

# CSC 212: Data Structures and Abstractions

## 02: C++ Review, Memory, and Pointers

Prof. Marco Alvarez

Department of Computer Science and Statistics  
University of Rhode Island

Fall 2025



# Compiling C++ programs

## Context

machine code	assembly	C++	Python
<pre>10110100 10110111 00101011 00011010 00010100 10111011 10001000 11110111 00101000 10101010 00101101 00010001 01010010 11101100 11010001 10010100 10010100 00100000 00000000 10100001 00110001 10101001 00010101 00101010 00100100 10010100 01110001 11110101 11101011 00101111 01010010 10000101 11111110 10101010 00101101 00010001 01010010 11101100 11010001 10010100 10010100 00100000 00000000 10100001 00110001 10101001 00010101 00101010 00100100 10010100 01110001 11110101 11101011 00101111 01010010 10000101 11111110 00101001 00000000 00000000 00000000 00000000 01010000 00010101 00001010 00101010 00101010 00100100 10011111</pre>	<pre>.equ STDOUT, 1 .equ SVC_WRITE, 64 .equ SVC_EXIT, 93  .text .global _start _start:     stp x29, x30, [sp, -16]!     mov x8, #STDOUT     ldr x1, =msg     mov x2, 13     mov x8, #SVC_WRITE     mov x29, sp     svc #0     ldp x29, x30, [sp], 16     mov x8, #0     mov x8, #SVC_EXIT     svc #0  msg: .ascii "Hello World\n" .align 4</pre>	<pre>#include &lt;iostream&gt;  int main () {     std::cout &lt;&lt; "Hello World!"     &lt;&lt; std::endl; }</pre>	<pre>print('Hello World')</pre>

→ increasing abstraction →

To illustrate the potential gains from performance engineering, consider multiplying two 4096-by-4096 matrices. Here is the four-line kernel of Python code for matrix-multiplication:

```
for i in xrange(4096):
    for j in xrange(4096):
        for k in xrange(4096):
            C[i][j] += A[i][k] * B[k][j]
```

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

# Program execution approaches

## • Compilation

- ✓ high level source **translated** into another language
  - often into a machine-specific instructions
  - translation occurs through multiple phases
- ✓ compilers can perform **optimizations** to make the code more efficient, resulting in faster execution (higher performance)
- ✓ e.g. C/C++ compilers

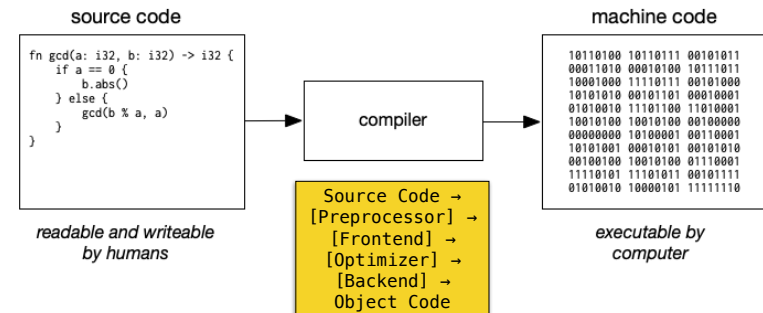
## • Interpretation

- ✓ “executing” a program directly from source
  - read code line by line, translate it into machine code, and execute
  - any language can be interpreted
- ✓ preferred when performance is not critical
- ✓ e.g. Javascript

5

# Compiling programs (simplified)

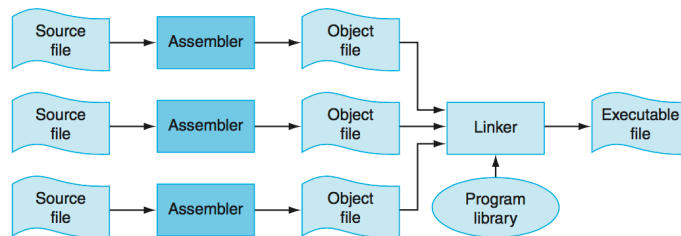
- Typically, “compiling” a program refers to the process of generating machine code from source code
  - ✓ the process takes several steps: **compile**, **assemble**, **link**



