Specification defined the first Fibonacci number as 1
  - We have observed working runs (e.g., `fib(2)`)
  - We have observed a failing run
  - We then read the code

- `fib(1)` failed // Hypothesis

Code	Hypothesis
`for (i = 9; …)`	Result depends on order of calls
`while (n > 1) {`	Loop check is incorrect
`int f;`	`f` is uninitialized

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
                 i, fib(i));
```

Prediction

- Propose a new condition or conditions
  - What will logically happen if your hypothesis is correct?
  - What data can be

- `fib(1)` failed // Hypothesis
  - // Result depends on order of calls
    - If `fib(1)` is called first, it will return correctly.
  - // Loop check is incorrect
    - Change to `n >= 1` and run again.
  - // `f` is uninitialized
    - Change to int `f = 1;`

Experiment

- Identical to the conditions of a prior run
  - Except with one condition changed
- Conditions
  - Program input, using a debugger, altering the code

- `fib(1)` failed // Hypothesis
  - If `fib(1)` is called first, it will return correctly.
    - Fails.
  - Change to `n >= 1`
    - `fib(1)=2`
    - `fib(0)=...`
  - Change to `int f = 1;`
    - Works. Sometimes a prediction can be a fix.

Observation

- What is the observed result?
  - Factual observation, such as “Calling `fib(1)` will return 1.”
  - The conclusion will interpret the observation(s)

- Don’t interfere.
  - `printf()` can interfere
  - Like quantum physics, sometimes observations are part of the experiment

- Proceed systematically.
  - Update the conditions incrementally so each observation relates to a specific change

Debugging Tools

Observing program state can require a variety of tools

- Debugger (e.g., `gdb`)
  - What state is in local / global variables (if known)
  - What path through the program was taken

- `valgrind`
  - Does execution depend on uninitialized variables
  - Are memory accesses ever out-of-bounds

Diagnosis

- A scientific hypothesis that explains current observations and makes future predictions becomes a theory
  - We'll call this a diagnosis

- Use the diagnosis to develop a fix for the defect
  - Avoid post hoc, ergo propter hoc fallacy
  - Or correlation does not imply causation

- Understand why the defect and fix relate

Fix and Confirm

- Confirm that the fix resolves the failure

- If you fix multiple perceived defects, which fix was for the failure?
  - Be systematic

David Amador
@DJ_Link

when a "simple" bug is found and we start looking at the code

12:22 PM · Jul 3, 2017

2,181 Retweets **3,400** Likes

Hal replacing a light bulb - Malcolm in the Middle, S03E06

Learn

- Common failures and insights
  - Why did the code fail?
  - What are my common defects?

- Assertions and invariants
  - Add checks for expected behavior
  - Extend checks to detect the fixed failure

- Testing
  - Every successful set of conditions is added to the test suite

Quick and Dirty

- Not every problem needs scientific debugging
  - Set a time limit: (for example)
    - 0 – 10 minutes – Informal Debugging
    - 10 – 60 minutes – Scientific Debugging
    - > 60 minutes – Take a break / Ask for help

Code Smells

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Lecture Plan

- Debugging
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments

Design

- A good design needs to achieve many things:
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

Design

Good Design does:

Complexity Management

&

Communication

Complexity

- There are well known limits to how much complexity a human can manage easily.

VOL. 63, NO. 2 MARCH, 1956

THE PSYCHOLOGICAL REVIEW

THE MAGICAL NUMBER SEVEN, PLUS OR MINUS TWO: SOME LIMITS ON OUR CAPACITY FOR PROCESSING INFORMATION [1]

GEORGE A. MILLER

Harvard University

Complexity Management

- However, patterns can be very helpful…

COGNITIVE PSYCHOLOGY 4, 55–81 (1973)

Perception in Chess[1]

WILLIAM G. CHASE AND HERBERT A. SIMON
Carnegie–Mellon University

This paper develops a technique for isolating and studying the perceptual structures that chess players perceive. Three chess players of varying strength — from master to novice — were confronted with two tasks: (1) A perception task, where the player reproduces a chess position in plain view, and (2) de Groot's (1965) short-term recall task, where the player reproduces a chess position after viewing it for 5 sec. The successive glances at the position in the perceptual task and long pauses in the memory task were used to segment the structures in the reconstruction protocol. The size and nature of these structures were then analyzed as a function of chess skill.

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- DRY
- Abstraction
- Information Hiding
- …

Managing Complexity

- Given the many ways to manage complexity
  - Design code to be testable
  - Try to reuse testable chunks

Complexity Example

- Split a cache access into three+ testable components
  - State all of the steps that a cache access requires
  - Which steps depend on the operation being a load or a store?

Complexity Example

- Split a cache access into three+ testable components
  - State all of the steps that a cache access requires

    Convert address into tag, set index, block offset

    Look up the set using the set index

    Check if the tag matches any line in the set

    If so, hit

    If not a match, miss, then

    Find the LRU block

    Evict the LRU block

    Read in the new line from memory

    Update LRU

    Update dirty if the access was a store

  - Which steps depend on the operation being a load or a store?

Designs need to be testable

- Testable design
  - Testing versus Contracts
  - These are complementary techniques

- Testing and Contracts are
  - Acts of design more than verification
  - Acts of documentation

Testing Example

- For your cache simulator, you can write your own traces
  - Write a trace to test for a cache hit

    L 50, 1
    L 50, 1

  - Write a trace to test dirty bytes in cache

    S 100, 1

Communication

When writing code, the author is communicating with:

- The machine
- Other developers of the system
- Code reviewers
- Their future self

Communication

There are many techniques that have been developed around code communication:

- Tests
- Naming
- Comments
- Commit Messages
- Code Review
- Design Patterns
- …

Naming

Avoid deliberately meaningless names:

Naming is understanding

“If you don’t know what a thing should be called, you cannot know what it is.

If you don’t know what it is, you cannot sit down and write the code.” - Sam Gardiner

Better naming practices

1. Start with meaning and intention
2. Use words with precise meanings (avoid “data”, “info”, “perform”)
3. Prefer fewer words in names
4. Avoid abbreviations in names
5. Use code review to improve names
6. Read the code out loud to check that it sounds okay
7. Actually rename things

Naming guidelines – Use dictionary words

- Only use dictionary words and abbreviations that appear in a dictionary.
  - For example: FileCpy -> FileCopy
  - Avoid vague abbreviations such as acc, mod, auth, etc..

Avoid using single-letter names

- Single letters are unsearchable
  - Give no hints as to the variable's usage

- Exceptions are loop counters
  - Especially if you know why `i`, `j`, etc were originally used

Limit name character length

“Good naming limits individual name length, and reduces the need for specialized vocabulary” – Philip Relf

Limit name word count

- Keep names to a four words maximum
- Limit names to the number of words that people can read at a glance.

- Which of each pair do you prefer?

