

# CS100 Recitation 1

GKxx

# 目录

- 开始
- 编译、运行一个 C 程序
- 初识函数
- `scanf` 和 `printf`
- 算术类型

开始

# 为什么要学“任何东西”？

为什么要学 C/C++ ？

- 为什么要学微积分/离散数学/XXXX/.....？

# 为什么要学“任何东西”？

为什么要学 C/C++ ？

- 为什么要学微积分/离散数学/XXXX/.....？

## 重走一个学科/领域从无到有的发现历程

- 理解学科中的基本思想、基本方法、里程碑、走过的弯路
- 锻炼学习的能力
- 应用、创新、革命 .....

# 怎样学好这门课

关键：把课件、课本等资料上的示例代码独立地实现一遍

- 作业也会有很多直接来自于课件（包括习题课）的题目
  - 看懂、掌握基本的知识
  - （尝试）独立地实现
  - 对比课件上的写法，有什么区别？哪种更好？
- 仅仅做到能“看懂”是**不够的**，只有独立实现才能内化为自己的知识。
  - 学数学也是如此。

# 关于学术诚信

我们采取相对宽松的规则。

- 简单题、教学题不查重，鼓励模仿和借鉴**好的写法、好的思路**。
  - 部分题目会设置与 TA 的线下 check 。
- **允许**帮别人 debug 。
- 鼓励使用 AI 工具辅助学习，比如让 ChatGPT/Copilot 帮你解释代码。
  - 注意谨慎求证。
- 但是禁止**直接抄袭**来自他人、网络、 AI 生成的代码。

## 关于 TA

有问题找 TA ， 任何问题都可以问。我们会在合理的范围内回答、提供帮助。



# 编译、运行一个 C 程序

# 编译、运行一个 C 程序

```
#include <stdio.h>

int main(void) {
    printf("Hello, world\n");
    return 0;
}
```

`gcc hello.c -o hello` 使用 `gcc` 编译，生成一个可执行文件。

- Windows 上叫 `hello.exe`，Linux/Mac 上叫 `hello`。

`./hello` (Linux/Mac), `.\hello` (Windows) 运行这个程序。

- 当然你也可以双击它，但它会“一闪而过”。

## 环境变量 PATH

当你执行 `gcc hello.c -o hello` 的时候，你唤醒的 `gcc` 是你安装在某处的 `gcc` (Linux/Mac) / `gcc.exe` (Windows)

例如，Windows 上的 `D:\mingw64\bin\gcc.exe`

终端如何认识到你想找的 `gcc` 是 `D:\mingw64\bin\gcc.exe` ？

## 环境变量 PATH

当你执行 `gcc hello.c -o hello` 的时候，你唤醒的 `gcc` 是你安装在某处的 `gcc` (Linux/Mac) / `gcc.exe` (Windows)