[https://www.uvm.edu/~cbcafier/cs1210/book/02\\_programming\\_and\\_the\\_python\\_shell/programming.html](https://www.uvm.edu/~cbcafier/cs1210/book/02_programming_and_the_python_shell/programming.html)

6

# Compiling/linking/running C programs



C++ programs can be compiled/linked through both IDEs and command-line tools.

- **Command Line:** Using compilers like g++ or clang++ gives you fine-grained control.
- **IDE:** IDEs like VS Code, Code::Blocks, or CLion handle compilation/linking behind the scenes. They typically use build systems like CMake, Make to manage the process automatically.

The command line gives you transparency and scriptability – you can see exactly what flags are being used and automate builds easily. IDEs provide convenience, debugging integration, and often better error visualization, but can sometimes obscure what's actually happening during the build process.

7

# Data representation

## Range of values

Data type	Size	Format	Value range
character	8	signed	-128 to 127
		unsigned	0 to 255
integer	16	signed	-32768 to 32767
		unsigned	0 to 65535
	32	signed	-2,147,483,648 to 2,147,483,647
		unsigned	0 to 4,294,967,295
	64	signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to 18,446,744,073,709,551,615

Data type	Smallest positive value (*)	Largest positive value (*)	Precision (**)
float	$\sim 1.401 \cdot 10^{-45}$	$\sim 3.403 \cdot 10^{+38}$	6-9 digits
double	$\sim 4.941 \cdot 10^{-324}$	$\sim 1.798 \cdot 10^{+308}$	15-17 digits

<https://en.cppreference.com/w/cpp/language/types>

9

## Standard integer types

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
signed char	signed char	at least 8	8	8	8	8
unsigned char	unsigned char					
short	short int	at least 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short	unsigned short int	at least 16	16	16	16	16
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int	unsigned int	at least 16	16	32	32	32
unsigned						
unsigned int	long int	at least 32	32	32	32	64
long						
long int						
signed long						
signed long int	unsigned long int	at least 32	32	32	32	64
unsigned long						
unsigned long int	long long int	at least 64	64	64	64	64
long long						
long long int						
signed long long						
signed long long int	unsigned long long int	at least 64	64	64	64	64
unsigned long long						
unsigned long long int	(C++11)					

10

## What is the output?

```
#include <iostream>

int main() {
    int d = 42;
    int o = 052;
    int x = 0x2a;
    int X = 0X2A;
    int b = 0b101010; // C++14

    std::cout << d << " " << o << " " << x
              << " " << X << " " << b << std::endl;

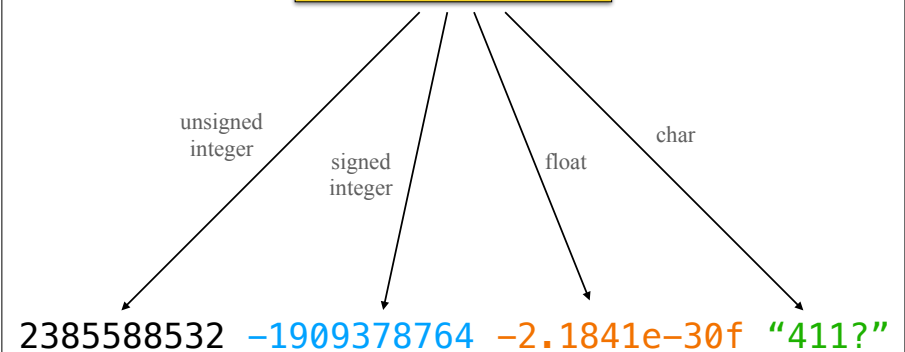
    return 0;
}
```

