CSC 212: Data Structures and Abstractions

02: C++ Review, Memory, and Pointers

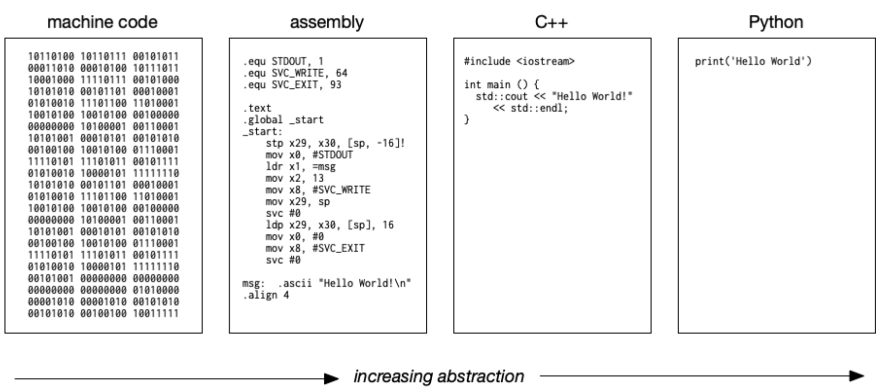
Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

Compiling C++ programs

Context



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]
```

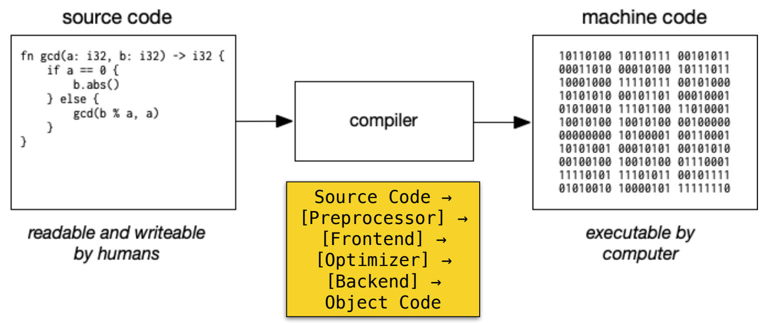
Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	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

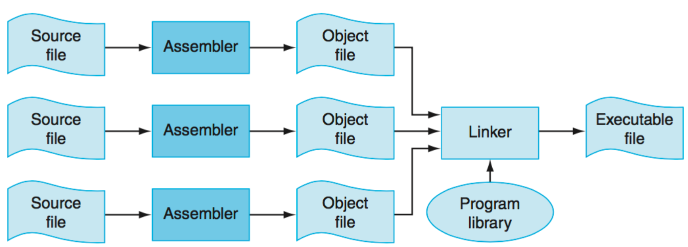
Compiling programs (simplified)

- 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

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.

From Computer Organization and Computer Design: The Hardware/Software Interface

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

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					
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

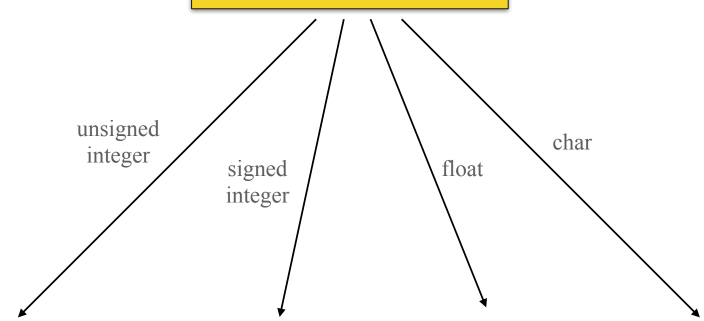
What is the output?

```
#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;
}
```


Variables are just bit sequences

`1000 1110 0011 0001 0011 0001 0011 0100`

0x8E313134



2385588532 −1909378764 −2.1841e−30f “411?”

Memory organization and pointers

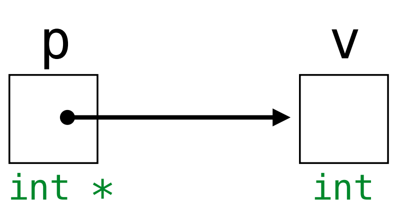
Memory organization

- 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.

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



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

```
*
```

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;
```


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		
…		

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		
…		

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		
…		

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 * 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

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
```


Example: changing a pointer within a function

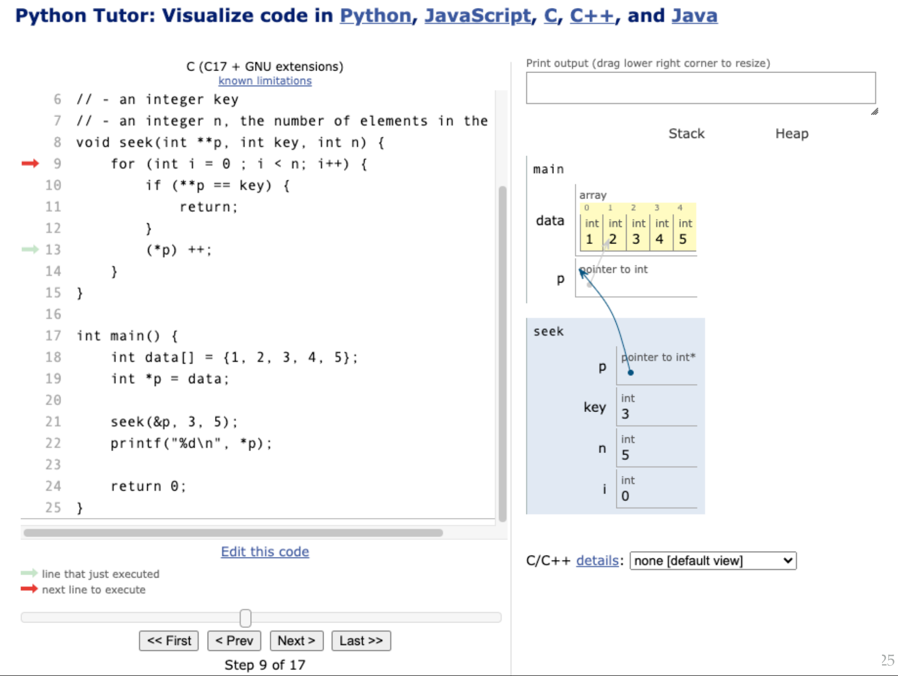
```
#include <stdio.h>

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(data, 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.

Example: changing a pointer within a function

```
// function to search for a key in an array
// 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, pass a pointer to the pointer.



Pointer safety and best practices in C++

- Null pointer initialization
  - proper initialization of pointers is crucial
  - use the modern `nullptr` keyword, which provides type safety and clarity over older methods like NULL or 0
- Memory leaks
  - occur when dynamically allocated memory isn't properly freed
- Dangling pointers
  - occur when they reference memory that has been freed or is no longer valid
- Buffer overflow
  - occur when pointers access memory beyond allocated boundaries, potentially corrupting adjacent memory or crashing
- Pointers and arrays
  - arrays decay to pointers to their first element in most contexts
  - array names are **constant** addresses (point to the first element)
  - **`sizeof`**: sizeof(array) returns the total size of the entire array in bytes, sizeof(pointer) returns the size of the pointer variable itself (typically 8 bytes on 64-bit systems, 4 bytes on 32-bit systems)
- Safety Guidelines
  - always initialize pointers before use
  - track memory allocation and deallocation carefully
  - validate pointer validity before dereferencing
  - understand the distinction between arrays and pointers