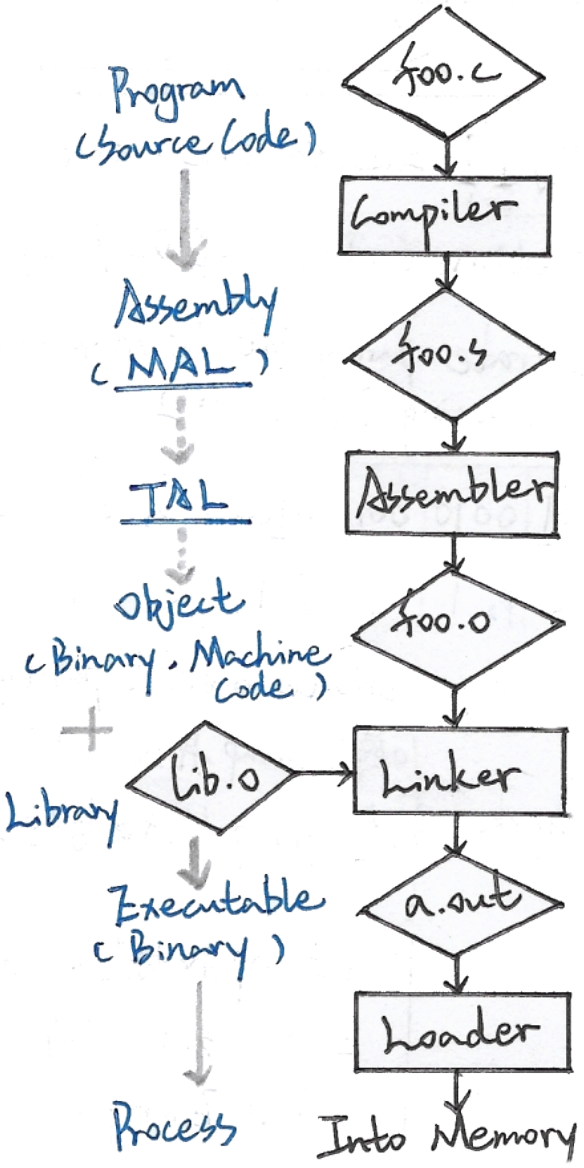
CA I: Discussion #1

See what *we* can *C*!

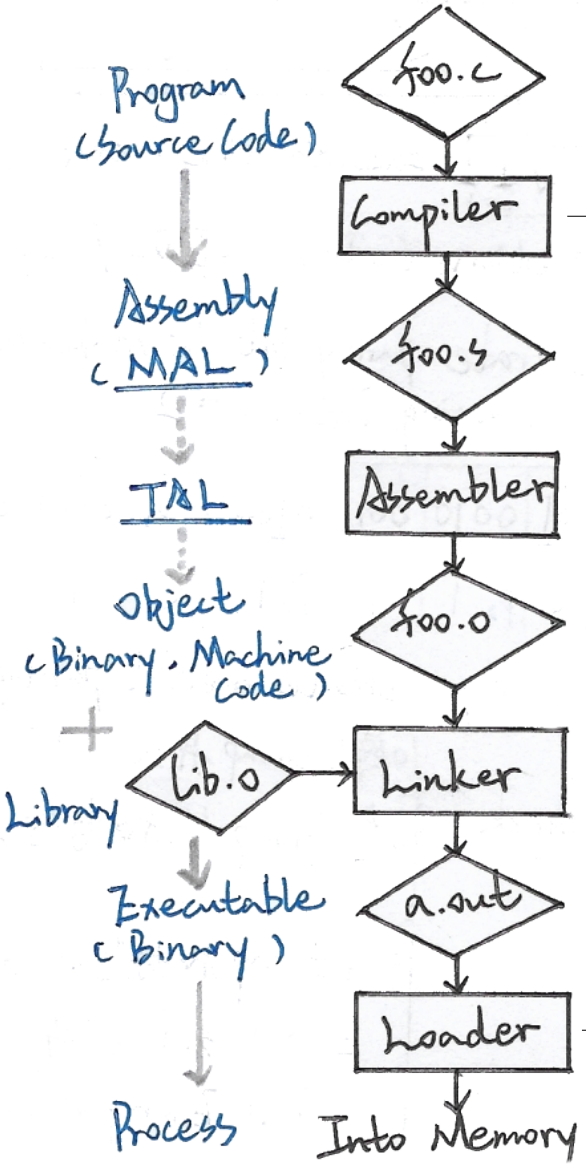
宋延杰、胡冠洲

Inspired by Berkeley CS 61C

An intuition to “CALL”