```
a1) arraysOfSetsOfLinesOfBlocks
a2) cache

b1) evictedData
b2) evictedDataBytes
```

Describe Meaning

- Use descriptive names.
- Avoid names with no meaning: `a`, `foo`, `blah`, `tmp`, etc

- There are reasonable exceptions:

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Use a large vocabulary

- Be more specific when possible:
  - `Person -> Employee`

- What is size in this `binaryTree`?

```
struct binaryTree {
  int size;
  …
};
```

Use problem domain terms

- Use the correct term in the problem domain's language.
  - Hint: as a student, consider the terms in the assignment

- In Assignment 5, consider the following:

```
username

secret_pin
```

Use opposites precisely

- Consistently use opposites in standard pairs
  - `first/end` -> `first/last`

Comments

Don’t Comments

- Don’t say what the code does
  - because the code already says that
- Don’t explain awkward logic
  - improve the code to make it clear
- Don’t add too many comments
  - it’s messy, and they get out of date

Awkward Code

Imagine someone (TA, employer, etc) has to read your code

- Would you rather rewrite or comment the following?

```
(*(void **)((*(void **)(bp)) + DSIZE)) = (*(void **)(bp + DSIZE));
```

- How about?

```
bp->prev->next = bp->next;
```

- Both lines update program state in the same way.

Do Comments

- Answer the question: why the code exists

- When should I use this code?
- When shouldn’t I use it?
- What are the alternatives to this code?

Why does this exist?

- Explain why a magic number is what it is.

```
// Each address is 64-bit, which is 16 + 1 hex characters
const int MAX_ADDRESS_LENGTH = 17;
```

- When should this code be used? Is there an alternative?

```
unsigned power2(unsigned base, unsigned expo){
    unsigned i;
    unsigned result = 1;
    for(i=0;i<expo;i++){
        result+=result;
    }
    return result;
}
```

How to write good comments

1. Write short comments of what the code will do.
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

```
// Initialize locals
// Pick a pivot value
// Reorder array around the pivot
// Recurse
```

How to write good comments

1. Write short comments of what the code will do.
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

2. Write that code.

3. Revise comments / code
    1. If the code or comments are awkward or complex
    2. Join / Split comments as needed

4. Maintain code and comments

Commit Messages

- Committing code to a source repository is a vital part of development
  - Protects against system failures and typos:
    - `cat foo.c` versus `cat > foo.c`
  - The commit messages are your record of your work
    - Communicating to your future self
    - Describe in one line what you did

  "Parses command line arguments"

  "fix bug in unique tests, race condition not solved"

  "seg list finished, performance is ..."

Debugging and Design

- Programs have defects
  - Be systematic about finding them
- Programs are more complex than humans can manage
  - Write code to be manageable
- Programming is not solitary, even if you are communicating with a grader or a future self
  - Be understandable in your communication

Recap

- Debugging
- Design

Next time: Linking

photo by unsplash user @davidclode

COMP201

Computer Systems & Programming

Lecture #25 – Linking

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Happy New Year !

- No labs this Friday!

Image: Futurama

Recap

- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments

Plan for Today

- Linking
- Case study: Library interpositioning

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class

Lecture Plan

- Linking
- Case study: Library interpositioning

Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Static Linking

Programs are translated and linked using a *compiler driver*:

```
linux> gcc -Og -o prog main.c sum.c
linux> ./prog
```

Why Linkers?

Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.

- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Why Linkers? (cont)

Reason 2: Efficiency

- Time: Separate compilation
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.

- Space: Libraries
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

Step 1: Symbol resolution

- Programs define and reference *symbols* (global variables and functions):
  - `void swap() {…}  /* define symbol swap */`
  - `swap();  /* reference symbol swap */`
  - `int *xp = &x;  /* define symbol xp, reference x */`

- Symbol definitions are stored in object file (by assembler) in *symbol table*.
  - Symbol table is an array of `structs`
  - Each entry includes name, size, and location of symbol.

- During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do? (cont)

Step 2: Relocation

- Merges separate code and data sections into single sections
- Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
- Updates all references to these symbols to reflect their new positions.

Let's look at these two steps in more detail....

Three Kinds of Object Files (Modules)

- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (`.so` file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

Executable and Linkable Format (ELF)

- Standard binary format for object files

- One unified format for
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- Generic name: ELF binaries

ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (`.o`, `exec`, `.so`), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, …
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space

ELF Object File Format (cont.)

- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- `.rel.data` section
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (`gcc -g`)
- Section header table
  - Offsets and sizes of each section

Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-**`static`** C functions and non-**`static`** global variables.

- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and global variables defined with the **`static`** attribute.
  - Local linker symbols are *not* local program variables

Step 1: Symbol Resolution

...that's defined here

Referencing a global...

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Defining a global

Linker knows nothing of `val`

Referencing a global...

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linker knows nothing of `i` or `s`

...that's defined here

Local Symbols

- Local non-static C variables vs. local static C variables
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss`, or `.data`

```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

- Compiler allocates space in `.data` for each definition of `x`
- Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

How Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
  - **Strong**: procedures and initialized globals
  - **Weak**: uninitialized globals

strong → `int foo=5;`

strong → `p1() { }`

p1.c

`int foo;` ← weak

`p2() { }` ← strong

p2.c

Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error

- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol

- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`

Linker Puzzles 1

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized `int`.
Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` will overwrite `y`!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Linker Puzzles 2

slido

- Which definitions do the references `main` or `x` refer to?

```
int main()
{}
```

```
int main;
int p2() {}
```

REF(`main.1`) → DEF(`main.1`)
REF(`main.2`) → DEF(`main.1`)

```
void main()
{}
```

```
int main = 1;
p2() {}
```

Error! Each module defines a strong symbol `main`

```
int x;
void main()
{}
```

```
double x=1.0;
int p2()
{}
```

REF(`x.1`) → DEF(`x.2`)
REF(`x.2`) → DEF(`x.2`)

Global Variables

- Avoid if you can

- Otherwise
  - Use `static` if you can
  - Initialize if you define a global variable
  - Use `extern` if you reference an external global variable

Step 2: Relocation

Relocation Entries

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

objdump -r -d main.o

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi          # %edi = &array
                        a: R_X86_64_32 array              # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13>     # sum()
                        f: R_X86_64_PC32 sum-0x4          # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

main.o

Relocated .text section

objdump -dx prog

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08       sub    $0x8,%rsp
  4004d4:       be 02 00 00 00    mov    $0x2,%esi
  4004d9:       bf 18 10 60 00    mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00    callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08       add    $0x8,%rsp
  4004e7:       c3                retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00          mov    $0x0,%eax
  4004ed:       ba 00 00 00 00          mov    $0x0,%edx
  4004f2:       eb 09                   jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                movslq %edx,%rcx
  4004f7:       03 04 8f                add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                add    $0x1,%edx
  4004fd:       39 f2                   cmp    %esi,%edx
  4004ff:       7c f3                   jl     4004f4 <sum+0xc>
  400501:       f3 c3                   repz retq
```

Using PC-relative addressing for `sum()`: `0x4004e8` = `0x4004e3` + `0x5`

Loading Executable Object Files

Executable Object File

Packaging Commonly Used Functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-fashioned Solution: Static Libraries

- **Static libraries** (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries

```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

```
% ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (`sin`, `cos`, `tan`, `log`, `exp`, `sqrt`, ...)

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”, z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries

"c" for "compile-time"

Using Static Libraries

- **Linker's algorithm for resolving external references:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
  - More on this when you learn about virtual memory

Dynamic Linking at Load-time

Dynamic Linking at Run-time

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c

Dynamic Linking at Run-time

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Linking Summary

- Linking is a technique that allows programs to be constructed from multiple object files.

- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- Understanding linking can help you avoid nasty errors and make you a better programmer.

Lecture Plan

- Linking
- Case study: Library interpositioning

Case Study: Library Interpositioning

- **Library interpositioning:** powerful linking technique that allows programmers to intercept calls to arbitrary functions
- Interpositioning can occur at:
  - Compile time: When the source code is compiled
  - Link time: When the relocatable object files are statically linked to form an executable object file
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

- **Security**
  - Confinement (sandboxing)
  - Behind the scenes encryption
- **Debugging**
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to Posix write functions (`write`, `writev`, `pwrite`)

Source: Facebook engineering blog post at `https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/`

Some Interpositioning Applications

- **Monitoring and Profiling**
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - Generating address traces

Example program

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

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```

int.c

- **Goal:** trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.
- **Three solutions:** interpose on the `lib` `malloc` and `free` functions at compile time, link time, and load/run time.

Compile-time Interpositioning

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n",
           (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```

Link-time Interpositioning

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

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Link-time Interpositioning

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```

- The "`-Wl`" flag passes argument to linker, replacing each comma with a space.
- The "`--wrap,malloc`" arg instructs linker to resolve references in a special way:
  - Refs to `malloc` should be resolved as `__wrap_malloc`
  - Refs to `__real_malloc` should be resolved as `malloc`

Load/Run-time Interpositioning

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Load/Run-time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.

Interpositioning Recap

- **Compile Time**
  - Apparent calls to `malloc/free` get macro-expanded into calls to `mymalloc/myfree`
- **Link Time**
  - Use linker trick to have special name resolutions
    - `malloc` → `__wrap_malloc`
    - `__real_malloc` → `malloc`
- **Load/Run Time**
  - Implement custom version of `malloc/free` that use dynamic linking to load library `malloc/free` under different names

Recap

- Linking
- Case study: Library interpositioning

- **Next time:** Managing the heap

photo by pixabay user @ulleo

COMP201

Computer Systems & Programming

Lecture #27 – Managing The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- Static Linking
- Symbol Resolution
- Relocation
- Static Libraries
- Shared Libraries

Plan for Today

- The Heap So Far
- What Is A Heap Allocator?
- Heap Allocator Requirements and Goals
- Bump Allocator
- Implicit Free List Allocator

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

—Ruth Anderson's UW CSE 351 class

Multiple Ways to Store Program Data

- Static global data
  - *Fixed size* at compile-time
  - Entire *lifetime of the program* (loaded from executable)
  - Portion is read-only (*e.g.* string literals)
- Stack-allocated data
  - Local/temporary variables
    - *Can* be dynamically sized (in some versions of C)
  - *Known lifetime* (deallocated on `return`)
- **Dynamic (heap) data**
  - Size known only at runtime (*i.e.* based on user-input)
  - Lifetime known only at runtime (long-lived data structures)

