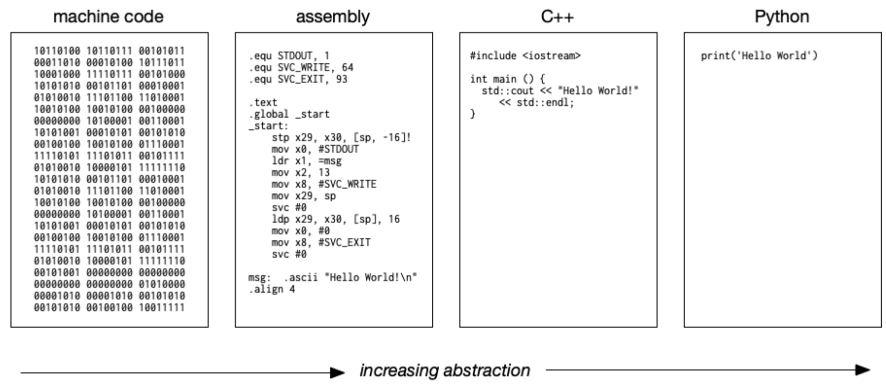
CSC 212

Data Structures and Abstractions (Spring 2025)
C++ Review, Memory, and Pointers

Prof. Marco Alvarez, University of Rhode Island

Context



https://www.uvm.edu/~cbcafier/cs1210/book/02_programming_and_the_python_shell/programming.html

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:

```python
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

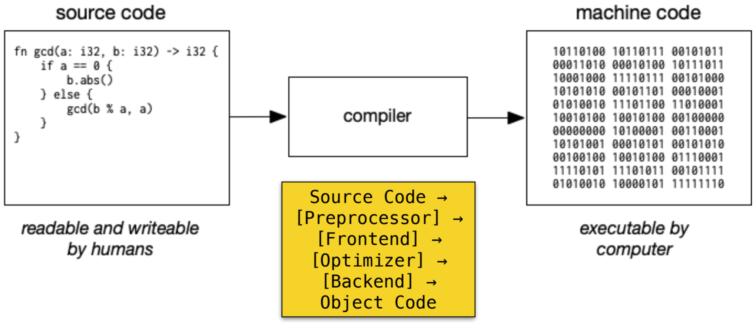
From: "There's plenty of room at the Top: What will drive computer performance after Moore's law? "

Program execution approaches

- Compilation
  - high level source **translated** into another language
    - most of the time into a low-level language
  - as code is translated at once, 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

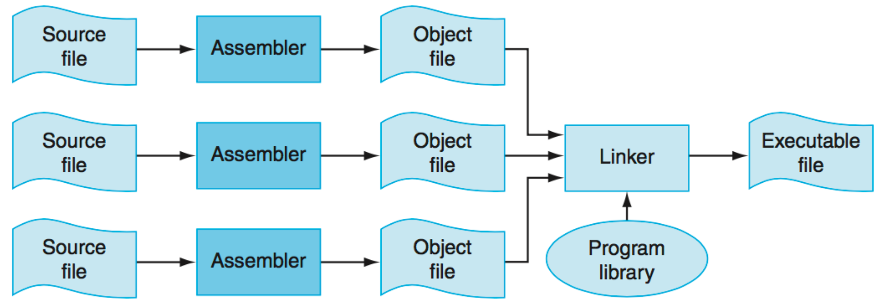
Compiling programs (simplified)

- 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



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

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;
}
```

Range of values (fundamental types)

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

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

Integral 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`						

Memory organization

Memory organization

- Memory as a **byte array**
  - used to store **data and instructions** for computer programs
  - contiguous sequence of bytes
  - each byte individually accessed via a **unique address**
- Memory address
  - **unique** numerical identifier for each byte in memory
  - **pointer** variables store memory addresses
  - provides indirect access to data stored at that location

Memory organization

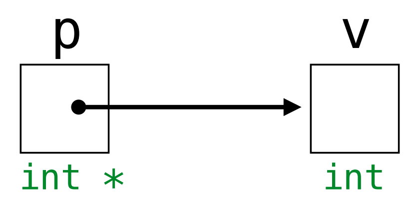
- Data representation in memory
  - variables stored as byte sequences
  - interpretation depends on type
    - integers, floating-point numbers, characters, etc.
- OS provides private address space to each **“process”**
  - process: a program being executed
  - address space: enormous arrays of bytes visible to the process
  - typically implemented through virtual memory

Pointers

Variables and pointers

- Every variable exists at a **memory address** (regardless of **scope**)
  - memory address corresponds to a unique location
- The compiler translates names to addresses when generating machine code

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



Pointers

- Must be declared before use
  - **pointer type** must be specified
- Pointer operators
  - **address-of operator**: get memory address of variable/object

    &

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

    *

Null pointers and arrays

- The NULL value (`0x00000000`)
  - represents the absence of value
  - reading/writting with a null value can generate a **segmentation fault** signal
  - in C++, it is safer to use `nullptr` (keyword)
- Pointers and arrays
  - arrays decay to pointers (to the first element) in most contexts, but they are not themselves pointers
  - array names provide the address of the first element, can't be treated as variables

Declaring pointers

```c
// 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

```c
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

```c
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

```c
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

- Can add values to pointers
  - treats addresses as unsigned integers
- Must be careful !
  - p+1 adds the size of pointed variable
  - p+1 does NOT add 1 "byte"
- Can use pointer arithmetic for array traversal

`a[i]      is equivalent to      *(a+i)`

Changing a pointer inside 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);
    printf("%d\n", *p);

    return 0;
}
```

does it work?

Using double pointers

```
// function to search for a key in an array
// - pointer to an array of integers
// - 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) ++;
    }
}
```

Using double pointers

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

    seek(&p, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

C (C17 + GNU extensions)
known limitations

```c
// - an integer key
// - an integer n, the number of elements in the
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);
    printf("%d\n", *p);

    return 0;
}
```

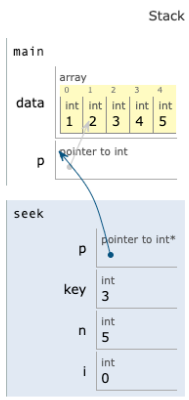
Edit this code

line that just executed
next line to execute

<< First < Prev Next > Last >>

Step 9 of 17

Print output (drag lower right corner to resize)



C/C++ details: none [default view]