

Lecture 0: Introduction to Computer Systems

CS 105

Spring 2026

<https://cs.pomona.edu/classes/cs105/>



Abstraction




Correctness

- **Example 1: Is $x^2 \geq 0$?**
- **Example 2: Is $(x + y) + z = x + (y + z)$?**

Performance

```
void copyij(int src[2048][2048],
            int dst[2048][2048]){
    int i,j;
    for (i = 0; i < 2048; i++){
        for (j = 0; j < 2048; j++){
            dst[i][j] = src[i][j];
        }
    }
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048]){
    int i,j;
    for (j = 0; j < 2048; j++){
        for (i = 0; i < 2048; i++){
            dst[i][j] = src[i][j];
        }
    }
}
```

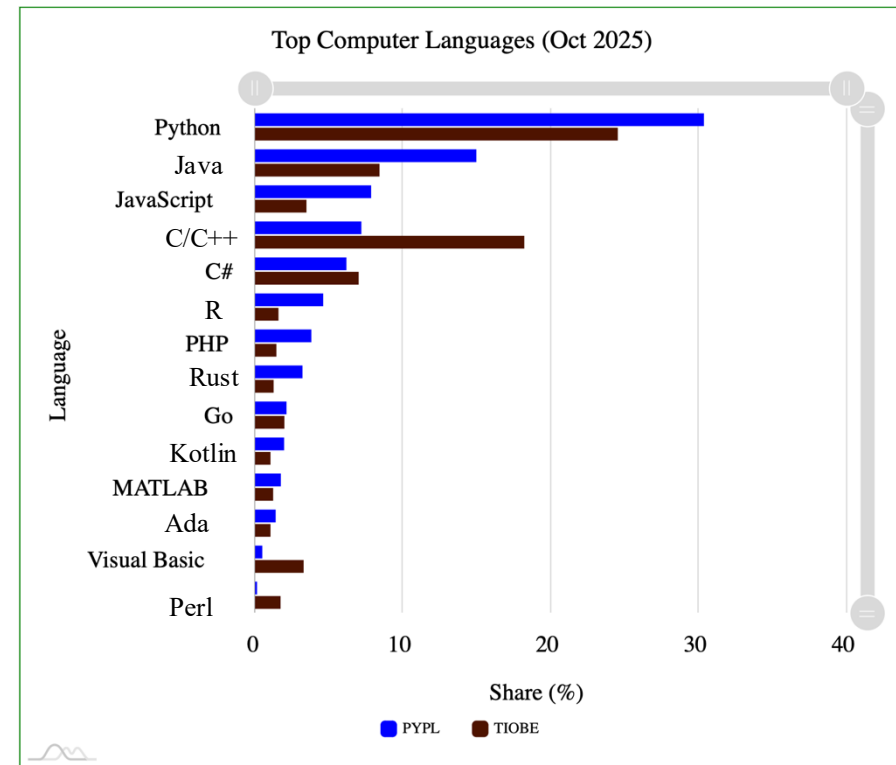
A diagram consisting of two black arrows crossing each other. One arrow originates from the closing brace of the inner loop in the 'copyij' function (the row-major loop) and points towards the opening brace of the inner loop in the 'copyji' function (the column-major loop). The other arrow originates from the opening brace of the inner loop in the 'copyij' function and points towards the closing brace of the inner loop in the 'copyji' function, illustrating the reordering of the nested loops.

Security

```
int buggy_authenticate() {  
    char password[4]; // allocate space to store a string  
    gets(password);   // initialize string from user input  
  
    return 0;         // always returns False  
}  
  
void example3() {  
    if(buggy_authenticate()){ // equivalent to if False  
        printf("The answer is 42\n"); // should never happen  
    } else {  
        printf("Unauthenticated User (correct behavior)\n");  
    }  
}
```

C

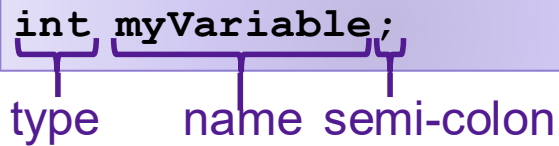
- compiled, imperative language that provides low-level access to memory
 - low overhead, high performance
-
- developed at Bell labs in the 1970s
 - C (and related languages) still commonly used today



