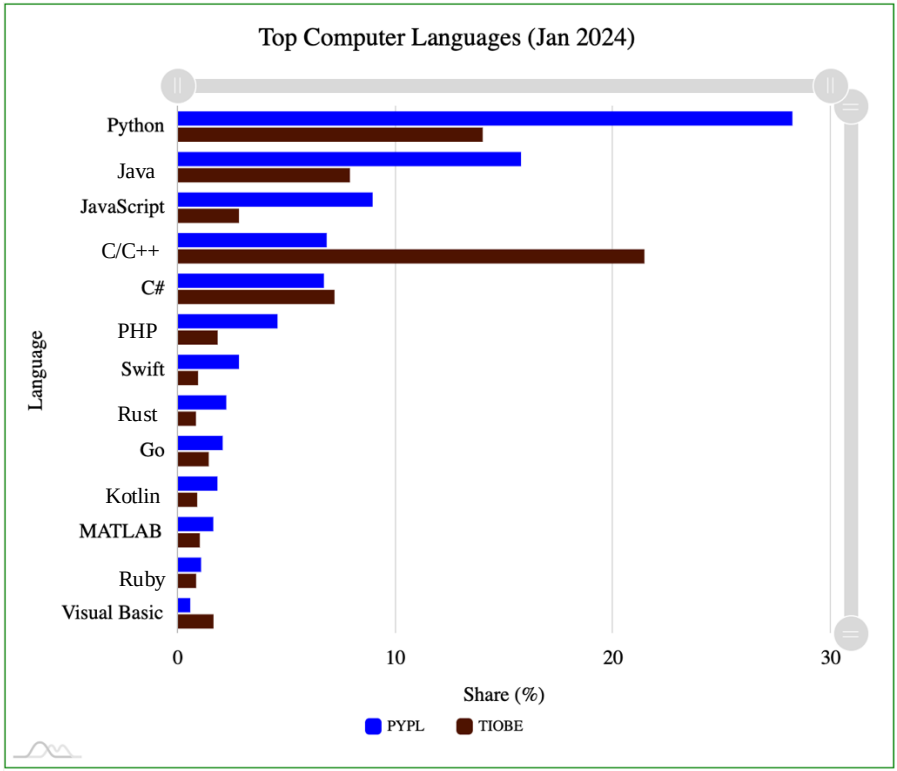
Lab 1: Introduction C

CS 105 Fall 2024

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;
```

type name semi-colon

- Assignment

```
myVariable = 47;
```

name value 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;
```

- Bitwise Binary Operations: &, |, ~, ^

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

Control Flow

Conditionals

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

While Loops

```
int x = 47;

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

Do-While Loops

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

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);
```

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;
}
```

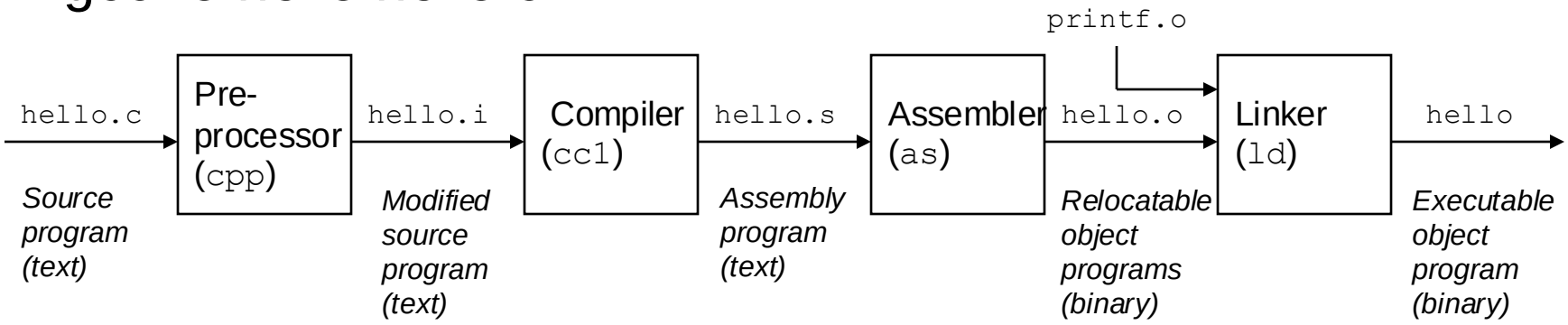
Printing

```
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

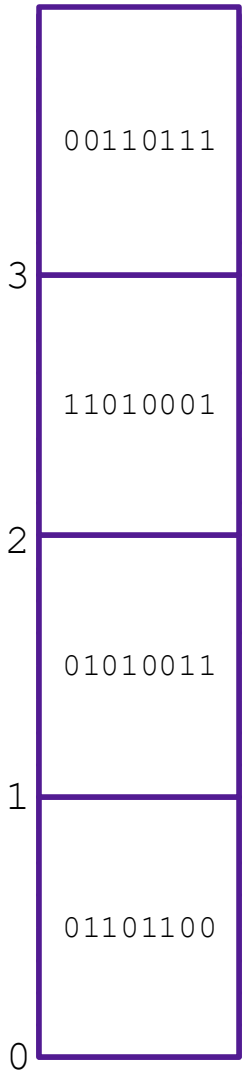
- ./hello

Example C Types

C Data Type	size (in bytes)
int	4
long	8
short	2
char	1
double	8
float	4

Review: Bytes and Memory

- **Memory** is an array of ~~bits~~ bytes
- 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