11

## Variables are just bit sequences

1000 1110 0011 0001 0011 0001 0011 0100

0x8E31313A



12

# Memory and pointers

## Memory organization

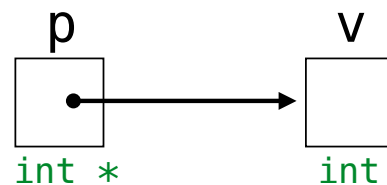
- Memory as a byte array
  - ✓ contiguous sequence of bytes
  - ✓ used to store **data and instructions** for computer programs
  - ✓ each byte individually accessed via a **unique address**
- Memory address
  - ✓ **unique** numerical identifier for each byte in memory, often displayed in hexadecimal notation
  - ✓ provides indirect access to data stored at that location
- Data representation in memory
  - ✓ variables stored as byte sequences
  - ✓ interpretation and number of bytes depends on type
    - integers, floating-point numbers, characters, etc.

14

## Variables and pointers

- Every variable exists at a **memory address**
  - ✓ regardless of **variable scope**
  - ✓ the compiler translates names to addresses when generating machine code

A **pointer** is just a variable that stores the memory address of another variable



15

## Pointers

- Declaration
    - ✓ like other variables, pointers must be declared before use
    - ✓ for each declaration, a pointer type must be specified
- ```
type *pointer_name;
```
- Pointer operators
    - ✓ **address-of** operator: get memory address of variable/object

&

- ✓ **dereference** operator: get value at given memory address

\*

16

## Declaring pointers

```
// can declare a single
// pointer (preferred)
int *p;

// can declare multiple
// pointers of the same type
int *p1, *p2;

// can declare pointers
// and other variables too
double *p3, var, *p4;
```

17

## Pointer operators

```
int main() {
    int var = 10;
    int *ptr;
    ptr = &var;
    *ptr = 20;

    // ...

    return 0;
}
```

32-bit words

| Address    | Value | Variable |
|------------|-------|----------|
| ...        |       |          |
| 0x91340A08 |       |          |
| 0x91340A0C |       |          |
| 0x91340A10 |       |          |
| 0x91340A14 |       |          |
| 0x91340A18 |       |          |
| 0x91340A1C |       |          |
| 0x91340A20 |       |          |
| 0x91340A24 |       |          |
| 0x91340A28 |       |          |
| 0x91340A2C |       |          |
| 0x91340A30 |       |          |
| 0x91340A34 |       |          |
| ...        |       |          |

18

## Pointer operators

```
int main() {
    int temp = 10;
    int value = 100;
    int *p1, *p2;

    p1 = &temp;
    *p1 += 10;

    p2 = &value;
    *p2 += 5;

    p2 = p1;
    *p2 += 5;

    return 0;
}
```

32-bit words

| Address    | Value | Variable |
|------------|-------|----------|
| ...        |       |          |
| 0x91340A08 |       |          |
| 0x91340A0C |       |          |
| 0x91340A10 |       |          |
| 0x91340A14 |       |          |
| 0x91340A18 |       |          |
| 0x91340A1C |       |          |
| 0x91340A20 |       |          |
| 0x91340A24 |       |          |
| 0x91340A28 |       |          |
| 0x91340A2C |       |          |
| 0x91340A30 |       |          |
| 0x91340A34 |       |          |
| ...        |       |          |

19

## Pointers and functions

```
void increment(int *ptr) {
    (*ptr) ++;
}

int main() {
    int var = 10;

    increment(&var);
    increment(&var);

    // ...

    return 0;
}
```

32-bit words

| Address    | Value | Variable |
|------------|-------|----------|
| ...        |       |          |
| 0x91340A08 |       |          |
| 0x91340A0C |       |          |
| 0x91340A10 |       |          |
| 0x91340A14 |       |          |
| 0x91340A18 |       |          |
| 0x91340A1C |       |          |
| 0x91340A20 |       |          |
| 0x91340A24 |       |          |
| 0x91340A28 |       |          |
| 0x91340A2C |       |          |
| 0x91340A30 |       |          |
| 0x91340A34 |       |          |
| ...        |       |          |