Variables

- Declaration

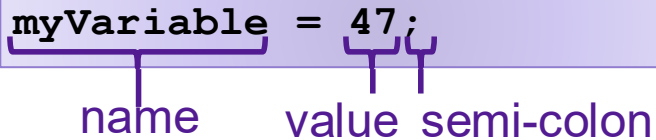
```
int myVariable;
```



The diagram shows the code `int myVariable;` with three purple curly braces underneath. The first brace is under `int` and is labeled `type`. The second brace is under `myVariable` and is labeled `name`. The third brace is under the semicolon `;` and is labeled `semi-colon`.

- Assignment

```
myVariable = 47;
```



The diagram shows the code `myVariable = 47;` with three purple curly braces underneath. The first brace is under `myVariable` and is labeled `name`. The second brace is under `47` and is labeled `value`. The third brace is under the semicolon `;` and is labeled `semi-colon`.

- Declaration and assignment

```
int myVariable = 47;
```

Operations

- Arithmetic Operations: +, -, *, /, %

```
int x = 47;  
int y = x + 13;  
y = (x * y) % 5;
```

- Boolean Operators: ==, !=, >, >=, <, <=

```
int x = (13 == 47);
```

- Logical Operations: &&, ||, !

```
int x = 47;  
int y = !x;  
y = x && y;
```

- (also bitwise operations... more on that next week)

Control Flow

Conditionals

```
int x = 13;
int y;
if (x == 47){
    y = 1;
} else {
    y = 0;
}
```

Do-While Loops

```
int x = 47;
do {
    x = x - 1;
} while (x > 0);
```

While Loops

```
int x = 47;

while (x > 0){
    x = x - 1;
}
```

For Loops

```
int x = 0;
for (int i=0; i < 47; i++){
    x = x + i;
}
```

Functions

Declaring a Function

```
int myFunction(int x, int y){  
  
    int z = x - 2*y;  
    return z * x;  
  
}
```

Calling a Function

```
int a;  
  
a = myFunction(47, 13);
```

Exercise 1

- Define a function `sum_interval` that takes two integers and returns an integer. If the second integer argument is greater than (or equal to) the first, it returns the sum of the integer values between those two numbers (inclusive). Otherwise it returns -1.

Main Functions

- By convention, main functions in C take two arguments:
 1. `int argc`
 2. `char** argv`
- By convention, main functions in C return an int
 - 0 if program exited successfully

```
int main(int argc, char** argv){  
    // do stuff  
  
    return 0;  
}
```

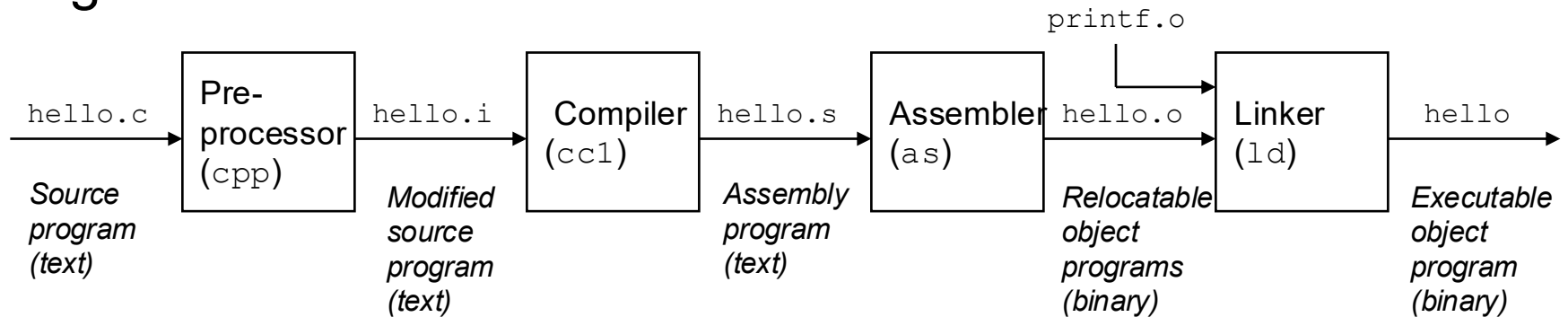
Printing

```
printf("Hello world!\n");  
  
printf("%d is a number\n", 13);  
  
printf("%d is a number greater than %f\n", 47, 3.14);
```