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
char*	8
int*	8
double*	8
⋮	8

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

& and * are inverses of one another

Exercise

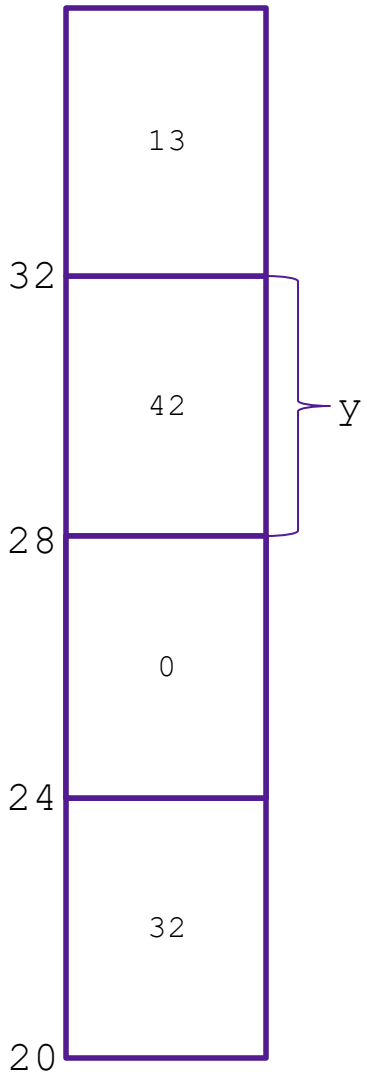
What does x evaluate to in each of the following?

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

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

3.
```
int* x = 24;
*x = 47;
```

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



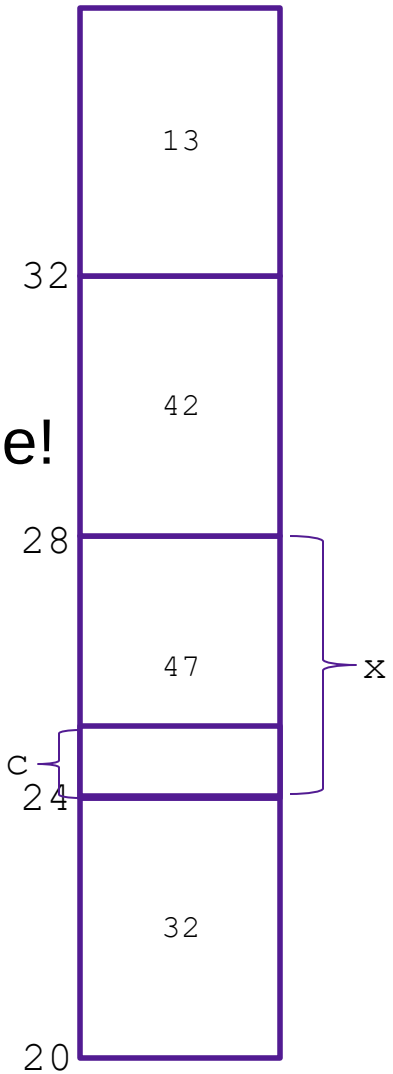
Casting between Pointer Types

- 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; // ptr == 24
char* ptr2 = (char*) ptr; // ptr2 == 24
int y = *ptr; // y == 47
char c = *ptr2; // c == ??
```



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`

Pointer Arithmetic

```
char* ptr = &my_char;    // assume ptr == 32
int* ptr2 = (int*) ptr; // ptr2 == 32

ptr += 1;                // ptr == 33
ptr2 += 1;               // ptr2 == 36
```

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

- **`array[k]`** is the same as **`*(array+k)`**

Exercise 2

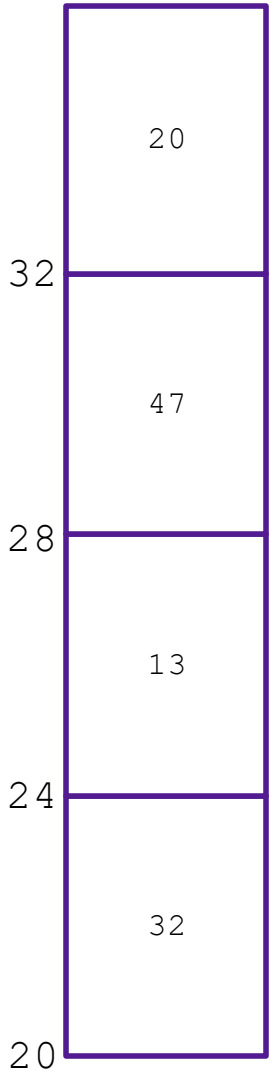
What does x evaluate to in each of the following?

1.
```
int* ptr = 20;
int* x = ptr+2;
```

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

3.
```
char* ptr = 20;
char* x = ptr+2;
```

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



Structs

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

```
typedef struct cell {
  int value;
  struct cell *next;
} cell_t;
```

- Usage:

```
cell_t c;
c.value = 42;
c.next = NULL;
```

- Usage with pointers:

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

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

Exercise 3

```
typedef struct example {
  int y;
  int z;
} example_t;
```

What does x evaluate to in each of the following?

1.
```
example_t* p = 20;
example_t ex = *p;
int x = ex.y;
```

2.
```
example_t* p = 20;
example_t ex = *(p+1);
int x = ex.z;
```

3.
```
example_t* p = 20;
int x = p->y;
```

4.
```
example_t* p = 20;
int x = (p+1)->z;
```