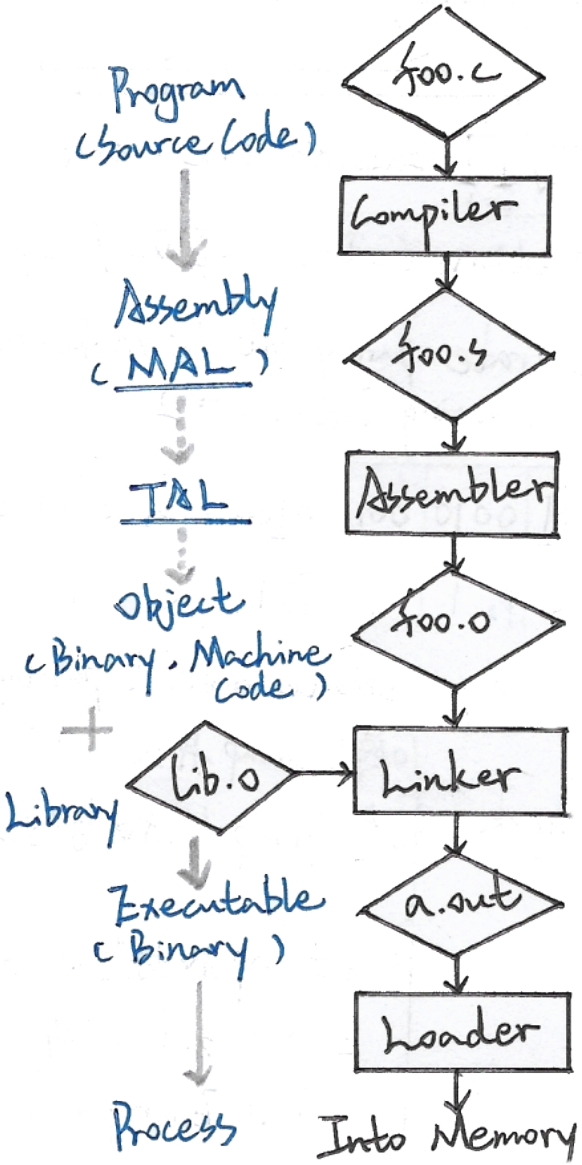
An intuition to “CALL”



→ **Computer Language & Compilers** (CS131)

→ A tiny part of **Operating Systems** (CS130)

An intuition to “CALL”



√ Write programs in high-level language (C)

√ Assembly (RISC-V) syntax & Underlying architecture

√ Assembling

√ Corresponding machine code

√ Relocating & Linking

√ Look inside a CPU

√ CA1 will cover

Contents

* Write Programs in C
  - Number Representation
  - Some Confusing Concepts
  - Debugging Techniques

* This discussion

Number Representation

- Bases
- Binary Signed Numbers
- Floating Point Numbers

Bases

Decimal 十进制

11_{ten}

Convention for human mathematics

Binary 二进制

1011_{two}

Best for electric circuits

Octal 八进制

13_{eight}

Seldom used

Hexadecimal 十六进制

$B_{sixteen}$

A compact form of Binary

Bases

Q: Can we have **Unary** 一进制 ?

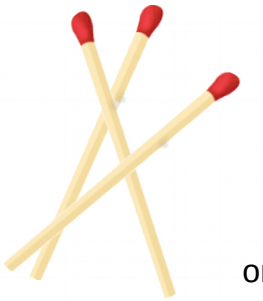
Bases

Q: Can we have **Unary** 一进制 ?

A: Sure we can! ;)

one

1_{ten}



one

3_{ten}

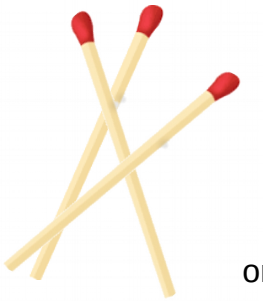
Bases

Q: Can we have **Unary** 一进制 ?