Compilation

compiler output name filename

- gcc -o hello hello.c



```
#include<stdio.h>

int main(int argc,
        char ** argv){

    printf("Hello
           world!\n");
    return 0;
}
```

```
...
int printf(const char *
           restrict,
           ...)
    __attribute__((__format__
    (__printf__, 1, 2)));
...
int main(int argc,
        char ** argv){

    printf("Hello
           world!\n");
    return 0;
}
```

```
pushq    %rbp
movq     %rsp, %rbp
subq     $32, %rsp
leaq     L_.str(%rip), %rax
movl     $0, -4(%rbp)
movl     %edi, -8(%rbp)
movq     %rsi, -16(%rbp)
movq     %rax, %rdi
movb     $0, %al
callq    _printf
xorl     %ecx, %ecx
movl     %eax, -20(%rbp)
movl     %ecx, %eax
addq     $32, %rsp
popq     %rbp
retq
```

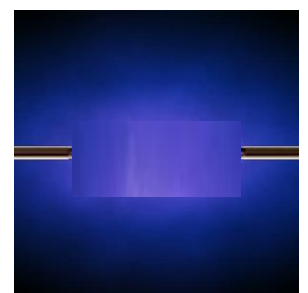
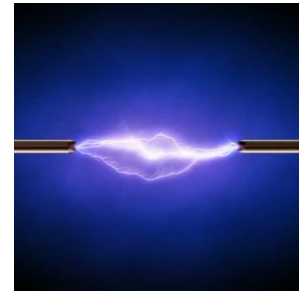
```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Running a Program

- `./hello`

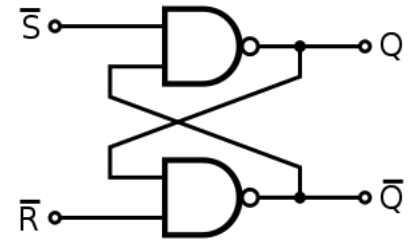
Bits

- a **bit** is a binary digit that can have two possible values
- can be physically represented with a two state device



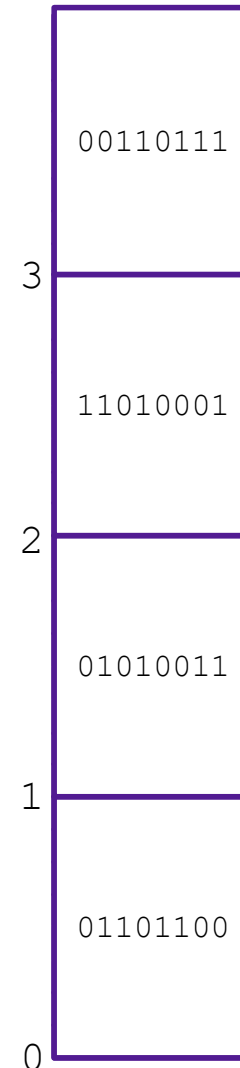
Storing bits

- Static random access memory (SRAM): stores each bit of data in a flip-flop, a circuit with two stable states
- Dynamic Memory (DRAM): stores each bit of data in a capacitor, which stores energy in an electric field (or not)
- Magnetic Disk: regions of the platter are magnetized with either N-S polarity or S-N polarity
- Optical Disk: stores bits as tiny indentations (pits) or not (lands) that reflect light differently
- Flash Disk: electrons are stored in one of two gates separated by oxide layers



Bytes and Memory

- **Memory** is an array of ^{bytes}~~bits~~
- A **byte** is a unit of eight bits
- An index into the array of memory is an **address**, **location**, or **pointer**
 - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
 - The value may be a single byte ...
 - ... or a multi-byte quantity starting at that address



Example C Types

C Data Type	x86-64
int	4
double	8
char	1

Pointers

- Pointers are addresses in memory (i.e., indexes into the array of bytes)
- Most pointers declare how to interpret the value at (or starting at) that address
- Example:

```
int myVariable = 47;  
int* ptr = &myVariable;
```

- Dereferencing pointers:

```
int var2 = *ptr
```

Pointer Types	x86-64
void*	8
int*	8
char*	8
:	8

& is an "address of" operator
* is a "value at" operator

& and * are inverses of one another

Exercise 2

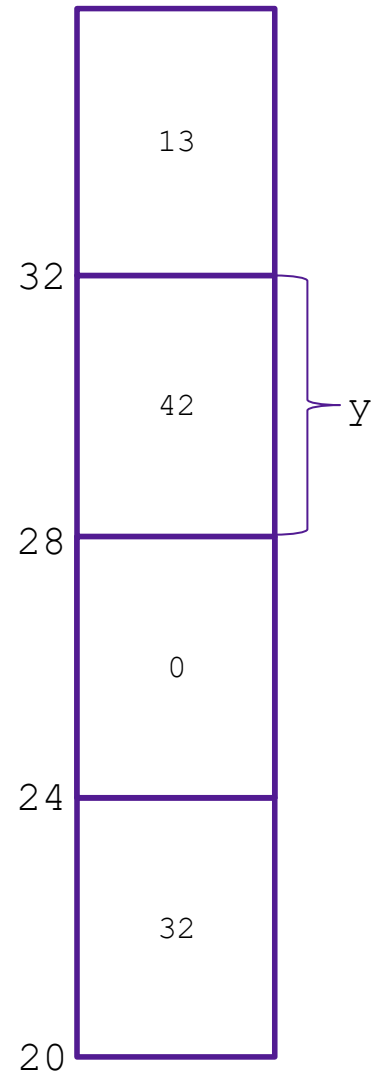
What does x evaluate to in each of the following?

1. `int* ptr = 32;`
`x = *ptr;`

2. `int y = 42; // assume allocated at address 28`
`x = &y;`

3. `int* ptr = 20;`
`x = **ptr; // same as *(*ptr)`

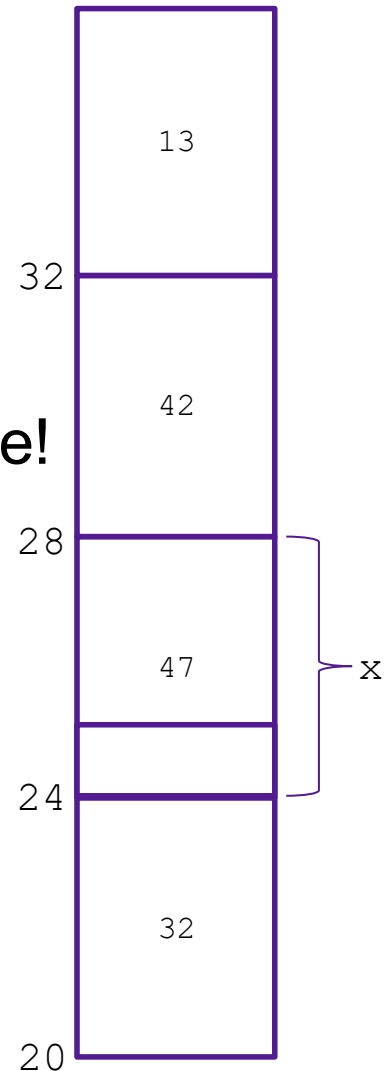
4. `int* x = 24;`
`*x = 47;`



Casting between Pointer Types

- You can cast values between different types
- This includes between different pointer types!
- Doesn't change value of address
- Does change what you get when you dereference!
- Example:

```
int x = 47; // assume allocated at address 24
int* ptr = &x;
char* ptr2 = (char*) ptr;
int y = *ptr
char c = *ptr2;
```



Pointer Arithmetic

