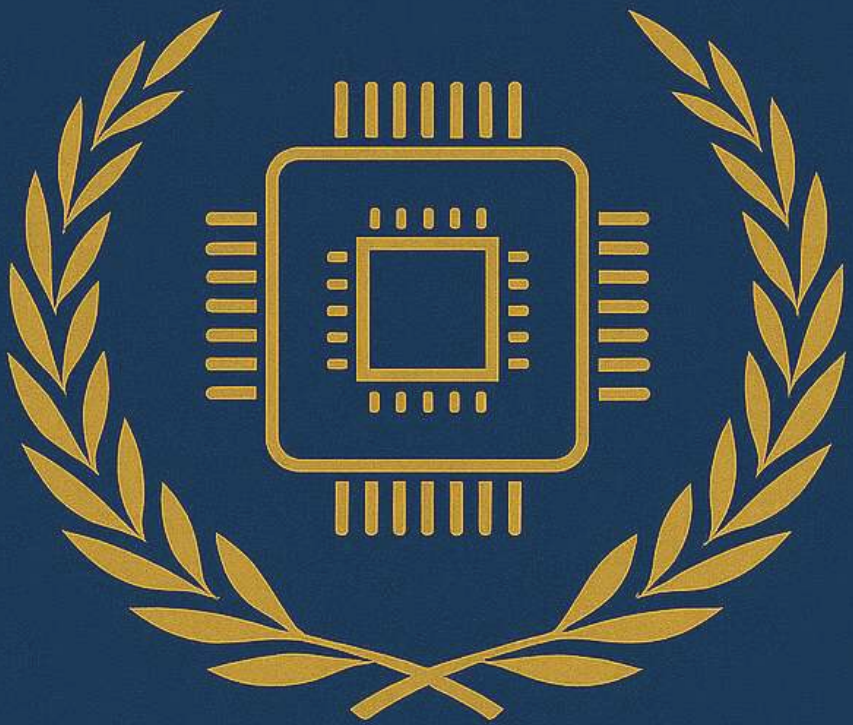
COMP201

BY AYKHAN AHMADZADA



COURSE NOTES

Computer Systems &
Programming

KOÇ UNIVERSITY

COMP201 · Computer Systems & Programming

Koç University — Spring 2025 | Department of Computer Engineering

INSTRUCTOR & TA

Instructor: Aykut Erdem

TA: N/A

SCHEDULE

Lectures: Tuesday & Thursday, 16:00–17:10 (Tower, Second Floor)

Labs: Friday — 14:00–15:40 (Lab A, SNA B242) • 16:00–17:40 (Lab B, SNA B149)

Delivery: In person. Lectures recorded and made available via KUHub Learn.

ASSESSMENT

- 9 Labs (lowest 2 dropped, no make-up): **21%**
- Class Participation: **5%**
- 5 Programming Assignments (A0 2%, A1–A4 4% each): **18%**
- Midterm Exam: **28%**
- Final Exam: **28%**

COURSE DESCRIPTION

Solid understanding of the principles and abstractions of computer systems and machine programs using C. Topics include the C language, compilation flow and runtime behavior, computer arithmetic, the relationship between C and its assembly translation, and practical Unix power-user skills (shell, version control, compilers, debuggers, profilers).

- Prerequisite: COMP132 (strong programming background recommended).
- Learning outcomes: memory management proficiency; compilation & runtime insights; computer arithmetic; C↔assembly relationship; Unix tooling.

TEXTS & REFERENCES

- R. E. Bryant & D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson (2016).
- B. W. Kernighan & D. M. Ritchie, *The C Programming Language*.
- A. Athalye, J. Gjengset, J. J. G. Ortiz, *The Missing Semester of Your CS Education*, MIT (2020).

Course Webpage: https://aykuterdem.github.io/classes/comp201.s25

This work is a personal academic compilation created for educational purposes as part of the COMP201 course at Koç University.

Compiled in Istanbul, Turkey.

To everyone who learned that pointers are power and responsibility.

COMP201

1. Introduction: Unix, the Command Line, and Basic C Programming

Objective: These notes cover **Lecture 1** (from Slide 22 through the end), focusing on **Unix and the Command Line**, as well as an **Introduction to the C Language**.

Unix and the Command Line

Unix is a family of multitasking, multiuser operating systems that share a common set of standards and tools. Many modern systems (e.g., Linux, macOS) trace their origins to Unix.

What is Unix?

Unix defines a standard environment and command set used widely for:

- **Server administration** (running websites, databases)
- **Software development** (compiling, debugging, version control)
- **Embedded systems** (Raspberry Pi, IoT devices)

What is the Command Line?

> **COMMAND-LINE INTERFACE:** A text-based interface to interact with a computer system by typing commands, rather than using graphical icons and menus.

A command-line interface (CLI) allows you to navigate directories, create/remove/edit files, and execute programs or scripts directly.

Command Line vs. GUI

- **Graphical User Interface (GUI)**: Uses icons, windows, and menus.
- **CLI**: Text-based, often requiring memorized commands or references.

Even though the CLI appears more "retro," it remains powerful and flexible for development tasks, scripting, and large-scale automation.

Why Use Unix / the Command Line?

- **Consistency**: One set of commands/tools (`ls`, `cp`, `rm`, etc.) works across many systems.
- **Versatility**: Easily handles repetitive tasks, advanced scripting, remote administration.
- **Efficiency**: Powerful command chaining (piping), quick file navigation, and automation.

Unix Commands Recap

Command	Description
ls	Lists files in the current folder
cd	Changes the current directory
mkdir	Creates a new directory/folder
rm	Removes a file or folder
man	Displays the manual for a command
vi/emacs	Opens a text editor in the terminal

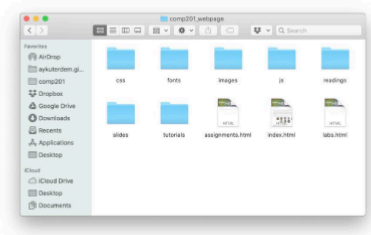
Example:

```
cd assignments   // change to the "assignments" folder
ls               // list files in the current folder
```

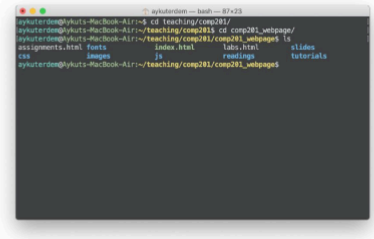
Command Line vs. GUI

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

The C Language

C was developed in the early 1970s to facilitate writing operating systems like Unix. It provides low-level access to memory, compiles to efficient machine code, and forms the basis for many modern languages (C++, Objective-C, Java).

C vs. C++ and Java

All three share basic syntax and structures (loops, conditionals). However:

- **C**: Procedural, minimal abstraction, direct memory manipulation.
- **C++**: Adds object-oriented features, large libraries, operator overloading, and templates.
- **Java**: Runs on a virtual machine with garbage collection, fully object-oriented, large standard library.

Programming Language Philosophies

> **PROCEDURAL PARADIGM:** Organizes code into procedures/functions operating on data, rather than bundling data and methods together.

C's design emphasizes performance and direct hardware control over high-level safety features.

Why C?

- **Efficiency**: Speed and minimal overhead (often used in OS kernels, embedded systems).
- **Portability**: Runs on nearly every platform.
- **Foundation**: Influenced many subsequent languages and is still widely used for systems programming.

Programming Language Popularity

C remains top-ranked in surveys (e.g., TIOBE index) due to its broad usage in high-performance and low-level applications.

Our First C Program

A simple "Hello, world!" in C:

```
/*
 * hello.c
 * Prints a welcome message.
 */
#include <stdio.h>  // for printf

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

- `#include <stdio.h>` provides `printf`.
- `main` returns an integer (0 = success).

- `argc` and `argv` allow for command-line arguments.

Familiar Syntax

C shares operators and control structures with C++/Java:

```
int x = 10;
for (int i = 0; i < x; i++) {
    if (i % 2 == 0) {
        printf("Index %d is even.\n", i);
    }
}
```

Boolean Variables

C uses `stdbool.h` for the `bool` type, with values `true` or `false`. Without `<stdbool.h>`, any nonzero integer is considered true, and 0 is false.

```
#include <stdbool.h>

bool condition = false;
if (condition) {
    // ...
}
```

Console Output (printf)

> PRINTF: Prints text to standard output, defined in <stdio.h>. Format placeholders match the argument types: %d for int, %s for string, %f for double, etc.

Example:

```
int num = 201;
printf("Welcome to COMP%d\n", num);  // "Welcome to COMP201"
```

Writing, Debugging, and Compiling

The typical C workflow involves:

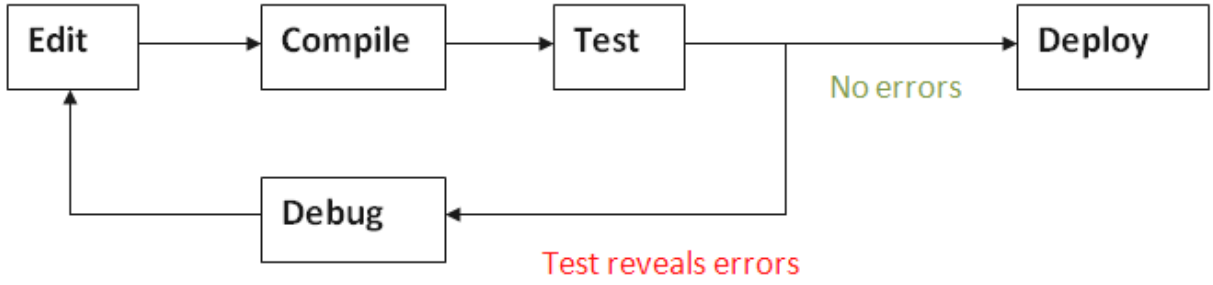
1. Editing code (e.g., `vi`, `emacs`, or another editor).
2. Compiling (e.g., `gcc`, `clang`).
3. Running the executable.
4. Debugging (e.g., using `gdb`, logging statements).

Example:

```
gcc hello.c -o hello  // compile
./hello               // run
```

Explanation of Each Part:

Keyword	Meaning
`gcc`	The **GNU Compiler Collection** command-line tool used to compile C programs.
`hello.c`	The **source code file** containing the C program that needs to be compiled.
`-o`	Specifies the **output file name** for the compiled executable.
`hello`	The **name of the generated executable file**. If omitted, the default output is `a.out`.



Demo: Compiling and Running a C Program

Steps:

1. Edit a source file (e.g., `hello.c`).

2. Compile with `gcc hello.c -o hello`.
3. Run via `./hello`.
4. If issues arise, use `gdb` or print statements to debug.

Recap

- **Unix & Command Line**: Essential for file management, building, and running programs.
- **C Language**: Procedural, low-level, efficient—ideal for learning how software interacts closely with hardware.

Next time, we'll delve deeper into how computers represent data in memory, manage processes, and handle more advanced system-level details.

Self Test

Self-Test: Lecture 1

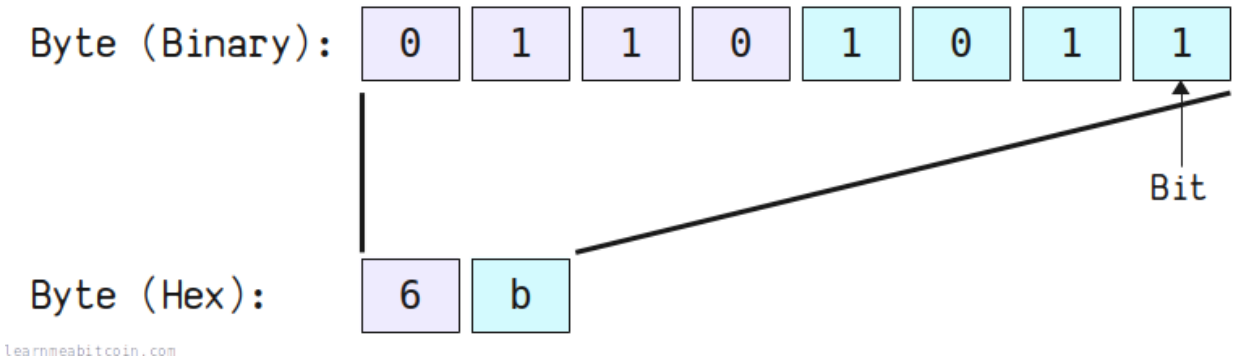
2. Bits and Bytes, Representing and Operating on Integers

Bits and Bytes

In a digital computer, the **bit** (binary digit) is the smallest unit of information. Each bit can be either 0 (off) or 1 (on). A **byte** is a group of 8 bits. Modern computer architectures typically organize memory in byte-addressable form, meaning each byte has a unique address in memory.

- Storing data (like text, images, audio) ultimately comes down to representing patterns of 0s and 1s.
- A single byte can hold values from **0** to **255** when using **unsigned** representation (2^8 possible patterns).

> **BYTE-ADDRESSABLE MEMORY:** Each byte in memory is assigned a unique numeric address, allowing the CPU to access or modify it.



Why Bits?

At the hardware level, electronic circuits use transistors that switch between two states (voltage high or low). Software uses these two states to form the conceptual "0 or 1" representation.

Base Conversions

Bits are naturally expressed as binary (base 2). However, we often convert between binary, decimal (base 10), and hexadecimal (base 16) for readability.

Binary (Base 2)

- Uses digits 0 and 1.
- For example, 1011_2 means:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}.$$

Decimal (Base 10)

- Our everyday number system uses digits 0–9.
- Converting from binary to decimal adds up powers of 2. Converting decimal to binary often uses repeated division by 2, keeping track of remainders.

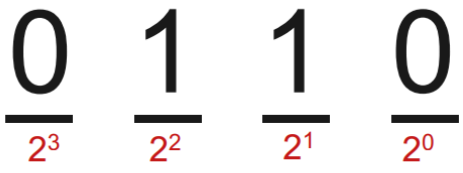
Base 2

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

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of $2 \leq 6$? $\mathbf{2^2=4}$
  - Now, what is the largest power of $2 \leq 6 - 2^2$? $\mathbf{2^1=2}$
  - $6 - 2^2 - 2^1 = 0$!



$= \mathbf{0}*8 + \mathbf{1}*4 + \mathbf{1}*2 + \mathbf{0}*1 = 6$

Hexadecimal (Base 16)

- Uses digits 0–9 plus A–F for 10–15.
- Each hex digit matches exactly 4 bits (binary):
  - For instance, `0xF` corresponds to `1111₂`.

- Notation: `0x3A` or `0X1F2B` typically indicates a hexadecimal number.

- Convenient for compressing large binary numbers: 32 bits can be expressed with just 8 hex digits.

> **HEX DIGIT:** A single symbol from $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$, each representing four bits.

Binary to Hexadecimal Table

MATH MONKS

Binary (Base 2)	Hexadecimal (Base 16)	Binary (Base 2)	Hexadecimal (Base 16)
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Multiplying by Base

$$1450 \times 10 = 14500$$

$$1100_2 \times 2 = 11000$$

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!

Integer Representations

Computers store integers in bit patterns. The common integer categories are:

1. **Unsigned integers**: Nonnegative (0 and above).
2. **Signed integers**: Include negative values, zero, and positive values.

In C, different integer types occupy different numbers of bytes, affecting the range of values they can hold. For instance:

- `int` on many systems is 4 bytes (32 bits), typically ranging from -2^{31} to $2^{31}-1$ if signed, or 0 to $2^{32}-1$ if unsigned.
- `long` (on 64-bit systems) is often 8 bytes (64 bits).

> **C DECLARATIONS:**
>
> ```
> int x;  // typically 4 bytes (32-bit)
> unsigned int y;  // 4 bytes (32-bit), but stores only nonnegative
> numbers
> ```

Different operating systems and compiler settings may alter these sizes (especially for `long`). The fundamental concept remains the same: a certain number of bits store the integer.

Unsigned Integers

An **unsigned integer** uses all bits to represent nonnegative numbers.

- For an 8-bit unsigned integer:
  - **Minimum** = 0 (binary 00000000)
  - **Maximum** = 255 (binary 11111111)
- For a 32-bit unsigned integer:
  - **Minimum** = 0
  - **Maximum** = $2^{32}-1$ = 4,294,967,295

Arithmetic on unsigned values is straightforward binary addition and subtraction. When a calculation exceeds the maximum representable value, the result "wraps around" from the top back to zero (an effect called **overflow**).

Example:

```
unsigned char a = 255; // 8-bit max
a = a + 1; // Overflow! Wraps around to 0
```

Signed Integers

Signed integers include **negative** values, zero, and **positive** values. The question is how to store "sign information" in bits. Several historical methods exist:

Sign-Magnitude Representation

> **SIGN-MAGNITUDE:** The most significant bit (MSB) indicates sign (0 = positive, 1 = negative). The remaining bits represent magnitude.

- Advantage: Conceptually simple for sign determination.
- Disadvantages:
  - Two ways to represent zero (+0 and −0).
  - Arithmetic operations are more complex because you must handle the sign bit separately.

Two's Complement (Modern Standard)

> **TWO'S COMPLEMENT:** A negative number is formed by inverting (flipping) all bits of its positive version (one's complement) and then adding 1.

1. If you have `+6` (binary `00000110` in 8 bits), you invert bits → `11111001`, then add 1 → `11111010`. This results in the binary representation for −6.
2. To go back from `-6`, perform the same two's complement steps.

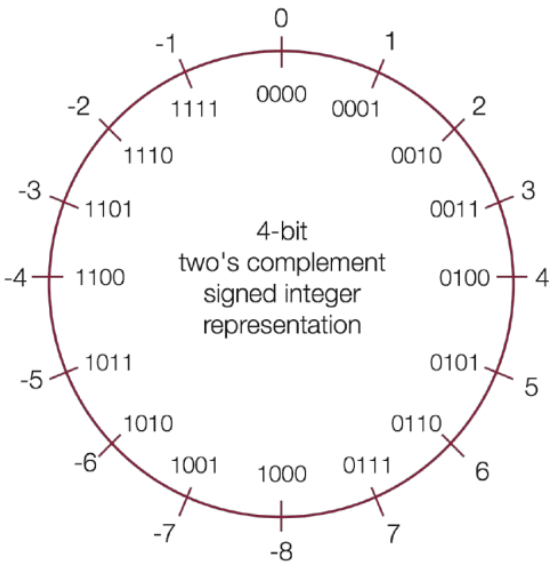
Two's complement provides:

- **One unique zero** (no separate +0 and −0).
- **Simple addition**: No special sign logic. Normal binary addition works for negative and positive.
- **Efficient hardware** for arithmetic.

In a 32-bit two's complement:

- The MSB still indicates negativity if it's 1.
- The range is typically -2^{31} through $+2^{31}-1$.

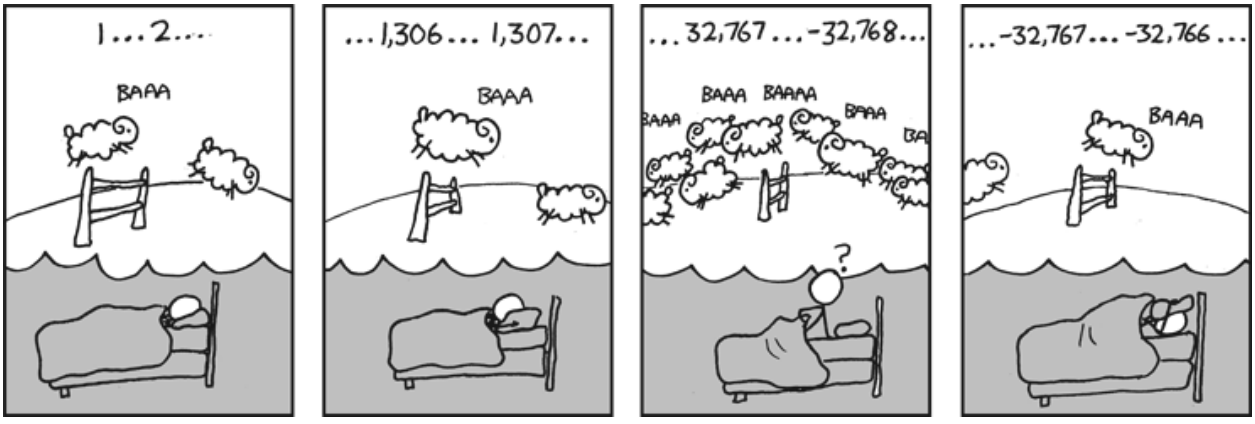
Two's Complement



Overflow

Overflow occurs if an arithmetic result doesn't fit within the available bits. The final result "wraps around" in a seemingly unpredictable way.

- **Unsigned overflow**: If you add 1 to 255 in an 8-bit unsigned, you get 0.
- **Signed overflow**: If you exceed $+2^{31}-1$ (for 32-bit), it wraps to a negative number.



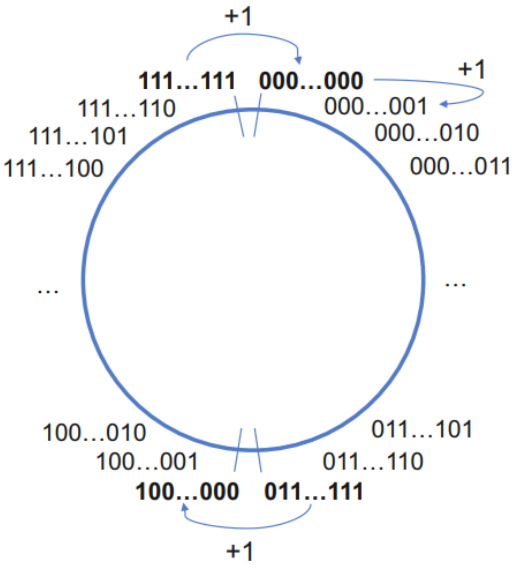
Real-World Examples

- **The Gandhi bug** in the game *Civilization*: When Gandhi's aggression rating (1) was reduced by 2, it underflowed from 1 to 255, making him extremely aggressive.

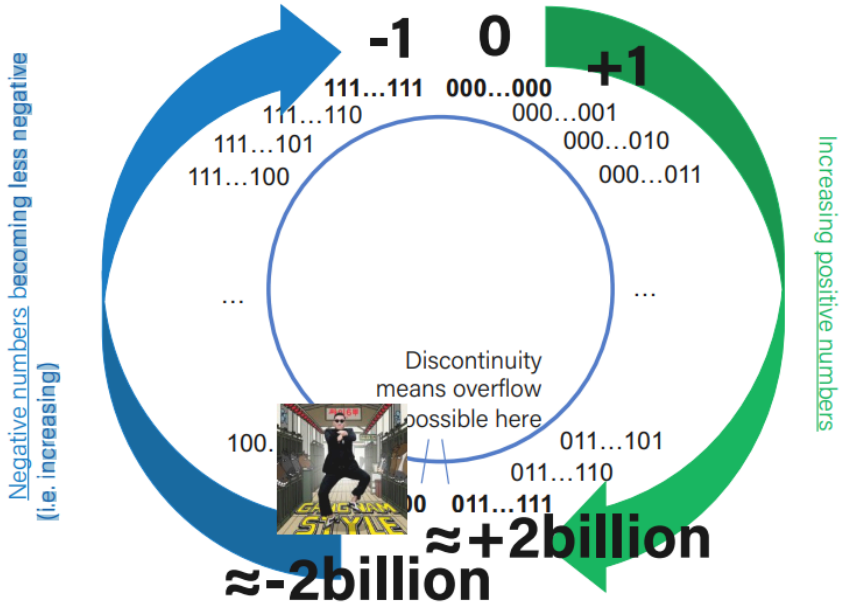
- **Windows 95** uptime limit: The system timer was stored in a 32-bit integer counting milliseconds. After ~49.7 days, it overflowed and crashed.

> **CAUSE:** Finite bits can't store infinitely large or small numbers.
>
> **RESULT**: Wrap-around or weird negative/positive flips.

Overflow



Signed Numbers



Casting and Combining Types

In C, operations involving different integer types can lead to implicit conversions:

- A **signed** value combined with an **unsigned** of the same size typically converts the signed operand to unsigned (if 32 bits each, for example). Negative signed values become large positive values when reinterpreted in unsigned form.
- Printing with the wrong format specifier (`%d` vs. `%u` vs. `%x`) interprets the same bits differently in output.

Example:

```
int n = -5;
unsigned int m = n;   // same bits, but interpreted as a large unsi
gned value
printf("n = %d, m = %u\n", n, m);
```

```
Output:
n = -5, m = 4294967291
```

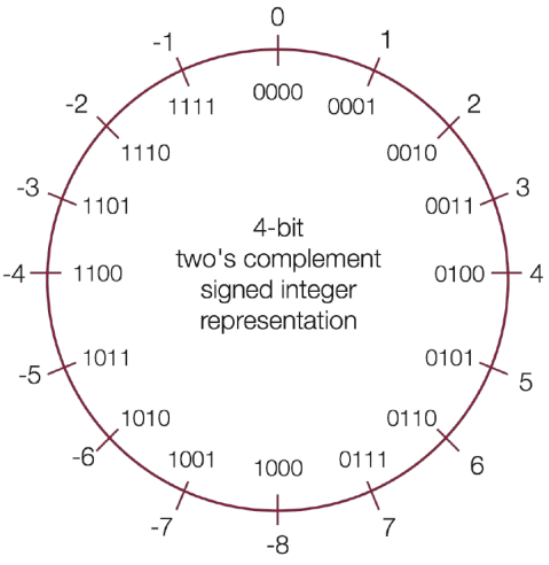
```
...Program finished with exit code 0
```

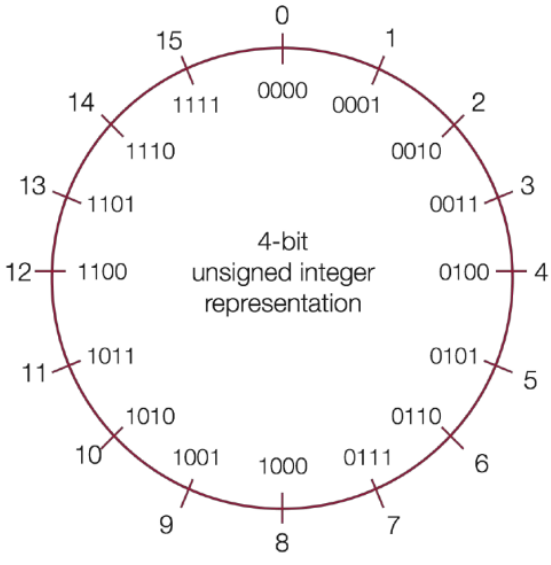
If `n`'s 32-bit pattern is `11111111 11111111 11111111 11111011`, then `m` sees that as 4,294,967,291 in decimal.

When comparing `signed` and `unsigned` in expressions, the signed operand is promoted to `unsigned`, which can lead to unexpected results (e.g., –1 might become a large positive unsigned).

The binary pattern for –1 in 32-bit two's complement is `0xFFFFFFFF`, which is 4,294,967,295 when interpreted as unsigned 32-bit.

Casting





[1]

Code Examples

Unsigned Overflow

```
#include <stdio.h>
int main(void) {
    unsigned int x = 4294967295U; // max 32-bit unsigned
    x = x + 1; // wraps around to 0
```

```
    printf("Wrapped: %u\n", x);   // prints 0
    return 0;
}
```

Two's Complement Negative

```
#include <stdio.h>

int main(void) {
    int positive = 6;       // 0000 0110 in 8-bit, conceptual
    int negative = ~6 + 1;  // two's complement: invert bits (~) an
d add 1
    printf("%d\n", negative);
    return 0;
}
```