```
int array[1024];

int* foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n*sizeof(int));
  return dyn;
}
```

COMP201 Topic 8: How do the core `malloc`/`realloc`/`free` memory-allocation operations work?

How do `malloc/realloc/free` work?

Pulling together all our COMP201 topics this semester:

- Testing
- Efficiency
- Bit-level manipulation
- Memory management
- Pointers
- Generics
- Assembly
- And more...

Learning Goals

- Learn the restrictions, goals and assumptions of a heap allocator
- Understand the conflicting goals of utilization and throughput
- Learn about different ways to implement a heap allocator

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Bump Allocator
- Implicit Free List Allocator

Running a program

- **Creates new process**
- **Sets up address space/segments**
- **Read executable file, load instructions, global data**
  Mapped from file into gray segments
- **Libraries loaded on demand**

- **Set up stack**
  Reserve stack segment, init `%rsp`, call main
- **`malloc` written in C, will init self on use**
  Asks OS for large memory region, parcels out to service requests

The Stack Revisited

Stack memory "goes away" after function call ends.

Automatically managed at compile-time by gcc

Today: The Heap

Heap memory persists until caller indicates it no longer needs it.

Managed by C standard library functions (`malloc`, `realloc`, `free`)

This lecture:
How does heap management work?

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Bump Allocator
- Implicit Free List Allocator

Revisited: Allocating Memory in C

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Good practices:
  - `ptr = (int*) malloc(n*sizeof(int));`
    - `sizeof` makes code more portable
    - `void*` is implicitly cast into any pointer type; explicit typecast will help you catch coding errors when pointer types don't match

Revisited: Allocating Memory in C

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; `NULL` indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Related functions:
  - `void* calloc(size_t nitems, size_t size)`
    - "Zeros out" allocated block
  - `void* realloc(void* ptr, size_t size)`
    - Changes the size of a previously allocated block (if possible)
  - `void* sbrk(intptr_t increment)`
    - Used internally by allocators to grow or shrink the heap

Revisited: Freeing Memory in C

- Need to `#include <stdlib.h>`
- `void free(void* p)`
  - Releases whole block pointed to by `p` to the pool of available memory
  - Pointer `p` must be the address *originally* returned by `m/c/realloc` (*i.e.* beginning of the block), otherwise system exception raised
  - Don't call `free` on a block that has already been released or on `NULL`

Memory Allocation Example in C

```c
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int));          /* allocate block of n ints */
  if (p == NULL) {                           /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)                        /* initialize int array */
    p[i] = i;
                                             /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                           /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)                    /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)                      /* print new array */
    printf("%d\n", p[i]);
  free(p);                                   /* free p */
}
```

Your role so far: Client

```
void *malloc(size_t size);
```

Returns a pointer to a block of heap memory of at least size bytes, or NULL if an error occurred.

```
void free(void *ptr);
```

Frees the heap-allocated block starting at the specified address.

```
void *realloc(void *ptr, size_t size);
```

Changes the size of the heap-allocated block starting at the specified address to be the new specified size. Returns the address of the new, larger allocated memory region.

Your role now: Heap Hotel Concierge

(aka **Heap Allocator**)

http://screencrave.com/wp-content/uploads/2014/03/the-grand-budapest-hotel-anderson-image-2.jpg

Types of heap allocators

- **Explicit allocator:** programmer allocates and frees space
  - Example: `malloc` and `free` in C

- **Implicit allocator:** programmer only allocates space (no free)
  - Example: garbage collection in Java, Caml, and Lisp

Dynamic memory allocation

- Allocator organizes heap as a collection of variable-sized blocks, which are either allocated or free
  - Allocator requests pages in the heap region; virtual memory hardware and OS kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
    - (Larger objects handled too; ignored here)

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 1: Hi! May I please have 2 bytes of heap memory?

Allocator: Sure, I've given you address 0x10.

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 1: Hi! May I please have 2 bytes of heap memory?

Allocator: Sure, I've given you address 0x10.

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 2: Howdy! May I please have 3 bytes of heap memory?

Allocator: Sure, I've given you address 0x12.

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 2: Howdy! May I please have 3 bytes of heap memory?

Allocator: Sure, I've given you address 0x12.

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need some of it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need parts of it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 3		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need of it.

Request 3: Hi again! I'd like to request the region of memory at 0x10 be reallocated to 4 bytes.

Allocator: Sure thing. I've given you address 0x15.

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need of it.

Request 3: Hi again! I'd like to request the region of memory at 0x10 be reallocated to 4 bytes.

Allocator: Sure thing. I've given you address 0x15.

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Bump Allocator
- Implicit Free List Allocator

Heap Allocator Functions

```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

All these heap allocators functions should work in harmony, and of course in an efficient manner! More on this later...