20

## Pointer arithmetic

### Core principle

- allows mathematical operations (**addition, subtraction**) with pointers, but works differently than regular arithmetic

### Key Rules

- add/subtract integer values to pointers ( $p + n$ )
  - adding  $n$  to a pointer  $p$  moves it forward by  $(n * \text{sizeof}(*p))$  bytes
- memory addresses are integers, typically displayed in hexadecimal format

**Warning:** adding 1 to a pointer means moving to the next element of the pointed-to type, not moving 1 byte forward in memory

- incorrect pointer arithmetic can lead to buffer overflows and undefined behavior
- always verify pointer bounds before arithmetic operations

21

## Pointer arithmetic

```
int arr[] = {1, 2, 3, 4, 5};
int *ptr = arr;
ptr++;    // advances ptr by 4 bytes
ptr += 2; // advances ptr by 8 bytes
```

Note: an **array name** is NOT a pointer variable but an immutable array identifier that automatically converts to a pointer in most contexts. In expressions and function calls, `arr` undergoes "pointer decay" and behaves as `&arr[0]`.

22

## Example: changing a pointer within a function

```
#include <iostream>
```

```
void seek(int *p, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (*p == key) {
            return;
        }
        p++;
    }
}

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(p, 3, 5);
    std::cout << *p << std::endl;

    return 0;
}
```

The pointer variable `p` in `seek()` is a copy. Any changes to `p` only affect this local copy. The original pointer `p` in `main()` remains unchanged.

32-bit words

| Address    | Value | Variable |
|------------|-------|----------|
| ...        |       |          |
| 0x91340A08 |       |          |
| 0x91340A0C |       |          |
| 0x91340A10 |       |          |
| 0x91340A14 |       |          |
| 0x91340A18 |       |          |
| 0x91340A1C |       |          |
| 0x91340A20 |       |          |
| 0x91340A24 |       |          |
| 0x91340A28 |       |          |
| 0x91340A2C |       |          |
| 0x91340A30 |       |          |
| 0x91340A34 |       |          |
| ...        |       |          |

23

## Example: changing a pointer within a function

```
// arguments:
// - pointer to a pointer (array)
// - an integer key
// - an integer n, the number of elements
void seek(int **p, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (**p == key) {
            return;
        }
        (*p)++;
    }
}

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(&p, 3, 5);
    std::cout << *p << std::endl;

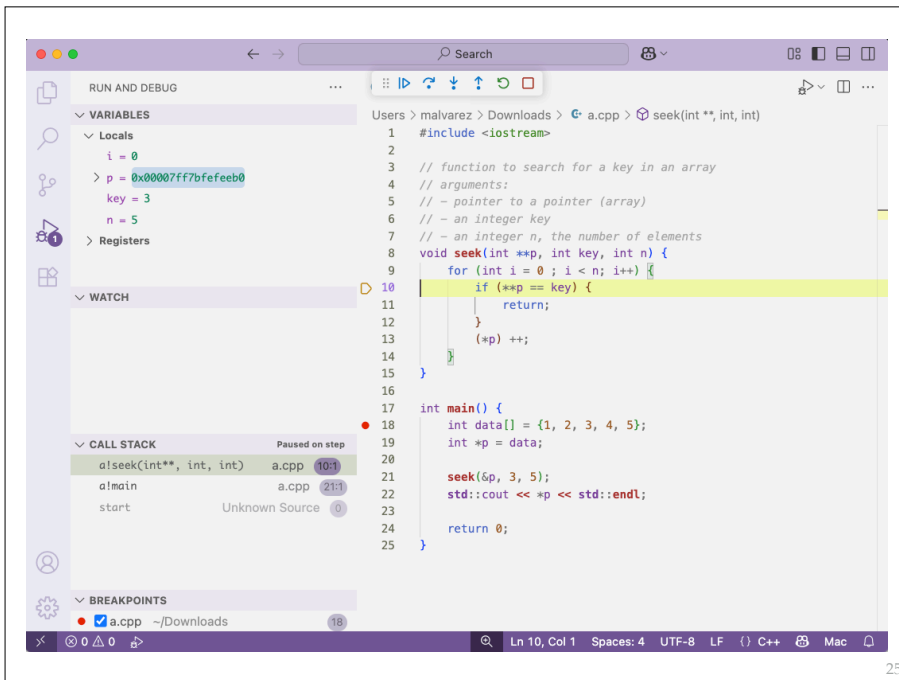
    return 0;
}
```