```
Output:
-6

...Program finished with exit code 0
```

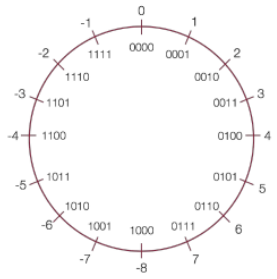
Depending on the system (usually 32-bit or 64-bit `int`), the logic remains the same, just with more bits involved.

Comparison Between Different Types

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

Summary

- **Bits and Bytes**: The fundamental representation of data in digital systems.
- **Base Conversions**: Binary → decimal → hex are useful to read and write integer values.
- **Unsigned Integers**: Store only nonnegative values; wrap around on overflow.
- **Signed Integers**: Typically use two's complement for negative representation.
- **Overflow**: Occurs when results exceed the representable range, causing wrap-around.
- **Casting and Combining Types**: Bit patterns stay the same, but the interpretation changes if you switch from signed to unsigned or use mismatched `printf` format specifiers.

Understanding these fundamentals is crucial for debugging low-level behaviors, ensuring correct arithmetic operations, and writing robust C programs that handle all edge cases.

Self Test

⚛ Self-Test: Lecture 2

4. Floating Points

Representing Real Numbers

Real numbers include fractions and decimals and have infinitely many possible values between any two integers. Unlike integers, a fixed-width representation must approximate real numbers.

- **Challenge:** There are infinitely many real numbers between any two integers.
- **Solution:** Use a fixed-width representation that sacrifices some accuracy for a finite, manageable format.
- **Approaches:** Two primary methods are used:
  - **Fixed Point Representation**
  - **Floating Point Representation**

> **REAL NUMBER REPRESENTATION:** A method to encode numbers with fractional parts in a fixed number of bits.

Fixed Point Representation

Fixed point representation is similar to the standard decimal representation but in binary. It extends the integer representation by adding a fixed number of bits for the fractional

part.

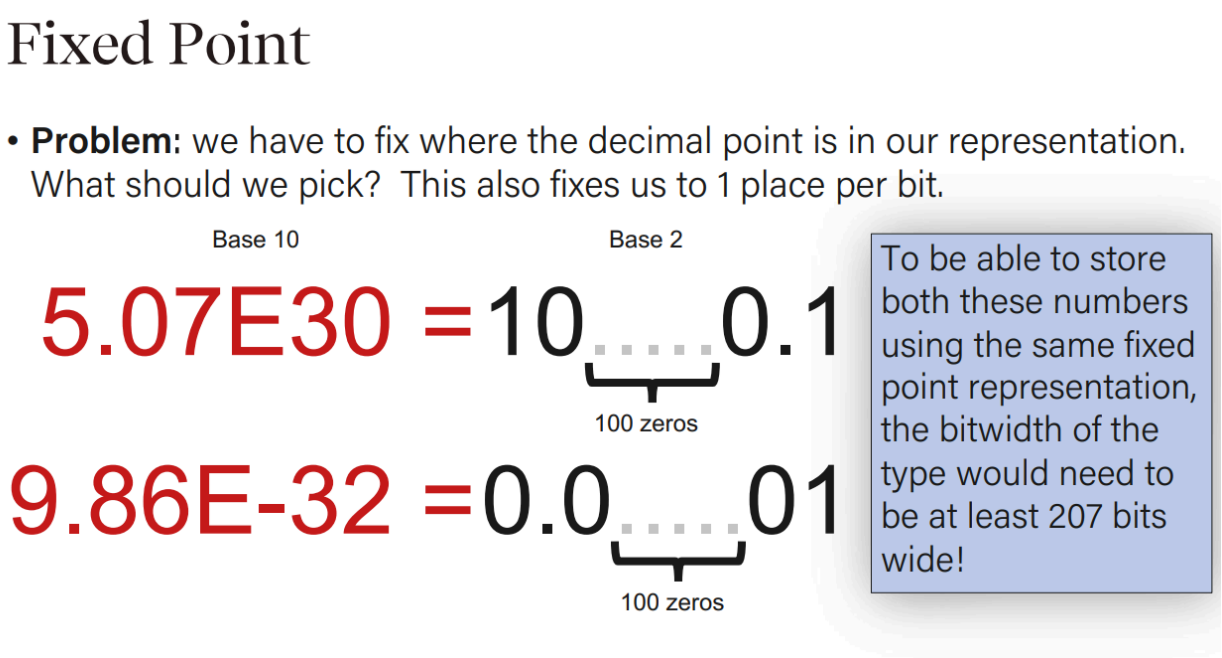
- **Concept:** The binary point is fixed in one position.
- **Pros:**
  - Arithmetic is straightforward.
  - Precision is predictable.
- **Cons:**
  - The location of the binary point is fixed, limiting the range.
  - To cover both very large and very small numbers, the bit-width must be increased significantly.

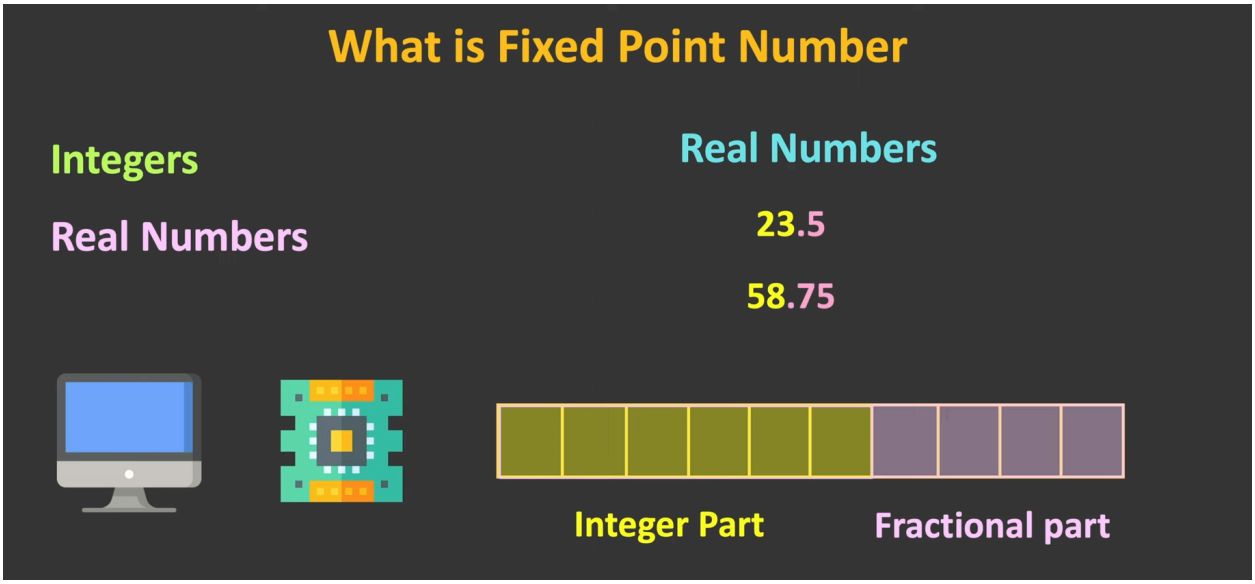
Example:

A fixed point number might be represented as:

$$1011.011$$

where the bits to the left represent the integer part and those to the right represent the fractional part.





Floating Point Representation

Floating point representation allows the decimal (or binary) point to "float," enabling a much wider range of values to be represented.

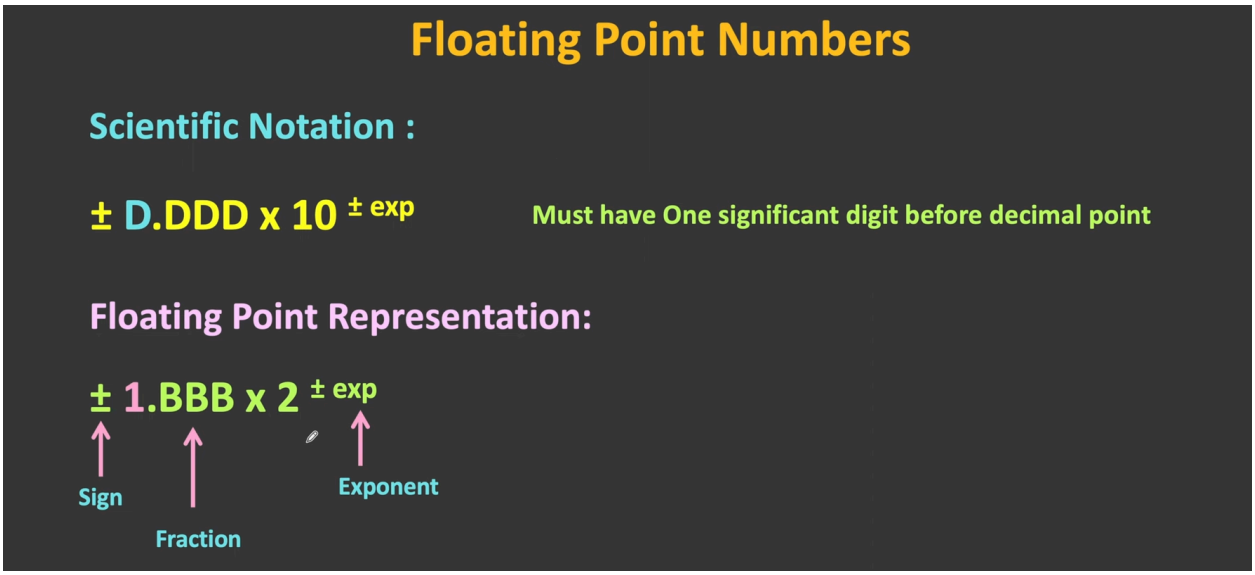
- **Format:** Numbers are expressed in the form:

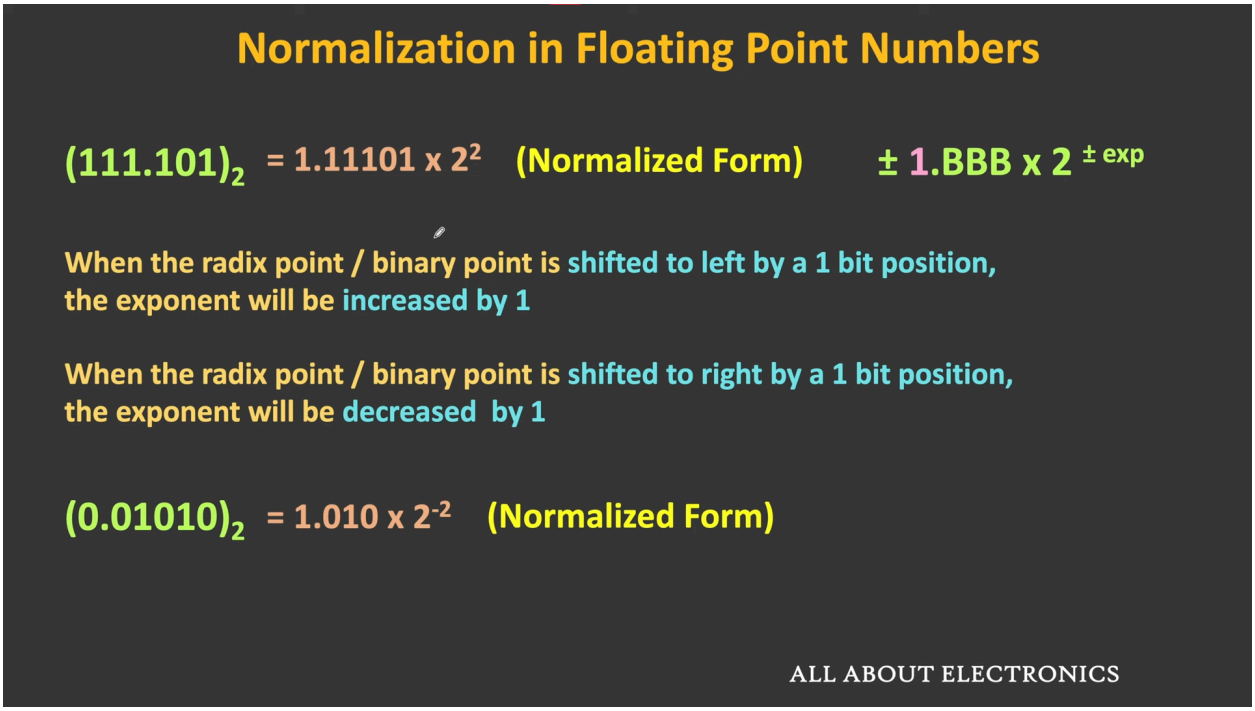
$$\pm x \times 2^{E}$$

  where x is the significand (or mantissa) and E is the exponent.

- **Advantages:**
  - Wide dynamic range: can represent very small and very large numbers.
  - Flexible: accommodates scientific notation.
- **Disadvantages:**
  - Not every real number can be represented exactly.
  - Rounding and precision issues occur due to the finite number of bits.

> **FLOATING POINT REPRESENTATION:** A method to represent real numbers that uses a significand and an exponent, allowing the decimal point to move, thereby enabling a wide range of values.





IEEE Floating Point Format

The IEEE 754 standard is widely used for floating point arithmetic in digital systems.

Structure of Single Precision (32-bit)

- **Sign Bit (1 bit):** 0 for positive, 1 for negative.
- **Exponent (8 bits):** Stored with a bias (127 for single precision).

- **Fraction (23 bits):** Represents the fractional part of the significand.

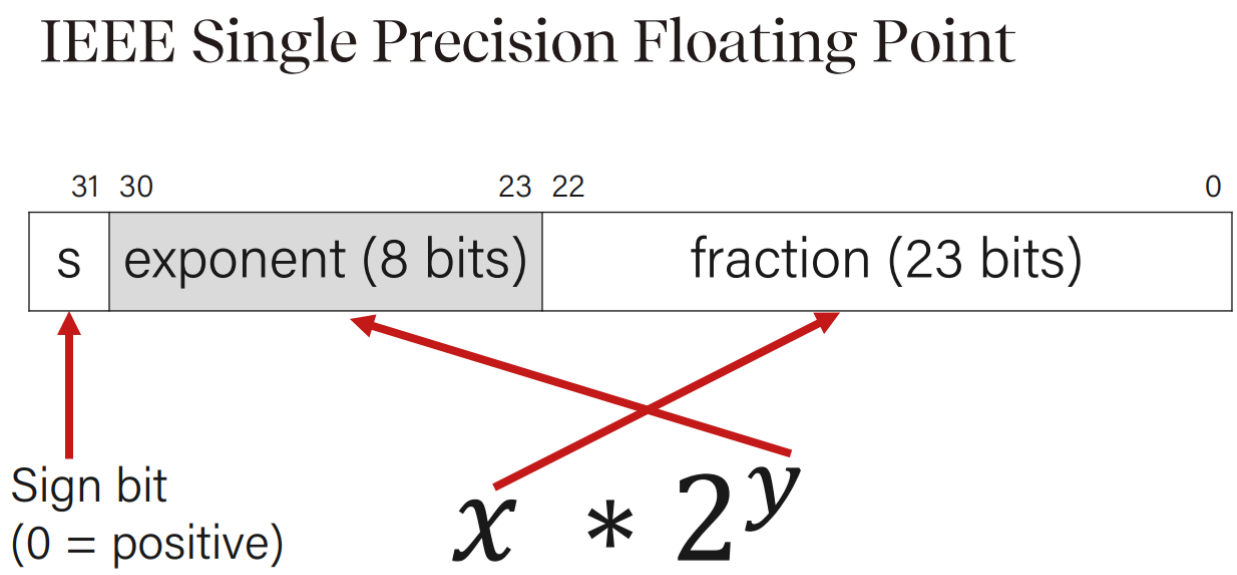
The number is represented as:

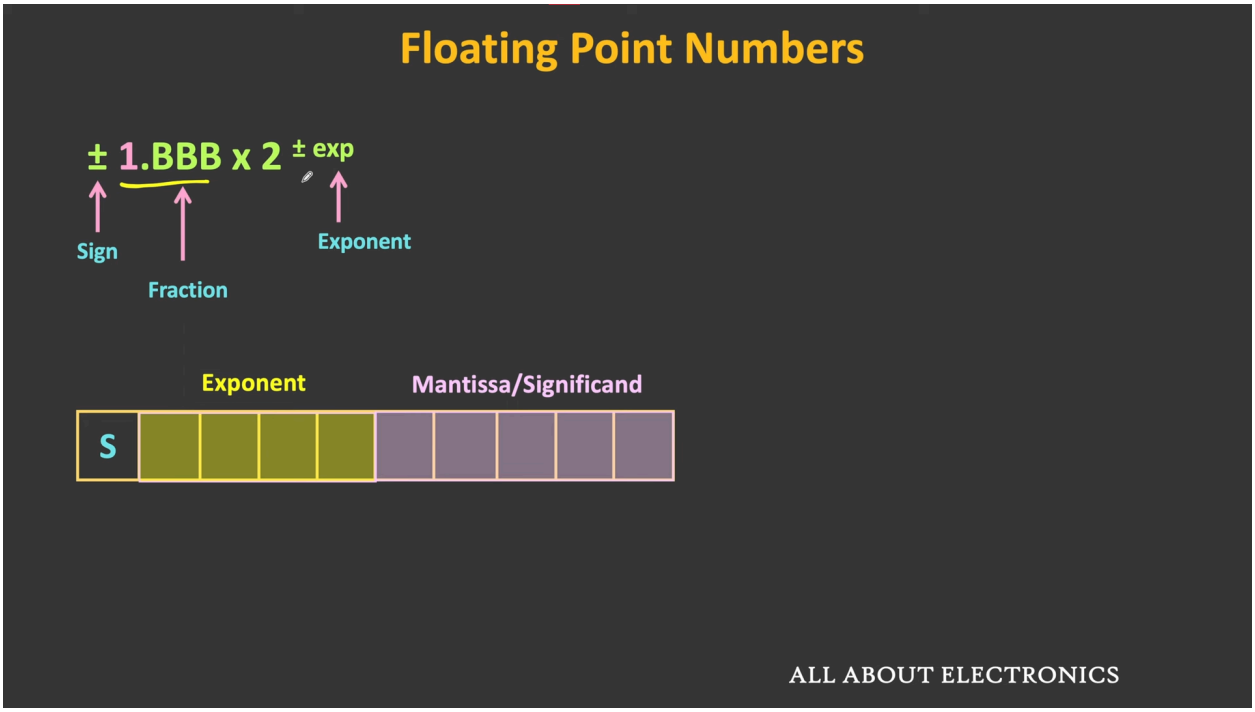
$$(-1)^s \times 1.f \times 2^{(e-127)}$$

where:

- s is the sign bit.
- e is the stored exponent.
- f is the fractional part (the bits to the right of the binary point).

> **IEEE SINGLE PRECISION FLOAT:** A 32-bit format that provides a balance between range and precision.





Exponent Field and Bias

- **Bias:** The exponent field in IEEE single precision has a bias of 127.
- **Actual Exponent:** Calculated by subtracting the bias from the stored exponent value.

$$\text{Actual Exponent} = e - 127$$

- This method allows both positive and negative exponents to be stored in an unsigned format.

> **NOTE:** The exponent is stored as an unsigned integer with a fixed bias, which simplifies comparison and arithmetic at the bit level.

IEEE 754

Half Precision (16 bits)

Single Precision (32 bits)

Double Precision (64 bits)

Quadruple Precision (128 bits)

Octuple Precision (256 bits)

ALL ABOUT ELECTRONICS

IEEE 754 Single Precision Format

$\pm 1.BBB \times 2^{\pm exp}$

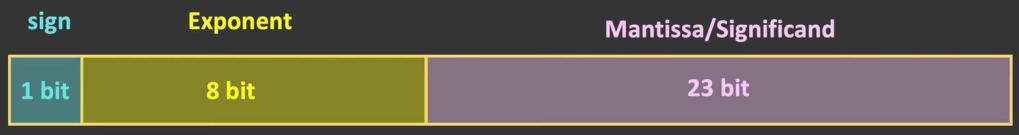
Sign

Fraction

Exponent

Exponent

8-bits → 0 to 255 (unsigned)



ALL ABOUT ELECTRONICS

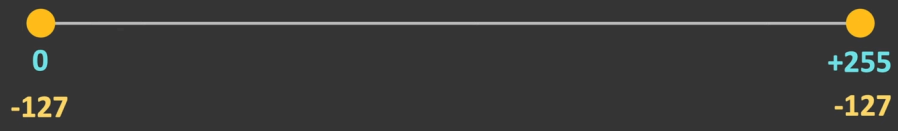
Biased Representation

$$Bias = 2^{n-1} - 1$$

n - no of bits

8 bits ➡ Bias = 127

In biased Representation



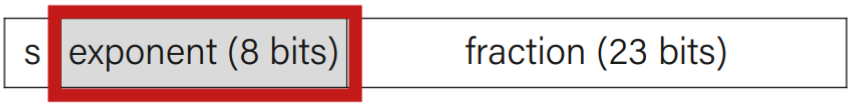
ALL ABOUT ELECTRONICS

Biased Representation

Actual Number	Biased Number	Biased Representation
-127	0	0000 0000
-126	1	0000 0001
.....	.....	.....
-1	126	0111 1110
0	127	0111 1111
1	128	1000 0000
.....		
127	254	1111 1110
128	255	1111 1111

ALL ABOUT ELECTRONICS

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 Field and Normalization

- **Normalization:** In normalized numbers, the significand is adjusted so that there is an implicit leading 1 before the binary point. This "hidden bit" provides an extra bit of precision.
- **Fraction Field:** Only the bits to the right of the binary point are stored.

$$\text{Stored Value} = 1.f$$

- **Denormalized Numbers:** When the exponent is all zeros, the number is denormalized; the implicit leading 1 is assumed to be 0, allowing representation of numbers very close to zero.

> **KEY POINT:** Normalization maximizes precision by ensuring that the significand is in a standard form.

1. Rule for Normalized vs. Denormalized

Condition	Type	Leading Digit in Mantissa
Exponent ≠ 0 (`1 ≤ exponent ≤ 254`)	**Normalized**	Implicit `1` (`1.xxx`)
Exponent = 0 (`00000000`)	**Denormalized**	Implicit `0` (`0.xxx`)

Key Difference:

- **Normalized numbers** → Always have an **implicit `1` before the mantissa.**
- **Denormalized numbers** → The leading `1` **is missing**, so the mantissa is `0.xxx`.

Floating Point Arithmetic and Pitfalls

Floating point arithmetic is not as straightforward as integer arithmetic due to several issues:

- **Alignment:** To add two floating point numbers, their exponents must be aligned, which may require shifting the significand and can lead to loss of precision.
- **Rounding:** The result of an operation may be rounded to fit into the available bits.
- **Non-Associativity:** Floating point addition is not necessarily associative. For example:

$$(3.14 + 1e20) - 1e20 \neq 3.14 + (1e20 - 1e20)$$

- **Over/Underflow:** Results that exceed the representable range become infinity or zero (or denormalized numbers), leading to unexpected behavior.

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.

> **PRACTICAL TIP:** Avoid using floating point numbers for high-precision financial calculations and be cautious when comparing floating point values for equality.

Floating Point in C

C provides two main floating point types:

- **float:** Single precision (32-bit)
- **double:** Double precision (64-bit)

Conversions and Casting:

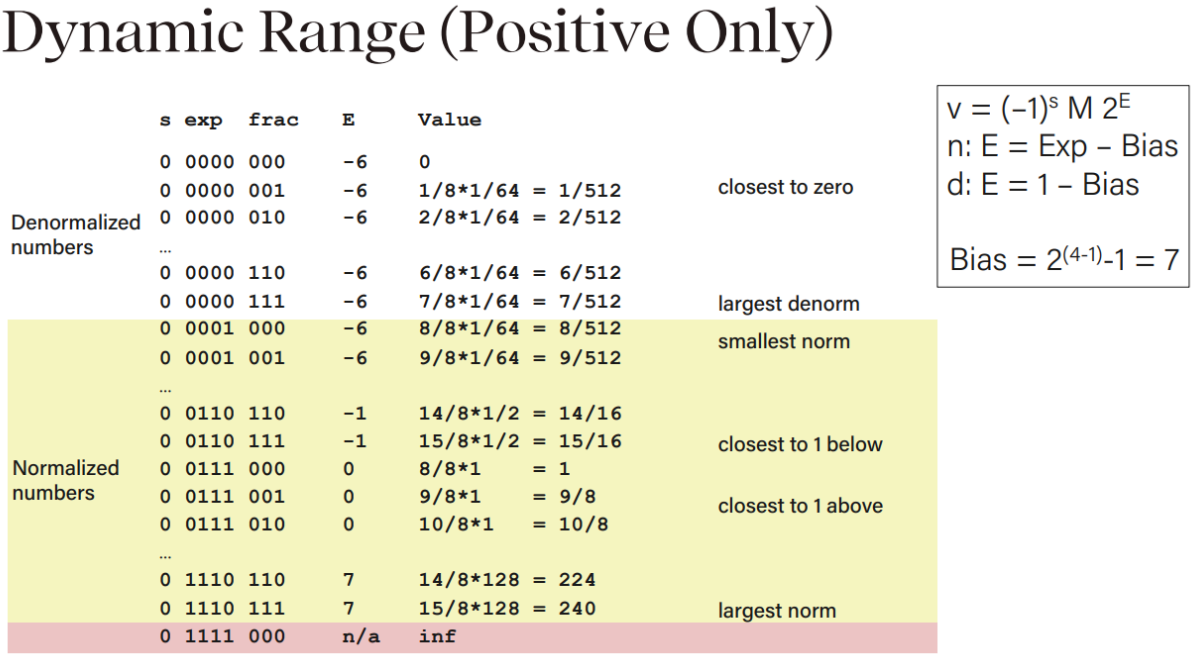
- Converting from float or double to int truncates the fractional part (rounding toward zero).
- Conversions may result in rounding errors or undefined behavior when the value exceeds the target type's range.

> **NOTE:** Choose the appropriate floating point type (float or double) based on the required precision and range for your application.

The Ariane 5 Example: Real-World Implications

On June 4, 1996, the Ariane 5 rocket self-destructed shortly after liftoff due to an overflow error:

- **Cause:** Conversion from a 64-bit floating point number to a 16-bit signed integer led to an overflow.
- **Context:** Software reused from Ariane 4 assumed that the horizontal velocity would always be within the range of a 16-bit integer. However, Ariane 5's higher speed invalidated this assumption.
- **Lesson:** Even well-established standards can fail if underlying assumptions are not revalidated in new contexts.



41

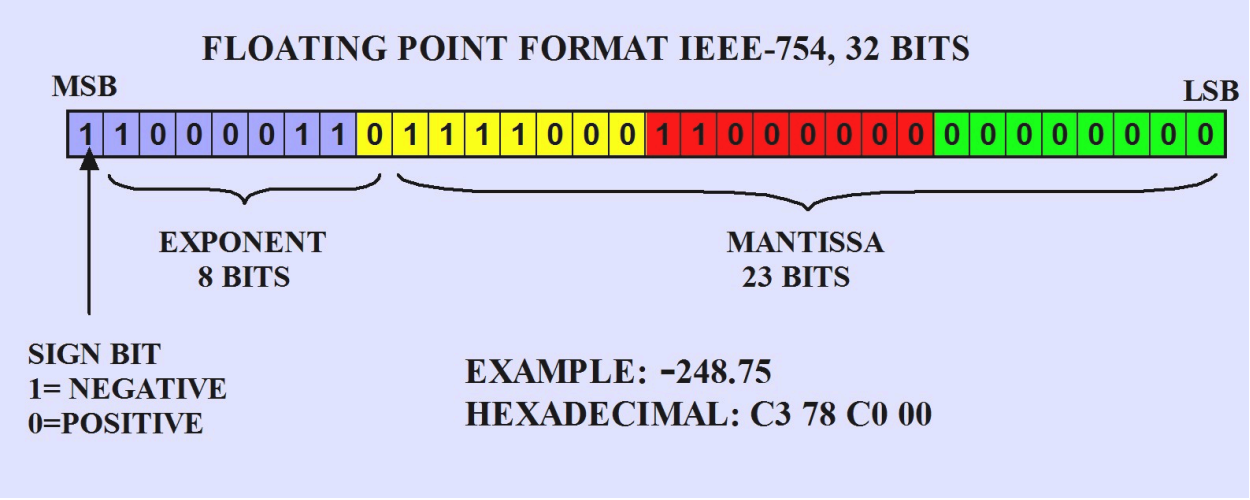
Summary and Conclusion

This lecture has covered:

- **Representing Real Numbers:** The challenges of representing an infinite set of real numbers in a fixed-width format.
- **Fixed Point Representation:** Its simplicity and inherent limitations.
- **Floating Point Representation:** The IEEE standard, including the structure of single precision (sign, exponent with bias, fraction), and normalization.

- **Floating Point Arithmetic:** Issues such as alignment, precision loss, non-associativity, and overflow/underflow.
- **Floating Point in C:** How C implements floating point types (float and double) and the implications of conversions.
- **Real-World Impact:** The Ariane 5 incident illustrates the critical importance of proper floating point handling in safety-critical systems.

Understanding these concepts is crucial for designing reliable digital systems and writing robust software that involves numerical computations.



Self Test

- Self-Test: Lecture 4

5. Chars and Strings in C

Objective & Scope
This note covers the essential concepts from the lecture on **Chars and Strings in C**. It explains how characters and strings are represented and manipulated in C, details the standard library functions for string operations.

Characters in C

> **CHAR:** A basic data type in C that represents a single character (or glyph).

- Typically 1 byte (8 bits) in size.
- Characters are stored as integer values using ASCII encoding:
  - For example, `'A'` is 65, `'a'` is 97, and `'0'` is 48.
- Arithmetic on characters is supported:

```
char uppercaseA = 'A'; // 65
int diff = 'c' - 'a';  // 2
```

> **UNICODE & UTF-8:** Modern systems often use Unicode to represent a vast range of characters.

- **UTF-8** is a variable-width encoding that efficiently stores common English characters while supporting many others.

ASCII

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	U+0000 NUL 0	U+0001 SOH 1	U+0002 STX 2	U+0003 ETX 3	U+0004 EOT 4	U+0005 ENQ 5	U+0006 ACK 6	U+0007 BEL 7	U+0008 BS 8	U+0009 HT 9	U+000A LF 10	U+000B VT 11	U+000C FF 12	U+000D CR 13	U+000E SO 14	U+000F SI 15
1_	U+0010 DLE 16	U+0011 DC1 17	U+0012 DC2 18	U+0013 DC3 19	U+0014 DC4 20	U+0015 NAK 21	U+0016 SYN 22	U+0017 ETB 23	U+0018 CAN 24	U+0019 EM 25	U+001A SUB 26	U+001B ESC 27	U+001C FS 28	U+001D GS 29	U+001E RS 30	U+001F US 31
2_	U+0020 SP 32	U+0021 ! 33	U+0022 " 34	U+0023 # 35	U+0024 $ 36	U+0025 % 37	U+0026 & 38	U+0027 ' 39	U+0028 (40	U+0029) 41	U+002A * 42	U+002B + 43	U+002C , 44	U+002D - 45	U+002E . 46	U+002F / 47
3_	U+0030 0 48	U+0031 1 49	U+0032 2 50	U+0033 3 51	U+0034 4 52	U+0035 5 53	U+0036 6 54	U+0037 7 55	U+0038 8 56	U+0039 9 57	U+003A : 58	U+003B ; 59	U+003C < 60	U+003D = 61	U+003E > 62	U+003F ? 63
4_	U+0040 @ 64	U+0041 A 65	U+0042 B 66	U+0043 C 67	U+0044 D 68	U+0045 E 69	U+0046 F 70	U+0047 G 71	U+0048 H 72	U+0049 I 73	U+004A J 74	U+004B K 75	U+004C L 76	U+004D M 77	U+004E N 78	U+004F O 79
5_	U+0050 P 80	U+0051 Q 81	U+0052 R 82	U+0053 S 83	U+0054 T 84	U+0055 U 85	U+0056 V 86	U+0057 W 87	U+0058 X 88	U+0059 Y 89	U+005A Z 90	U+005B [91	U+005C \ 92	U+005D] 93	U+005E ^ 94	U+005F _ 95
6_	U+0060 ` 96	U+0061 a 97	U+0062 b 98	U+0063 c 99	U+0064 d 100	U+0065 e 101	U+0066 f 102	U+0067 g 103	U+0068 h 104	U+0069 i 105	U+006A j 106	U+006B k 107	U+006C l 108	U+006D m 109	U+006E n 110	U+006F o 111
7_	U+0070 p 112	U+0071 q 113	U+0072 r 114	U+0073 s 115	U+0074 t 116	U+0075 u 117	U+0076 v 118	U+0077 w 119	U+0078 x 120	U+0079 y 121	U+007A z 122	U+007B { 123	U+007C | 124	U+007D } 125	U+007E ~ 126	U+007F DEL 127

Common ctype.h Functions

- `isalpha(ch)`: Checks if `ch` is a letter.
- `islower(ch)`, `isupper(ch)`: Determine the case of `ch`.
- `isspace(ch)`: Checks for whitespace (e.g., space, tab, newline).
- `isdigit(ch)`: Checks if `ch` is a digit.
- `toupper(ch)` and `tolower(ch)`: Convert characters between upper and lower case.

C Strings

> **C STRING:** A sequence of characters stored in an array that ends with a null character ('\0').

- There is no dedicated string type in C; strings are simply arrays of `char`.
- The null terminator indicates where the string ends and must be accounted for in memory allocation.

String Length

- `strlen(str)` computes the number of characters before the null terminator.

```
int length = strlen("Hello");  // returns 5
```

- **Note:** `strlen` is O(n) because it scans the string until `'\0'`.

Passing Strings to Functions

- When a string (a `char[]`) is passed to a function, it decays into a pointer (`char *`) to its first element.

```
void processString(char *str) {
    // Modifications here affect the original string.
}
```

Common String Operations

Comparing Strings

- Use `strcmp(str1, str2)` or `strncmp(str1, str2, n)` to compare string contents.

```
if (strcmp("Hello", "World") == 0) {
    // Strings are identical.
}
```

Copying Strings

- `strcpy(dst, src)` copies the entire string (including the null terminator).
- **Buffer Overflows:** Ensure that the destination array has enough space.

```
char destination[6];
strcpy(destination, "Hello"); // Valid if destination is large e
nough.
```

- `strncpy(dst, src, n)` copies at most `n` characters. It may not append `'\0'` if the source is longer than `n`, so manual termination is often required:

```
char buf[6];
strncpy(buf, "Hello, world!", 5);
buf[5] = '\0';  // Ensure proper termination.
```

Concatenating Strings

- `strcat(dst, src)` appends `src` to the end of `dst`.
- `dst` must have sufficient space for the concatenated result.

```
char greeting[13];
strcpy(greeting, "Hello ");
strcat(greeting, "World!");
// greeting now contains "Hello World!"
```

Working with Substrings

- Because C strings are pointers, you can create substrings by pointer arithmetic:

```
char word[] = "racecar";
char *sub = word + 4;  // Points to "car"
```

- To make an independent substring, use `strncpy` and manually add the null terminator:

```
char subword[5];
strncpy(subword, word, 4);
subword[4] = '\0'; // subword now contains "race"
```

Key Takeaways

- **Characters** in C are stored as small integers (typically using ASCII) and can be manipulated using standard arithmetic and ctype functions.

- **C Strings** are arrays of characters terminated by `'\0'`. They require careful handling to avoid buffer overflows.
- **Standard Library Functions** (from `string.h`) enable common operations like comparison, copying, and concatenation, but always ensure that the destination buffers are large enough and properly null-terminated.
- **Pointer Arithmetic** facilitates efficient substring operations, though modifications affect the original string memory.

Understanding these fundamentals is crucial for safe and effective text manipulation in C.

6. More Strings, Pointers

Recap: C Strings and Common String Functions

> **C STRING:** A C string is an array of characters terminated by a null character ('\\0'). Functions such as strlen depend on this terminator to determine the string's length.

Common Functions in `string.h`

- `strlen(str)`
  Returns the number of characters in a string before the null terminator.