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

Heap Allocator Requirements

A heap allocator must...

1. **Handle arbitrary request sequences of allocations and frees**
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

A heap allocator cannot assume anything about the order of allocation and free requests, or even that every allocation request is accompanied by a matching free request.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. **Keep track of which memory is allocated and which is available**
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

A heap allocator marks memory regions as **allocated** or **available**. It must remember which is which to properly provide memory to clients.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. **Decide which memory to provide to fulfill an allocation request**
4. Immediately respond to requests without delay

A heap allocator may have options for which memory to use to fulfill an allocation request. It must decide this based on a variety of factors.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. **Immediately respond to requests without delay**

A heap allocator must respond immediately to allocation requests and should not e.g. prioritize or reorder certain requests to improve performance.

Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay
5. **Return addresses that are 8-byte-aligned (must be multiples of 8).**

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Utilization

- The primary cause of poor utilization is **fragmentation**. **Fragmentation** occurs when otherwise unused memory is not available to satisfy allocation requests.
- In this example, there is enough aggregate free memory to satisfy the request, but no single free block is large enough to handle the request.
- **In general:** we want the largest address used to be as low as possible.

Request 6: Hi! May I please have 4 bytes of heap memory?

Allocator: I'm sorry, I don't have a 4 byte block available...

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	*Free*	Req. 2	*Free*	Req. 3	*Free*	Req. 4	*Free*	Req. 5	*Free*

Utilization

Question: what if we shifted these blocks down to make more space? Can we do this?

A. YES, great idea!

B. YES, it can be done, but not a good idea for some reason (e.g. not efficient use of time)

C. NO, it can't be done!

Utilization

Question: what if we shifted these blocks down to make more space? Can we do this?

- **No -** we have already guaranteed these addresses to the client. We cannot move allocated memory around, since this will mean the client will now have incorrect pointers to their memory!

Fragmentation

- Poor memory utilization is caused by **fragmentation**
  - Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
  - Two types: **internal** and **external**

- **Recall:** Fragmentation in structs
  - Internal fragmentation was wasted space inside of the struct (between fields) due to alignment
  - External fragmentation was wasted space between struct instances (*e.g.* in an array) due to alignment

- Now referring to wasted space in the heap inside or between allocated blocks

Internal Fragmentation

- For a given block, **internal fragmentation** occurs if payload is smaller than the block

- Causes:
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)
  - Explicit policy decisions (*e.g.* return a big block to satisfy a small request)
- Easy to measure because only depends on past requests

External Fragmentation

= 8-byte word

- For the heap, **external fragmentation** occurs when allocation/free pattern leaves "holes" between blocks
  - That is, the aggregate payload is non-continuous
  - Can cause situations where there is enough aggregate heap memory to satisfy request, but no single free block is large enough

- Don't know what future requests will be
  - Difficult to impossible to know if past placements will become problematic

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

These are seemingly conflicting goals – for instance, it may take longer to better plan out heap memory use for each request.

Heap allocators must find an appropriate balance between these two goals!

Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Other desirable goals:
Locality ("similar" blocks allocated close in space)
Robust (handle client errors)
Ease of implementation/maintenance

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Bump Allocator
- Implicit Free List Allocator

Bump Allocator

Let's say we want to entirely prioritize throughput, and do not care about utilization at all. This means we do not care about reusing memory. How could we do this?

1. Utilization

Never reuses memory

2. Throughput

Ultra fast, short routines

Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request and does nothing on a free request.
- Throughput: each `malloc` and `free` execute only a handful of instructions:
  - It is easy to find the next location to use
  - Free does nothing!
- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18
c	0x20

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18
c	0x20

Bump Allocator

```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

Variable	Value
a	0x10
b	0x18
c	0x20
d	NULL

Summary: Bump Allocator

- A bump allocator is an extreme heap allocator – it optimizes only for **throughput**, not **utilization**.
- Better allocators strike a more reasonable balance. How can we do this?

Questions to consider:

1. How do we keep track of free blocks?
2. How do we choose an appropriate free block in which to place a newly allocated block?
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
4. What do we do with a block that has just been freed?

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert a freed block into the heap?

Lecture Plan

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Bump Allocator
- Implicit Free List Allocator

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.
- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free.
- The header should be 8 bytes (or larger).
- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated o
- When we allocate a block, we look throug
  we update its header to reflect its allocate
- When we free a block, we update its heac
- The header should be 8 bytes (or larger).
- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

This is larger than the 4 byte headers specified in the book, as this makes it easier to satisfy the alignment constraint and store information!.

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28
e	0x48

Representing Headers

e.g. with 8-byte alignment, possible values for size:

```
00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
...
```

- For each block we need: **`size`, is-allocated?**
  - Could store using two words, but wasteful
- Standard trick
  - If blocks are aligned, some low-order bits of `size` are always 0
  - Use lowest bit as an allocated/free flag (fine as long as aligning to K>1)
  - When reading `size`, must remember to mask out this bit!

Format of allocated and free blocks:

`a = 1`: allocated block
`a = 0`: free block

`size`: block size (in bytes)

`payload`: application data (allocated blocks only)

If `x` is first word (header):