Solution: to modify the original pointer, can pass a pointer to pointer.

32-bit words

| Address    | Value | Variable |
|------------|-------|----------|
| ...        |       |          |
| 0x91340A08 |       |          |
| 0x91340A0C |       |          |
| 0x91340A10 |       |          |
| 0x91340A14 |       |          |
| 0x91340A18 |       |          |
| 0x91340A1C |       |          |
| 0x91340A20 |       |          |
| 0x91340A24 |       |          |
| 0x91340A28 |       |          |
| 0x91340A2C |       |          |
| 0x91340A30 |       |          |
| 0x91340A34 |       |          |
| ...        |       |          |

24



# C/C++ memory layout

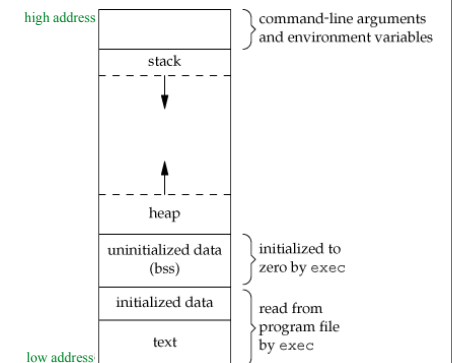
## Memory layout

- What is the C/C++ memory model?
  - ✓ a formal specification that defines how programs interact with memory (rules for memory operations — ordering, visibility, synchronization)
  - implementation details are handled by the compiler and CPU architecture
- Memory layout
  - ✓ memory is organized into multiple segments (where data is stored)
  - ✓ each segment serves a specific purpose and has different properties

27

## Memory layout (segments)

- Text Segment (code)
  - ✓ contains **instructions** generated by the compiler
  - ✓ marked as **read-only**
- Data (global/static variables)
  - ✓ contains multiple subsections (e.g. initialized data, uninitialized data, constant data)
  - ✓ size determined at compilation, addresses resolved during linking
- Heap
  - ✓ grows upward (low to high addresses)
  - ✓ dedicated to **dynamic memory allocation**
  - ✓ requires explicit management by the programmer
- Stack (local variables, function parameters)
  - ✓ grows downward (high to low addresses)
  - ✓ no explicit deallocation required



28

```
#include <iostream>

// global variable
float pi = 3.1416;
// constant global variable
const int min = 100;
// uninitialized global variable
int sum;
```

```
void foo(int arg) {
    // local variable
    int i = 1;
    std::cout << "address of arg\t" << &arg << std::endl;
    std::cout << "address of i\t" << &i << std::endl;
}
```

```
int main() {
    // heap variable
    int *A = new int[10];
```

```
    std::cout << "address of pi\t" << &pi << std::endl;
    std::cout << "address of min\t" << &min << std::endl;
    std::cout << "address of sum\t" << &sum << std::endl;
    std::cout << "value of A\t" << A << std::endl;
    std::cout << "address of A\t" << &A << std::endl;
    std::cout << "address of main\t" << (void*) &main << std::endl;
    std::cout << "address of foo\t" << (void*) &foo << std::endl;
    foo(5);
```

```
    delete [] A;
```

```
    return 0;
}
```

`./prog | sort -k 4`

```
address of foo 0x0000000105499fc0
address of main 0x000000010549a0f0
address of min 0x000000010549af48
address of pi 0x000000010549c000
address of sum 0x000000010549c004
value of A 0x00007fd29f705e90
address of i 0x00007ff7baa66438
address of arg 0x00007ff7baa6643c
address of A 0x00007ff7baa66460
```

NOTE: (64-bit addresses)

Can you tell what are the memory locations grouped by different colors?