A: Sure we can! ;)

one

1_{ten}



one

3_{ten}

one

183_{ten}

Binary Signed Numbers

Sign & Magnitude

$01011001 = +1011001_{two}$

$11011001 = -1011001_{two}$

One's Complement

(反码)

Rule: -x = ~x

$01011001 = +1011001_{two}$

$10100110 = -1011001_{two}$

Binary Signed Numbers

Sign & Magnitude

$01011001 = +1011001_{two}$

$11011001 = -1011001_{two}$

One's Complement

(反码)

Rule: -x = ~x

$01011001 = +1011001_{two}$

$10100110 = -1011001_{two}$

Two's Complement

(补码)

Rule: -x = ~x + 1

$01011001 = +1011001_{two}$

$10100111 = -1011001_{two}$

100000000

Binary Signed Numbers

Sign & Magnitude

$01011001 = +1011001_{two}$

$11011001 = -1011001_{two}$

One's Complement

(反码)

Rule: -x = ~x

$01011001 = +1011001_{two}$

$10100110 = -1011001_{two}$

Two's Complement

(补码)

Rule: -x = ~x + 1

$01011001 = +1011001_{two}$

$10100111 = -1011001_{two}$

100000000

Q: How many *ZERO*s?

Floating Point Numbers

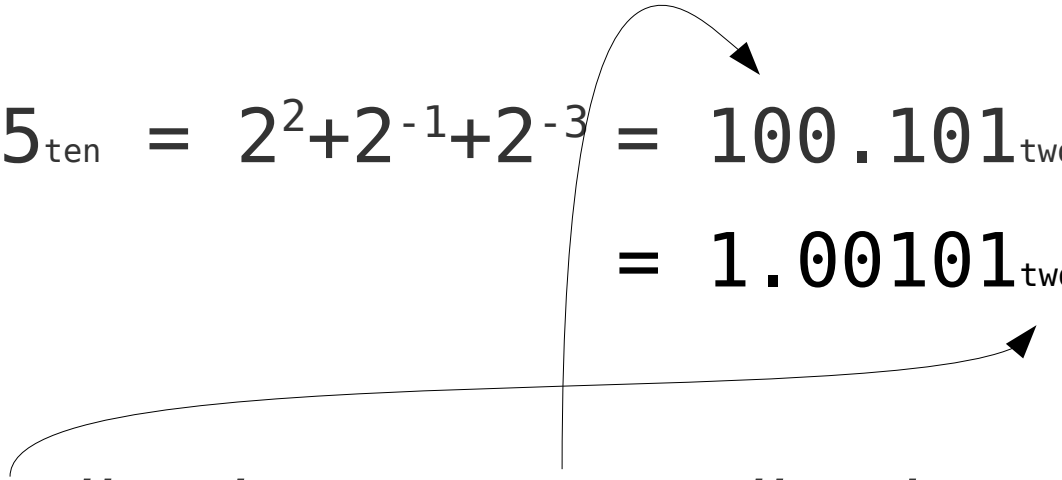
$$4.625_{ten} = 2^{2}+2^{-1}+2^{-3} = 100.101_{two}$$

$$= 1.00101_{two} \times 2^{2}$$

Floating Point Numbers

$$4.625_{ten} = 2^{2}+2^{-1}+2^{-3} = 100.101_{two}$$

$$= 1.00101_{two} \times 2^{2}$$



- *Normalized* vs. *Unnormalized*

Floating Point Numbers

$$4.625_{ten} = 2^{2}+2^{-1}+2^{-3} = 100.101_{two}$$

$$= 1.00101_{two} \times 2^{2}$$

- *Normalized* vs. *Unnormalized*
- Need a standard for this:
  - IEEE 754 (Most widely accepted)
  - You will learn that later in this course

C: Some Clarifications

- Type Qualifiers
- Pointers
- Structures

Type Qualifiers

`static`

- Static variables goes into *.bss/.data* memory segment
- Static functions are limited to be visible only in its own file

```
1  static int a = 1;
2  static int b;
3
4  static void
5  swap(int arr[], unsigned a, unsigned b)
6  {
7      int tmp = arr[a];
8      arr[a] = arr[b];
9      arr[b] = tmp;
10 }
11
12 int
13 main(void)
14 {
```

Type Qualifiers

extern

- Use an external (from another file) global variable
- Needed when multiple source files want to share a global variable

```c
/* print_add_one.c */

#include <stdio.h>
#include "secret.h"     /* `secret_int` defined in this file. */

extern int secret_int;

int
main(void)
{
    printf("%d\n", secret_int + 1);
    return 0;
}
```

```c
/* print_square.c */

#include <stdio.h>
#include "secret.h"     /* `secret_int` defined in this file. */

extern int secret_int;

int
main(void)
{
    printf("%d\n", secret_int * secret_int);
    return 0;
}
```

Type Qualifiers

`volatile`

- Tells the compiler:

  “Do NOT optimize me!”

```
1  int
2  calc_time_elapsed(void)
3  {
4      int i, j;
5      volatile double vd = 0.0;
6
7      /*
8       *  Stopwatch start code.
9       *  ...
10      */
11
12     for (i = 0; i < 10000; ++i)
13         vd += vd * n;
14
15     /*
16      *  Stopwatch pause code.
17      *  ...
18      */
19
```