```
x = size | a;

a = x & 1;

size = x & ~1;
```

Implicit Free List Allocator

How can we choose a free block to use for an allocation request?

- **First fit:** search the list from beginning each time and choose first free block that fits.
- **Next fit:** instead of starting at the beginning, continue where previous search left off.
- **Best fit:** examine every free block and choose the one with the smallest size that fits.

- First fit/next fit easier to implement
- What are the pros/cons of each approach?

Implicit List: Finding a Free Block

- **First fit**
  - Can take time linear in total number of blocks
  - In practice can cause "splinters" at beginning of list
- **Next fit**
  - Like first-fit, but **search list starting where previous search finished**
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- **Best fit**
  - Search the list, choose the **best** free block: large enough AND with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Usually worse throughput

Practice 1

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?

[24 byte payload, free] [16 byte payload, free] [8 byte payload, allocated for A]

```
void *b = malloc(8);
```

[8 byte payload, allocated for B] [8 byte payload, free] [16 byte payload, free] [8 byte payload, allocated for A]

Practice 2

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **best-fit** approach?

[24 byte payload, free] [8 byte payload, free] [8 byte payload, allocated for A]

```
void *b = malloc(8);
```

[24 byte payload, free] [8 byte payload, allocated for B] [8 byte payload, allocated for A]

Implicit Free List Summary

For **all blocks**,

- Have a header that stores size and status.
- Our list links *all* blocks, allocated (A) and free (F).

Keeping track of free blocks:

- **Improves memory utilization** (vs bump allocator)
- **Decreases throughput** (worst case allocation request has O(A + F) time)
- Increases design complexity ☺

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

A. Throw into allocation for `e` **as extra padding?** *Internal fragmentation – unused bytes because of padding*

Splitting Policy

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

A. Throw into allocation for **e** as extra padding?
B. Make a "zero-byte free block"? *External fragmentation – unused free blocks*

Revisiting Our Goals

Questions we considered:

1. How do we keep track of free blocks? **Using headers!**
2. How do we choose an appropriate free block in which to place a newly allocated block? **Iterate through all blocks.**
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block? **Try to make the most of it!**
4. What do we do with a block that has just been freed? **Update its header!**

Coalescing

```
void *e = malloc(24);   // returns NULL!
```

We do not need to worry about this problem for the implicit allocator, but investigate this for the *explicit* allocator! (More about this later).

In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

Variable	Value
a	0x10

In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

Variable	Value
a	0x10
b	0x28

The implicit allocator can always move memory to a new location for a **`realloc`** request. The *explicit* allocator must support in-place **`realloc`** (more on this later).

Summary: Implicit Allocator

- An implicit allocator is a more efficient implementation that has reasonable **throughput** and **utilization** due to its recycling of blocks.

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse?
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

Recap

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator

Next: Explicit Free List Allocator, Wrapping Up

Photo: Dennis Ritchie and Ken Thompson

COMP201

Computer Systems & Programming

Lecture #28 – More Heap Allocators, Wrapping Up

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator

Recap: Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Other desirable goals:
Locality ("similar" blocks allocated close in space)
Robust (handle client errors)
Ease of implementation/maintenance

Recap: Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request and does nothing on a free request.
- Throughput: each `malloc` and `free` execute only a handful of instructions:
  - It is easy to find the next location to use
  - Free does nothing!
- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹

Recap: Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.
- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free.
- The header should be 8 bytes (or larger).
- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

Plan for Today

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

—Ruth Anderson's UW CSE 351 class

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

This is inefficient – it triples the size of *every* header, when we just need to jump from one free block to another. And even if we just made free headers bigger, it's complicated to have two different header sizes.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure?

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure? *More difficult to access in a separate place – prefer storing near blocks on the heap itself.*

Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>

Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>

Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>
- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free <u>and allocated.</u> (why?)

Explicit Free Lists

Allocated block:

(same as implicit free list)

Free block:

- Use list(s) of free blocks, rather than implicit list of all blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer) for coalescing

Explicit Free List Allocator

- This design builds on the implicit allocator, but also stores pointers to the next and previous free block inside each free block's payload.
- When we allocate a block, we look through just the free blocks using our linked list to find a free one, and we update its header and the linked list to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free and update the linked list.

This **explicit** list of free blocks increases request throughput, with some costs (design and internal fragmentation)

Explicit Free List: List Design

How do you want to organize your explicit free list?
(compare utilization/throughput)

A. Address-order (each block's address is less than successor block's address) — Better memory util, Linear free

B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list — Constant free (push recent block onto stack)

C. Other (e.g., by size, etc.) — (more later)

Explicit Free List: List Design

Note that the doubly-linked list *does not have to be in address order*.

Implicit vs. Explicit: So Far

Implicit Free List

- 8B header for size + alloc/free status
- Allocation requests are worst-case linear in total number of blocks
- Implicitly address-order

Explicit Free List

- 8B header for size + alloc/free status
- Free block payloads store prev/next free block pointers
- Allocation requests are worst-case linear in number of free blocks
- Can choose block ordering

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

We'd like to be able to merge adjacent free blocks back together. How can we do this?

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Hey, look! I have a free neighbor. Let's be friends! ☺

Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

The process of combining adjacent free blocks is called coalescing.

For your explicit heap allocator, you should coalesce if possible when a block is freed. **You only need to coalesce the most immediate right neighbor.**

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on `free()`.**
3. Can we avoid always copying/moving data during `realloc`?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on `free()`.**
3. Can we avoid always copying/moving data during `realloc`?

`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.

`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 48);
```

a's earlier request was too small, so we added padding. Now they are requesting a larger size we can satisfy with that padding! So `realloc` can return the same address.