What happens if you run the program multiple times?

29

# Dynamic memory allocation

## • Static vs dynamic memory

- ✓ **static memory** (stack): size known at **compile-time**
- ✓ **dynamic memory** (heap): size determined at **runtime**

Programmer responsibility to pair every new with corresponding delete

## • Why dynamic memory?

- ✓ useful for variable-sized data (e.g., user input, large arrays)
- ✓ complex data structures (linked lists, trees, graphs)

## • C++ operators

| new                                                                                                                                                                                                                                                                         | delete                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• allocates memory on the heap at runtime</li> <li>• returns pointer to the allocated memory location</li> <li>• two forms: single object allocation and array allocation</li> <li>• throws exception if allocation fails</li> </ul> | <ul style="list-style-type: none"> <li>• deallocates memory previously allocated with new</li> <li>• calls destructor for objects before freeing memory</li> <li>• must match allocation type: single delete for single objects, array delete for arrays</li> <li>• does not set pointer to null after deallocation</li> </ul> |

## • Critical rules

- ✓ every **new** must have exactly one matching **delete**
- ✓ deleting the same pointer twice causes undefined behavior
- ✓ accessing deleted memory leads to undefined behavior

30

# Using new/delete

```
int* p = new int;           // allocate single int
int* arr = new int[10];     // allocate array of 10 ints
```

```
delete p;                   // free single object
delete [] arr;              // free array
```

```
int *ptr1 = new int[100];
int *ptr2 = ptr1; // both pointers reference the same address

delete [] ptr1; // array is freed
delete [] ptr2; // ERROR: attempting to delete already freed memory
```

31

# Pointer safety issues

## • Null pointers

- ✓ dereferencing a null pointer causes *undefined behavior*
- ✓ example:

```
int *p = nullptr;
*p = 5;
```

## • Uninitialized pointers

- ✓ using a pointer before assigning it a valid address results in unpredictable behavior
- ✓ example:

```
int *p;
*p = 10;
```

## • Dangling Pointers

- ✓ occur when a pointer refers to memory that has already been freed or gone out of scope
- ✓ example: returning the address of a local variable from a function

32



## Pointer safety issues

- **Memory Leaks**
  - ✓ dynamically allocated memory that is never freed accumulates and wastes memory
  - ✓ in long-running systems, this can exhaust available memory
- **Buffer overflow**
  - ✓ writing past the end of an allocated block corrupts adjacent memory and may lead to crashes or exploitable vulnerabilities
  - ✓ example: indexing out of array bounds with a pointer
- **Pointer/Array confusion**
  - ✓ arrays decay to pointers, but they are not the same, array names are **constant** addresses
  - ✓ `sizeof(array)` => total size in bytes of all elements
  - ✓ `sizeof(pointer)` => size of the pointer variable itself (e.g., 8 bytes on a 64-bit machine)
  - ✓ misunderstanding this leads to incorrect memory usage and errors

33

## Best practices

- **Initialization**
  - ✓ always initialize pointers, use `nullptr` instead of `0` or `NULL`
- **Prefer smart pointers**
  - ✓ (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) instead of raw pointers for dynamic memory — not covered in class
  - ✓ they automatically manage lifetime and prevent leaks/dangling references
- **Avoid manual memory management**
  - ✓ use containers (`std::vector`, `std::string`, `std::array`) instead of raw arrays

34

## Important C++ topics to review

- Memory model and pointers
- Dynamic memory allocation
- Classes and objects
- References
- Templates
- STL containers

35