```
int len = strlen("Hello");  // len is 5
```

- `strcmp(str1, str2)` **and** `strncmp(str1, str2, n)`

  Compare two strings lexicographically.

  - `strcmp` compares until a difference is found or a `'\0'` is reached.
  - `strncmp` compares at most `n` characters.

```
if (strcmp("apple", "banana") < 0) {
    // "apple" comes before "banana"
```

```
}
```

- `strchr(str, ch)` **and** `strrchr(str, ch)`
  - `strchr` returns a pointer to the first occurrence of `ch` in `str`.
  - `strrchr` returns a pointer to the last occurrence.

```
char *p = strchr("Daisy", 'a');  // p points to "aisy"
```

- `strstr(haystack, needle)`

  Searches for the substring `needle` in `haystack` and returns a pointer to its first occurrence, or `NULL` if not found.

```
char *sub = strstr("Daisy Dog", "Dog");  // sub points to "Dog"
```

- `strcpy(dst, src)` **and** `strncpy(dst, src, n)`
  - `strcpy` copies the source string (including the null terminator) to the destination array.
  - `strncpy` copies at most `n` characters; if `src` is longer, no `'\0'` is appended automatically.

```
char dest[6];
strcpy(dest, "Hello");  // Correct if dest is large enough.
// If dest is too small, this will cause a buffer overflow.
```

- `strcat(dst, src)` **and** `strncat(dst, src, n)`

  Concatenate `src` onto the end of `dst`, ensuring the result is null-terminated. The destination must be large enough to hold the combined string.

```
char greeting[13];
strcpy(greeting, "Hello ");
strcat(greeting, "World!");  // greeting becomes "Hello World!"
```

- `strspn(str, accept)` **and** `strcspn(str, reject)`

- `strspn` returns the length of the initial segment of `str` containing only characters in `accept`.
- `strcspn` returns the length of the initial segment that contains none of the characters in `reject`.

```
int span = strspn("Daisy Dog", "aDeoi");  // span might be 3
```

Searching in Strings

Searching functions help locate characters or substrings:

- **Character Search:**

  Use `strchr` to find the first occurrence and `strrchr` for the last occurrence of a character.

- **Substring Search:**

  Use `strstr` to locate a substring within another string.

- **Span Functions:**

  `strspn` and `strcspn` measure how many characters at the beginning of a string meet a certain condition (either belonging or not belonging to a set).

Strings as Parameters and Arrays of Strings

- When a C string (a `char[]`) is passed as a parameter, it automatically decays to a pointer (`char *`) to its first element.

```
void processString(char *str) {
    // Operations here affect the original string.
}
```

- An array of strings can be defined using pointers:

```
char *stringArray[] = { "Hello", "Hi", "Hey there" };
printf("%s\n", stringArray[0]);  // Prints "Hello"
```

Pointers in C

> **POINTER:** A variable that stores a memory address. Pointers are essential for dynamic memory management and for passing large data efficiently.

Basics of Pointers

- **Declaration and Initialization:**

```
int x = 2;
int *xPtr = &x;  // xPtr stores the address of x
printf("%d", *xPtr);  // Dereferencing xPtr prints 2
```

- **Pass-by-Reference via Pointers:**

  Since C passes all parameters by value, pointers are used to allow functions to modify the original variable.

```
void modify(int *p) {
    *p = 3;
}

int main() {
    int x = 2;
    modify(&x);
    printf("%d", x);  // x is now 3
    return 0;
}
```

Pointer Arithmetic

- Pointers can be incremented or decremented to traverse arrays:

```
char *str = "apple";  // Assume it points to a string literal
printf("%s\n", str);    // prints "apple"
```

```
printf("%s\n", str + 1);  // prints "pple"
printf("%s\n", str + 3);  // prints "le"
```

- Using the subscript notation (e.g., `str[3]`) is equivalent to using pointer arithmetic and dereferencing (`(str + 3)`).

Strings in Memory

Understanding how strings are stored is crucial:

- **Char Arrays (`char[]`):**
  - Declared as an array, they reside in stack memory.
  - Their contents can be modified.

```
char str[6];
strcpy(str, "apple");  // Modifiable copy stored on the stack.
```

- **String Literals (`char *`):**
  - Declared as a pointer initialized to a literal, they reside in the read-only data segment.
  - Attempting to modify them (e.g., `myString[0] = 'h';`) leads to undefined behavior (often a segmentation fault).

```
char *myString = "Hello, world!";
// myString[0] = 'h'; // Not allowed.
```

- **Reassignment:**
  - Arrays cannot be reassigned to point to new memory (e.g., `str = anotherStr;` is illegal when `str` is declared as an array).
  - Pointers can be reassigned:

```
char *pStr = "apple";
pStr = "banana";  // Valid: pStr now points to a different liter
```

```
al.
```

- **Pointer to an Array:**

  When you declare a pointer and assign it to an array, it points to the first element of the array.

```
char arr[6];
strcpy(arr, "apple");
char *p = arr;  // p points to 'a'
```

Exercises: char* vs. char[] Behavior

Consider these scenarios:

1. **Reassigning an Array:**

```
char str[7];
strcpy(str, "Hello1");
// str = str + 1; // Compile error: you cannot reassign an arra
y.
```

2. **Modifying a String Literal:**

```
char *str = "Hello2";
// str[1] = 'u'; // May cause a segmentation fault because strin
g literals are read-only.
```

3. **Using a Pointer to a Modifiable Array:**

```
char arr[7];
strcpy(arr, "Hello3");
char *p = arr;
p = p + 1;      // Allowed: p now points to the second characte
r.
p[1] = 'u';     // Modifies the underlying array.
```

```
printf("%s", p);  // May print "ellu3" (depending on the modific
ation).
```

These examples illustrate the differences in behavior between arrays and pointers, especially regarding reassignment and modification.

Key Takeaways

- **C Strings** are null-terminated arrays of characters. Their functions depend on the presence of the null terminator.
- **String Functions:**
  - Use functions like `strlen`, `strcmp`, `strchr`, etc., to operate on strings.
  - Be mindful that these functions assume proper null termination.
- **Pointers:**
  - Pointers store memory addresses and are used to efficiently pass data to functions.
  - Pointer arithmetic allows traversal and creation of substrings.
- **Memory Segments:**
  - Strings declared as arrays are stored on the stack and are modifiable.
  - String literals (assigned to `char *`) reside in read-only memory and should not be modified.
- **Parameter Passing:**
  - When passing a string to a function, the array decays to a pointer, meaning modifications within the function affect the original data.
- **Arrays vs. Pointers:**
  - Arrays cannot be reassigned, while pointers can be redirected to point elsewhere.
  - Understanding these differences is crucial for avoiding common pitfalls such as segmentation faults and buffer overflows.

This comprehensive overview should serve as a solid foundation for understanding more about strings, pointers, and memory management in C.

7. Arrays and Pointers

Arrays and Pointers – Comprehensive Note

This note covers the key concepts from the lecture on arrays and pointers. It reviews common string library functions, the fundamentals of pointers, how character arrays work in memory, the differences between arrays and pointers for strings, pointer arithmetic, and parameter passing. Detailed examples and exercises illustrate the behavior of strings in memory, modifications via pointers, and the use of double pointers to modify pointer variables.

Recap: Common string.h Functions

- `strlen(str)`

  Returns the number of characters in a C string (up to, but not including, the null terminator).

- `strcmp(str1, str2)` **and** `strncmp(str1, str2, n)`

  Compare two strings lexicographically.

  - Returns 0 if the strings are identical.
  - Returns a negative value if `str1` comes before `str2` in alphabetical order.

    - Returns a positive value if `str1` comes after `str2`.
    - `strncmp` stops comparing after at most `n` characters.
- `strchr(str, ch)` **and** `strrchr(str, ch)`

  Search for a character in a string.

    - `strchr` returns a pointer to the first occurrence of `ch`.
    - `strrchr` returns a pointer to the last occurrence.
    - Returns `NULL` if the character is not found.
- `strstr(haystack, needle)`

  Searches for the first occurrence of the substring `needle` in `haystack` and returns a pointer to its start, or `NULL` if not found.
- `strcpy(dst, src)` **and** `strncpy(dst, src, n)`

  Copy the source string to the destination (including the null terminator).

    - `strncpy` copies at most `n` characters and does not necessarily add a null terminator.
- `strcat(dst, src)` **and** `strncat(dst, src, n)`

  Concatenate `src` onto the end of `dst`.

    - `strncat` appends at most `n` characters and always adds a null terminator.
- `strspn(str, accept)` **and** `strcspn(str, reject)`
    - `strspn` returns the length of the initial segment of `str` containing only characters from `accept`.
    - `strcspn` returns the length of the initial segment of `str` containing no characters from `reject`.

Recap: Pointers

- **Definition:** A pointer is a variable that stores a memory address. In C, pointers allow you to pass around the address of a memory instance rather than copying its contents.
- **Key Points:**

- One pointer (typically 8 bytes on a 64-bit system) can represent any memory location.
- Pointers are essential for dynamic memory allocation on the heap.
- They allow functions to modify data in its original memory location because C does not have true pass-by-reference (only pass-by-value of pointers).

Example:

```
int x = 2;
int *xPtr = &x;    // xPtr holds the address of x
printf("%d", *xPtr);  // Dereferences xPtr to print 2
```

Recap: Character Arrays

- **Storage:** When you declare a character array (e.g., `char str[6];`), contiguous memory is allocated on the stack for all its characters.
- **Modification:** The contents of a character array created this way can be modified because they reside in writable memory.

Example:

```
char str[6];
strcpy(str, "apple");  // Copies "apple" into the array allocated o
n the stack
```

Recap: String Parameters

- **Conversion:** When passing a `char[]` to a function, it is implicitly converted to a `char *`.
- **Usage:** All string functions (like those in string.h) accept `char *` parameters, so you can pass a character array directly.
- **Representation:** Although `char []` and `char *` both represent strings in usage (accessing characters via index, printing, using library functions), under the hood they differ. Arrays refer to a fixed block of memory while pointers can be reassigned.

Recap: Strings in Memory

Important points about string memory:

- If a string is created as a `char[]`, its characters reside in stack memory and can be modified.
- You cannot assign a new value to a `char[]` variable because it refers to a fixed block of memory.
- Passing a `char[]` as a parameter automatically converts it to a `char *`.
- A string created as a `char *` that points to a literal (e.g., `char *p = "Hello";`) resides in a read-only data segment, so its characters should not be modified.
- A `char *` is reassignable; you can change what it points to.
- Adding an offset to a C string pointer gives you a substring that starts that many characters past the beginning.
- Modifications made to a string through a pointer parameter persist outside the function since both the caller and callee refer to the same memory.

Difference Between char[] and char *

- **char[] (Array):**
  - Declared with a fixed size (e.g., `char str[7];`).
  - Memory is allocated on the stack.
  - Cannot be reassigned to point to another location.
- **char * (Pointer):**
  - Can be assigned to point to a string literal (e.g., `char *pStr = "Hello";`).
  - The pointer is reassignable.
  - When pointing to a literal, the string is in a read-only segment and should not be modified.

Example Comparison:

```
char aString[] = "Hello, world!";  // Array, modifiable characters
in stack memory.
char *pString = "Hello, world!"; // Pointer, points to a constant s
tring in data segment.
```

Arrays and Pointers

- You can set a pointer equal to an array; it will point to the first element.

Example:

```
char str[6];
strcpy(str, "apple");
char *ptr = str;  // ptr now points to str[0]
```

- Equivalently, you can write:

```
char *ptr = &str[0];
```

- Avoid writing confusing expressions like:

```
char *ptr = &str;  // Although equivalent in some contexts, this ca
n be misleading.
```

Pointer Arithmetic

- **Concept:** Pointer arithmetic allows you to adjust the pointer by a number of elements.

Example:

```
char *str = "apple";       // Suppose str points to address 0xff0.
char *str2 = str + 1;      // Points to the second character ('p').
char *str3 = str + 3;      // Points to the fourth character ('l').
printf("%s", str);   // Prints "apple"
```

```
printf("%s", str2);  // Prints "pple"
printf("%s", str3);  // Prints "le"
```

- **Bracket Notation:**
  Using `str[index]` is equivalent to `*(str + index)` and accesses the character at that offset.

String Behavior and Modifications

- When a function receives a string parameter (as a `char *`), it gets a copy of the pointer. Both the caller and callee refer to the same memory.
- Changes made to the characters of the string in the function will persist outside the function.

Example:

```
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    printf("%s", str);  // Prints "apply"
}
```

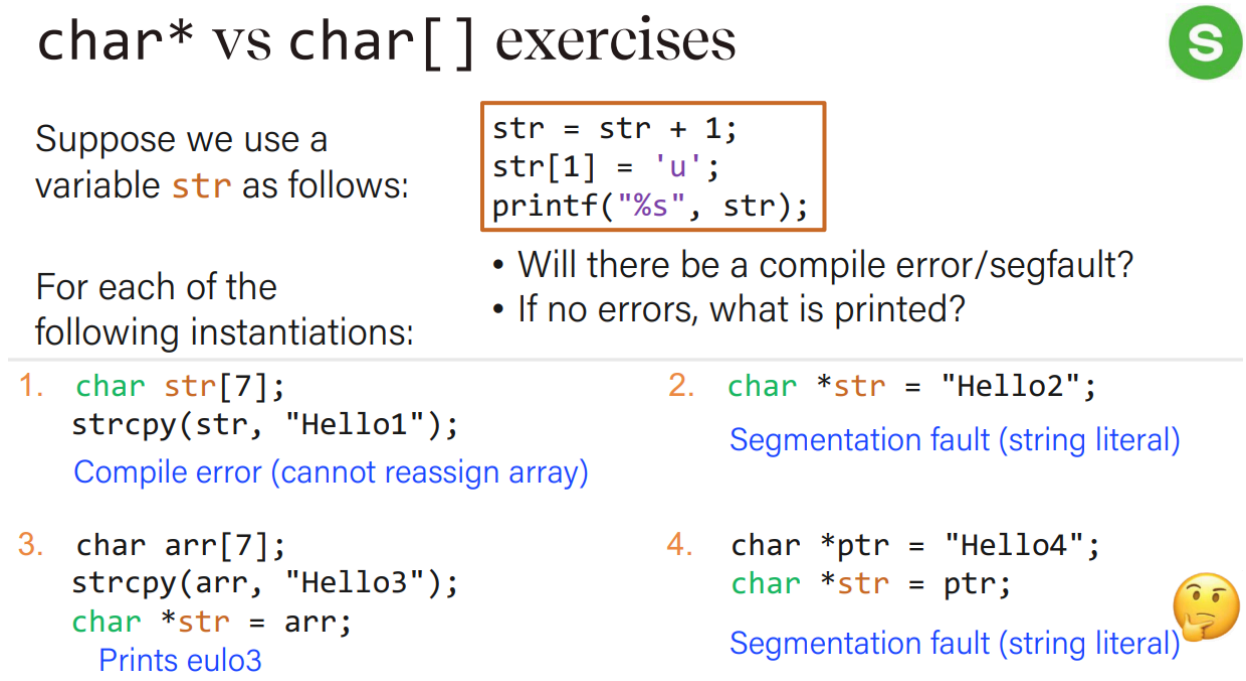
Exercises: char* vs. char[] and Modifiability

Several exercises illustrate the differences between string pointers and arrays, including:

- Attempting to reassign an array (which leads to compile errors).
- Reassigning a pointer to a string literal (which can lead to segmentation faults if modifications are attempted).
- Using a character array to allow modifications and then observing the results when using pointer arithmetic and reassignment.

Common Observations:

- Arrays cannot be reassigned; for example, `str = str + 1;` is illegal if `str` is declared as an array.
- A `char *` pointing to a string literal should not be modified, or it may cause a segmentation fault.
- When a character array is used and then its address is passed to a pointer variable, modifications made via the pointer affect the array.



Difference Between C Arrays and Pointers for Strings

In **C**, strings are represented as **character arrays (`char arr[]`)** or **pointers (`char *ptr`)**. Below is a detailed comparison:

Feature	**Character Array (`char arr[]`)**	**Pointer to String (`char *ptr`)**
Definition	`char str[] = "Hello";`	`char *str = "Hello";`
Memory Allocation	Allocated **on the stack** (modifiable).	String **literal stored in read-only memory** (modification may cause a crash).

Feature	**Character Array (`char arr[]`)**	**Pointer to String (`char *ptr`)**
Modifiable?	✅ Yes, can modify individual characters (`str[0] = 'h'`).	⚠️ **No (if pointing to a string literal).** Modifying `*str` leads to **undefined behavior**.
Storage Location	Stored in **stack or global memory**.	Stored in **read-only memory** (if a string literal) or heap (if `malloc` is used).
Size Determination	Can use `sizeof(str)` to get the total **allocated** size.	`sizeof(str)` gives only **pointer size (4 or 8 bytes)**, not string length.
Reassignment?	❌ No, `arr = "New"` is **invalid** (array name is fixed).	✅ Yes, `ptr = "New"` is **valid** (pointer can be reassigned).
Example of Modification	`str[0] = 'h'; // Works`	`ptr[0] = 'h'; // ❌ Undefined behavior (if pointing to a literal)`
Dynamic Allocation?	❌ No, arrays have a **fixed size**.	✅ Yes, can allocate dynamically using `malloc()`.
Use in Functions	Passes the **entire array reference** (modifications persist).	Passes only the **pointer address** (efficient).
Example Function Call	`void print(char str[])`	`void print(char *str)`

Example Code

1️⃣ Character Array (Modifiable)

```
char str[] = "Hello";
str[0] = 'h';  // ✅ Works
printf("%s", str);  // Output: "hello"
```

2️⃣ Pointer to String (Immutable if pointing to a literal)

```
char *str = "Hello";
str[0] = 'h';  // ❌ Undefined behavior (segfault risk)
```

3️⃣ Pointer with Dynamic Allocation (Safe & Modifiable)

```
char *str = malloc(10);
strcpy(str, "Hello");
str[0] = 'h';  // ✅ Works
printf("%s", str);  // Output: "hello"
free(str);
```

🚀 **Use arrays when the size is fixed, and pointers when flexibility is needed!** 🔥

C Parameters and Passing Values vs. Pointers

- **Passing by Value:**
  When you pass a simple data type (like an int or char) to a function, C passes a copy of that value.

Example:

```
void printSquare(int x) {
    int square = x * x;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // Prints 9
}
```

- **Passing by Pointer:**
  When you want a function to modify the actual instance of a variable, you pass its address.

Example for modifying a variable:

```
void doubleNum(int *x) {
    *x = (*x) * (*x);
}
```

```
int main(int argc, char *argv[]) {
    int num = 2;
    doubleNum(&num);  // Now num becomes 4
    printf("%d", num);
}
```

- **Strings as Parameters:**
  Passing a string (as a `char []`) to a function converts it to a `char *`, meaning that any modifications made to the string inside the function will persist outside.

Exercises: Print Square and Flip Case

Print Square Exercise:

The function should take an integer and print its square.

```
void printSquare(int x) {
    int square = x * x;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // Expected output: 9
}
```

Since the function only performs a calculation without modifying the original value, passing by value is appropriate.

Flip Case Exercise:

To flip the case of a character, use a pointer so that the modification affects the original variable.

```
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);  // Expected output: 'G'
}
```

Here, passing the address of the character allows the function to change its value.

Pointers Summary

- If an operation does not require modifying the input, pass the data type directly.
- To modify a specific instance, pass the pointer (address) to that instance.
- A function that takes an address can access and modify the actual memory content.
- Avoid setting a function parameter to a new value if you intend to modify the caller's instance; such assignments only change the function's local copy of the pointer.

Example Pitfall:

```
void advanceStr(char *str) {
    str += 2;  // This only modifies the local copy of the pointer.
}
```

The above does not change the pointer in the caller's context.

Double Pointers and Modifying Pointer Variables

Sometimes you want to modify the pointer itself (not just the data it points to). This is done using a double pointer.

Skip Spaces Example:
The function `skipSpaces` modifies the string pointer to skip any leading spaces.

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(&str);
    printf("%s", str);  // Expected output: "hello"
    return 0;
}
```

Here, a double pointer is used so that the function can update the caller's pointer directly.

Summary of Key Concepts

- **String Library Functions:** Familiarity with functions like strlen, strcpy, strcmp, etc., is crucial for string manipulation.
- **Pointers:** Pointers store memory addresses, enabling efficient data manipulation, memory allocation, and parameter passing.
- **Character Arrays vs. Pointers:** Arrays allocate fixed, modifiable memory on the stack, while pointers can be reassigned and may point to immutable data.
- **Pointer Arithmetic:** Allows you to navigate through memory by adjusting the pointer based on the size of the data type.
- **Parameter Passing:** Passing by value copies data; passing by pointer (or address) allows functions to modify the original data.
- **Double Pointers:** Essential for modifying pointer variables within functions.

This comprehensive note incorporates all the detailed information from the slides on arrays, pointers, string behavior, and parameter passing. It is designed to be a complete resource on these topics, ensuring that all key concepts and examples are included for effective understanding.

8. The Stack and The Heap

The Stack and The Heap – Comprehensive Note 📚

This note covers the fundamental concepts of memory management in C, focusing on the use of pointers, arrays, the stack, and the heap. It includes detailed explanations, examples, and exercises to illustrate pointer practice, pointer arithmetic, how arrays are stored in memory, and the differences between stack and heap allocation. The note also explains how to properly allocate and free memory on the heap to avoid memory leaks.

Pointers Practice and "* Wars" Stories

Pointers are variables that store memory addresses. They are essential for passing data by reference and for dynamic memory allocation.

- In variable declaration, the asterisk () creates a pointer.
    - Example:
      Here, `ch` stores a character, `cptr` stores the address of a character, and `strptr` stores the address of a pointer to a char.

```
char ch = 'r';
char *cptr = &ch;
char **strptr = &cptr;
```

- When reading from or writing to memory, the dereference operator () accesses the value at the address.
    - Example:

```
char ch = 'r';
ch = ch + 1;         // Increment ch: now 's'
char *cptr = &ch;
*cptr = *cptr + 1;   // Increment value pointed to by cptr:
now 't'
```

- A more advanced example shows modifying the pointer itself using a double pointer:
    - Example:

```
char ch = 'r';
char *cptr = &ch;
char **strptr = &cptr;
*strptr = *strptr + 1;  // This modifies the pointer cptr it
self (its value, i.e., the address)
```

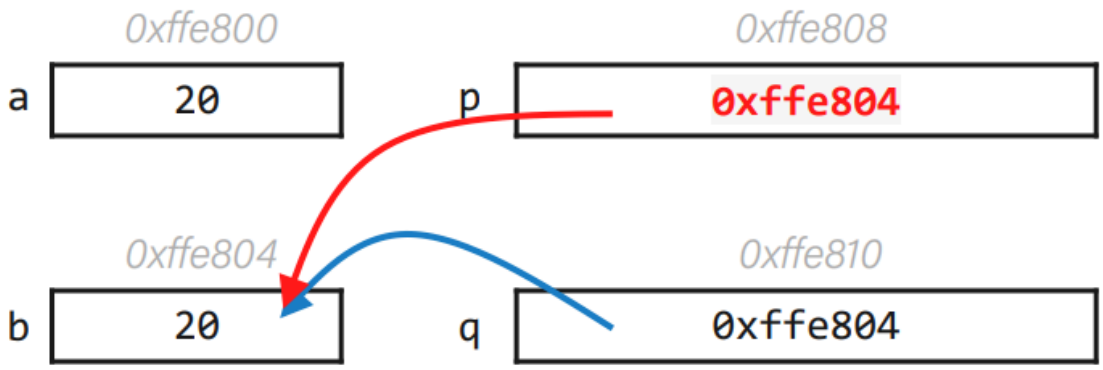
  Diagrams and pen-and-paper exercises (labeled as "* Wars: Episode I/II") help in visualizing how values and addresses change through pointer operations.

- **Pen and Paper Exercise ("A * Wars Story"):**
  Consider the function:

```
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
```

```
    *p = *q;
    p = q;
}
```

Initially, `p` points to `a` and `q` points to `b`. The statement `*p = *q;` makes the value of `a` become 20, and then `p = q;` makes `p` point to `b`. Diagrams are recommended to keep track of addresses (e.g., 0xffe800, 0xffe804) and values.



Arrays in Memory

Arrays in C are blocks of contiguous memory allocated on the stack. Key points include:

- Declaring an array (e.g., `char str[6];`) allocates space for the entire array.
- When using functions like `strcpy`, the contents of the array are copied into this contiguous block.
- The array variable (e.g., `str`) refers to the entire block and is not a pointer itself. For example, `sizeof(str)` returns the total size of the array in bytes.
- You cannot reassign an entire array (e.g., `nums = nums2;` is illegal).

Arrays as Parameters

When an array is passed to a function, C automatically converts it to a pointer to its first element. This means:

- In the function, you lose information about the original array size; `sizeof` on the parameter returns the size of a pointer.
- Both the caller and callee refer to the same memory, so modifications within the function persist outside.

Example:

```
void myFunc(char *myStr) {
    // Operates on the same memory as passed from main
}

int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    myFunc(str);
    // str still holds the modified value, if any changes were made
}
```

Arrays of Pointers

Arrays can also be arrays of pointers. For instance:

```
char *stringArray[5];
```

This declaration reserves space for 5 pointers to char. Each element can point to a string literal or a dynamically allocated string. This is useful for grouping multiple strings (e.g., command-line arguments).

```
void printArgs(char *argv[]) {
    while (*argv) {
        printf("%s\\n", *argv);
        argv++;
    }
}
```

Here, `argv` is effectively `char **argv` — a pointer to (pointers to `char`). It still decays to a pointer to the first element (`argv[0]`), which itself is a pointer to `char`.

Pointer Arithmetic

Pointer arithmetic is based on the size of the type to which the pointer points. Key ideas:

- Adding an integer to a pointer moves the pointer by that number of elements, not bytes.
- Example with characters:

```
char *str = "apple";   // Suppose str is at address 0xff0, point
s to 'a'
char *str1 = str + 1;              // Now points to 0xff1, which
holds 'p'
char *str3 = str + 3;              // Now points to 0xff3, which
holds 'l'

printf("%s", str);   // Prints "apple" starting at address 0xff0
printf("%s", str1);  // Prints "pple" starting at address 0xff1
printf("%s", str3);  // Prints "le"   starting at address 0xff3
```

- For an integer array, pointer arithmetic scales by `sizeof(int)`.

```
int nums[] = {52, 23, 34, 12};
// Suppose the array starts at address 0x1000:
// nums[0] (52) is at address 0x1000,
// nums[1] (23) is at address 0x1004,
// nums[2] (34) is at address 0x1008, and so on.
int *numsPtr = nums;      // Points to nums[0] (52) at address 0
x1000
int *numsPtr1 = nums + 1; // Points to nums[1] (23) at address 0
x1004 (i.e., 0x1000 + sizeof(int))
int diff = numsPtr1 - nums;   // diff equals 1, meaning they are
one element apart
```

Bracket notation (e.g., `str[i]`) is just syntactic sugar for pointer arithmetic: `*(str + i)`.

String Behavior and Modifications

Several important points about strings and pointers:

- **Creating a String as a char Array:**

  When you declare a string as a `char[]`, its memory is allocated on the stack and is modifiable.

- **Creating a String as a char Pointer:**

  When you declare a string as a `char *` pointing to a literal, the string resides in a read-only data segment. Modifying it leads to undefined behavior (often a segmentation fault).

- **Passing Strings to Functions:**

  Whether you pass a `char[]` or a `char *`, a copy of the pointer is passed. Therefore, if you modify the string via the pointer, changes persist outside the function.

- **Adding an Offset:**

  Adding an offset to a pointer gives you a substring. For example, `str + 1` returns a pointer starting from the second character.

Exercises in the lecture illustrate common pitfalls, such as attempting to reassign an array (which causes compile errors) and the difference between modifying a string literal versus a character array.

C Parameters: Pass by Value vs. Pass by Pointer

- **Pass by Value:**

  A function that receives an int or char gets a copy of the value. Changes inside the function do not affect the original variable.

  Example:

```
void printSquare(int x) {
    int square = x * x;
    printf("%d", square);
```

```
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // Prints 9
}
```

- **Pass by Pointer:**

  To modify the original variable, you pass its address. The function can then dereference the pointer and modify the data at that address.

  Example:

```
void doubleNum(int *x) {
    *x = (*x) * (*x);
}

int main(int argc, char *argv[]) {
    int num = 2;
    doubleNum(&num);  // num becomes 4
    printf("%d", num);
}
```

- **Passing Strings:**

  When passing strings (arrays), the function receives a pointer to the first element. Any modifications made are reflected in the caller's memory.

Exercises: char* vs. char[] and Parameter Passing

Several exercises demonstrate the differences:

- Reassigning a `char[]` (an array) is not allowed.
- A `char *` pointing to a literal should not be modified.
- When using pointer arithmetic on strings, remember that a pointer points to the memory address of the first character, and arithmetic operations advance the pointer by the size of the element type.

Flip Case Example:

```
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);  // Prints 'G'
}
```

This exercise demonstrates modifying a specific instance by passing its address.

Double Pointers and Modifying Pointer Variables

Sometimes, you want to modify the pointer itself rather than the data it points to. This is achieved using a double pointer.

Skip Spaces Example:

```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(&str);
    printf("%s", str);  // Prints "hello"
    return 0;
}
```

Here, a double pointer (`char **`) is used so that the function can update the caller's pointer, effectively skipping the initial spaces.

The Stack and the Heap

The Stack

- **Definition:**

  The stack is where local variables and function parameters are stored. Each function call pushes a new frame onto the stack, and when the function returns, its frame is removed.

- **Properties:**
  - The stack grows downward as functions are called and shrinks upward when they return.
  - Local variables in the stack are automatically cleaned up when a function finishes.
  - Recursive function calls consume stack space, and too deep recursion can cause a stack overflow.
  - 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!

- **Example – Function Calls and Recursion:**

  Consider a recursive function for computing factorial:

```
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));  // Computes 24
    return 0;
}
```

Each recursive call has its own stack frame.

- **Memory Diagram:**

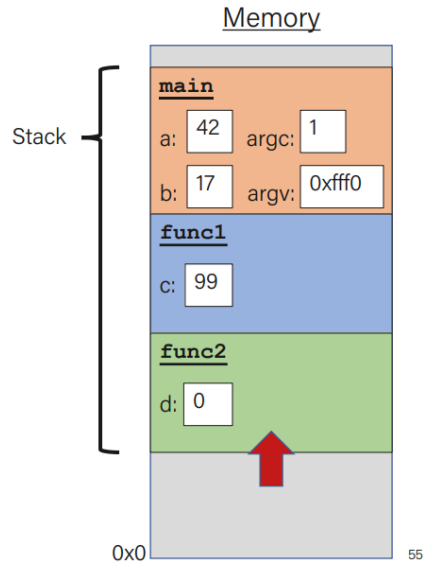
  Diagrams in the lecture show the stack layout for functions like `main`, `func1`, and `func2`, with variables `a`, `b`, `c`, and `d` stored in their respective frames.