`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a `realloc` is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a `realloc` is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

Now we can still return the same address.

realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.

realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.

`realloc`: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.

`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.
- If you can't do an in-place `realloc`, then you should move the data elsewhere.

Practice 3

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

[24 byte payload, allocated for B] [16 byte payload, free] [16 byte payload, allocated for A]

```
free(B);
```

[48 byte payload, free] [16 byte payload, allocated for A]

Practice 4

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

[16 byte payload, allocated for A] [32 byte payload, free] [16 byte payload, allocated for B]

```
realloc(A, 24);
```

[24 byte payload, allocated for A] [24 byte payload, free] [16 byte payload, allocated for B]

Going beyond: Explicit list w/size buckets

- Explicit lists are much faster than implicit lists.
- However, a first-fit placement policy is still linear in total # of free blocks.
- What about an explicit free list **sorted by size** (e.g., as a tree)?
- What about several explicit free lists **bucketed by size**? (below)

Read B&O Section 9.9.14!

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

Wouldn’t it be nice…

- If we never had to free memory?
- Do you free objects in Java?
  - Reminder: *implicit* allocator

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Garbage Collection (GC) (Automatic Memory Management)

- **Garbage collection:** automatic reclamation of heap-allocated storage – application never explicitly frees memory

```
void foo() {
   int* p = (int*) malloc(128);
   return;  /* p block is now garbage! */
}
```

- Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…

- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers in registers/stack/globals)

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

Classical GC Algorithms

- **Mark-and-sweep collection** (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

We've covered *a lot* in just 14 weeks! Let's take a look back.

Our COMP201 Journey

Course Overview

1. **Bits and Bytes -** *How can a computer represent integer and float numbers?*
2. **Chars and C-Strings -** *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap –** *How can we effectively manage all types of memory in our programs?*
4. **Generics -** *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. **Assembly -** *How does a computer interpret and execute C programs?*
6. **Cache Memories -** *How does the memory system is organized as a hierarchy of different storage devices with unique capacities?*
7. **Optimization-** *How we can optimize our code to improve efficiency and speed? The optimizations GCC can perform.*
8. **Linking-** *How to construct programs from multiple object files?*
9. **Heap Allocators -** *How do core memory-allocation operations like malloc and free work?*

First Day

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

First Day

- The **command-line** is a text-based interface to navigate a computer, instead of a Graphical User Interface (GUI).

Graphical User Interface

Text-based interface

Bits And Bytes

Key Question: *How can a computer represent integer numbers?*

Bits And Bytes

Why does this matter?

- Limitations of representation and arithmetic impact programs!
- We can also efficiently manipulate data using bits.

https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Floats

- IEEE Floating Point is a carefully-thought-out standard. It's complicated, but engineered for their goals.
- Floats have an extremely wide range, but cannot represent every number in that range.
- Some approximation and rounding may occur! This means you definitely don't want to use floats e.g. for currency.
- Associativity does not hold for numbers far apart in the range
- Equality comparison operations are often unwise.

C Strings

Key Question: *How can a computer represent and manipulate more complex data like text?*

- Strings in C are arrays of characters ending with a null terminator!
- We can manipulate them using pointers and C library functions (many of which you could probably implement).

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

C Strings

Why does this matter?

- Understanding this representation is key to efficient string manipulation.
- This is how strings are represented in both low- and high-level languages!
  - C++: https://www.quora.com/How-does-C++-implement-a-string
  - Python: https://www.laurentluce.com/posts/python-string-objects-implementation/

Pointers, Stack and Heap

Key Question: *How can we effectively manage all types of memory in our programs?*

- Arrays let us store ordered lists of information.
- Pointers let us pass addresses of data instead of the data itself.
- We can use the stack, which cleans up memory for us, or the heap, which we must manually manage.

Stack And Heap

Why does this matter?

- The stack and heap allow for two ways to store data in our programs, each with their own tradeoffs, and it's crucial to understand the nuances of managing memory in any program you write!
- Pointers let us pass around references to data efficiency

Generics

Key Question: *How can we use our knowledge of memory and data representation to write code that works with any data type?*

- We can use `void *` to circumvent the type system, `memcpy`, etc. to copy generic data, and function pointers to pass logic around.

Why does this matter?

- Working with any data type lets us write more generic, reusable code.
- Using generics helps us better understand the type system in C and other languages, and where it can help and hinder our program.

Assembly Language

Key Question: *How does a computer interpret and execute C programs?*

- GCC compiles our code into *machine code instructions* executable by our processor.
- Our processor uses registers and instructions like **mov** to manipulate data.

Assembly Language

Why does this matter?

- We write C code because it is higher level and transferrable across machines. But it is not the representation executed by the computer!
- Understanding how programs are compiled and executed, as well as computer architecture, is key to writing performant programs (e.g. fewer lines of code is not necessarily better).
- We can reverse engineer and exploit programs at the assembly level!

Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from *really fast* (registers) to *really slow* (disk).
- As data is more frequently used, it ends up in faster and faster memory.

Caching

Why does this matter?

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Temporal locality**
  - Repeat access to the same data tends to be co-located in TIME
  - Intuitively: things I have used recently, I am likely to use again soon
- **Spatial locality**
  - Related data tends to be co-located in SPACE
  - Intuitively: data that is near a used item is more likely to also be accessed

Linking

- Linking is a technique that allows programs to be constructed from multiple object files.
- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

Why does this matter?