Pointers

- Why pointers?
  - Operating memory chunks (say, Arrays)
  - Non-copying argument passing

Pointers

- Why pointers?
  - Operating memory chunks (say, Arrays)
  - Non-copying argument passing
  - Function pointers

```
1  float
2  get_add(float a, float b)
3  {
4      return a + b;
5  }
6
7  float
8  get_mul(float a, float b)
9  {
10     return a * b;
11 }
12
13 float
14 operate(float a, float b, float (*func)(float, float))
15 {
16     return (*func)(a, b);
17 }
```

Pointers: Friends & Enemies

Match the following MEMORY SAFETY issues:

Dangling pointer

```
int *ptr = malloc(5 * sizeof(int));
/* Do something and just jump out. */
```

Wild pointer

```
int ptr[5] = {3, 5, 7, 2, 9};
ptr[5] = 11;
```

Index out of bound

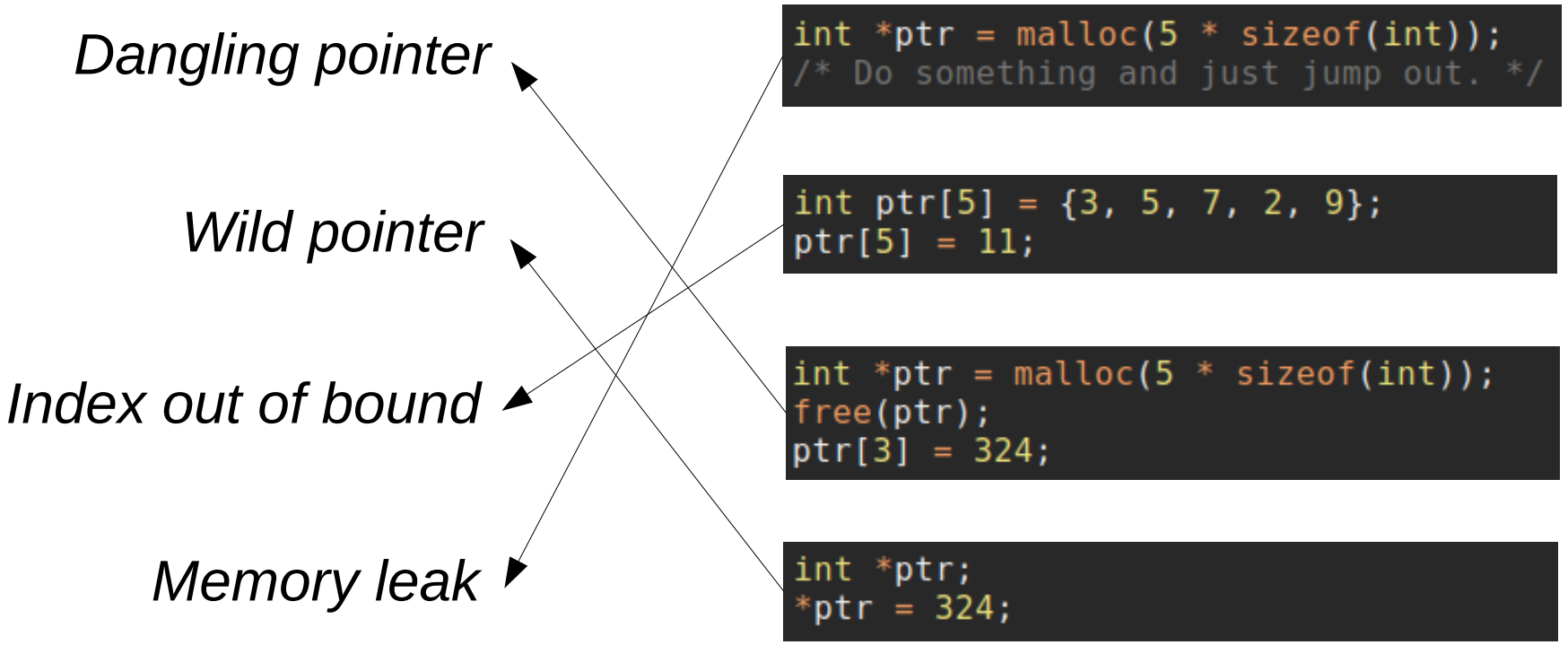
```
int *ptr = malloc(5 * sizeof(int));
free(ptr);
ptr[3] = 324;
```

Memory leak

```
int *ptr;
*ptr = 324;
```

Pointers: Friends & Enemies

Match the following MEMORY SAFETY issues:



Pointers: Friends & Enemies

Match the following MEMORY SAFETY issues:

Structures

`struct`

```c
#include <stdio.h>
#include <inttypes.h>

struct student
{
    uint8_t age;
    int32_t id;     /* `int` triggers alignment */
    char name[13];
};

int
main(void)
{
    printf("%lu\n", sizeof(struct student));

    return 0;
}
```

Q1: The output (in Bytes, of course) will be?