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;
}
```

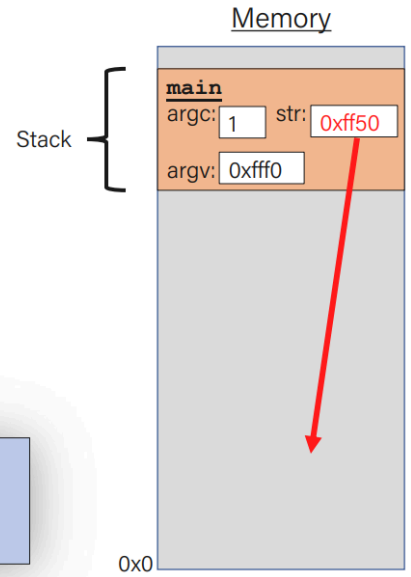


The Stack

```
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 Heap and Dynamic Memory

- **Definition:**

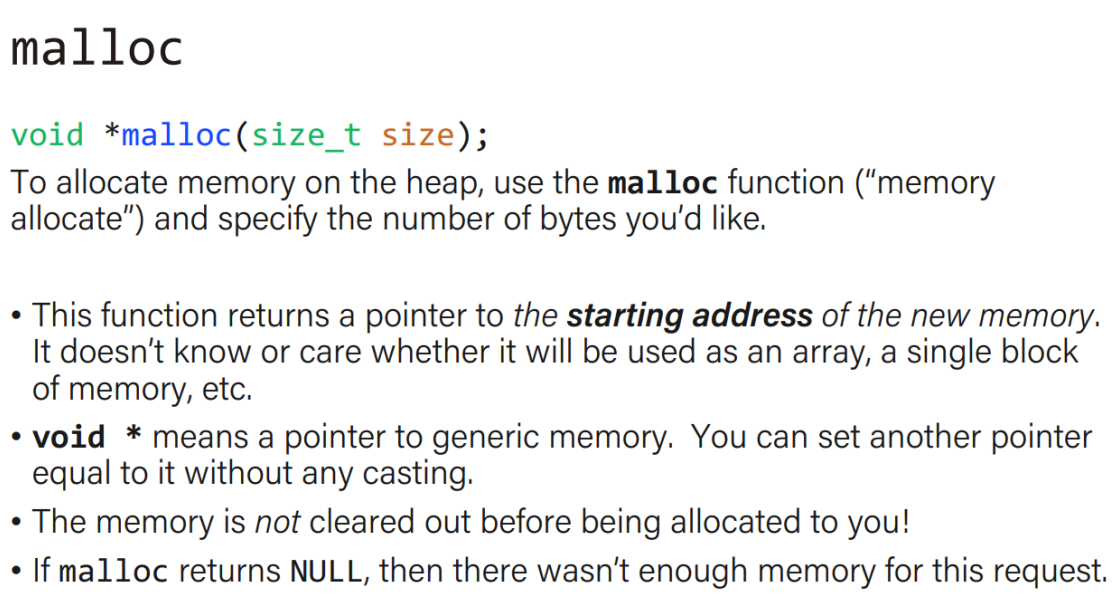
  The heap is a memory region that you manage manually. Memory allocated on the heap remains allocated until you explicitly free it.

- **Dynamic Memory Allocation:**

  - **malloc:** Allocates a specified number of bytes and returns a pointer to the beginning of the block. It does not initialize the memory.

    ```
    char *new_str = malloc(sizeof(char) * (num + 1));
    ```

    Always check if the allocation was successful (e.g., using `assert(new_str != NULL);`).

  - **calloc:** Similar to malloc but initializes the allocated memory to zero.

    ```
    int *scores = calloc(20, sizeof(int));
    ```

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!

- **strdup:** Allocates memory on the heap and duplicates a given string.

```
char *str = strdup("Hello, world!");
```

This function makes it easier to obtain a modifiable copy of a string literal.

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';
```

- **Freeing Memory:**

  Memory allocated on the heap must be freed using the `free()` function to avoid memory leaks. Each allocated block should be freed only once.

```
char *bytes = malloc(4);
// Use the memory...
free(bytes);
```

Freeing memory multiple times or freeing a pointer that was not allocated with malloc/calloc can lead to undefined behavior.

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);
```

- **Memory Leaks:**

  A memory leak occurs when allocated heap memory is not freed. Tools like Valgrind help detect memory leaks.

- **Exercise – Array of Multiples Using malloc:**

  Write a function that returns an array of the first `len` multiples of a given number.

```
int *array_of_multiples(int mult, int len) {
    int *arr = malloc(sizeof(int) * len);
    assert(arr != NULL);
    for (int i = 0; i < len; i++) {
        arr[i] = mult * (i + 1);
    }
```

```
    return arr;
}
```

Cleaning Up and Memory Management

- **Freeing Heap Memory:**

  After you are done using memory allocated on the heap, always free it.

```
char *str = strdup("Hello!");
// Use the string...
free(str);
```

- **Common Pitfalls:**
  - Freeing the same block of memory twice.
  - Freeing only part of an allocated block.
  - Not freeing memory, which leads to memory leaks.
- **Exercise on Freeing Memory:**

  The lecture includes examples where memory allocated inside a loop must be freed within the loop, and then later the overall allocated memory (like a duplicated string) must also be freed.

Where should we free memory below so that all memory is freed properly?

```
1   char *str = strdup("Hello");
2   assert(str != NULL);
3   char *ptr = str + 1;
4   for (int i = 0; i < 5; i++) {
5       int *num = malloc(sizeof(int));
6       assert(num != NULL);
7       *num = i;
8       printf("%s %d\n", ptr, *num);
9       free(num);
10  }
11  printf("%s\n", str);
12  free(str);
```

Summary and Key Takeaways

- **Pointers and Arrays:** Understand the differences between arrays (fixed, stack-allocated) and pointers (reassignable, can point to heap or read-only data).
- **Pointer Arithmetic:** This allows you to navigate through an array by moving the pointer by increments of the data type size.
- **Parameter Passing:** Passing by value versus passing by pointer is crucial for determining whether modifications persist outside a function.
- **Double Pointers:** Use them when you need to modify a pointer variable itself (e.g., skipping spaces in a string).
- **The Stack vs. The Heap:** The stack is for local, temporary storage with automatic cleanup, while the heap is for dynamic memory that you must manage manually.
- **Dynamic Memory Functions:** Use malloc, calloc, and strdup to allocate memory on the heap and always use free to release it.
- **Memory Leaks and Debugging:** Memory leaks can cause long-term issues; tools like Valgrind help ensure your program cleans up after itself.

This note integrates all the key details from the lecture on the stack and the heap, covering pointers, arrays, pointer arithmetic, and dynamic memory management in C. Enjoy studying and happy coding!

9. Realloc, Freed Memory, and Memory Leaks in C

Objective & Scope

This lecture explores dynamic memory management in C. The focus is on using functions such as **malloc**, **calloc**, **realloc**, **strdup**, and **free**, while understanding common memory bugs and the differences between stack and heap memory.

Recap: Arrays Of Pointers

Arrays of pointers allow you to group multiple strings or data items without storing all the data contiguously.

> **ARRAY OF POINTERS:** An array that stores pointers, each pointing to separate memory locations (e.g., strings).
>
> ```
> char *stringArray[5]; // Space for 5 pointers to char
> ```

- **Usage:** Manage collections of strings or dynamically allocated objects.
- **Memory Layout:** Each pointer may reference data in different memory areas, such as the heap or static storage.

Recap: Pointer Arithmetic

Pointer arithmetic in C adjusts addresses based on the size of the data type pointed to—not in raw bytes.

> **POINTER ARITHMETIC:** The process of moving a pointer by an offset multiplied by the size of its data type.
>
> ```
> char *str = "apple";
> char *str1 = str + 1; // Points to 'p'
> ```
>
> *Example with int:*
>
> ```
> int *nums = ...;
> int *nums1 = nums + 1; // Moves by sizeof(int)
> ```

- **Array Indexing:** The syntax `ptr[i]` is equivalent to `(ptr + i)`.
- **Safety Considerations:** Ensure pointer arithmetic stays within the bounds of allocated memory to avoid undefined behavior.

Recap: The Stack

The stack is the memory area where local variables and function parameters are stored. Memory on the stack is automatically managed.

> **STACK:** A region of memory used for local variables and function calls. It grows downward and is deallocated when a function returns.

- **Lifetime:** Local variables exist only during the execution of the function.
- **Common Pitfall:** Returning the address of a local (stack) variable results in undefined behavior.

Recap: The Heap

The heap is used for dynamic memory allocation, allowing data to persist beyond the scope of a single function call until explicitly freed.

> **HEAP:** Memory that is allocated during runtime and remains allocated until it is manually deallocated using functions like free.

- **Usage:** Ideal for large or variable-sized data that must persist after the function exits.
- **Management:** The programmer is responsible for both allocation and deallocation, making proper error checking crucial.

Recap: malloc

The **malloc** function allocates a block of memory on the heap and returns a pointer to it.

> MALLOC:
>
> ```
> void *malloc(size_t size);
> ```
>
> Allocates `size` bytes on the heap and returns a pointer to the allocated memory, or `NULL` if allocation fails.

- **Initialization:** Memory allocated by **malloc** is not automatically zeroed.
- **Error Handling:** Always check that the pointer returned by **malloc** is not `NULL` before using it.

Recap: Always Assert with the Heap

Using assertions after allocation ensures that the program terminates early if memory allocation fails.

> **ASSERT:** A debugging aid that checks a condition (e.g., non-NULL pointer) and aborts the program if the condition is false.

- **Robust Programming:** Helps catch allocation failures immediately.
- **Example:**

```
int *arr = malloc(sizeof(int) * len);
assert(arr != NULL);
```

Other Heap Allocations: calloc

> **CALLOC:** The **calloc** function allocates memory for an array and initializes all bits to zero.
>
> ```
> void *calloc(size_t nmemb, size_t size);
> ```
>
> Allocates memory for `nmemb` elements, each of size `size`, and sets all bytes to zero.

- **Performance:** Generally slower than **malloc** due to initialization.
- **When to Use:** Ideal when you need a clean, zero-initialized memory block.

Other Heap Allocations: strdup

> **STRDUP:** The **strdup** function duplicates a string by allocating enough memory on the heap and copying the content.
>
> ```
> char *strdup(const char *s);
> ```
>
> Returns a pointer to a new, null-terminated string that is a duplicate of `s`.

- **Convenience:** Eliminates manual memory allocation and copying.
- **Memory Management:** Remember to free the duplicated string when it is no longer needed.

Implementing strdup

A custom implementation of **strdup** demonstrates how dynamic memory and string copying work together.

> **Custom strdup Implementation:**
>
> ```
> char *myStrdup(const char *str) {
>     char *heapStr = malloc(strlen(str) + 1);
>     assert(heapStr != NULL);
>     strcpy(heapStr, str);
> ```

```
    return heapStr;
}
```

- **Safety:** Check allocation success with **assert**.
- **Null-Termination:** Ensure that the copied string is properly terminated.

Cleaning Up with free

> **FREE:** The **free** function is used to release memory that was previously allocated on the heap.
>
> ```
> void free(void *ptr);
> ```
>
> Frees the memory block pointed to by `ptr`. Only pointers returned by allocation functions should be freed.

- **Double-Free Error:** Freeing the same memory twice can lead to undefined behavior.
- **Ownership:** Only free the memory you are responsible for and that was allocated dynamically.

Memory Leaks

Memory leaks occur when allocated memory is not properly freed, potentially leading to resource exhaustion.

> **MEMORY LEAK:** A situation in which memory is allocated but not deallocated, eventually exhausting available heap memory.

- **Detection:** Tools like Valgrind can help identify memory leaks.
- **Prevention:** Ensure every allocated block has a corresponding **free** call.

realloc

The **realloc** function resizes an existing memory block. It may extend the current block or allocate a new block and free the old one.

> **REALLOC:**
>
> ```
> void *realloc(void *ptr, size_t size);
> ```
>
> Resizes the memory block pointed to by `ptr` to `size` bytes and returns a pointer to the new memory block.

Further Understanding

- **In-Place vs. Relocation:** If there is enough space, **realloc** expands the block in place; otherwise, it moves the data to a new location.
- **Usage Example:**

```
char *str = strdup("Hello");
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

This analogy helps conceptualize how dynamic memory management functions operate:

- **malloc:** Checking into a hotel room (allocating memory).
- **realloc:** Expanding your room by connecting adjacent rooms or moving to a larger suite.
- **free:** Checking out of the hotel (deallocating memory).
- **Responsibility:** Just as you must check out to avoid charges, you must free allocated memory to avoid leaks.
- **Consequences:** Failure to manage your “room” (memory) properly can lead to errors and wasted resources.

Heap Allocation Interface: A Summary

The key functions for dynamic memory management in C are:

- `malloc(size)`
- `calloc(nmemb, size)`
- `realloc(ptr, size)`
- `strdup(s)`
- `free(ptr)`

Engineering Principles: Stack vs Heap

Understanding the trade-offs between stack and heap memory is essential:

- **Stack:**
  - **Pros:** Fast allocation/deallocation, automatic management.
  - **Cons:** Limited size (typically around 8MB) and less flexible.
- **Heap:**
  - **Pros:** Larger, more flexible, and suitable for dynamic data.
  - **Cons:** Requires manual management, prone to leaks and errors.

Further Understanding

- **When to Use:**
  - Use the stack for local variables and fixed-size data.
  - Use the heap for large, dynamic, or persistent data.
- **Design Considerations:** Choose based on performance, safety, and memory requirements.

Pointers and Working with Dynamic Memory

Dynamic memory management is error-prone if not handled carefully. Common issues include:

- **Use-After-Free:** Accessing memory after it has been deallocated.

- **Double-Free:** Freeing the same memory block twice.
- **Insufficient Allocation:** Allocating too little memory.
- **Incorrect Pointer Arithmetic:** Leading to out-of-bounds access.
- **Returning Local Addresses:** Returning pointers to stack variables.

> **MEMORY BUGS:** Errors in dynamic memory handling that can result in undefined behavior, program crashes, or security vulnerabilities.

- **Debugging:** Use tools like Valgrind to detect memory errors.
- **Best Practices:** Initialize pointers, check allocation results, and clearly document memory ownership.

Exercises and Common Errors

Exercise 1: Improper Pointer Assignment

- **Issue:** A function allocates memory and assigns it to a local pointer, but the caller's pointer remains unchanged.
- **Example:**

```
void myfunc(int *arr) {
    int *p_arr = malloc(2 * sizeof(int));
    p_arr[0] = 42;
    p_arr[1] = 24;
    arr = p_arr; // Does not modify the caller's pointer
}

int main(void) {
    int *arr = NULL;
    myfunc(arr);
    // arr remains NULL, leading to undefined behavior when accesse
d.
    free(arr);
```

```
    return 0;
}
```

- **Lesson:** Use pointers-to-pointers if you need to modify the caller's pointer.

Exercise 2: Incorrect Allocation Size

- **Issue:** Allocating insufficient memory by using the wrong `sizeof` expression.
- **Example:**

```
int myfunc(int **array, int n) {
    int **int_array = malloc(n * sizeof(int)); // Incorrect: should
use sizeof(int*)
    *array = int_array;
    return 0;
}
```

- **Lesson:** Always use the correct type size when allocating memory.

Final Summary & Takeaways

- **Dynamic Memory Management:**
  - Use **malloc**, **calloc**, **realloc**, and **strdup** to allocate memory dynamically.
  - Always free allocated memory to avoid memory leaks.
- **Pointer Arithmetic:**
  - Understand that pointer arithmetic is based on the size of the data type.
- **Memory Bugs:**
  - Common pitfalls include use-after-free, double-free, insufficient allocation, and returning pointers to local variables.
- **Best Practices:**
  - Always check allocation results.
  - Use assertions to catch errors early.

- Choose between stack and heap based on the specific needs of your program.

10. C Generics and Void Pointers

Overview: Generics in C

> **GENERICS:** The practice of writing functions that operate on any data type, thereby reducing code duplication and simplifying maintenance. In C, generics are achieved using void pointers **(void *)** along with functions like **memcpy** and **memmove**.

- **Benefits:**
  - Code reuse: Write one function that works for multiple data types.
  - Easier maintenance: Fix bugs or make improvements in one place.
- **Common Applications:**
  - Sorting and searching arrays of any type.
  - Generic swap functions for data elements.
  - Manipulating user-defined structures.

Generic Swap Function

Traditional Swap Functions

For specific types, swap functions are defined separately:

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

For strings:

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
```

The Challenge for Generics

To write a single swap function for any type, we must:

- Use `void *` pointers since the data type is not known.
- Determine the size (in bytes) of the data to swap.
    - Use a temporary storage buffer and copy the raw bytes.

Implementation Using `memcpy`

> ***GENERIC SWAP (Using void):** A generic swap function takes two pointers and the number of bytes to swap.
>
> **Function Prototype:**
>
> ```
> void swap(void *data1ptr, void *data2ptr, size_t nbytes);
> ```

Step-by-Step Implementation:

1. Allocate temporary storage as an array of `char` of size `nbytes` (since `char` is 1 byte).
2. Copy `nbytes` from the first pointer into the temporary storage using `memcpy`.

3. Copy `nbytes` from the second pointer to the first pointer.
4. Copy `nbytes` from the temporary storage to the second pointer.

Code Example:

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

void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}

int main(void) {
    int x = 2, y = 5;
    swap(&x, &y, sizeof(x));
    printf("After swap: x = %d, y = %d\n", x, y);

    short s1 = 10, s2 = 20;
    swap(&s1, &s2, sizeof(s1));
    printf("After swap: s1 = %d, s2 = %d\n", s1, s2);

    char *str1 = "Hello";
    char *str2 = "World";
    swap(&str1, &str2, sizeof(str1));
    printf("After swap: str1 = %s, str2 = %s\n", str1, str2);

    return 0;
}
```

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 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` points to the location contained in `dest`. (It also returns **dest**).

```c
int main()
{
  //                1,2,1,2,3,4,5,6,7,8
  int source[10] = {1,2,3,4,5,6,7,8,9,10};
  int destination[10];

  memmove(source+2, source, sizeof(int) * 8);

  for (int i = 0; i < 10; i++)
    printf("destination[%d]=%d\n", i, destination[i]);

  for (int i = 0; i < 10; i++)
    printf("source[%d]=%d\n", i, source[i]);
```

Generic Array Swap (Swap Ends)

Problem Statement

Write a function that swaps the first and last elements of an array of any data type.

Challenges

- **Pointer Arithmetic with `void *`:**

  Arithmetic cannot be directly performed on `void *` pointers because C does not know the size of the elements.

- **Solution:**

  Cast the `void *` pointer to a `char *` pointer so that arithmetic is done in bytes.

Implementation Strategy

1. Add an additional parameter for the element size.
2. Compute the address of the last element as:

$$\text{last element address} = (char*)arr + (nelems - 1) \times elem\_bytes$$

3. Use the generic swap function to swap the first and last elements.

Code Example:

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

void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}

void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    // Cast arr to char* for byte-wise pointer arithmetic
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}

int main(void) {
    int nums[] = {5, 2, 3, 4, 1};
    size_t nelems = sizeof(nums) / sizeof(nums[0]);
    swap_ends(nums, nelems, sizeof(nums[0]));
    printf("After swap_ends: nums[0] = %d, nums[%zu] = %d\n", nums
[0], nelems - 1, nums[nelems - 1]);

    // Example with strings
    char *strs[] = {"Hi", "Hello", "Howdy"};
    nelems = sizeof(strs) / sizeof(strs[0]);
    swap_ends(strs, nelems, sizeof(strs[0]));
    printf("After swap_ends: strs[0] = %s, strs[%zu] = %s\n", strs
[0], nelems - 1, strs[nelems - 1]);

    return 0;
}
```

Generics Pitfalls

> **VOID POINTER PITFALLS:** Although `void *` allows generic programming, it lacks type safety. C cannot check the type of data pointed to by a `void *`, making errors such as incorrect element size or misinterpreting memory content possible.

Common Pitfalls:

- **Dereferencing Issues:**

  You cannot directly dereference a `void *` because the compiler does not know the size of the data.

- **Pointer Arithmetic:**

  Arithmetic on `void *` is not allowed; casting to a `char *` is necessary.

- **Incorrect Size Parameter:**

  Failing to pass the correct number of bytes can lead to data corruption or memory errors.

- **Memory Overlap:**

  `memcpy` does not support overlapping regions; use `memmove` if overlap is possible.

> **ADVICE:** Always verify that the element size passed to generic functions is accurate, and use explicit casts to ensure correct pointer arithmetic.

```
// /*
//  * COMP201
//  * Lecture R13
//  *§
//  * This program implements a generic swap function that
//  * works for any variable type.  It also shows how you
//  * can call the function incorrectly and what happens in
//  * memory if you do so.
//  */

 #include <stdio.h>
 #include <string.h>

 /* This is a generic swap function that can swap the data pointed
```

```
  * to by the two pointers, of the given size in bytes.
  */
 void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
     char temp[nbytes];
     memcpy(temp, data1ptr, nbytes);
     memcpy(data1ptr, data2ptr, nbytes);
     memcpy(data2ptr, temp, nbytes);
 }

 int main(int argc, char *argv[]) {
     // Example 1
     int x = 0xffffffff;
     int y = 0xeeeeeeee;
     printf("BEFORE: x = 0x%x, y = 0x%x\n", x, y);
     //swap(&x, &y, sizeof(x));
      swap(&x, &y, sizeof(short));  // what happens if we do this?
     printf("AFTER: x = 0x%x, y = 0x%x\n", x, y);

     // Example 2
     char string1[10] = "Hello";
     char string2[10] = "Goodbye";
     printf("BEFORE: string1: %s\n", string1);
     printf("BEFORE: string2: %s\n", string2);
     //swap(string1, string2, sizeof(string1));
     swap(string1, string2, sizeof(int));  // what happens if we do
this?
     printf("AFTER: string1: %s\n", string1);
     printf("AFTER: string2: %s\n", string2);
     return 0;
 }
```

```
Output:
BEFORE: x = 0xffffffff, y = 0xeeeeeeee
AFTER: x = 0xffffeeee, y = 0xeeeeffff
BEFORE: string1: Hello
```

```
BEFORE: string2: Goodbye
AFTER: string1: Goodo
AFTER: string2: Hellbye
```

Final Summary & Takeaways

- **Generics in C** allow the creation of functions that work with any data type using `void *` pointers.
- A **generic swap function** can be implemented using `memcpy` to handle arbitrary data types, provided the element size is known.
- **Generic array operations** (such as swapping the first and last elements) require careful pointer arithmetic using casts to `char *` to work with raw bytes.
- **Pitfalls:**
  - Lack of type safety with `void *`
  - The necessity of accurate element size specification
  - Use of `memcpy` vs. `memmove` in overlapping regions

11. Function Pointers and Generics in C

Recap: Generics in C

> ***VOID POINTER (void):** A generic pointer that can point to any data type. However, since C does not perform type checking on void *, pointer arithmetic and dereferencing require explicit casts.

Key points:

- Use `memcpy` or `memmove` to copy arbitrary data.
- To perform arithmetic, cast a `void *` to a `char *` because `sizeof(char)` is 1 byte.
- Generic functions (like a generic swap) reduce code duplication by handling different data types.

Example: Generic Swap Function

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
```

```
    memcpy(data2ptr, temp, nbytes);
}
```

This function swaps the bytes at two memory locations regardless of the data type, provided the number of bytes is specified.

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
```

Introduction to Function Pointers

> **FUNCTION POINTER:** A variable that holds the address of a function and allows functions to be passed as parameters or assigned to variables.
>
> **General Syntax:**
>
> `[return type] (*[name])([parameter types])`

Function pointers enable writing generic algorithms that can delegate type-specific operations (such as comparisons or printing) to user-provided functions.

Declaring a Function Pointer

For a comparison function that compares two generic elements:

```
bool (*compare_fn)(void *a, void *b);
```

Or, using the more common integer-returning comparison (similar to `strcmp`):

```
int (*compare_fn)(void *a, void *b);
```

Generic Bubble Sort Using Function Pointers

Motivation

Bubble sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are out of order. To make bubble sort generic, the algorithm must:

- Work on an array of any type.
- Rely on a user-supplied comparison function to decide if two elements are in the correct order.

Generic Bubble Sort Prototype

```
void bubble_sort(void *arr, int n, int elem_size_bytes, int (*compa
re_fn)(void *a, void *b));
```

Implementation Outline

1. **Accessing Elements:**

   Calculate the address of the i-th element using:

   ```
   void *p_elem = (char *)arr + i * elem_size_bytes;
   ```

2. **Comparison:**

   Use the passed comparison function to compare adjacent elements.

3. **Swapping:**

   Call the generic `swap` function to exchange elements when needed.

Example Implementation

```
void bubble_sort(void *arr, int n, int elem_size_bytes, int (*compa
re_fn)(void *, void *)) {
    bool swapped;
    do {
        swapped = false;
        for (int i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_b
ytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem) > 0) {  // Com
pare returns >0 if out-of-order
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
                swapped = true;
            }
        }
    } while (swapped);
}
```

Example: Integer Comparison Function

```
int integer_compare(void *a, void *b) {
    int int_a = *(int *)a;
    int int_b = *(int *)b;
    return int_a - int_b;
}
```

Usage:

```
int nums[] = {4, 2, -5, 1, 12, 56};
int count = sizeof(nums) / sizeof(nums[0]);
bubble_sort(nums, count, sizeof(nums[0]), integer_compare);
```

Additional Generic Operations

Generic Array Rotation (Swap Ends)

Swapping the first and last elements of an array generically requires pointer arithmetic with a specified element size.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    // Cast to char* for byte arithmetic
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

Usage Example:

```
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

Generic Printing and Counting Matches

Generic functions can also accept function pointers for printing elements or counting matches in an array. For example:

Count Matches Prototype

```
int count_matches(void *base, int nelems, int elem_size_bytes, bool
(*match_fn)(void *));
```

```
#include <stdbool.h>

// Callback functions to be used in count_matches
bool match_less_than_three(void *ptr) {
    return *(int *)ptr < 3;
}

bool match_nonnegative(void *ptr) {
```

```
    return *(int *)ptr >= 0;
}
```

Example Implementation

```
int count_matches(void *base, int nelems, int elem_size_bytes, bool
(*match_fn)(void *)) {
    int count = 0;
    for (int i = 0; i < nelems; i++) {
        void *elem_ptr = (char *)base + i * elem_size_bytes;
        if (match_fn(elem_ptr)) {
            count++;
        }
    }
    return count;
}
```

This function iterates over a generic array and uses a match function to determine if each element satisfies a condition.

Standard Library Usage and Function Pointers

> **STANDARD LIBRARY FUNCTIONS:** Functions such as `qsort`, `bsearch`, `lfind`, and `lsearch` in the C standard library use function pointers for comparing elements. These functions require the caller to supply a comparison function that follows a specific signature, allowing the functions to work with any data type.

- **Example:**

```
qsort(base, nelems, elem_size_bytes, compare_fn);
```

This demonstrates how function pointers are integral to writing flexible, generic code in C.

- **`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.

- **`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.

Final Summary & Takeaways

- **Generics:**
  - `void *` pointers enable writing code that works with any data type.
  - Operations such as swapping and array manipulation are implemented by treating memory as a sequence of bytes.
- **Function Pointers:**
  - Allow passing functions as parameters to perform type-specific operations (e.g., comparisons).
  - Have a standard syntax and are used in many standard library functions.
- **Generic Bubble Sort:**
  - Illustrates how to combine generic data handling with function pointers to create a reusable sorting algorithm.
- **Generic Utility Functions:**

    - Beyond sorting, generic functions can be written for printing, counting matches, and array rotations.
- **Key Pitfalls:**
    - `void *` lacks type safety; correct element sizes must be provided.
    - Pointer arithmetic with `void *` requires casting to `char *`.
    - Always verify that function pointers match the expected signatures.

12. Structs, const, and Generic Stack

Objective & Scope

This note introduces two fundamental topics in C programming:

- The use of **const** and **structs** to create robust, maintainable code.
- The design and implementation of a **generic stack** data structure that works with any data type.

These topics build on previous lectures covering generics, void pointers, and generic swap functions. Prerequisites include familiarity with basic memory operations (e.g., memcpy) and previous exposure to generic programming techniques in C.

Recap of Generics So Far

> **GENERICS:** The use of `void *` pointers and functions like `memcpy` / `memmove` enables us to write code that operates on data of any type.
>
> **KEY IDEA:** By combining void pointers with function pointers, we can create reusable algorithms (e.g., generic swap and bubble sort) that delegate type-specific operations to user-provided functions.

The const Keyword

Global and Local Constants

> **CONST VARIABLE:** A variable declared with `const` cannot be modified after initialization.
>
> *Example:*
>
> ```
> const double PI = 3.1415;
> const int DAYS_IN_WEEK = 7;
> ```

const with Pointers

> **CONST POINTER TO DATA:** A declaration like `const char *s` means that the characters pointed to by `s` cannot be modified through `s`.
>
> *Example:*
>
> ```
> char buf[6];
> strcpy(buf, "Hello");
> const char *s = buf;
> // s[0] = 'h';  // Error: cannot modify data via s
> ```
>
> However, the pointer itself can be changed:
>
> ```
> s = "World";  // Valid: changing where s points is allowed
> ```

const in Function Parameters

Using `const` in function parameters signals that the function will not modify the data pointed to by the parameter.

Example:

```
int countUppercase(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
```

```
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}
```

Here, `str` is declared as a `const char *` to prevent modification of its content.

Structs in C

Defining and Using Structs

> **STRUCT:** A user-defined data type that groups related variables (members) under one name.
>
> *Example:*
>
> ```
> struct date {
>     int month;
>     int day;
> };
>
> struct date today;
> today.month = 1;
> today.day = 28;
> ```

Typedef and Struct Initialization

Wrapping a struct definition in a `typedef` allows you to create variables without repeatedly writing the keyword `struct`.

> Example:
>
> ```
> typedef struct date {
>     int month;
>     int day;
> ```

```
} date;

date new_years_eve = {12, 31};
```

Passing Structs to Functions

When a struct is passed by value, a copy is made. To modify the original struct, pass a pointer.

> By Value:
>
> ```
> void advance_day(date d) {
>     d.day++;
> }
> ```
>
> *By Reference:*
>
> ```
> void advance_day(date *d) {
>     d->day++; // equivalent to (*d).day++;
> }
>
> int main(void) {
>     date my_date = {1, 28};
>     advance_day(&my_date);
>     printf("%d\\n", my_date.day);  // Output: 29
>     return 0;
> }
> ```

Arrays of Structs

Arrays of structs are declared like any other arrays. They can be initialized either in full or field-by-field.

> Example:
>
> ```
> typedef struct my_struct {
>     int x;
> ```

```
    char c;
} my_struct;

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Generic Stack Implementation

Motivation and Overview

> **STACK:** A data structure that supports last-in, first-out (LIFO) operations: push, pop, and peek.
>
> **GOAL:** Create a generic stack that can store elements of any type.

From Type-Specific to Generic Stack

Traditional implementations for specific types (e.g., an int stack) are straightforward:

> Example:
>
> ```
> typedef struct int_node {
>     struct int_node *next;
>     int data;
> } int_node;
>
> typedef struct int_stack {
>     int nelems;
>     int_node *top;
> } int_stack;
> ```

For a **generic stack**, we must:

- Use a `void *` pointer in each node to store data of any type.
- Store the element size in the stack structure for correct memory operations.