- Understanding linking can help you avoid nasty errors and make you a better programmer.

Heap Allocators

Key Question: *How do core memory-allocation operations like malloc and free work?*

- A *heap allocator* manages a block of memory for the heap and completes requests to use or give up memory space.
- We can manage the data in a heap allocator using headers, pointers to free blocks, or other designs

Why does this matter?

- Designing a heap allocator requires making many design decisions to optimize it as much as possible. There is no perfect design!
- All languages have a “heap” but manipulate it in different ways.

COMP201 Learning Goals

The goals for COMP201 are for students
to gain **mastery** of

- writing C programs with complex use of memory and pointers
- an accurate model of the address space and compile/runtime behavior of C programs

to achieve **competence** in

- translating C to/from assembly
- writing programs that respect the limitations of computer arithmetic
- finding bottlenecks and improving runtime performance
- working effectively in a Unix development environment

and have **exposure** to

- a working understanding of the basics of cache memories

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Tools and Techniques

- Unix and the command line
- Coding Style
- Debugging (GDB)
- Testing (Sanity Check)
- Memory Checking (Valgrind)

Unix And The Command Line

Unix and command line tools are extremely popular tools outside of COMP201 for:

- Running programs (web servers, python programs, remote programs...)
- Accessing remote servers (Amazon Web Services, Microsoft Azure, Heroku...)
- Programming embedded devices (Raspberry Pi, etc.)

<u>Our goal for COMP201 was to help you become proficient in navigating Unix</u>

Coding Style

- Writing clean, readable code is crucial for any computer science project
- Unfortunately, a fair amount of existing code is poorly-designed/documented

<u>Our goal for COMP201 was to help you write with good coding style, and read/understand/comment provided code.</u>

Debugging (GDB)

- Debugging is a crucial skill for any computer scientist
- <u>Our goal for COMP201 was to help you become a better debugger</u>
  - narrow in on bugs
  - diagnose the issue
  - implement a fix
- Practically every project you work on will have debugging facilities
  - Python: “PDB”
  - IDEs: built-in debuggers (e.g. QT Creator, Eclipse)
  - Web development: in-browser debugger

Testing (Sanity Check)

- Testing is a crucial skill for any computer scientist
- <u>Our goal for COMP201 was to help you become a better tester</u>
  - Writing targeted tests to validate correctness
  - Use tests to prevent regressions
  - Use tests to develop incrementally

Memory Checking and Profiling

- Memory checking and profiling are crucial for any computer scientist to analyze program performance and increase efficiency.
- Many projects you work on will have profiling and memory analysis facilities:
  - Mobile development: integrated tools (XCode Instruments, Android Profiler, etc.)
  - Web development: in-browser tools

Tools

You'll see manifestations of these tools throughout projects you work on. We hope you can use your COMP201 knowledge to take advantage of them!

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- Larger Applications
- What's Next?
- Q&A

After COMP201, you are prepared to take a variety of classes in various areas. What are some options?

Role within COMP Curriculum

Courses

How is an operating system implemented? (take COMP304!)

- Threads
- User Programs
- Virtual Memory
- Filesystem

How is a programming language designed? (take COMP301!)

- Lexical analysis
- Parsing
- Semantic Analysis
- Code Generation

Courses

What are the principles of designing computer hardware (take COMP303)

- Computer organization

How can we write programs that execute on special hardware? (take COMP317!)

- Embedded systems

How can applications communicate over a network? (take COMP416!)

- How can we weigh different tradeoffs of network architecture design?
- How can we effectively transmit bits across a network?

Machine Learning

Can we speed up machine learning training with reduced precision computation?

- https://www.top500.org/news/ibm-takes-aim-at-reduced-precision-for-new-generation-of-ai-chips/
- https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/

How can we implement performant machine learning libraries?

- Popular tools such as TensorFlow and PyTorch are implemented using C!
- https://pytorch.org/blog/a-tour-of-pytorch-internals-1/
- https://www.tensorflow.org/guide/extend/architecture

Web Development

How can we efficiently translate Javascript code to machine code?

- The Chrome V8 JavaScript engine converts Javascript into machine code for computers to execute: https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964
- The popular Node.js web server tool is built on Chrome V8

How can we compile programs into an efficient binary instruction format that runs in a web browser?

- WebAssembly is an emerging standard instruction format that runs in browsers: https://webassembly.org
- You can compile C/C++/other languages into WebAssembly for web execution

Programming Languages / Runtimes

How can programming languages and runtimes efficiently manage memory?

- Manual memory management (C/C++)
- Reference Counting (Swift)
- Garbage Collection (Java)

How can we design programming languages to reduce the potential for programmer error?

- Haskell/Swift "Optionals"

How can we design portable programming languages?

- Java Bytecode: https://en.wikipedia.org/wiki/Java_bytecode

Theory

How can compilers output efficient machine code instructions for programs?

- Languages can be represented as regular expressions and context-free grammars
- We can model programs as control-flow graphs for additional optimization

Security

How can we find / fix vulnerabilities at various levels in our programs?

- Understand machine-level representation and data manipulation
- Understand how a computer executes programs
- macOS High Sierra Root Login Bug: https://objective-see.com/blog/blog_0x24.html

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- Larger Applications
- What's Next?
- Q&A

Course Evaluations

- I hope you can take the time to fill out the end-semester COMP201 course evaluation.
- I sincerely appreciate any feedback you have about the course and read every piece of feedback we receive.
- I am always looking for ways to improve!

Q&A