Q2: What if we there's no *`id`* field?

Q3: Where are the members located?

Structures

`struct`

```c
#include <stdio.h>
#include <inttypes.h>

struct student
{
    uint8_t age;
    int32_t id;     /* `int` triggers alignment */
    char name[13];
};

int
main(void)
{
    printf("%lu\n", sizeof(struct student));

    return 0;
}
```

Q1: The output (in Bytes, of course) will be? A1: *24*

Q2: What if we there's no *`id`* field? A2: *14*

Q3: Where are the members located?

Structures

union

```c
#include <stdio.h>
#include <inttypes.h>

union a_number  /* size of a union equals the */
{               /* size of the biggest member */
    uint8_t a;
    int32_t b;
    int64_t c;
};

int
main(void)
{
    printf("%lu\n", sizeof(union a_number));

    return 0;
}
```

Q: The output (in Bytes, of course) will be?

Structures

union

```c
#include <stdio.h>
#include <inttypes.h>

union a_number  /* size of a union equals the */
{               /* size of the biggest member */
    uint8_t a;
    int32_t b;
    int64_t c;
};

int
main(void)
{
    printf("%lu\n", sizeof(union a_number));

    return 0;
}
```

Q: The output (in Bytes, of course) will be?

A: *8*

Structures

Struct with bit-field specifications

```c
#include <stdio.h>
#include <inttypes.h>

struct student
{
    uint8_t age : 5;
    int32_t id : 27;
};

int
main(void)
{
    printf("%lu\n", sizeof(struct student));

    return 0;
}
```

Q: The output (in Bytes, of course) will be?

Structures

Struct with bit-field specifications

```c
#include <stdio.h>
#include <inttypes.h>

struct student
{
    uint8_t age : 5;
    int32_t id : 27;
};

int
main(void)
{
    printf("%lu\n", sizeof(struct student));

    return 0;
}
```

Q: The output (in Bytes, of course) will be? A: *4*

C: Debugging Techniques

- *printf()*
- *assert()*
- *gdb*: the GNU Project Debugger
- *valgrind*

Naive Ways (but sometimes convenient)

printf

你们应该很熟练了

assert

Pre-assert some preliminary conditions, so that some apparent bugs can be caught easily.

```c
#include <stdio.h>
#include <assert.h>

void
print_age(int age)
{
    assert(age >= 0 && age < 150);
}

int
main(void)
{
    print_age(-2);

    return 0;
}
```

GDB: Really Powerful

- Start with following **Berkeley 61C – Lab1**:

  https://cs61c.org/labs/lab01/

  You are no longer naive if you can answer all the questions in *Exercise 2 - ACTION ITEM*.

- **Official Manual** (learn it comprehensively):

  https://sourceware.org/gdb/current/onlinedocs/gdb/

GDB: Really Powerful

- Start with following **Berkeley 61C – Lab1**:

  https://cs61c.org/labs/lab01/

  You are no longer naive if you can answer all the questions in *Exercise 2 - ACTION ITEM*.

- **Official Manual** (learn it comprehensively):

  https://sourceware.org/gdb/current/onlinedocs/gdb/

A streamlined GDB **cheatsheet** I made:

(Download if you need)

https://josehu.com/files/basic-gdb-usage.pdf

Valgrind: Easy, Handy

- Also follow **Berkeley 61C – Lab1**:

  https://cs61c.org/labs/lab01/#exercise-4-valgrinding-away

- **Official Quick-start** (learn it over):

  http://valgrind.org/docs/manual/QuickStart.html

Valgrind: Easy, Handy

- Also follow **Berkeley 61C – Lab1**:

  https://cs61c.org/labs/lab01/#exercise-4-valgrinding-away

- **Official Quick-start** (learn it over):

  http://valgrind.org/docs/manual/QuickStart.html

Valgrind is really a handy and easy tool, only need to:

- Understand the memcheck outputs
- Grasp basic options (e.g. `--leak-check=yes`)

Thanks!

See now what *you* can *C*!

宋延杰、胡冠洲

Inspired by Berkeley CS 61C