Generic Stack Data Structures

> **GENERIC STACK STRUCTS:**
>
> *Definition:*

```
typedef struct node {
    struct node *next;
    void *data;
} node;

typedef struct stack {
    int nelems;
    size_t elem_size_bytes;
    node *top;
} stack;
```

Generic Stack Operations

Creating a Stack

> **stack_create:** Allocates a new stack with the specified element size.
>
> *Code Example:*

```
stack *stack_create(size_t elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

Pushing onto the Stack

When pushing, the stack must allocate memory for a copy of the element to ensure the data persists.

> **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++;
}
```

Popping from the Stack

Instead of returning the popped element (which may cause memory management issues), the caller provides a memory location to copy the data.

stack_pop:

```
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        // Handle error (e.g., exit or return error code)
        fprintf(stderr, "Cannot pop from empty stack\\n");
        exit(1);
    }
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
}
```

Example Usage of the Generic Stack

Example: Pushing and popping integers.

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

```
#include <string.h>

// (Assume stack and node struct definitions and functions are defi
ned as above)

int main(void) {
    stack *int_stack = stack_create(sizeof(int));
    int value;

    int x = 7;
    stack_push(int_stack, &x);

    x = 42;
    stack_push(int_stack, &x);

    // Pop the top element into 'value'
    stack_pop(int_stack, &value);
    printf("Popped: %d\\n", value);

    // Clean up remaining elements...
    while (int_stack->nelems > 0) {
        stack_pop(int_stack, &value);
        printf("Popped: %d\\n", value);
    }

    free(int_stack);
    return 0;
}
```

Final Summary & Takeaways

- **const Keyword:**
    - Used to declare variables and pointers that should not be modified.
    - Essential for defining immutable data and ensuring safe function contracts.

- **Structs in C:**
  - Allow creation of custom data types grouping related variables.
  - `typedef` can simplify struct usage.
  - Passing structs by pointer enables modification of the original data.
- **Generic Stack Implementation:**
  - Generic stacks leverage `void *` to store any data type and require element size to manage memory.
  - Key operations (push, pop, create) must carefully manage dynamic memory to avoid leaks.
  - A generic stack improves code reusability and forms the basis for other generic data structures.
- **Generics and Function Pointers:**
  - Function pointers and generic programming techniques are foundational for writing flexible and reusable C code.

13. Compiling C Programs

Objective & Scope

This note details the processes involved in compiling C programs using GCC, along with an introduction to Make and Makefiles. The lecture covers material on GCC's internal pipeline—preprocessor, compiler, assembler, and linker—as well as how Make automates building projects.

GNU

GNU: "GNU's Not Unix"

- GNU is a Unix-like operating system. That means it is a collection of many programs: applications, libraries, developer tools, even games. The development of GNU, started in Jan 1984, is known as the GNU Project. Many of the programs in GNU are released under the auspices of the GNU Project; those we call GNU packages.
- The program in a Unix-like system that allocates machine resources and talks to the hardware is called the "kernel". GNU is typically used with a kernel called Linux. This combination is the GNU/Linux operating system. GNU/Linux is used by millions, though many call it "Linux" by mistake.
- GNU's own kernel, The Hurd, was started in 1990 (before Linux was started). Volunteers continue developing the Hurd because it is an interesting technical project.

– *taken from www.gnu.org*

42

The GCC (Gnu Compiler Collection) Compilation Process

Below is the 4 stages of compilation:

- **Preprocessing:**

  Handles directives such as file inclusion, macro expansion, and conditional compilation. It produces a modified source file ready for actual compilation.

- **Compiling:**

  Converts the preprocessed code into assembly language by parsing and optimizing the code.

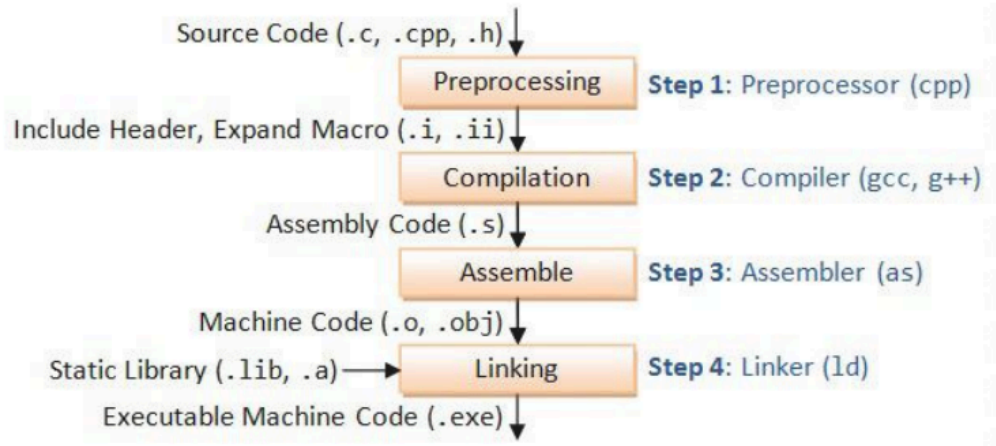
- **Assembling:**

  Translates the assembly code into object (machine) code, producing intermediate binary files, which are the binary formats that the computer can directly execute.

- **Linking:**

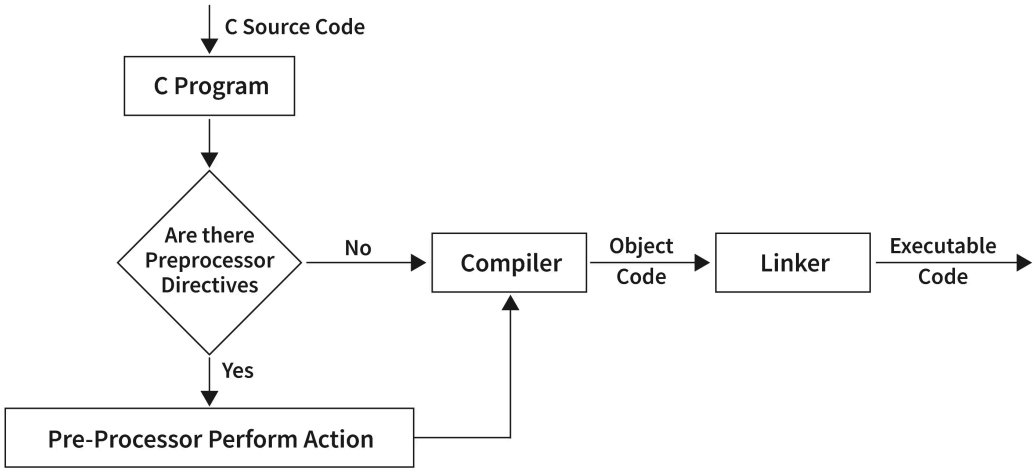
  The linker takes the intermediate binary file (or multiple object files, if your program is split over several source files) along with any libraries your code depends on. Then it combines object files and libraries to produce the final executable program

The GNU Compiler Collection (GCC)



The Preprocessor

> **PREPROCESSOR:** Handles directives such as `#define` and `#include`. It performs macro substitution and file inclusion, effectively "pasting" the contents of header files into the source.
>
> - **Object Macros:**
>
> ```
> #define BUFFER_SIZE 1024
> foo = (char *) malloc(BUFFER_SIZE);
> ```
>
> - **Function Macros:**
>
> ```
> #define min(X,Y) ((X) < (Y) ? (X) : (Y))
> #define twice(X) (2*(X))
> ```
>
> - **Importing Files:**
>
> The `#include` directive includes the content of header files.



SCALER Topics

When you run the command:

```
gcc -E -o hello.i hello.c
```

1. `E` **Option**
   - Tells GCC to run **only the C preprocessor** phase.
   - This expands macros and includes, but stops before compilation.
2. `o` **Option**
   - Specifies the **output file name**.
   - Here, `hello.i` will contain the **preprocessed C code**.

As a result, you'll find that `hello.i` holds all the code after preprocessing—making it useful for debugging or inspecting macro expansions and `#include` directives.

The Compiler (output: assembly code)

COMPILER: Transforms the preprocessed code into assembly code. Its primary function is parsing the C source code and generating corresponding assembly instructions.

- **Demo Command:**

```
gcc -S hello.i
```

  - **Purpose:**

    The `-S` option tells GCC to compile the input file down to **assembly language**.

  - **What It Does:**

    It converts the preprocessed source (in this case, `hello.i`) into an assembly file (commonly named `hello.s`) and stops before generating object code (before sending to assembler).

The Assembler (output: object code) and ELF

ASSEMBLER: Converts assembly code into machine code, resulting in an object file (e.g., `hello.o`).

- **ELF (Executable and Linkable Format):**

  A cross-platform standard that represents object code and executable files. It includes several sections:

  - **.text:** Executable code.
  - **.data:** Global or static variables with predefined values.
  - **.rodata:** Read-only data.
  - **.bss:** Uninitialized global or static variables.
  - **.comment:** Meta information about the object file.

- **Demo Commands:**

```
as -o hello.o hello.s
readelf -e hello.o
```

1. **Assembling:**

   Command: `as -o hello.o hello.s`

   - **Purpose:** Converts the assembly file (`hello.s`) into an object file (`hello.o`).
   - **Details:**
     - `as` is the GNU assembler, which translates assembly language into machine code.
     - `o hello.o` sets the output file name to `hello.o`.
     - The produced object file is in ELF format and is ready for linking.

2. **Inspecting the Object File:**

   Command: `readelf -e hello.o`

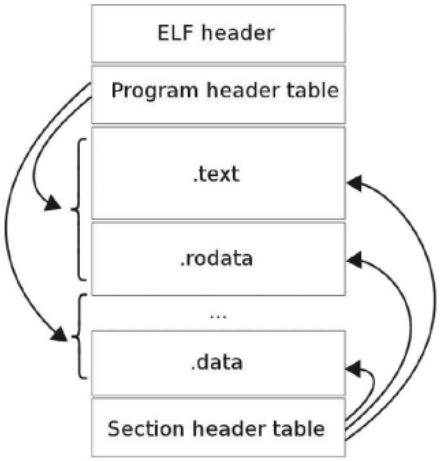
   - **Purpose:** Displays all (and only) the header information in the ELF object file (`hello.o`).
   - **Details:**
     - `readelf` is a tool used to examine the contents of ELF files.
     - The `e` option (or `-all`) outputs all header information, including the ELF header, section headers, and program headers.
     - This command is useful for verifying and debugging the structure of your object file.

Table below outlines several common **sections** in an ELF (Executable and Linkable Format) file produced by assemblers and compilers on Unix-like systems. Each section has a specific purpose in the final binary

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




```
nm hello.o
```

- **Purpose:**

  The command `nm hello.o` is used to display the symbol table of the object file `hello.o`. It shows the symbols defined in and referenced by the file.

- **What It Does:**
  - Lists symbols (such as functions and global variables) with their addresses.
  - Identifies the type of each symbol (e.g., text for code, data for initialized variables, bss for uninitialized variables).
  - Flags undefined symbols that need to be resolved during the linking phase.

Conclusion:

These two tools offer complementary views of an ELF file:

- **readelf:** Displays detailed header information, including the ELF header, section headers, program headers, and other metadata about the file's structure. It helps you understand how the file is organized and how the sections are laid out.
- **nm:** Lists the symbol table, showing you the names and types of symbols (functions, variables, etc.) defined in or referenced by the ELF file. It helps you see how symbols are used within the file.

Together, these tools let you inspect almost all the important metadata and symbol information in an ELF file, though they don't display the raw binary content of each section.

The Linker (output: executable)

When the assembler generates object files (typically in ELF format on Unix-like systems), the linker then takes these ELF object files, along with any libraries, and combines them into a final executable (or shared library), which is also usually in ELF format. The linker resolves symbols and rearranges sections so that the resulting ELF file is properly structured for execution by the operating system's loader.

> **LINKER:** Combines object files into a single executable and resolves references to external functions and libraries.
>
> - **Static Linking:**
>
> The machine code of external functions used in your program is copied into the executable (files usually have a ".a" extension).
>
> - **Dynamic Linking:**

Only an offset table is created in the executable. The operating system loads the machine code needed for external functions during execution (files usually have a ".so" extension).

- **Demo Command:**

```
ld --dynamic-linker /lib64/ld-linux-x86-64.so.2 hello.o -o he
llo -lc --entry main
```

  - `ld`: Invokes the GNU linker.
  - `--dynamic-linker /lib64/ld-linux-x86-64.so.2`: Specifies the dynamic linker (loader) to use at runtime. In this case, it points to the 64-bit Linux dynamic linker.
  - `hello.o`: The input object file generated by the assembler (or compiler).
  - `-o hello`: Sets the output file name to `hello`, which will be the final executable.
  - `-lc`: Links against the standard C library (`libc`), ensuring that standard functions (like those from `printf` or `malloc`) are available in the executable.
  - `--entry main`: Specifies the entry point of the executable. Here, the linker will set the starting function to `main`.
  - **Note:** You may not get this command working, because it will be slightly different on different Linux distributions

After linking, the executable is run (e.g., `./hello`) to demonstrate that all components have been integrated correctly.

Make and Makefiles

Introduction to Make

MAKE: A build automation tool that reads a Makefile—a set of rules that defines how to compile and link a program. *"GNU Make is a tool which controls the generation of executables... from the program's source files."*

- **Purpose:** Automate the build process **by rebuilding only what is necessary**.
- **Advantages:**

- General (usable for more than just C code)
- Fast (only rebuilds modified dependencies)
- Shareable (users compile by simply running `make`)

Structure of a Makefile

MAKEFILE RULE: Each rule contains a target, dependencies, and the commands (recipes) to build the target. **MAKEFILE = List of Rules.**

```
target: dependencies
    command(s)
```

- **Example for a simple C program:**

  ```
  simple: simple.c
      gcc -o simple simple.c
  clean:
      rm -rf simple
  ```

- Usage

  ```
  make simple
  make clean
  ```

Note: Commands must be indented with a tab.

Advanced Makefile Example

REALISTIC MAKEFILE: For a project with multiple source files:

```
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

.PHONY: clean all
```

Variables such as `CC`, `CFLAGS`, and automatic variables like `$@` (target) and `$^` (prerequisites) simplify the build process.

Generic Makefile Template

TEMPLATE: A generic Makefile for small projects:

```
PROGRAMS = hello

CC = gcc
CFLAGS = -g -Wall -O0 -std=gnu99
LDFLAGS = -lm

$(PROGRAMS): %: %.c
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

.PHONY: clean all
all: clean $(PROGRAMS)

clean:
    rm -f $(PROGRAMS) *.o
```

This template can be extended to include libraries or additional targets as needed.

Final Summary & Takeaways

- The GCC compilation process involves distinct phases: **preprocessing**, **compiling**, **assembling**, and **linking**. Each phase transforms the C source code closer to a runnable executable.
- **Make and Makefiles** automate the build process by specifying dependencies and recipes, ensuring that only the necessary components are rebuilt when changes occur.
- Understanding these processes enhances practical skills in compiling, debugging, and organizing larger projects.

14. Introduction to x86-64 Assembly

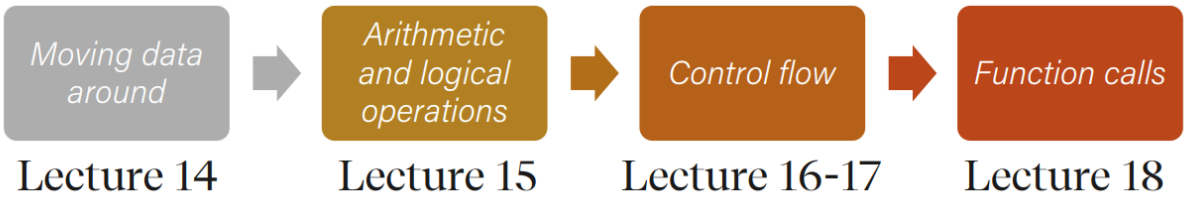
Learning Assembly

> **FOCUS AREAS:**
>
> - Moving data
> - Arithmetic & logical operations
> - Control flow
> - Function calls

Learning Goals

> **LEARNING GOALS:**
>
> - Understand what assembly language is and its importance
> - Recognize x86-64 assembly format
> - Master the `mov` instruction for data movement

Learning Assembly



Plan

1. Overview: GCC & Assembly
2. Demo: Disassembling an executable
3. Registers & assembly-level abstraction
4. The `mov` instruction

Bits All the Way Down

- **Data types:**
  - Integers (unsigned, 2's complement)
  - Floating-point (IEEE single/double)
  - Char (ASCII)
  - Address (unsigned long)
  - Aggregates (arrays, structs)
- **Code:** machine-encoded bits; assembly is human-readable form.

GCC and Assembly

> **GCC:** Compiler that translates C (and other languages) to machine code.
>
> - High-level abstractions are lowered to bits.

> - Assembly is textual representation of machine code.
> - One C statement can map to multiple assembly instructions.

Demo: Looking at an Executable

- Command: `objdump -d <executable>`
- Examine:
  - Function labels & start addresses
  - Instruction bytes (hex)
  - Mnemonics & operands

Our First Assembly: `sum_array`

```
int sum_array(int arr[], int nelems) {
  int sum = 0;
  for (int i = 0; i < nelems; i++) {
    sum += arr[i];
  }
  return sum;
}
```

Disassembly:

```
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
```

Explanation

- **Label & Address:** `sum_array` at `0x4005b6`
- **Hex Bytes:** Machine code (e.g., `ba 00 00 00 00`). This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths. Executed by the CPU. These machine codes are given as hexadecimal instructions, so that when you convert hexadecimals int binary number (1 and 0s), you will see the real machine code.
- **Mnemonic:** Instruction (e.g., `mov $0x0,%edx`). This is the assembly code: "human-readable" versions of each machine code instruction. Assembler converts those into machine code. Each instruction has an operation name **("opcode")**.
- **Operands:**
  - `$` → "immediate", constant
  - `%` → register, a storage location on the CPU
  - Memory addressing forms (e.g., `(%rdi,%rcx,4)`)

Assembly Abstraction

> **ABSTRACTION:**
>
> - C hides machine details; assembly exposes raw operations.
> - Assembly/machine code is processor-specific, no type checking.

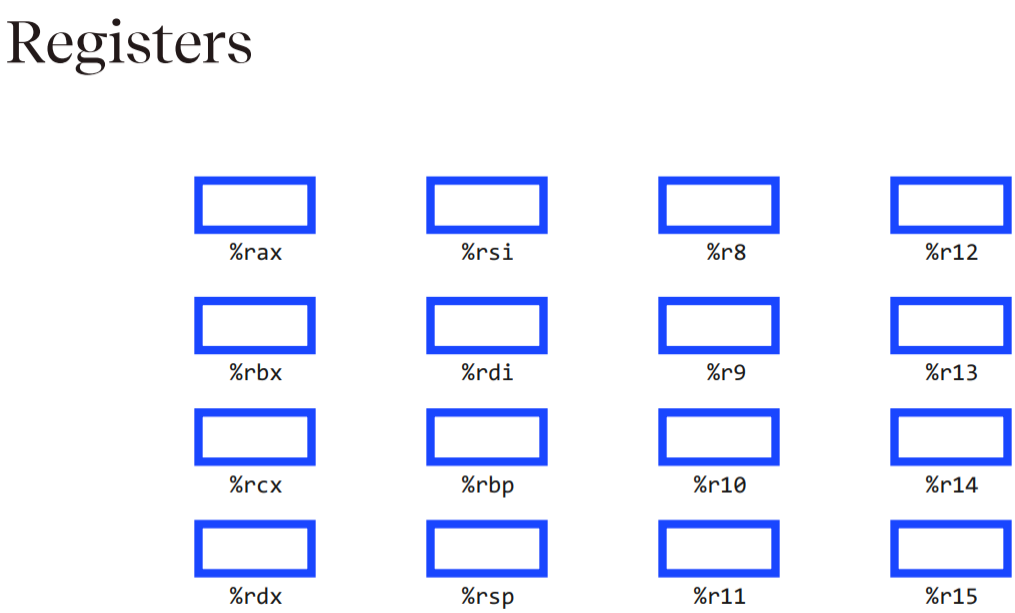
• Here's what the "assembly-level abstraction" of C code might look like:

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*

Registers

> **REGISTER:** 64-bit CPU storage for fast data access, parameters, and returns.

Bottom line: Registers are the CPU's ultra-fast storage "slots" that your assembly instructions use to pass data around, control program flow, and keep track of status. Registers are **not** located in memory! They are fast read/write memory slot right on the **CPU** that can hold variable values.



General-Purpose Registers

There are 16 **general-purpose** registers used in normal integer and pointer code.

- `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`
- `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`

Last 8 general-purpose registers, `%r8` – `%r15`, are added in the 64-bit extension, used for additional arguments, temporaries, etc.

Category	Register Names	Primary Use
General-Purpose	`RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RBP`, `RSP`	Integer arithmetic, passing arguments, stack/frame management
Pointer/Index	`RSI`, `RDI`, `RBP`, `RSP`	Source/destination pointers in memory operations and stack ops

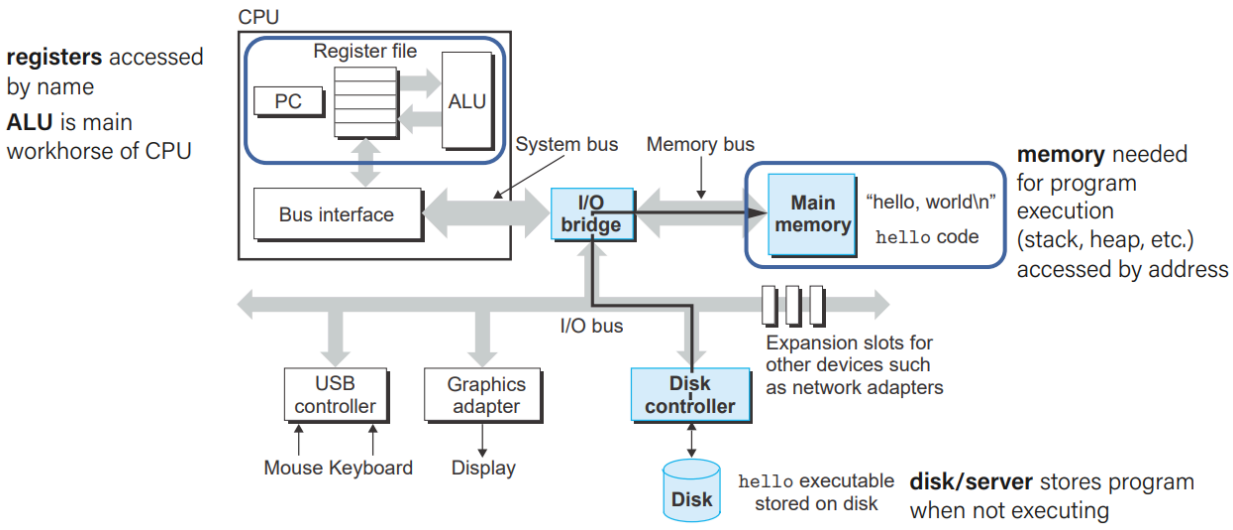
Category	Register Names	Primary Use
Instruction Ptr.	`RIP`	Holds the address of the next instruction to execute
Flags	`RFLAGS`	Status and control flags (zero, carry, overflow, interrupt enable, etc.)
SIMD/Vector	`XMM0` – `XMM15`	Floating-point and 128-bit vector operations (SSE, SSE2, etc.)
Control/System	`CR0` – `CR4`, MSRs	CPU mode, paging, cache control, model-specific registers

- 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 vs. Assembly

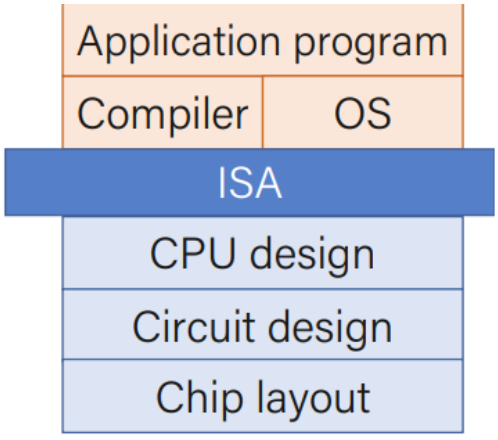
- **Assembly:** Human-readable mnemonics.
- **Machine code:** Hexadecimal bytes.
- Sequential instructions occupy sequential addresses.

Computer architecture



Instruction Set Architecture (ISA)

> **ISA:** A contract between program/compiler and hardware, defining CPU operations, data formats, and control.
>
> - Defines operations that the processor (CPU) can execute
> - Data read/write/transfer operations
> - Control mechanisms
> - x86-64 evolves from Intel's 1978 design, retaining legacy support.
> - Dictates register names and sizes.



The `mov` Instruction

> **mov src,dst:** Copy data from src to dst.

Operand Types

- **Immediate:** `$Imm`
- **Register:** `%reg`
- **Memory:** `Imm(base,index,scale)`

Operand Forms

Immediate

> **mov $0x104, %rax** — load constant into %rax.

Register

> **mov %rbx, %rax** — copy between registers.

Absolute Address

> **mov 0x104, %rax** — load from memory address.

Indirect

> **mov (%rbx), %rax** — load from address in register.

Base + Displacement

> **mov 0x10(%rax), %rdx** — load from base + offset. RAX + 0x10 to RDX.

Indexed

> **mov (%rax, %rdx), %rcx** — load from base + index. (value in RAX) + (value in RDX)

Indexed + Displacement

> **mov 0x10(%rax, %rdx), %rcx.** (%rcx ← M[RAX + RDX + 0x10]) *M = "the **memory** at address"

Scaled Indexed

> **mov (, %rdx,4), %rax** — load from scale*index (scale*RDX).

Scaled Indexed + Displacement

> **mov 0x4(,%rdx, 4), %rax.**

Base + Scaled Indexed

> **mov (%rax, %rdx, 2), %rcx.**

Full Form

> **Imm(base,index,scale) ≡ address Imm + R[base] + R[index]*scale.**

Most General Operand Form

$$\texttt{Imm}(r_b, r_i, s)$$

is equivalent to...

$$\texttt{Imm} + R[r_b] + R[r_i] * s$$

In the AT&T form

```
disp(base, index, scale)
```

– here:

- `disp` (the displacement) is the constant before the parentheses (e.g. `0x10`).
- `base` is the first register inside the parentheses (e.g. `%rax`).

- `index` is the second register inside (e.g. `%rdx`).
- `scale` (if you have one) is the third element; if you omit it, it defaults to 1.

Example:

```
0x10(%rax, %rdx)
```

- **Displacement** = `0x10`
- **Base** = `%rax`
- **Index** = `%rdx`
- **Scale** = 1 (implicit)

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	$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	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." 57

Goals of Indirect Addressing

> **PURPOSE:** Provide flexible memory references (arrays, pointers) via displacement, base, index, and scale.

Final Summary & Takeaways

> **SUMMARY**: Covered assembly intro: data models, GCC workflow, disassembly, registers, ISA, mov, addressing modes, and practice.
>
> **KEY TAKEAWAYS:**
>
> - Assembly maps high-level constructs to CPU ops
> - Registers are fast, limited storage
> - Addressing modes enable complex memory access
> - Mastery of `mov` and addressing is foundational

15. Arithmetic and Logic Operations

Plan

- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations

Data Sizes

> **DATA SIZE:** Assembly terminology for data units.
>
> - **Byte:** 1 byte
> - **Word:** 2 bytes
> - **Double word:** 4 bytes
> - **Quad word:** 8 bytes

> **SUFFIXES:** Instruction suffix indicates data size.
>
> - **b** — byte (8-bit)

> - **w** — word (16-bit)
> - **l** — double word (long) (32-bit)
> - **q** — quad word (64-bit)

C Type ↔ Suffix ↔ Intel Data Type

C Type	Suffix	Size (bytes)	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

> **REGISTER SUBREGISTERS:** Each 64-bit register has smaller aliases.

General-Purpose Registers

64-bit	32-bit	16-bit	8-bit
`%rax`	`%eax`	`%ax`	`%al`
`%rbx`	`%ebx`	`%bx`	`%bl`
`%rcx`	`%ecx`	`%cx`	`%cl`
`%rdx`	`%edx`	`%dx`	`%dl`
`%rsi`	`%esi`	`%si`	`%sil`
`%rdi`	`%edi`	`%di`	`%dil`
`%rbp`	`%ebp`	`%bp`	`%bpl`
`%rsp`	`%esp`	`%sp`	`%spl`
`%r8`	`%r8d`	`%r8w`	`%r8b`
`%r9`	`%r9d`	`%r9w`	`%r9b`

64-bit	32-bit	16-bit	8-bit
`%r10`	`%r10d`	`%r10w`	`%r10b`
`%r11`	`%r11d`	`%r11w`	`%r11b`
`%r12`	`%r12d`	`%r12w`	`%r12b`
`%r13`	`%r13d`	`%r13w`	`%r13b`
`%r14`	`%r14d`	`%r14w`	`%r14b`
`%r15`	`%r15d`	`%r15w`	`%r15b`

Register Responsibilities

> **COMMON USAGE:**
>
> - `%rax` — return value
> - `%rdi` — first function argument
> - `%rsi` — second function argument
> - `%rdx` — third function argument
> - `%rip` — instruction pointer (address of next instruction)
> - `%rsp` — stack pointer (top of stack)

`mov` Variants

> **MOV SIZES:** mov may be suffixed to specify operand size:
>
> - `movb` — byte
> - `movw` — word
> - `movl` — double word
> - `movq` — quad word

> **NOTE:** movl to a register zero-extends the upper 32 bits.

Practice #1: `mov` and Data Sizes

For each, choose the correct suffix (`b`, `w`, `l`, or `q`):

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)`

Answers:

```
movl %eax, (%rsp)
movw (%rax), %dx
movb $0xff, %bl
movb (%rsp,%rdx,4), %dl
movq (%rdx), %rax
movw %dx, (%rax)
```

`movabsq` Instruction

> **movabsq:** Load a 64-bit immediate into a register.
>
> - `movq` supports only 32-bit immediates as source.
> - Use `movabsq $IMM64, %reg`.