```
int* ptr = &myVariable;  
char* ptr2 = (char*) ptr;  
  
ptr += 1;  
ptr2 += 1;
```

- Location of `ptr+k` depends on the type of `ptr`
- adding 1 to a pointer `p` adds `1*sizeof(*p)` to the address

Exercise 3

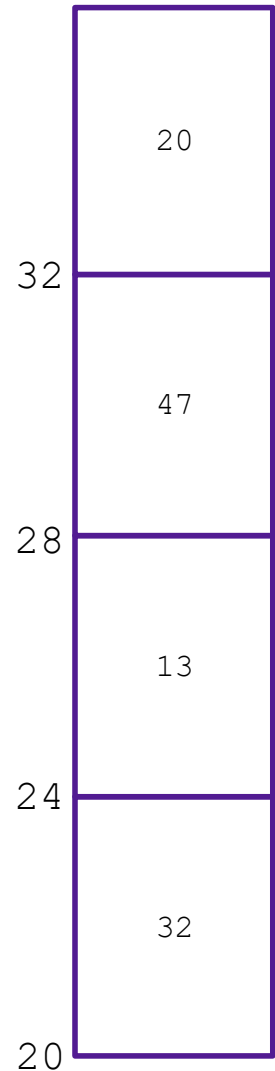
What does x evaluate to in each of the following?

1. `int* ptr = 20;`
`int* x = ptr+1;`

2. `int* ptr = 20;`
`int x = *(ptr+2)`

3. `char* ptr = 20;`
`char* x = ptr+1;`

4. `char* ptr = 20;`
`int x = *((int*)(ptr + 4));`



Arrays

- Contiguous block of memory
- Random access by index
 - Indices start at zero

- Declaring an array:

```
int array1[5]; // array of 5 ints named array1  
  
char array2[47]; // array of 47 chars named array2  
  
int array3[7][4]; // two dimensional array named array3
```

- Accessing an array:

```
int x = array1[2]; // array[k] is the same as *(array+k)
```

- Arrays are pointers!
 - The array variable stores the address of the first element in the array

Strings

- Strings are just arrays of characters
 - aka strings are just pointers
- declared as type `char*`
- End of string is denoted by null byte `\0`

Structs

- Heterogeneous records, like objects
- Typical linked list declaration:

```
typedef struct node {  
    int value;  
    struct node* next;  
} node_t;
```

- Usage:

```
node_t n;  
n.value = 42;  
n.next = NULL;
```

- Usage with pointers:

```
node_t* p;  
p->value = 42;  
p->next = NULL;
```

`p->next` is an
abbreviation for
`(*p).next`

LOGISTICS

The Course in a Nutshell

- Two Professors (Prof. Birrell + Prof. Thomas) + 4 TAs
- Textbooks (not required)
 - Bryant and O'Halloran, *Computer Systems: A Programmer's Perspective*, **third edition**, Pearson, 2016
 - Arpaci-Dusseau and Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, online, 2018
- Classes
 - Monday and Wednesday, 2:45-4pm in Edmunds 101
- Labs
 - Wednesdays 7-8:15pm in Edmunds 229 ← **Starts today**
 - Thursdays 9:35-10:50am in Edmunds 105 ← **Starts tomorrow**
- Office Hours TBD
- Mentor Sessions TBD

Grading

- Assignments (10)

- Introduced during labs, Due Tuesdays at 11:59pm
- Tremendous fun, work in pairs
- 10 late days

- Check-ins (5)

- 2-3 question quizzes (13 topics total)
- February 18/19, March 11/12, April 8/9, April 23/24, May 4
- Can improve grade on any topics(s) with "Extra Chance Check-in" (may take after any later check-in or during final exam time)
- Final Exam Time: May 14 @ 2-5pm (most people), May 7 @ 2-5pm (grads)

- Grades

- Must successfully complete all the assignments
- Beyond that, 50% assignments, 45% check-ins, 5% participation

Course website

<https://cs.pomona.edu/classes/cs105>



- All information is on the course website
- All course materials get posted on the course website
- Links from the course page:
 - Gradescope, for submitting assignments and seeing grades
 - Additional resources