Example:

```
movabsq $0x0011223344556677, %rax
```

Practice #2: `mov` and Upper Bytes

Determine how each modifies the upper bytes of `%rax` (initial `%rax = 0`):

1. `movabs $0x0011223344556677, %rax` → `%rax = 0x0011223344556677`
2. `movb $-1, %al` → `%rax = 0x00112233445566FF`
3. `movw $-1, %ax` → `%rax = 0x001122334455FFFF`

4. `movl $-1, %eax` → `%rax = 0x00000000FFFFFFFF`
5. `movq $-1, %rax` → `%rax = 0xFFFFFFFFFFFFFFFF`

`movz` and `movs`

> **ZERO-EXTEND (movz):** Fills upper bytes with zeros
>
> **SIGN-EXTEND (`movs`):** Fills upper bytes by sign-extending the source's MSB

Zero-Extend Variants

Instruction	Description
`movzbw`	byte → word (zero-extend)
`movzbl`	byte → double word
`movzwl`	word → double word
`movzbq`	byte → quad word
`movzwq`	word → quad word

> **Operation:** R ← ZeroExtend(S)

Sign-Extend Variants

Instruction	Description
`movsbw`	byte → word (sign-extend)
`movsbl`	byte → double word
`movswl`	word → double word
`movsbq`	byte → quad word
`movswq`	word → quad word
`movslq`	double word → quad word
`cltq`	`%eax` → sign-extend in `%rax`

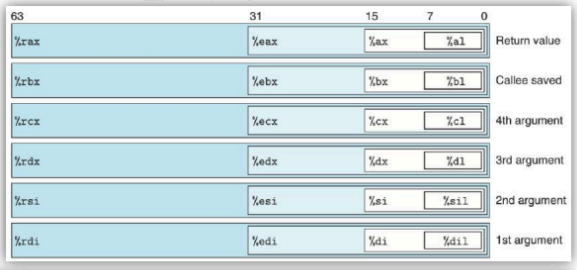
> **Operation:** R ← SignExtend(S)

Code Reference: `full_divide`

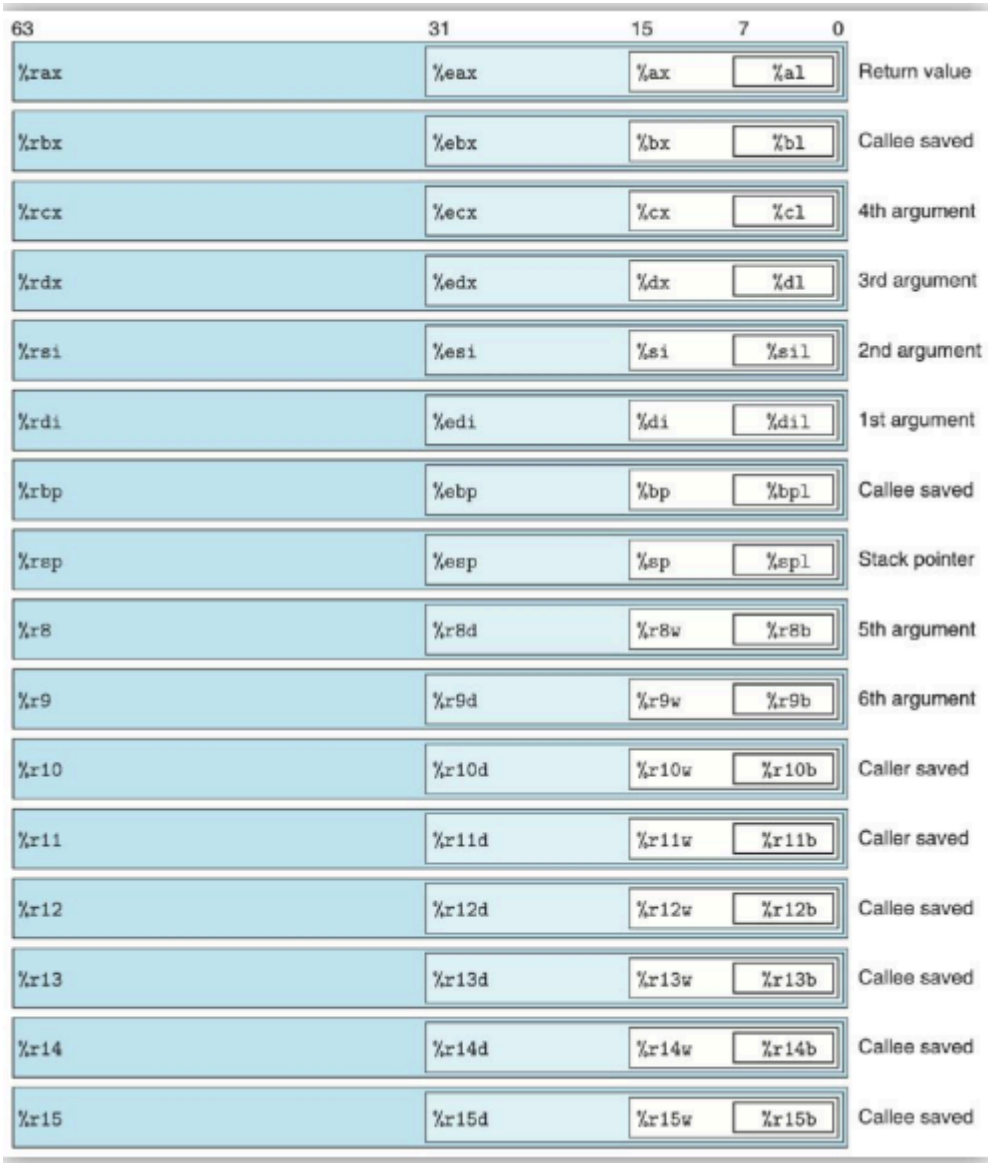
```
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

-------

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



The `lea` Instruction

> **lea src, dst:** Load Effective Address.
>
> - Copies the **address** computed by `src` into `dst`, instead of dereferencing.
> - Same operand forms as `mov`.

Example:

```
lea 6(%rax), %rdx     # %rdx ← 6 + R[%rax]
```

```
mov 6(%rax), %rdx     # %rdx ← M[6 + R[%rax]]
```

`lea` vs. `mov` Examples

Operands	`mov` Interpretation	`lea` Interpretation
`6(%rax), %rdx`	Load M[6 + R[%rax]] into `%rdx`	`%rdx` ← 6 + R[%rax]
`(%rax,%rcx), %rdx`	Load M[R[%rax] + R[%rcx]] into `%rdx`	`%rdx` ← R[%rax] + R[%rcx]
`(%rax,%rcx,4), %rdx`	Load M[R[%rax] + 4·R[%rcx]] into `%rdx`	`%rdx` ← R[%rax] + 4·R[%rcx]
`7(%rax,%rax,8), %rdx`	Load M[7 + R[%rax] + 8·R[%rax]] into `%rdx`	`%rdx` ← 7 + R[%rax] + 8·R[%rax]

A Note About Operand Forms

- Many instructions share the same address operand forms that **`mov`** uses.
  - E.g. `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`

Unary Instructions

> **Unary Instructions:** Operate on 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`	Bitwise NOT

Examples: `incq 16(%rax)`, `dec %rdx`, `not %rcx`.

Binary Instructions

> **Binary Instructions:** Operate on two operands (register/memory, immediate). Destination cannot be memory if source is memory.

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply (trunc.)
`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)`.

Large Multiplication

> **Full 128-bit product:**
>
> - `imulq S` — signed full multiply → `R[%rdx]:R[%rax] ← R[%rax] * S`
> - `mulq S` — unsigned full multiply

> **Two-operand imul S, D:** truncated result in D.

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

To summarize:

- `imul`: Truncates the result to fit in a 64-bit register.
- `mulq`: Produces a 128-bit result, storing it across two registers (`%rdx` and `%rax`).

Division & Remainder

> **Dividend / Divisor = Quotient + Remainder**
>
> - Dividend high 64 bits in `%rdx`, low 64 bits in `%rax`.
> - Divisor → operand.
> - Quotient → `%rax`, Remainder → `%rdx`.

Instruction	Effect
`idivq S`	Signed divide:
	`R[%rax] ← (R[%rdx]:R[%rax]) ÷ S`; `R[%rdx] ← (R[%rdx]:R[%rax]) mod S`
`divq S`	Unsigned divide (same mapping).
`cqto`	Sign-extend `%rax` into `%rdx` for a 128-bit dividend.

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

- Terminology: **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.

To summarize:

- **Division and Remainder Terminology**: The dividend divided by the divisor equals the quotient plus the remainder.
- **Registers for Division**:

- The dividend's high-order 64 bits are stored in `%rdx`, and the low-order 64 bits are in `%rax`.
- The quotient is placed in `%rax`, and the remainder is in `%rdx`.

- **Division Instructions**:
  - `idivq` is for signed division.
  - `divq` is for unsigned division.
  - `cqto` sign-extends the 64-bit value in `%rax` to fill both `%rax` and `%rdx`.
- **x86-64 Limitations**: Most divisions use only 64-bit divisors, with up to 128-bit dividend support using the two registers.

Shift Instructions

> **Syntax:** sal/shl/sar/shr k, D where k = immediate or %cl.

Instruction	Effect	Description
`sal k, D`	`D ← D << k`	Left shift
`sar k, D`	`D ← D >>ᵃ k`	Arithmetic right
`shr k, D`	`D ← D >>ˡ k`	Logical right

> **Shift Amount (%cl):** Only low-order $\log_2$(width) bits of %cl are used.

16. x86-64 Condition Codes & Control Flow

Lecture Plan

- Practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics

Reverse Engineering Practices

Follow along at: https://godbolt.org/z/QQj77g

Reverse Engineering Example

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}
```

```
add_to:
  movslq %edx, %rdx
  movl  %edi, %eax
  addl  (%rsi,%rdx,4), %eax
  ret
```

- `%edi` = `x`; `%rsi` = `arr`; `%edx` = `i`
- `movslq %edx, %rdx` ⟹ sign-extend `i`
- `movl %edi, %eax` ⟹ `sum = x`
- `addl (%rsi,%rdx,4), %eax` ⟹ `sum += arr[i]`
- `ret` ⟹ `return sum`

Learning Assembly

- Moving data around
- Arithmetic & logical operations
- Control flow
- Function calls

Executing Instructions

> **Execution:**
>
> - Instructions & data reside in memory.
> - CPU fetches bytes, decodes, executes.
> - `%rip` holds the address of the next instruction.

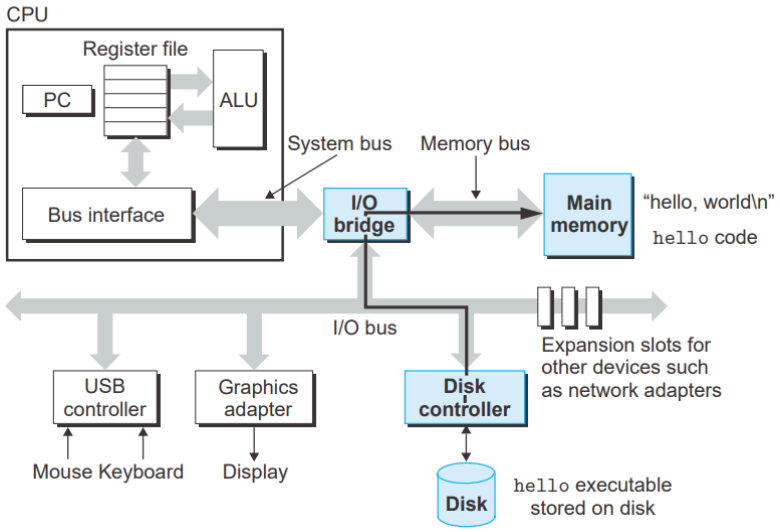
Register Responsibilities

> **Special Registers:**
>
> - `%rax` — return value
> - `%rdi` — 1st argument

- `%rsi` — 2nd argument
- `%rdx` — 3rd argument
- `%rip` — program counter (next instruction address)
- `%rsp` — stack pointer

Instructions Are Just Bytes!

> Machine code is stored as raw bytes; assembly is a mnemonic overlay.



`%rip` – Program Counter

> **%rip:** Program counter holding the address of next instruction.
>
> - Automatically advances by instruction length.
> - Can be changed by jump instructions.

%rip

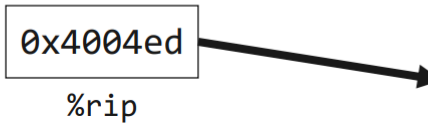
```
00000000004004ed <loop>:
4004ed: 55                    push   %rbp
4004ee: 48 89 e5              mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
4004fc: eb fa                 jmp    4004f8 <loop+0xb>
```

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

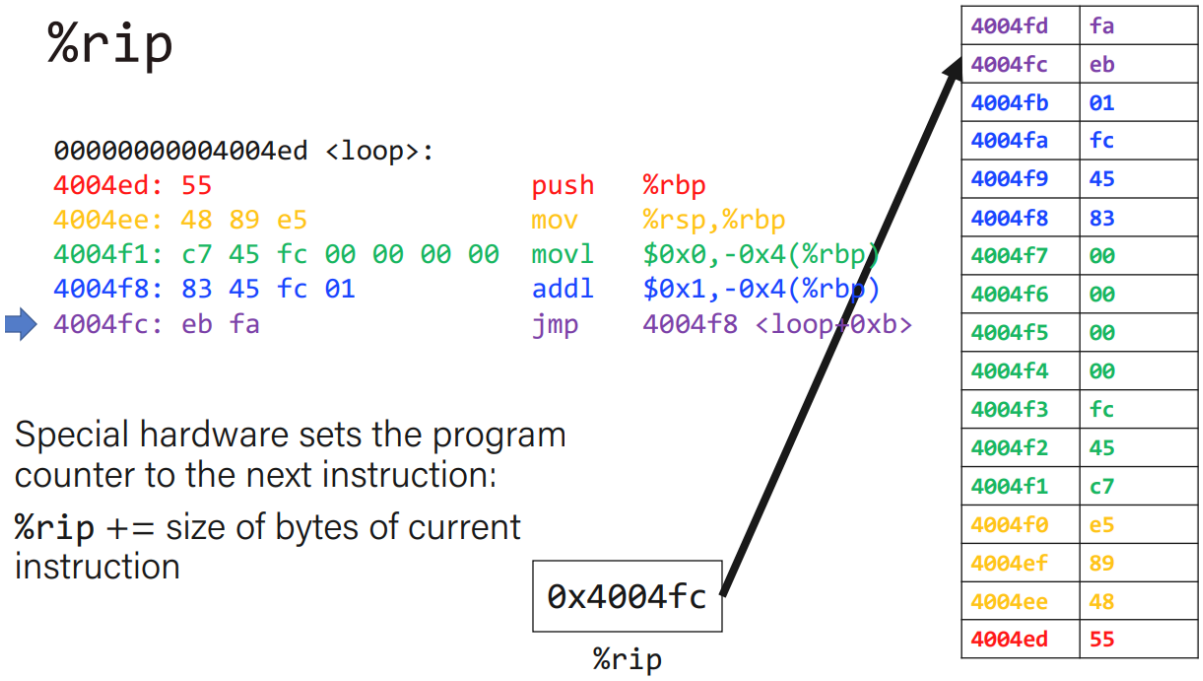
%rip

```
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.



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



42

Going In Circles

> Loops are implemented by "interfering" with %rip via **jump** instructions.

Jump!

> `jmp target` — unconditional jump to target.
>
> `jmp *%rax` — **indirect** jump to address in `%rax`.

`jmp`

- **Direct**: `jmp Label`
- **Indirect**: `jmp *Operand`

"Interfering" with `%rip`

> Unconditional jumps allow repetition or skips, forming loops.

Control

Control Flow: C's if/else/while/for → assembly's **cmp** + conditional jumps.

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	Condition
`je`	`jz`	Equal / zero (ZF=1)
`jne`	`jnz`	Not equal / not zero (ZF=0)
`js`		Negative (SF=1)
`jns`		Nonnegative (SF=0)
`jg`	`jnle`	Signed > (ZF=0 ∧ SF=OF)
`jge`	`jnl`	Signed ≥ (SF=OF)
`jl`	`jnge`	Signed < (SF≠OF)
`jle`	`jng`	Signed ≤ (ZF=1 ∨ SF≠OF)
`ja`	`jnbe`	Unsigned > (CF=0 ∧ ZF=0)
`jae`	`jnb`	Unsigned ≥ (CF=0)
`jb`	`jnae`	Unsigned < (CF=1)
`jbe`	`jna`	Unsigned ≤ (CF=1 ∨ ZF=1)

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]
```

Condition Codes

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction? 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. Alongside normal registers, the CPU also has single-bit condition code registers. They store the results of the most recent arithmetic or logical operation. Here are the most common condition codes:

> **Flags register bits:**
>
> - **CF**: Carry flag (unsigned overflow, The most recent operation generated a carry out of the
>   most significant bit. Used to detect overflow for unsigned operations)
> - **ZF**: Zero flag (result == 0, The most recent operation yielded zero)
> - **SF**: Sign flag (result < 0, The most recent operation yielded a negative value.)
> - **OF**: Overflow flag (signed overflow, The most recent operation caused a two's-complement
>   overflow-either negative or positive)

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
```

> **cmp** S1, S2 → computes S2 − S1, sets flags, discards result.

Instruction	Description
`cmpb`	Compare byte

Instruction	Description
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

`TEST` Instruction

- **Syntax**: `test S1, S2`
- **Operation**: Computes `S1 & S2`, sets flags, and discards the result.
- **Use case**: Often used to check the sign or zero of a value, for example:
  - `test %reg, %reg`

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!

Flags Behavior

- **Arithmetic/Logical Instructions**: Update **CF**, **ZF**, **SF**, and **OF**.
- `lea`: Does not modify any flags.
- **Logical Operations (e.g., `xor`)**: Clear **CF** and **OF (CF = OF = 0)**.
- **Shifts**:

    - Set **CF** to the last bit shifted out.
    - Clear **OF (OF = 0)**.
- `inc` / `dec`: Update **OF** and **ZF**, but leave **CF** unchanged.

Final Recap

> **Topics:**
>
> - Reverse Engineering C→assembly
> - Execution model & `%rip`
> - Control flow via condition codes & jumps
>
> **Next Time:** Conditional branches in depth

17. More Control Flow

Lecture Plan

- **If statements (cont'd.)**
- **Loops**
  - While loops
  - For loops
- **Other Instructions That Depend On Condition Codes**

Loops and Control Flow

Example: `while (i < 100)`

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
400570:    mov    $0x0, %eax       # i = 0
400575:    jmp    40057a           # jump to test
400577:    add    $0x1, %eax       # i++
40057a:    cmp    $0x63, %eax      # compare i to 99
40057d:    jle    400577           # if i ≤ 99, jump back to add
40057f:    repz   retq             # return
```

- `%eax` holds `i`.
- `mov $0x0, %eax` initializes `i` to 0.
- `jmp` unconditionally jumps to the comparison.
- `add $0x1, %eax` increments `i`.
- `cmp $0x63, %eax` computes `i - 99`, setting flags (e.g., SF=1 while `i<99`).
- `jle` ("jump if less or equal") tests ZF or SF≠OF and loops if `i ≤ 99`.
- When `i` becomes 100, the loop exits and the function returns.

Common While Loop Construction

> **Pattern**:
>
> 1. **Init**
> 2. **jmp** to test
> 3. **Body**
> 4. **Test** (`cmp` + conditional jump)
> 5. **Loop back** if condition holds

> **Pseudocode:**
>
> ```
> while (test) {
>   body
> }
> ```

> **Assembly Skeleton:**

```
    init
    jmp test
body:
    <body instructions>
test:
    cmp   <…>
    jl    body
    ret
```

Common For Loop Construction

C Syntax:

```
for (init; test; update) {
  body
}
```

Assembly AS While-Loop:

```
    init
    jmp   test
body:
    <body>
    <update>
test:
    cmp   <…>
    jl    body
    ret
```

For compilation, `for` is lowered to a `while(test){ body; update; }` form.

Back to Our First Assembly (`sum_array`)

```
int sum_array(int arr[], int nelems) {
   int sum = 0;
   for (int i = 0; i < nelems; i++) {
      sum += arr[i];
   }
   return sum;
}
```

```
4005b6:    mov    $0x0, %edx           # i = 0
4005bb:    mov    $0x0, %eax           # sum = 0
4005c0:    jmp    4005cb               # jump to test
4005c2:    movslq %edx, %rcx           # sign-extend i
4005c5:    add    (%rdi,%rcx,4), %eax  # sum += arr[i]
4005c8:    add    $0x1, %edx           # i++
4005cb:    cmp    %esi, %edx           # compare i to nelems
4005cd:    jl     4005c2               # if i < nelems, loop
4005cf:    repz   retq                 # return
```

1. `sum` is in `%eax`.
2. `i` is in `%edx`.
3. The instruction `add (%rdi,%rcx,4), %eax` implements `sum += arr[i]`.
4. `cmp %esi, %edx` tests `i < nelems`, and `jl` jumps when true (signed `<`).

Condition Code–Dependent Instructions

Three instruction classes read CPU flags set by arithmetic/logical ops:

1. **Conditional jumps** (`je`, `jl`, etc.)
2. `set` **instructions** (set a byte register to 0/1)
3. **Conditional moves (new versions of** `mov`**)** (`cmov…`)

`set`: Read Condition Codes

> **Purpose:** Write 1 or 0 into a byte register (e.g., %al) based on flags.
>
> - Destination: single-byte register or memory.
> - Does not alter other bytes of the register—commonly zero-extended after via `movzbl`.

Example:

```
int small(int x) {
    return x < 16;
}
```

```
cmp   $0xf, %edi      # compare x to 15
setl  %al             # %al = 1 if x<16 else 0
movzbl %al, %eax      # zero-extend into %eax
ret
```

Instr	Synonym	Condition
`sete`	`setz`	Equal / zero (ZF=1)
`setne`	`setnz`	Not equal / non-zero (ZF=0)
`sets`		Negative (SF=1)
`setns`		Nonnegative (SF=0)
`setg`	`setnle`	Greater (signed >) (ZF=0 ∧ SF=OF)
`setge`	`setnl`	≥ (signed ≥) (SF=OF)
`setl`	`setnge`	< (signed <) (SF≠OF)
`setle`	`setng`	≤ (signed ≤) (ZF=1 ∨ SF≠OF)
`seta`	`setnbe`	Above (unsigned >) (CF=0 ∧ ZF=0)
`setae`	`setnb`	≥ (unsigned ≥) (CF=0)
`setb`	`setnae`	Below (unsigned <) (CF=1)
`setbe`	`setna`	≤ (unsigned ≤) (CF=1 ∨ ZF=1)

`cmov` : Conditional Move

> **Purpose:** Move src→dst if a condition holds, without branching.
>
> - `dst` must be a register.
> - Often used for C's ternary operator.

Example:

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp    %edi, %esi      # compare x to y
mov    %edi, %eax      # assume x
cmovle %esi, %eax      # if x≤y, move y
ret
```

Instr	Synonym	Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF=1)
`cmovne`	`cmovnz`	Not equal / non-zero (ZF=0)
`cmovs`		Negative (SF=1)
`cmovns`		Nonnegative (SF=0)
`cmovg`	`cmovnle`	> (signed >) (ZF=0 ∧ SF=OF)
`cmovge`	`cmovnl`	≥ (signed ≥) (SF=OF)
`cmovl`	`cmovnge`	< (signed <) (SF≠OF)
`cmovle`	`cmovng`	≤ (signed ≤) (ZF=1 ∨ SF≠OF)
`cmova`	`cmovnbe`	Above (unsigned >) (CF=0 ∧ ZF=0)
`cmovae`	`cmovnb`	≥ (unsigned ≥) (CF=0)
`cmovb`	`cmovnae`	Below (unsigned <) (CF=1)
`cmovbe`	`cmovna`	≤ (unsigned ≤) (CF=1 ∨ ZF=1)

Ternary Operator

> **Syntax:** condition ? (expression If True) : (expression If False)

> **Semantics:** Evaluates one of two expressions based on a test—often lowered to `cmp` + `cmov` in assembly.

Practice: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

```
leal   3(%rdi), %eax    # bias for signed division
testl  %edi, %edi       # set flags based on x
cmovns %edi, %eax       # if x ≥ 0, restore x
sarl   $2, %eax         # arithmetic divide by 4
ret
```

Biasing ensures that `-14/4` rounds toward zero (result `-3`).

Practice: Fill In The Blank

```
long loop(long a, long b) {
    long result = _______;
    while (_________) {
        result = ____________;
        a = _________;
    }
    return result;
}
```

```
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
```

Answers:

- `result = 1;`
- `a < b;`
- `result = result * (a + b);`
- `a = a + 1;`

Recap

- Assembly Execution & `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Conditional Jumps, `set`, `cmov`
- Loops: While & For
- If statements (cont'd.)
- Other instructions depending on flags
- **Next Time:** Function calls in assembly

18. x86-64 Procedures

Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Recursion Example

%rip

> **%rip** is the instruction pointer, holding the address of the next instruction.
>
> - Offsets (`<+n>`) are relative to function start.
> - Unconditional jumps (`jmp`) use a signed byte to adjust `%rip`.
>   - Instructions are variable-length bytes.

- Without jumps, hardware adds the instruction's byte size to `%rip`.

Loop Example:

```
0x400570 <+0>:  mov   $0x0,%eax
0x400575 <+5>:  jmp   0x40057a <loop+10>
0x400577 <+7>:  add   $0x1,%eax
0x40057a <+10>: cmp   $0x63,%eax
0x40057d <+13>: jle   0x400577 <loop+7>
0x40057f <+15>: repz  retq
```

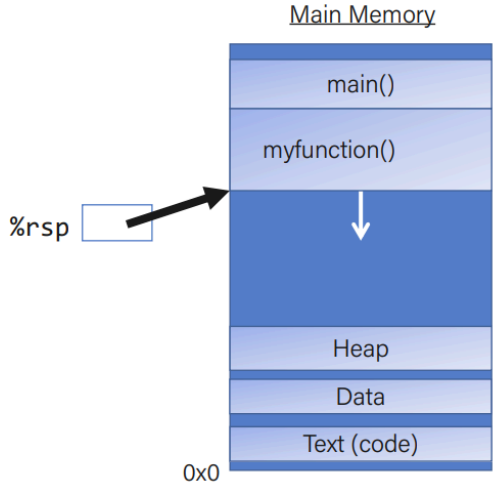
How do we call functions in assembly?

Requirements:

1. **Pass Control:** Transfer `%rip` to callee, then resume.
2. **Pass Data:** Place parameters, retrieve return value.
3. **Manage Memory:** Allocate/deallocate stack space.

%rsp

`%rsp` is the stack pointer, pointing to the top of the stack (stack grows downward).



40

push

> `pushq` S:
>
> - `%rsp -= 8`
> - `M[%rsp] = S`
> - Equivalent to `subq $8,%rsp` + `movq S,(%rsp)`.

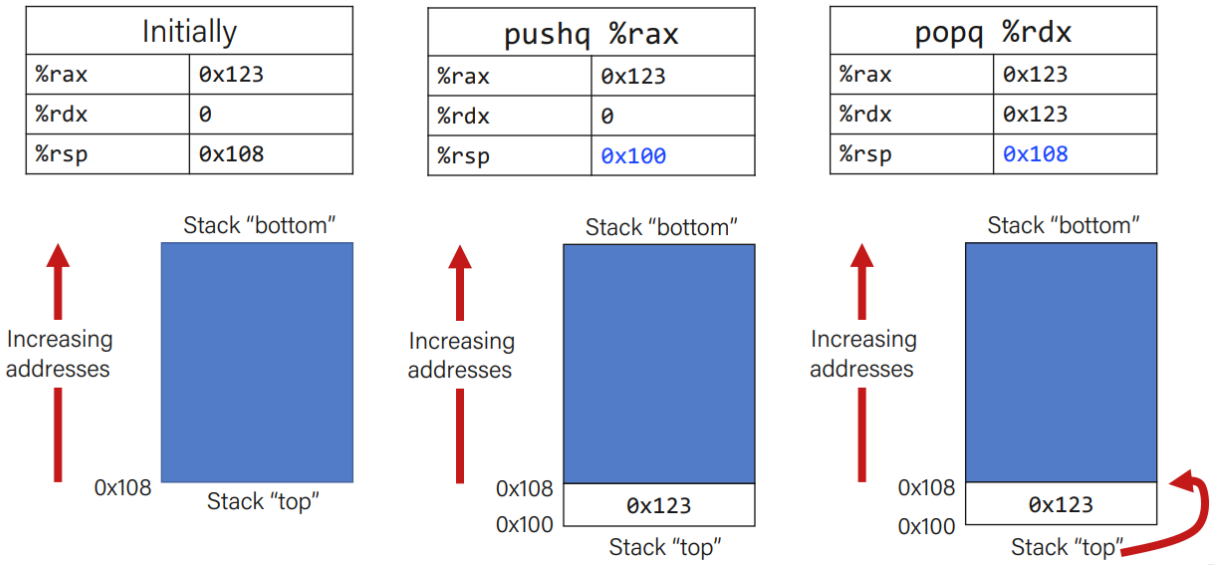
pop

> `popq` D:
>
> - `D = M[%rsp]`
> - `%rsp += 8`
> - Equivalent to `movq (%rsp),D` + `addq $8,%rsp`.

Stack Example

Initial: `%rsp = 0x108`, `%rax = 0x123`

```
pushq %rax        # %rsp → 0x100; [0x100] = 0x123
popq  %rdx        # %rdx = 0x123; %rsp → 0x10
```



Calling Functions In Assembly

> Caller invokes callee, handling `%rip`, parameters, and stack frame.

Remembering Where We Left Off

> `callq` Label pushes return address (next `%rip`) onto stack, jumps to Label. `retq` pops that address into `%rip`, resuming caller.

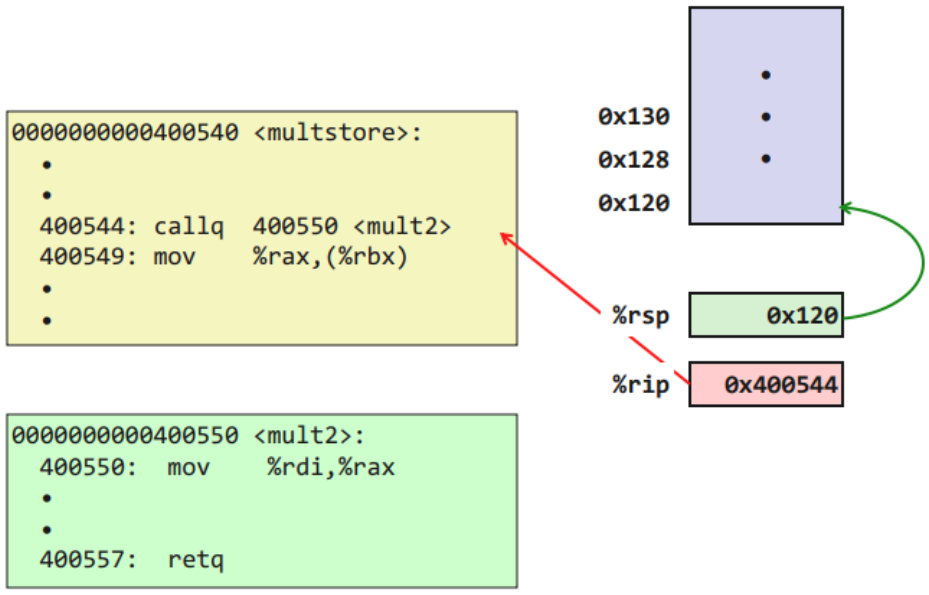
Example: Remembering Where We Left Off

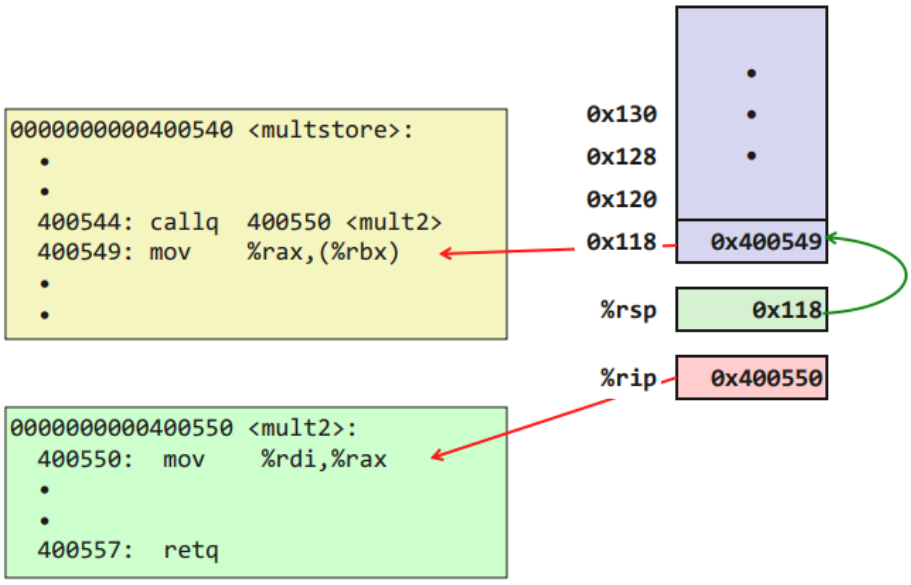
```
void multstore
 (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

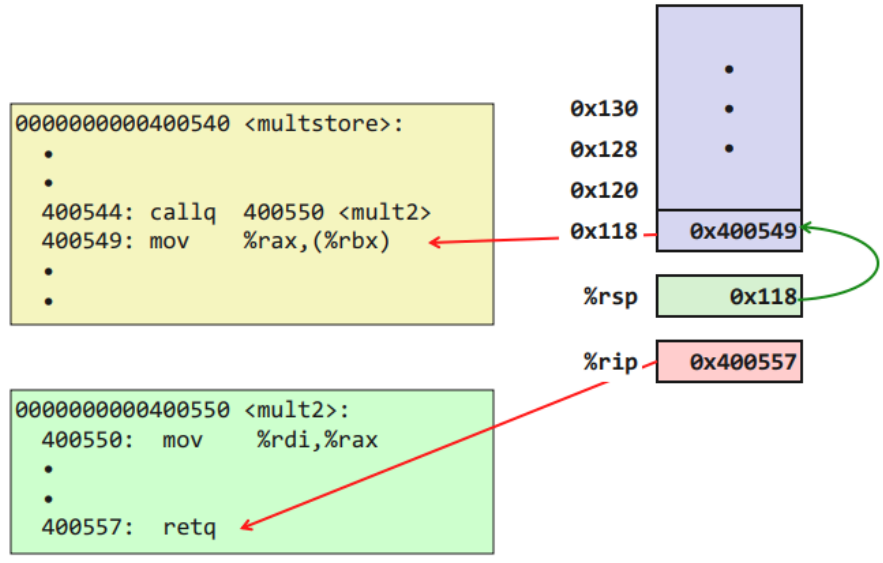
```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: mov    %rdx,%rbx             # Save dest
  400544: callq  400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: retq                         # Return
```

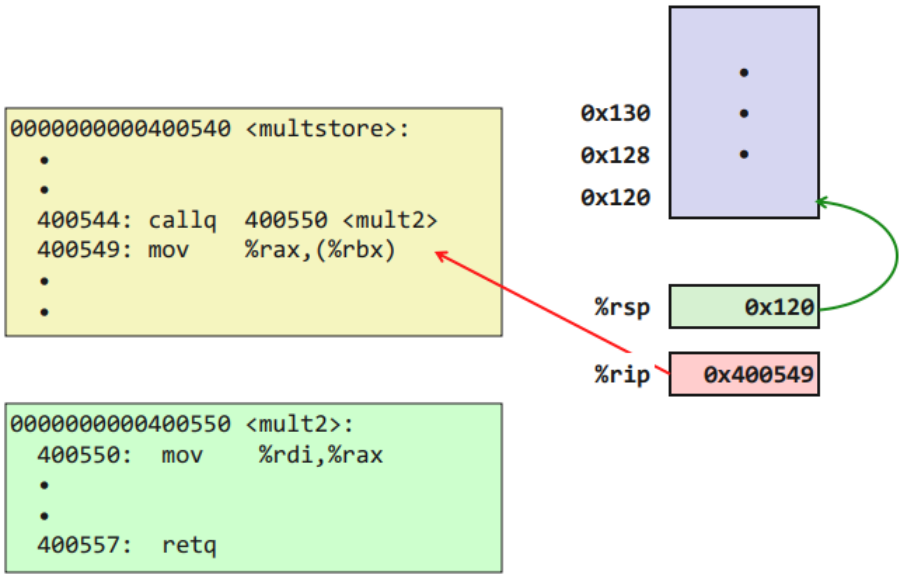
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax            # a
  400553:  imul   %rsi,%rax            # a * b
  400557:  retq                        # Return
```




0000000000400540 <multstore>:
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
0000000000400550 <mult2>:
400550: mov %rdi,%rax
400557: retq
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400550


0000000000400540 <multstore>:
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
0000000000400550 <mult2>:
400550: mov %rdi,%rax
400557: retq
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400557



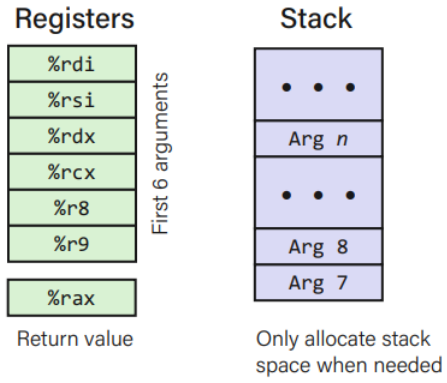
Parameters and Return

Argument Registers:

`%rdi,%rsi,%rdx,%rcx,%r8,%r9` for first six args.

Additional args: pushed onto stack in reverse order.

Return value: in `%rax`.

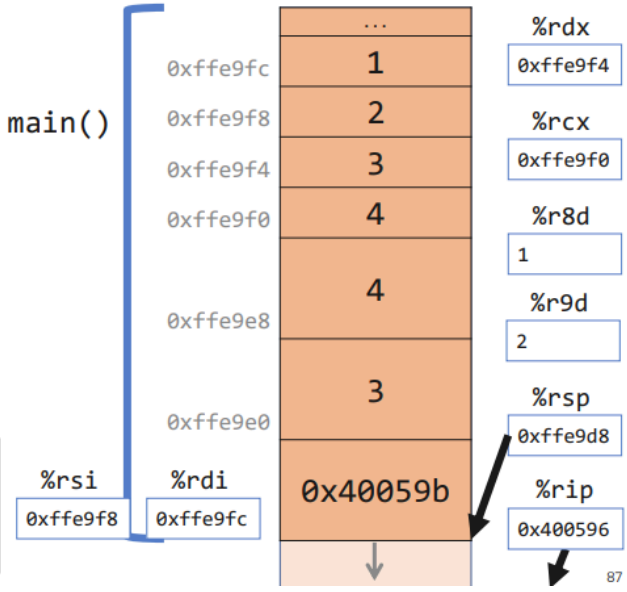


Example 2: Parameters and Return

```
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
…
```



87

Local Storage

> **Locals live on stack if:**
>
> - Out of registers
> - Address-of operator used
> - Arrays/structs

```
// Assume swap_add is defined elsewhere:
long swap_add(long *p1, long *p2);

long caller(void) {
    long arg1 = 534;
    long arg2 = 1057;
    long sum  = swap_add(&arg1, &arg2);
    return sum;
}
```

```
subq $0x10,%rsp
movq $534,   (%rsp)    # arg1
movq $1057, 8(%rsp)    # arg2
leaq 8(%rsp),%rsi      # &arg2
movq %rsp,   %rdi      # &arg1
call swap_add
```

Register Restrictions

> **Caller-saved (volatile):** `%rax, %rcx, %rdx, %rsi, %rdi, %r8–%r11`
>
> **Callee-saved (non-volatile):** `%rbx, %rbp, %r12–%r15`

- **Caller** saves volatile if needed across calls.
- **Callee** saves non-volatile if it uses them.

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.

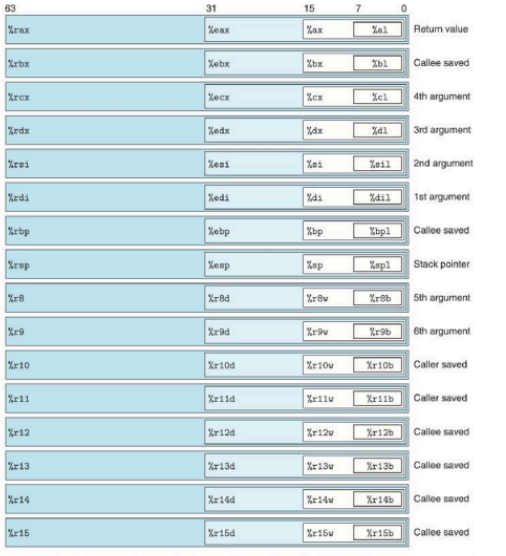


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Caller/Callee

> A function may be both caller and callee in nested calls.

Caller-Owned (Calle Saved) Registers

> Callee must preserve these (push/pop around use).

```
push %rbp
push %rbx
...
pop  %rbx
pop  %rbp
```

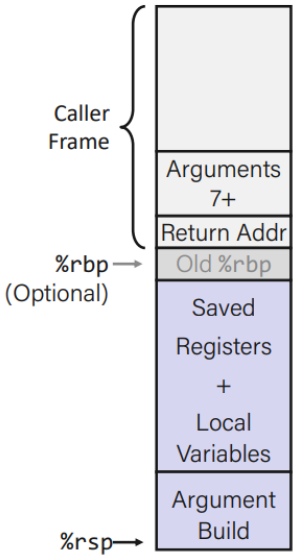
Callee-Owned (Caller Saved) Registers

> Caller must preserve these if it needs their values after a call.

```
push %r10
push %r11
call func
pop  %r11
pop  %r10
```

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



Example of Everything Learned about Assembly

Recap

- `%rip` revisited
- Function calls: stack, control, data, locals
- Register conventions
- Recursion example

19. Data and Stack Frames

Arrays

> **Array:** A contiguous block of memory holding elements of the same type. Access via base address plus offset.

Allocation

- Declaration `T A[L]` reserves `L * sizeof(T)` bytes contiguously.
- Multi-dimensional arrays use **row-major** order:

$$\text{Address}(A[i][j]) = A + \big(i \times C + j\big) \times \text{sizeof}(T).$$

Access

- One-dimensional: `A[i]` at `A + i*K` where `K = sizeof(T)`.
- Two-dimensional:

$$A[i][j] \;\longrightarrow\; A + \big(i \times C + j\big)\, K.$$

- Multi-level (pointer arrays): load pointer then apply offset.

> **Pointer Arithmetic:** In assembly, index scaling uses the addressing mode (%base, %index, scale).

Example: 1D Array Access

```
int get_digit(int *z, int idx) { return z[idx]; }
```

```
# %rdi = z, %rsi = idx
movl   (%rdi,%rsi,4), %eax   # load z[idx]
ret
```

Structures & Alignment

> **Structure:** Memory layout of fields in declaration order, with padding to satisfy each field's alignment.

Layout Rules

- Fields placed in order; compiler inserts **padding** so each field's offset is a multiple of its alignment.
- **Overall size** of the struct is padded to a multiple of the largest field alignment.

Common Types & Alignments (x86-64)

- 1 byte: `char` (align 1)
- 2 bytes: `short` (align 2)
- 4 bytes: `int`, `float` (align 4)
- 8 bytes: pointers, `double` (align 8)

Accessing Members

- Compute member address: `base + offset`.
- For array-in-struct: combine array indexing and struct offset.

> **Example:**

```
struct rec {
  int a[4];
  int i;
  struct rec *next;
};
int *get_ap(struct rec *r, size_t idx) {
  return &r->a[idx];
}
```

```
# %rdi = r, %rsi = idx
leaq   (%rdi,%rsi,4), %rax   # address of r->a[idx]
ret
```

Floating-Point Operations

> **XMM Registers:** 16 registers (%xmm0–%xmm15), each 128 bits, used for SIMD FP.

Calling Convention

- **Arguments:** `%xmm0`, `%xmm1`, ...
- **Return:** `%xmm0`
- **Caller-saved**: all XMM registers.

Scalar & SIMD Instructions

- **Single-precision:**
  - Scalar add: `addss %xmm1, %xmm0`
  - SIMD add (4 floats): `addps %xmm1, %xmm0`
- **Double-precision:**
  - Scalar add: `addsd %xmm1, %xmm0`
  - SIMD add (2 doubles): `addpd %xmm1, %xmm0`

Memory Referencing

- Load/store between memory and XMM: `movss` / `movsd` (scalar), `movaps` / `movapd` (aligned SIMD).

Example: Double Increment

```
double dincr(double *p, double v) {
  double x = *p;
  *p = x + v;
  return x;
}
```

```
# %rdi = p, %xmm0 = v
movapd  %xmm0, %xmm1      # copy v
movsd   (%rdi), %xmm0     # x = *p
addsd   %xmm0, %xmm1      # t = x + v
movsd   %xmm1, (%rdi)     # *p = t
ret
```

- **General-purpose registers (GPRs):**
  - 64 bits wide
  - Used for addresses, loop counters, function arguments (on x86-64), integer arithmetic, etc.
- **XMM registers:**
  - 128 bits wide
  - Designed for data-parallel (SIMD) operations on multiple floats or integers at once

Each SSE instruction has two letters after the mnemonic to tell you:

1. **Packed vs. Scalar**
   - `p` = **Packed** (operate on all lanes in parallel)
   - `s` = **Scalar** (operate on just the lowest lane, leave the rest untouched)
2. **Precision**
   - `s` = **Single-precision** (32-bit floats)
   - `d` = **Double-precision** (64-bit floats)

- `addss %xmm0, %xmm1`
  - **add** scalar single-precision: adds the low-32-bit float in `%xmm0` to the low-32-bit float in `%xmm1` and writes the result back into the low lane of `%xmm1`.
- `addps %xmm0, %xmm1`
  - **add** packed single-precision: adds all four 32-bit floats in `%xmm0` to the four in `%xmm1` element-wise.
- `addsd %xmm0, %xmm1`
  - **add** scalar **d**ouble-precision: same idea, but for just one 64-bit float.
- `addpd` would be **add packed double** (two 64-bit floats in parallel).

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
```

Final Takeaways

- **Arrays:** contiguous, row-major, address = base + index×sizeof(type).
- **Structures:** ordered fields with padding for alignment; total size padded to largest alignment.
- **Floating-Point:** use XMM regs, follow calling convention, choose scalar vs. SIMD instructions appropriately.

20. Security Vulnerabilities

Floating-Point Operations & SIMD

> **XMM Registers:** Sixteen 128-bit registers (%xmm0–%xmm15) used for floating-point and SIMD.

> **SSE vs AVX:**
>
> - **SSE3:** Handles scalar and packed single-precision floats (4 lanes) or doubles (2 lanes).
> - **AVX:** Extends SIMD width and instruction set (not detailed here).

> **Calling Convention:** FP args in %xmm0, %xmm1, ...; result in %xmm0; all XMM are caller-saved.

Scalar & SIMD Instructions

- **Single-precision scalar:** `addss %xmm1, %xmm0`
- **Single-precision SIMD:** `addps %xmm1, %xmm0`
- **Double-precision scalar:** `addsd %xmm1, %xmm0`
- **Packed double:** `addpd %xmm1, %xmm0`

> **Zeroing XMM:** xorpd %xmm0, %xmm0 sets %xmm0 to 0.

Example: Simple FP Routines

```
float  fadd(float  x, float  y) { return x + y; }
double dadd(double x, double y) { return x + y; }
```

```
# %xmm0 = x, %xmm1 = y
addss  %xmm1, %xmm0   # for fadd
ret
addsd  %xmm1, %xmm0   # for dadd
ret
```

Linux Memory Layout

> **Memory Segments:**
>
> - **Text:** executable code (read-only)
> - **Data:** global/static variables, constants
> - **Heap:** dynamic allocations (`malloc`)
> - **Stack:** function frames, grows downward (8 MB limit)

Stack Frame Structure

- **Prologue:**

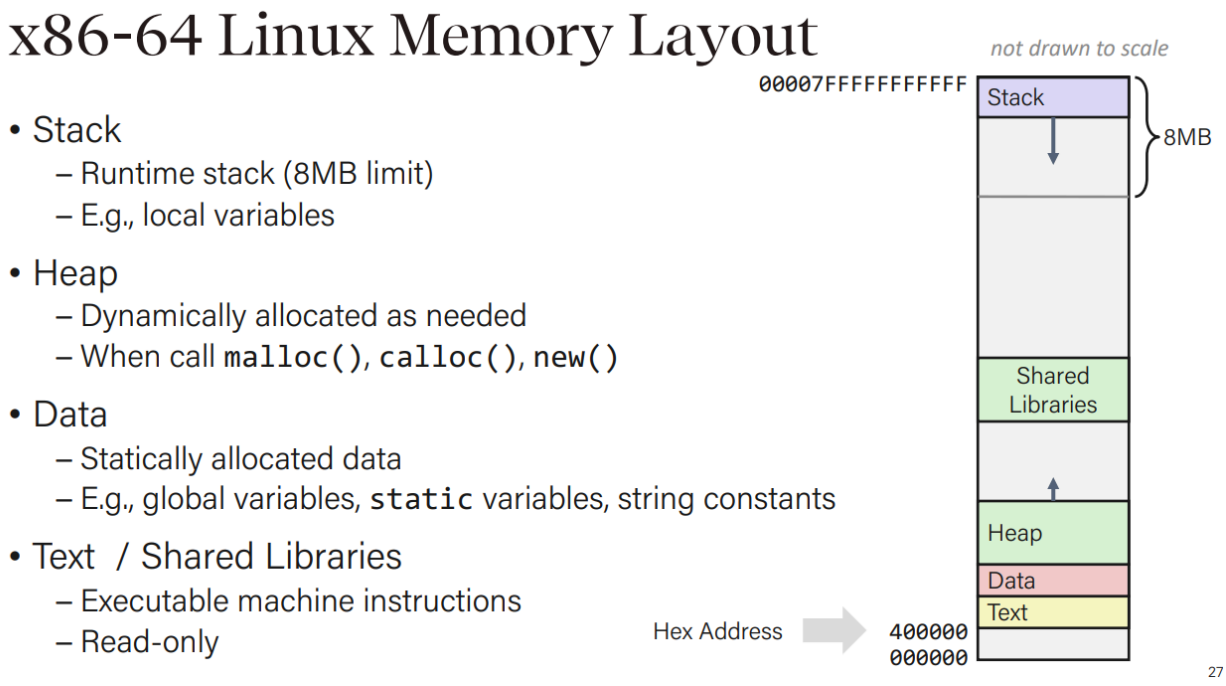
```
push   %rbp
mov    %rsp, %rbp
sub    $N, %rsp      # reserve locals/spills
```

- **Epilogue:**

```
mov    %rbp, %rsp
pop    %rbp
```

```
ret
```

- **Layout:** return address @[`%rbp+8`], saved regs, local buffers @[`%rbp-...`].



Buffer-Overflow Vulnerabilities

> **Buffer Overflow:** Writing beyond an array's bounds, corrupting adjacent stack data (return addresses, canaries).

Out-of-Bounds Struct Write

```
typedef struct {
  int   a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d        = 3.14;
  s.a[i]     = 1073741824;  // no bounds check
```

```
  return s.d;
}
```

- **Behavior:** Writing `s.a[2]` or beyond corrupts `s.d` or stack metadata, altering returned value or crashing.

Classic `gets` Based Overflow

```
void echo() {
  char buf[4];
  gets(buf);     // reads unlimited bytes
  puts(buf);
}
```

- **Attack:** An input longer than 4 bytes overwrites the saved `%rbp` and return address, enabling control-flow hijack.

Exploitation & Mitigations

Code-Injection & ROP

> **Code Injection:** Embedding machine code in input and redirecting execution to it.
>
> **Return-Oriented Programming (ROP):** Chaining short instruction sequences ("gadgets") ending in `ret` to perform complex actions without injecting new code.

Common Protections

- **Safe APIs:**
  - `fgets` instead of `gets`
  - `strncpy` / `snprintf` instead of `strcpy` / `sprintf`
- **Address Space Layout Randomization (ASLR):** Randomizes stack, heap, libraries.
- **Non-Executable Stack (NX bit):** Prevents execution in writable segments.
- **Stack Canaries:**
  - Compiler (`fstack-protector`) inserts a known value before return address

- Verified before function return; crash on mismatch

Final Takeaways

- Floating-point and SIMD require correct use of XMM regs and instructions.
- Understanding memory segments and stack frames is essential to identify overflow risks.
- Buffer overflows remain critical vulnerabilities but can be mitigated by safe coding, hardware and OS defenses, and compiler features.

21. Cache Memories

The Memory Abstraction

Writing & Reading Memory

> **LOAD (Read):** Transfer a word from memory to a register, e.g.

```
movq A(%rsp), %rax   # Read the 8-byte value at address %rsp + A in
to %rax
```

> **STORE (Write):** Transfer a register's value to memory, e.g.

```
movq %rax, A(%rsp)   # Write %rax into memory at address %rsp + A
```

Traditional Bus Structure & Transactions

A bus carries address, data, and control signals between CPU and memory:

1. **Address Phase:** CPU places address A on the bus.
2. **Memory Access:** Memory reads A and drives the data word onto the bus.

3. **Data Phase:** CPU reads the data word into the register.

Storage Technologies & Trends

SRAM vs DRAM

> **SRAM (Static RAM):** Fast (~4 ns), expensive, no refresh needed, used for caches.
>
> **DRAM (Dynamic RAM):** Slower (~60 ns), cheaper, requires periodic refresh, used for main memory.

Enhanced DRAMs

> **SDRAM:** Synchronous control via clock.
>
> **DDR SDRAM (DDR, DDR2, DDR3, DDR4):** Double-data-rate transfers; distinguished by prefetch buffer width.

Nonvolatile Memories

> **Flash Memory:** Electrically erasable, block-level erase, wears out after ~10^5 cycles; used in SSDs.
>
> **3D XPoint & Emerging NVMs:** Higher endurance, persistent storage.

Magnetic Disks

> **Magnetic Disk:** Electromechanical access, nonvolatile, high capacity, slower (seek ≈ 9 ms + rotational latency).

- **Access Time Formula:**

$$T_{\text{access}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

- **Example:**

$$T_{\text{seek}} = 9\ \text{ms},\ T_{\text{rotation}} = \tfrac{1}{2} \times \tfrac{60}{7200} \times 1000 \approx 4\ \text{ms},\ T_{\text{transfer}} \approx 0.02\ \text{ms}.$$

Solid State Disks (SSDs)

> **Page/Block Structure:** Pages (4 KB–512 KB), Blocks (32–128 pages).

> **Erase-before-Write Constraint:** Must erase an entire block before writing.
>
> **Wear Leveling:** Controller distributes writes evenly to extend endurance.

The CPU–Memory Performance Gap

> **Trend:** CPU speed doubles roughly every 18 months, while DRAM latency improves only ~7 % per year, widening the gap.

1. **Diverging Speeds**
   - Over the past few decades, **CPU cycle times** (and effective execution rates) have dropped into the **sub-nanosecond** range.
   - Meanwhile, **DRAM access latencies** sit in the **tens of nanoseconds**, SSDs in the **tens of microseconds**, and spinning disks in the **milliseconds**.
2. **Resulting Bottleneck**
   - A CPU can execute hundreds of instructions in the time it takes just to fetch a single word from main memory.
   - This means **CPUs spend a lot of time stalled**, waiting for data, rather than doing useful work.
3. **Why It's a Problem**
   - Even though raw compute has gotten dramatically faster, applications are often **memory-bound**—their performance is limited by how quickly they can get data, not by how fast they can process it.
   - Without addressing this gap, you can't fully utilize modern CPUs.

Locality of Reference

> **Principle of Locality:** Programs tend to reuse data/instructions near in time (temporal) or address (spatial).

- **Temporal Locality:** Recently accessed items likely reused soon.
- **Spatial Locality:** Nearby addresses likely accessed together.

Example:

```
int sum = 0;
for (i = 0; i < n; i++)
```

```
    sum += a[i];   // stride-1 ⇒ good spatial locality
```

1. Temporal Locality

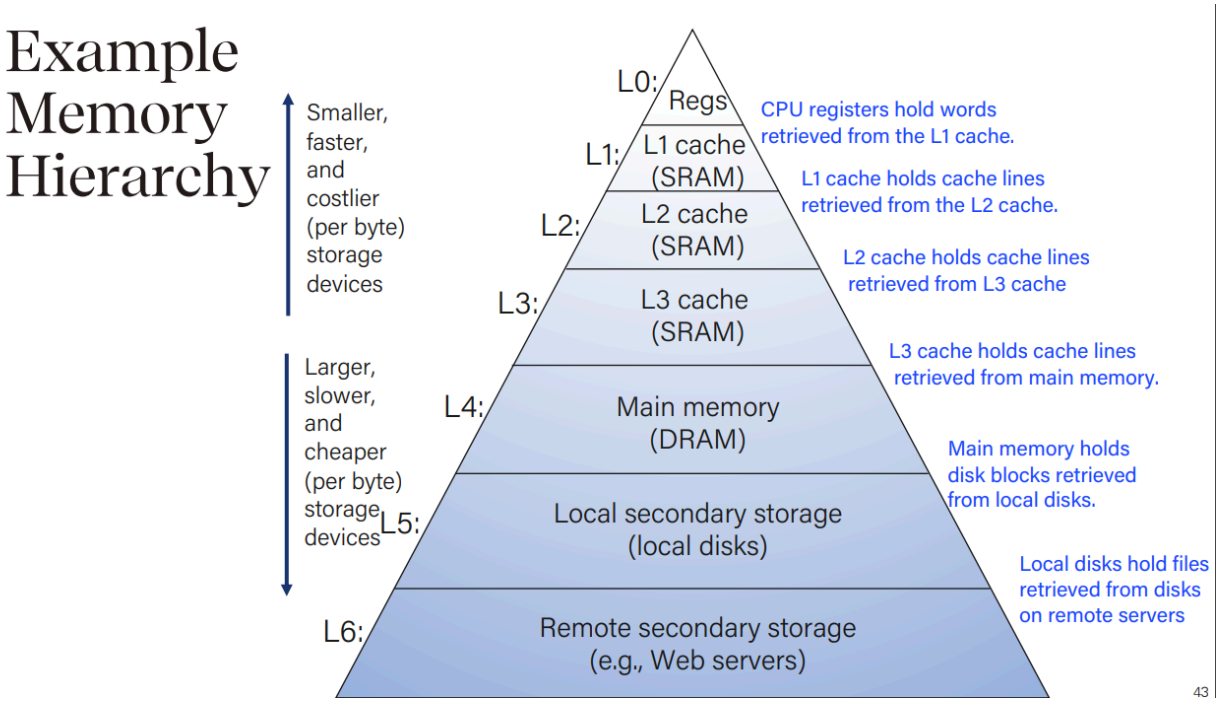
This is the idea that if you use a piece of data (or execute a given instruction) right now, you're very likely to use it again in the near future. Hardware takes advantage of this by keeping recently accessed cache lines in fast on-chip SRAM. On the next access, the CPU can hit the L1 or L2 cache rather than waiting tens of nanoseconds for DRAM.

2. Spatial Locality

This means that if you access an address in memory, you're likely to access nearby addresses soon afterward. Caches exploit this by fetching not just the single word you asked for but an entire "cache line" (e.g. 64 bytes) around it. Prefetchers also detect sequential patterns—like scanning through an array—and start loading future cache lines ahead of time.

The Memory Hierarchy

> **Memory Hierarchy:** Storage levels from fastest/smallest to slowest/largest; each upper level acts as a cache for the next.
>
> **Levels:**
>
> 1. CPU Registers
> 2. L1 Cache (SRAM)
> 3. L2/L3 Cache (SRAM)
> 4. Main Memory (DRAM)
> 5. Local Secondary Storage (SSD/HDD)
> 6. Remote Storage (Network)



Cache Basics

> **Cache:** Small, fast memory holding a subset of blocks from a larger device to provide low-latency access.

- **Hit:** Data found in cache → low latency.
- **Miss:** Data not in cache → fetch from lower level.

> **Miss Types:**
>
> **Cold (Compulsory):** First reference to a block.
>
> **Conflict:** Multiple blocks map to the same cache location.
>
> **Capacity:** Working set exceeds cache size.

Cache Use Cases

- Hardware MMU/TLB for address translation
- Web browser cache (pages)
- OS buffer cache (disk blocks)

Cache Organization

Address Breakdown & Block Size

> **Block Size (B):** Bytes transferred per cache fill (power of 2).

- **Offset (b bits = $\log_2$ B):** Byte index within the block.
- **Index (s bits = $\log_2$ (number of sets)):** Cache set selector.
- **Tag (t bits = m – s – b):** Remaining high-order bits.

Practice Example (6-bit address, B = 4):

Address 0x15 (0b010101) ⇒ offset = 01_2 = 1, block number = 5.

Mapping & Replacement

> **Direct-Mapped:** One location per block.
>
> **Set-Associative:** Blocks map to a set; replacement policy selects victim.
>
> **Fully-Associative:** Any block can go anywhere; high hardware cost.
>
> **Replacement Policies:** LRU, FIFO, Random.

Multi-Level Cache Example

Example (Intel Core i7):

- L1 i-cache/d-cache: 32 KB, 8-way, ~4 cycles
- L2 unified: 256 KB, 8-way, ~10 cycles
- L3 unified: 8 MB, 16-way, ~40–75 cycles
- Block size: 64 bytes

Performance Metrics

> **Miss Rate (MR):** misses ÷ accesses
>
> **Hit Time (HT):** time to access cache + tag check
>
> **Miss Penalty (MP):** extra time on miss

> **Typical Values:** L1 MR = 3–10 %, L2 MR < 1 %, L1 HT ≈ 4 cycles, L2 HT ≈ 10 cycles, MP ≈ 50–200 cycles.

Final Summary & Takeaways

- **Locality** underpins cache effectiveness.
- **Hierarchy** balances speed, capacity, and cost across levels.
- **Cache parameters** (block size, capacity, associativity) and **policies** (placement, replacement) determine performance.
- **Key metrics** (MR, HT, MP) guide design and tuning.
- **Common pitfalls:** Poor data layout (high stride), conflict misses, suboptimal block sizes.

22. More Cache Memories

Cache Organization and Mapping

Block Size (B)

> **BLOCK SIZE:** Number of bytes transferred per cache fill (power of 2, e.g., 64 bytes).

- **Offset bits (b):** $\log_2 B$ = number of low-order bits used to select a byte within a block.

Cache Size (C) and Sets (S)

> **CACHE SIZE (C):** Total capacity in bytes (e.g., 32 KiB).
>
> **NUMBER OF SETS (S):** $C / (B \times E)$, where E is associativity (ways).

- **Index bits (s):** $\log_2 S$ = bits used to select the cache set.
- **Tag bits (t):** Remaining bits $t = m - s - b$ (m = address width).

Replacement Policies

> **REPLACEMENT POLICY:** Determines which block to evict on a miss when a set is full.

- **LRU (Least Recently Used)**
- **FIFO (First-In, First-Out)**
- **Random**

Performance Metrics

> **MISS RATE (MR):** misses / total accesses
>
> **HIT TIME (HT):** time to access cache and perform tag check
>
> **MISS PENALTY (MP):** additional time on a miss to fetch from lower level

- **Typical Values:**
  - L1: MR = 3–10 %, HT ≈ 4 cycles
  - L2: MR < 1 %, HT ≈ 10 cycles
  - MP ≈ 50–200 cycles

Multi-Level Caches

> **MULTI-LEVEL CACHE:** Multiple cache levels (L1, L2, L3) balance hit time vs. miss rate.

- **Example (Intel Core i7):**
  - L1 i-cache/d-cache: 32 KB, 8-way, 4 cycles
  - L2 unified: 256 KB, 8-way, 10 cycles
  - L3 unified: 8 MB, 16-way, 40–75 cycles
  - **Block size:** 64 bytes for all levels.

Write Policies

> **WRITE-THROUGH:** Writes update lower level immediately (consistent but higher latency).
>
> **WRITE-BACK:** Defers write to lower level until eviction; uses **dirty bit** to track modified blocks.
>
> **WRITE-ALLOCATE:** On write miss, fetches block into cache before writing.

> **NO-WRITE-ALLOCATE:** On write miss, writes directly to lower memory without caching.

Write Hit

– **What it means:** The CPU wants to store (write) to an address, and that address's block is **already in the cache.**

– **What happens:**

1. You update the byte(s) right there in the cache.
2. If it's a **write-back** cache, you just mark that cache line **dirty** (meaning "this line has changed").
3. If it's a **write-through** cache, you also send the same update down to main memory immediately.
   – **Why it's fast:** You never had to go fetch anything—you just changed data in the fast on-chip cache.

Write Miss

– **What it means:** The CPU wants to store to an address, but that block isn't in the cache yet.

– **Two main options** (depending on your cache policy):

1. Write-Allocate (a.k.a. fetch-on-write)

1. **Fetch** the entire block from memory into the cache (just like a read-miss).
2. **Update** the desired byte in that newly fetched cache line.
3. If write-back, mark it dirty; if write-through, also push the write to memory.
   – **Why use it?** If you're going to touch (read or write) that block again soon, having it in cache pays off.

2. No-Write-Allocate (a.k.a. write-around)

1. **Skip** loading the block into cache.
2. **Send** the write straight to main memory.
3. The cache remains unchanged for that address.
   – **Why use it?** If you're only writing once and unlikely to reuse that data, you save cache space.

1. Write-Allocate (Fetch-on-Write)

- **What happens:**
  1. On a write-miss you **load** the entire block into the cache (just like you would on a read-miss).
  2. You then **update** that block in cache at the correct offset and (if it's write-back) mark it dirty.
- **Why:**
  - If you're likely to touch that block again soon—either reading or writing—having it in cache pays off.
  - Commonly paired with **write-back**: you fetch once, do many updates in cache, then write back on eviction.
- **Analogy:**
  You walk to the filing cabinet to pull the folder onto your desk **before** you modify it, because you know you'll need to refer to it again.

---

2. No-Write-Allocate (Write-Around)

- **What happens:**
  1. On a write-miss you **do not** load the block into cache.
  2. You send the write straight to the next level (L2 or DRAM).
  3. The cache remains unchanged for that address.
- **Why:**
  - If you're only going to write once and never read it soon, there's no point filling your cache with it.
  - Often paired with **write-through**: you update memory immediately anyway, so skipping cache on a miss saves space.
- **Analogy:**
  You simply walk to the cabinet, jot down your note in the folder right there, and leave it – you don't bother bringing it back to your desk.

1. **Write-through**
   - Every store you do in the cache immediately goes down to memory as well.
   - This keeps RAM perfectly up to date but can still be slow if you do lots of writes.
2. **Write-back**
   - On a store (write hit), the cache marks that line **dirty** and only updates the on-chip copy.
   - The CPU continues working fast.
   - **Later**, when that cache line gets evicted (kicked out to make room for something else), the cache notices the dirty bit and then writes it to main memory.
   - This batches up multiple writes into one DRAM transaction, which is more efficient.

The Memory Mountain

> **MEMORY MOUNTAIN:** A 3D surface plotting read throughput (MB/s) vs. working set size and stride to quantify spatial and temporal locality.

Test Function (C pseudocode):

```
long data[MAXELEMS];
int test(int elems, int stride) {
    long acc0=0, acc1=0, acc2=0, acc3=0;
    long limit = elems - 4*stride;
    for (long i = 0; i < limit; i += 4*stride) {
        acc0 += data[i];
        acc1 += data[i+stride];
        acc2 += data[i+2*stride];
        acc3 += data[i+3*stride];
    }
    for (long i = limit; i < elems; i++) {
        acc0 += data[i];
    }
    return acc0 + acc1 + acc2 + acc3;
}
```

Methodology:

1. Call `test()` once to warm caches.
2. Call `test()` again and measure read throughput.

Final Summary & Takeaways

- **Cache parameters** (B, C, E) and bit fields (b, s, t) define mapping behavior.
- **Replacement policies** (LRU, FIFO, Random) affect conflict miss rates.
- **Performance metrics** (MR, HT, MP) guide design and tuning.
- **Write policies** trade consistency vs. performance.
- **Memory Mountain** visualizes how working set size and stride impact throughput.

23. Optimization

Plan

- **Writing Cache-Friendly Code**
- **Compiler Optimization Techniques**

Writing Cache-Friendly Code

Loop Interchange for Spatial Locality

> **SPATIAL LOCALITY:** Accessing memory addresses that are contiguous.

```
/* ijk order */
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    double sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
```

```
  }
}
```

- Poor locality on `b[k][j]` (column-wise).
- Interchange to `kij` or `ikj` to traverse contiguous rows of `b`.

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;
  }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

```
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];
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

```
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;
  }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

Blocking (Tiling) for Temporal Locality

> **BLOCKING:** Partition loops into smaller tiles that fit in cache to maximize data reuse.

```
for (int i = 0; i < n; i += B)
  for (int j = 0; j < n; j += B)
    for (int k = 0; k < n; k += B)
      /* B×B mini-block multiply */
      for (int ii = i; ii < i+B; ii++)
        for (int jj = j; jj < j+B; jj++) {
          double sum = 0.0;
          for (int kk = k; kk < k+B; kk++)
            sum += a[ii][kk] * b[kk][jj];
```

```
            c[ii][jj] += sum;
        }
```

- Choose `B` so that `3·B² < CacheSize`.
- Reduces miss rate from $O(n^3)$ to $O(n^3/(4B))$.

Compiler Optimization Techniques

What Is Optimization

> **OPTIMIZATION:** The process of improving program efficiency in time or space, often aided by compiler transformations.

GCC Optimization Levels

- `O0` — No optimization (baseline).
- `O2` — Enable most safe, standard optimizations.
- `O3` — Aggressive optimizations (may increase code size).
- `Os` — Optimize for code size.
- `Ofast` — Disregard some language standards for speed.

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Common GCC Optimizations

> **CONSTANT FOLDING:** Compute constant expressions at compile time.
>
> **COMMON SUB-EXPRESSION ELIMINATION:** Reuse previously computed expressions.
>
> **DEAD CODE ELIMINATION:** Remove code with no effect on program output.
>
> **STRENGTH REDUCTION:** Replace expensive operations (e.g., multiply/divide) with cheaper ones (add/shift).
>
> **CODE MOTION:** Hoist invariant code out of loops.
>
> **TAIL RECURSION:** Convert tail-recursive calls into loops.
>
> **LOOP UNROLLING:** Expand loop bodies to reduce control overhead.

Limitations of GCC Optimization

- Cannot optimize across unknown function calls (e.g., repeated `strlen()` inside loops).
- May not hoist calls when data-dependence is unclear.
- Algorithmic improvements often yield greater gains than micro-optimizations.

Final Summary & Takeaways

- **Cache-Friendly Coding:** Loop interchange and blocking dramatically improve memory reuse.
- **Compiler Flags:** Use `O2` as a default; higher levels (e.g., `O3`) for performance-critical code.
- **Profile First:** Identify hotspots with tools like `callgrind` before manual tuning.
- **Balance Effort:** Prioritize algorithmic complexity before low-level optimizations.

24. Linking

Linking Overview

What Is Linking?

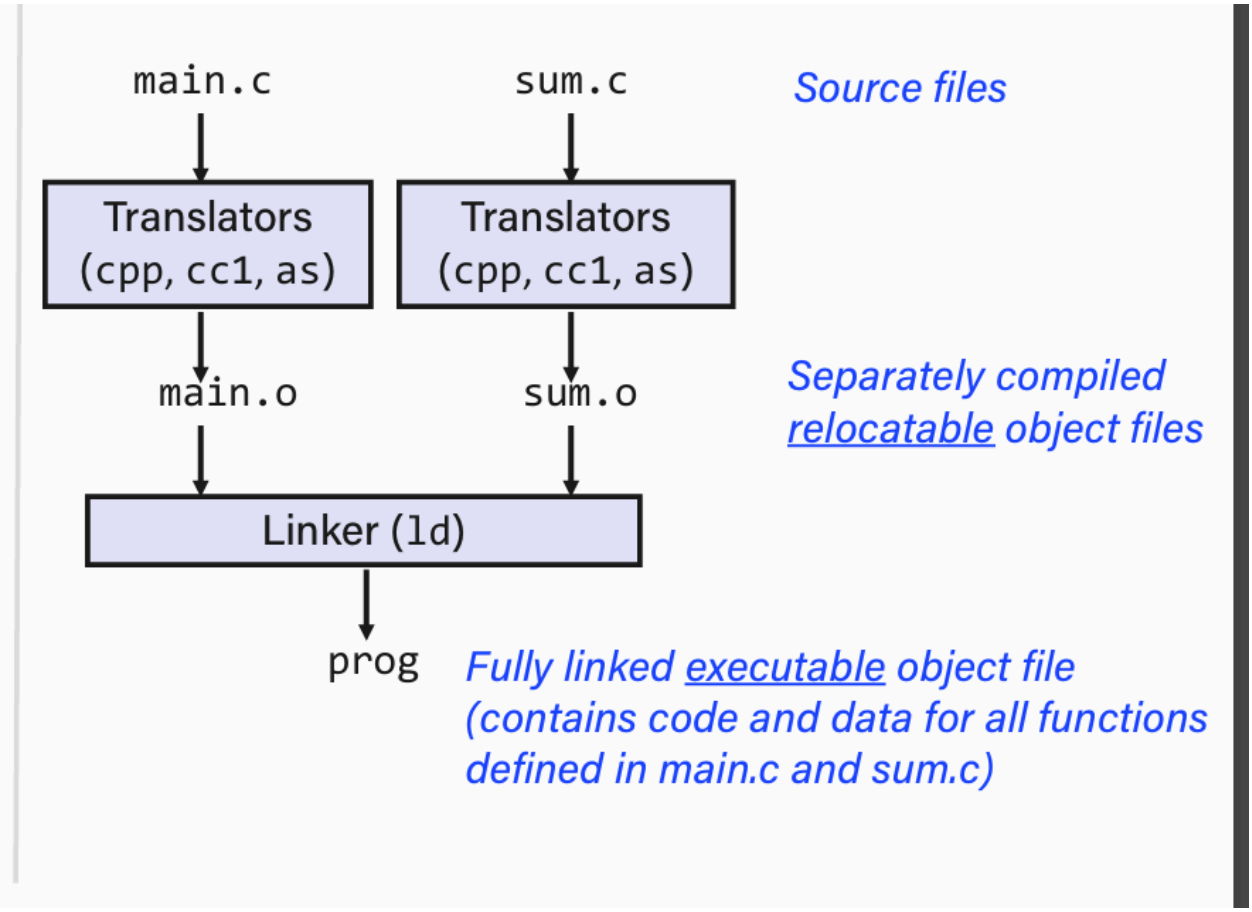
> **LINKING:** The process of taking one or more relocatable object files and combining them into a single executable or shared library by resolving symbol references and adjusting addresses.

- Enables modular development: compile each source file independently.
- Produces final binaries containing only the code and data needed at run time.

Linker Role in Toolchain

1. **Compilation & Assembly:**
    - Source files (`.c`) → Compiler frontend (preprocessing → parsing → codegen) → Compiler backend (assembly) → Assembler → Produces relocatable object files (`.o`).
2. **Linking:**
    - Linker (`ld`) takes `.o` files (and static libraries) to produce:
        - **Executable Object File** (e.g., `a.out` , `prog`), or

- **Shared Object File** (dynamically loadable `.so`).



Step 1: Symbol Resolution

Symbol Concepts

> **SYMBOL:** A name that identifies a function or global variable in code.
> **DEFINITION (Definition Site):** The object file section where a symbol's storage or code is allocated (e.g., int foo = 5;, void bar() { ... }).
> **REFERENCE (Reference Site):** A use of a symbol declared externally (e.g., calling an external function or accessing a global variable).

Symbol Tables in Object Files

- Each relocatable object file (`.o`) contains a **symbol table** listing:
  - **Name:** ASCII identifier (`foo`, `sum`, `array`).

- **Section & Offset:** Where the symbol resides (e.g., `.text`, `.data`, `.bss`).
- **Size & Visibility:** Size in bytes, and whether the symbol is global (external) or local (static).

> **SYMBOL RESOLUTION:** The linker's process of matching each undefined (external) symbol reference to exactly one definition across all input object files and libraries.

- If a reference has no matching definition → **undefined symbol error**.
- If multiple strong definitions exist → **duplicate symbol error**.
- Weak vs. strong symbols: uninitialized globals are "weak," initialized globals and functions are "strong."
  - Rule: one strong definition allowed; linking picks the one strong symbol, ignoring weak duplicates.

Example: Resolving `sum` and `array`

```
// main.c
int sum(int *a, int n);      // reference to sum
int array[2] = {1, 2};       // definition of array

int main() {
    int val = sum(array, 2); // sum: reference; array: reference
    return val;
}
// sum.c
int sum(int *a, int n) {     // definition of sum
    int i, s = 0;
    for (i = 0; i < n; i++) s += a[i];
    return s;
}
```

- **Linker Behavior:**
  1. In `main.o`, sees reference to `sum` and to `array`.
  2. In `sum.o`, sees definition of `sum`.

3. In `main.o`, sees definition of `array`.
4. Linker resolves:
    - `sum` reference → `sum` definition in `sum.o`.
    - `array` reference → `array` definition in `main.o`.

Step 2: Relocation

What Is Relocation?

> **RELOCATION:** Adjusting symbol addresses and placeholder references in object code so that instructions and data pointers refer to the correct absolute memory locations in the final executable.

- Each `.o` file's sections (`.text`, `.data`, `.bss`) begin at offset 0 relative to that file.
- The linker concatenates sections from multiple `.o` files, computing final base addresses for each section.
- Every instruction or data reference with a relocation entry is updated to reflect the final address of the target symbol.

Relocation Entries

- **Relocation Record:** In the `.rel.text` or `.rel.data` section of a relocatable file, containing:
    - **Offset:** Byte offset within the section where adjustment is needed.
    - **Type:** Type of relocation (e.g., absolute, PC-relative).
    - **Symbol:** Name/index of the symbol whose final address is used.
- During linking, the linker reads these records, computes each symbol's final address, and patches the instruction operand or data word at the given offset.

Object File Types

Relocatable Object File (`.o`)

> **RELOCATABLE (.o):** Contains code and data in sections that can be combined with other relocatable files.

- Sections:
  - `.text` (machine code)
  - `.rodata` (read-only constants)
  - `.data` (initialized globals)
  - `.bss` (uninitialized globals; allocated at load time)
  - `.symtab` (symbol table)
  - `.rel.text`, `.rel.data` (relocation info)
- Produced by the assembler (`as`) from a single translated source file.

Executable Object File (e.g., `a.out`, `prog`)

> **EXECUTABLE:** Contains code and data with all symbols resolved and addresses fixed; ready to be loaded by the OS loader.

- Sections:
  - ELF header, Program header table (for runtime loader)
  - `.text`, `.rodata`, `.data`, `.bss` (merged across modules)
  - Optional debug sections (`.debug`, `.symtab`) if compiled with `g`.

Shared Object File (`.so`)

> **SHARED OBJECT (.so):** A special relocatable file intended for dynamic linking at load or run time.

- Contains exportable symbols and relocation entries that the dynamic loader (`ld-linux.so`) processes when an executable is run.
- Can be loaded by multiple processes simultaneously, saving memory.
- ABI versioning and SONAME used to manage compatibility.

The ELF Format

ELF Basics

> **ELF (Executable and Linkable Format):** Standard binary format on Linux for all object files.

- Unified format for `.o`, executables, and `.so`.
- Consists of:
    1. **ELF Header:** Magic number, bit-width (32/64), endianness, file type, target architecture.
    2. **Program Header Table (executables only):** Information for runtime loader: segment addresses, sizes, permissions.
    3. **Section Header Table:** Describes each section's name, type, offset, size (e.g., `.text`, `.data`, `.symtab`, `.rel.text`).
    4. **Sections:**
        - `.text` — machine code
        - `.rodata` — read-only constants
        - `.data` — initialized globals
        - `.bss` — uninitialized globals (occupies no file space)
        - `.symtab` — symbol table entries
        - `.rel.text`, `.rel.data` — relocation entries
        - `.debug*` — debug information (optional)

Static Libraries

What Is a Static Library?

> **STATIC LIBRARY (.a):** An archive of multiple relocatable object files packaged together, used to resolve external references at link time.

- Common usage: grouping related functions (e.g., `libc.a`, `libm.a`, `libvector.a`).

- The linker searches archives in command-line order and extracts only those object files that satisfy currently unresolved symbols.

Creating and Using Static Libraries

1. **Compile Modules Individually:**

```
gcc -c addvec.c    # produces addvec.o
gcc -c multvec.c   # produces multvec.o
```

2. **Archive into Library:**

```
ar rcs libvector.a addvec.o multvec.o
```

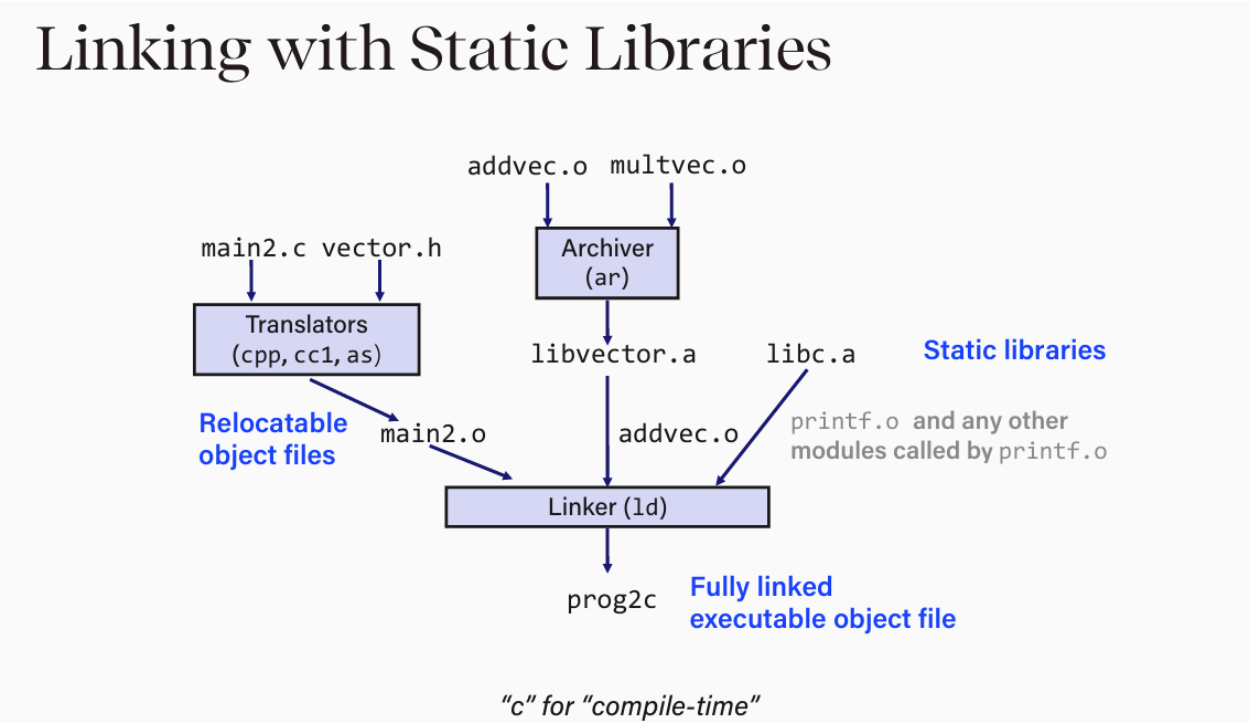
3. **Link with Library:**

```
gcc -o prog main.o -L. -lvector -lm
```

   - Order matters: unresolved references from `main.o` must come before `lvector`.
   - The linker only pulls object files from `libvector.a` that define needed symbols.

Advantages & Limitations

- **Advantages:**
  - Space efficiency: executables include only used functions.
  - Convenience: group related modules.
- **Limitations:**
  - Duplicate code across different executables (each static binary has its own copy).
  - Cannot update library code without relinking executables.



Shared (Dynamic) Libraries

What Is a Shared Library?

> **SHARED LIBRARY (.so):** A relocatable object file that is loaded and linked at load time or run time, allowing code sharing across multiple processes.

- Dynamically linked by the loader (`ld-linux.so`) when the executable starts, or by explicit calls to `dlopen()` at run time.

Building and Linking Shared Libraries

1. **Compile with Position-Independent Code (PIC):**

```
gcc -fPIC -c addvec.c   # addvec.o contains PIC
gcc -fPIC -c multvec.c  # multvec.o contains PIC
```

2. **Create Shared Object:**

```
gcc -shared -o libvector.so addvec.o multvec.o
```

3. **Link Executable Dynamically:**

```
gcc -o prog main.o -L. -lvector
```

   - At load time, the dynamic linker searches for `libvector.so` in library paths, loads it, resolves symbols, and performs necessary relocations.

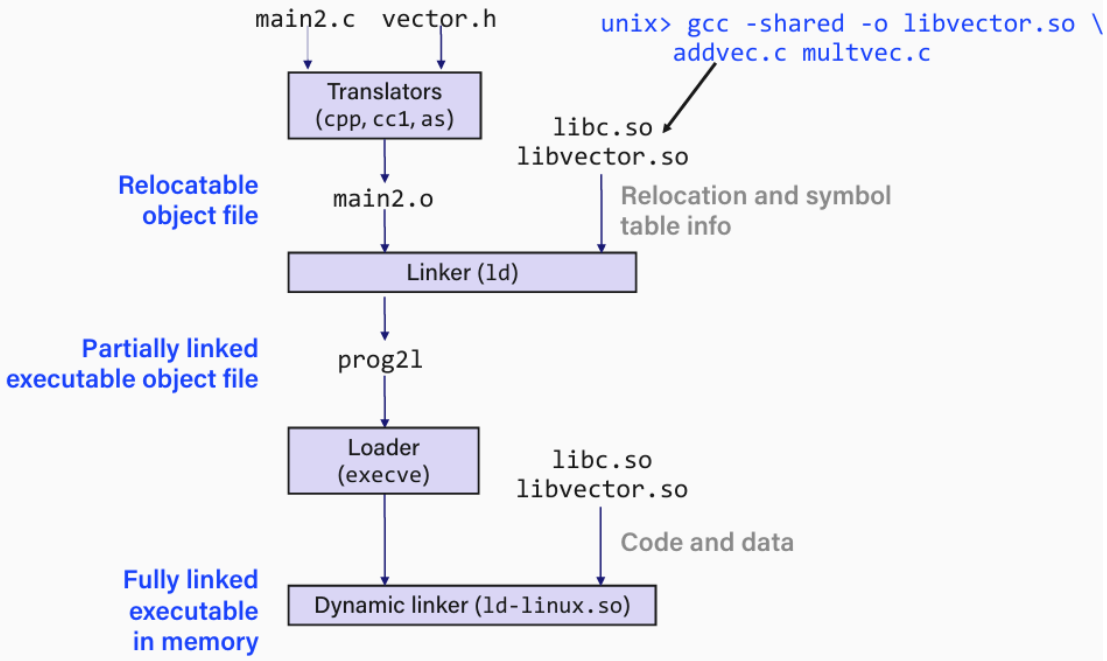
Load-Time vs. Run-Time Dynamic Linking

- **Load-Time Linking:**
  - Occurs when the program is started (via `execve`).
  - The dynamic linker resolves undefined symbols against loaded shared libraries, relocates code for position differences, and then transfers control to `main`.
- **Run-Time Linking (dlopen):**
  - A running program can load a shared library with `dlopen("libvector.so", RTLD_LAZY)`.
  - Retrieve function pointers with `dlsym()`, call routines, then unload with `dlclose()`.

Advantages & Trade-offs

- **Advantages:**
  - Single copy of library code in memory shared by all processes.
  - Easier library updates: fix a bug in `libfoo.so` and all executables using it benefit without relinking.
- **Trade-offs:**
  - Slight load-time overhead for dynamic symbol resolution.
  - Potential for “dependency hell” if incompatible versions are loaded at run time.

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

```
    ...

    /* 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

Common Linking Errors & Puzzles

Duplicate Symbol Definitions

> Occurs when two or more object files (or libraries) each provide a strong definition of the same symbol.

- **Example:**

```
// In a.c
int x;      // weak (uninitialized) definition of x
void p1() { }

// In b.c
int x;      // weak definition of x
void p2() { }
```

  - Linking succeeds, both `x` definitions are identical weak symbols — one is chosen arbitrarily.

- If one were `int x = 7;` (strong) and the other `int x;` (weak), the strong definition is chosen.
- Two strong definitions (e.g., `int x = 7;` in both) → **linker error: duplicate symbol**.

Undefined References

> Occurs when a symbol is referenced but not defined in any input file or library.

- **Fixes:**
  1. Add the missing object file or library to the link line.
  2. Ensure correct order: object files referencing library symbols must appear before `l<lib>` on the linker command line.

Relocation Errors

> Occurs when a relocation entry cannot be processed because the target symbol is missing or incompatible.

- **Common Causes:**
  - Mixing position-dependent and position-independent code incorrectly.
  - Attempting to statically link PIC objects without `fPIC`.
  - Mismatched architectures (e.g., compiling for x86_64 but linking with i386 libraries).

Final Summary & Takeaways

- **Linking Stages:**
  1. **Symbol Resolution:** Match symbol references to definitions, enforce one strong definition, handle weak symbols.
  2. **Relocation:** Adjust addresses in code and data based on final section placements.
- **Object File Categories:**
  - **Relocatable (`.o`):** Input to linker; contains symbol tables and relocation entries.

- **Executable:** Fully linked binary ready for loading.
- **Shared Object (`.so`):** Dynamically linked library loaded at run time or load time.

- **Static vs. Dynamic Libraries:**
  - **Static (`.a`):** Linked at compile/link time, duplicate code in each executable.
  - **Shared (`.so`):** Loaded by dynamic linker, one copy of code shared by multiple processes, can be updated independently.
- **Link-Time Errors:**
  - Duplicate strong symbols → linker error.
  - Undefined references → missing input file or library.
  - Relocation failures → architecture or PIC mismatches.
- **Good Practices:**
  - Use `static` keyword for internal-linkage variables/functions to avoid unintended symbol exports.
  - Organize libraries: put frequently used functions in shared libraries when appropriate.
  - Always place libraries (`l`) after object files in link command.
  - Use versioned SONAMEs for shared libraries to manage compatibility.

25. Wrap-Up

Recap: Core Topics Covered

1. Bits and Bytes

> **Representation:** How integers and floating-point values are encoded in binary.
>
> - **Integers:** Signed (two's complement) and unsigned representations; overflow and bitwise operations.
> - **Floats:** IEEE-754 format for single and double precision; rounding, precision limits, and pitfalls (e.g., floating-point comparisons).

2. Characters and C Strings

> **C Strings:** Arrays of char terminated by a null byte ('\\0').
>
> - **Operations:** `strlen`, `strcpy`, `strcmp`, pointer manipulation.
> - **Implications:** Memory safety (buffer overflows), efficient string traversal, and the importance of the null terminator.

3. Pointers, Stack, and Heap

Pointers: Variables that store memory addresses; dereferencing and pointer arithmetic.

Stack Allocation: Automatic (local) variables, function call frames, cleanup on return.

Heap Allocation: Dynamic memory via `malloc` / `free`; fragmentation and manual management.

- **Trade-Offs:** Stack is fast and auto-managed; heap is flexible but requires careful allocation and deallocation.

4. Generics in C

Void Pointers (void *): Type-agnostic pointers for data abstraction.

Memcpy & Function Pointers: Copying arbitrary data blocks; passing behavior via function pointers.

- **Use Cases:** Implementing generic data structures (e.g., linked lists, dynamic arrays) without compile-time type information.

5. Assembly Language

Compilation Workflow: C source → assembly (.s) → object (.o) → executable.

Registers & Instructions: `mov`, `add`, `call`, `ret`; calling conventions and RTL (Register Transfer Language).

- **Stack Frames:** Layout of saved registers, return addresses, and local variables; understanding `push` / `pop` and frame pointers.

6. Cache Memories

Memory Hierarchy: Registers → L1/L2/L3 cache → DRAM → secondary storage.

Locality of Reference:

- **Temporal:** Reuse recently accessed data.
- **Spatial:** Access contiguous addresses.

Cache Parameters: Block size, associativity, hit/miss rates, write policies.

- **Strategies:** Loop restructuring (interchange, blocking) to improve cache performance.

7. Optimization Techniques

Loop Transformations:

- **Loop Interchange:** Reorder nested loops to access data in cache-friendly order.
- **Blocking/Tiling:** Break large loops into cache-sized chunks to maximize data reuse.

Compiler Optimizations:

- **Constant Folding, Dead Code Elimination, Common Subexpression Elimination, Strength Reduction, Loop Unrolling, Code Motion.**
- **GCC Flags:** `O2` (standard optimizations), `O3` (aggressive), `Os` (size-optimized), `Ofast` (unsafe but fast).

8. Linking

Separate Compilation: Source files → compiled object files.

Static Linking: Combine `.o` files (and `.a` archives) into a single executable; symbols resolved at link time.

Dynamic Linking (Shared Libraries):

- `.so` **Files:** Position-Independent Code (PIC), loaded at run time by the dynamic loader (`ld-linux.so`).
- **Advantages:** Single shared copy in memory, easier updates, reduced executable size.

Relocation & Symbol Resolution: Adjust addresses and resolve external references; handle weak vs. strong symbols, duplicate definitions, and undefined references.

COMP201 Tools and Techniques

Unix and the Command Line

Shell Proficiency: Navigating directories (cd), listing files (ls), file permissions (chmod/chown), process management (ps/kill).

Text Processing: `grep`, `awk`, `sed`, `sort`, `uniq` for filtering and transforming text.

Build Systems:

- **Makefiles:** Define targets, dependencies, and build commands.
- `gcc` **Invocation:** Common flags for compilation (`O2`, `g`, `Wall`, library linking `l<name>`).

Coding Style

Code Readability: Consistent indentation, meaningful variable/function names, modular functions.

Commenting Practices: Brief, descriptive comments for non-obvious logic; header comments for file/module purpose.

Error Handling: Check return values from system/library calls, handle `errno`, use assertions (`assert`) for invariants.

Debugging with GDB

Breakpoints & Watchpoints:

- `break <location>`, `watch <expression>` to halt execution on conditions.

Stepping: step (into function calls), next (over calls), continue (resume).

Inspecting State:

- `print <variable>`, `info registers`, `backtrace` for call stacks.

Core Dumps:

- Enable core files via `ulimit -c unlimited`; analyze with `gdb <exec> core`.

Memory Checking with Valgrind

valgrind --leak-check=full <executable>: Detects memory leaks, unreachable blocks, and improper frees.

- **Invalid Reads/Writes, Use-after-free, Double Free.**

> **Memory Errors:**
>
> **Massif Tool:** valgrind --tool=massif for heap profiling; visualize allocation over time.

Profiling with Callgrind

> **valgrind --tool=callgrind <executable>:** Records function call counts and instruction counts.
>
> **Analysis:**
>
> - Use `kcachegrind` or `qcachegrind` to visualize hotspots and call graphs.

> **Optimization Guidance:** Focus on "expensive" functions or loops consuming the most instructions or cache misses.

Final Takeaways

- **Foundation in C and Systems:** Grasp of low-level data representation, memory hierarchy, and linking processes empowers you to write efficient, safe, and portable code.
- **Toolchain Mastery:** Proficiency with Unix/CLI, Makefiles, GDB, Valgrind, and profilers is essential for debugging, analyzing, and optimizing real-world applications.
- **Performance Mindset:** Understanding how code maps to hardware (caches, pipelines) guides algorithmic and code-level optimizations.
- **Lifelong Learning Path:** The concepts and skills from COMP201 serve as a springboard into specialized areas—embedded systems, operating systems, compilers, networking, databases, security, HPC, and beyond.
- **Practice and Exploration:** Continuously apply these tools and techniques in projects, open-source contributions, and research to deepen your expertise and adapt to evolving technologies.