例如，Windows 上的 `D:\mingw64\bin\gcc.exe`

终端如何认识到你想找的 `gcc` 是 `D:\mingw64\bin\gcc.exe` ？

- 将 `D:\mingw64\bin` 添加到环境变量 `PATH`。

## 在 VSCode/CLion 的终端中执行以上指令

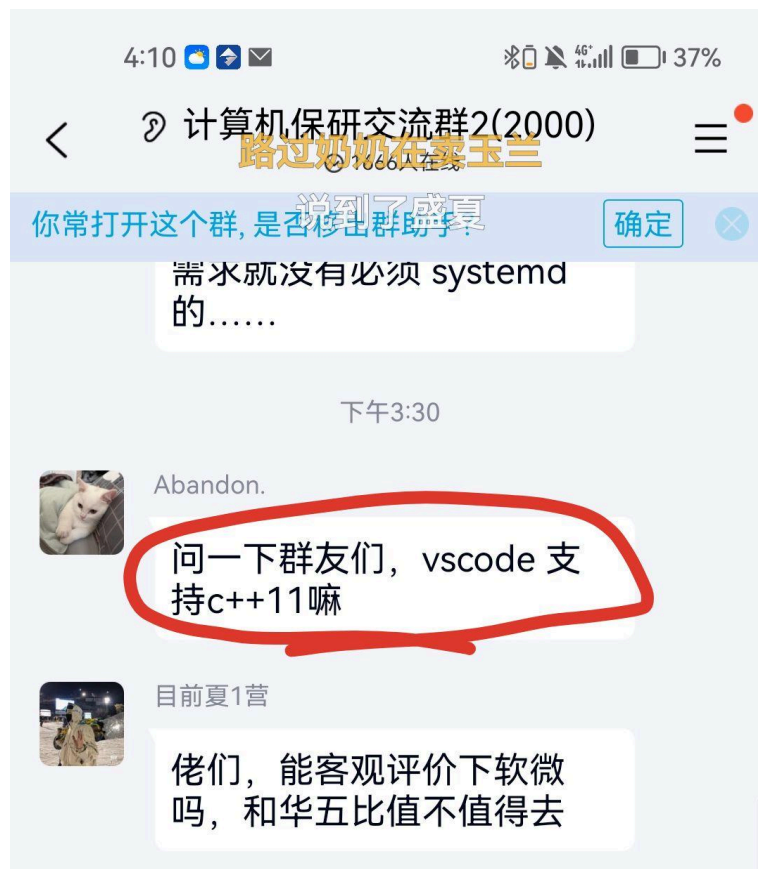
VSCode: `ctrl` + ``` 开启终端，默认 working directory 为当前打开的文件夹。

CLion: 最下面 "Terminal"/"终端"，或 `Alt` + `F12`

可以先 `cd` 到方便的位置（如果需要的话）

# 编译、运行、调试一个 C 程序

VSCode/CLion 本身不具备任何编译、运行、调试的功能！它只是把编译器、可执行文件和 GDB 的结果打在公屏上，然后给你提供一些方便操作的按钮。



# 初识函数

## 定义一个函数

语法： `ReturnType FunctionName(Parameters) { FunctionBody }`

- 数学上，函数是  $f : S \mapsto T$
- 一个函数的**参数**的类型对应  $S$ ，**返回值**类型对应  $T$ 。

例：下取整函数  $f(x) = \lfloor x \rfloor$  的参数和返回值类型分别是什么？



# 定义一个函数

语法： `ReturnType FunctionName(Parameters) { FunctionBody }`

- 数学上，函数是  $f : S \mapsto T$
- 一个函数的**参数**的类型对应  $S$ ，**返回值**类型对应  $T$ 。

例：下取整函数  $f(x) = \lfloor x \rfloor$  的参数和返回值类型分别是什么？

```
int floor(double x) {  
    // ...  
}
```

## 定义一个函数

一个函数可以接受多个参数 ( $f : S_1 \times S_2 \times \cdots \times S_n \mapsto T$ )，**每个参数的类型都需要写出来。**

- 哪怕它和前一个参数类型相同，也不可以省去。 `(int x, int y)` 不可以省略为 `(int x, y)`

```
ReturnType myFunction(Type1 a1, Type2 a2, Type3 a3) {  
    // ...  
}
```

## 定义一个函数

一个函数可以没有返回值，这样的函数返回值类型为 `void`。

和数学函数不同，C 函数不仅仅是“接受一些参数、返回一个值”，它实际上就是将一小段具有特定功能的代码单独拎出来，并且给它起个名字。

# 定义一个函数

一个函数可以不接受参数 ( $f : \emptyset \mapsto T$ )，写法是在参数列表里放一个 `void`

```
ReturnType do_some_work(void) {  
    // ...  
}
```

和 C++ 不同，**参数列表空着并不代表不接受参数**，而是表示 "accepting unknown parameters"。这项规则自 C23 起才被删除。

- 这是 C 的一个历史遗留问题。上古时期的 C 并不注重参数类型检查，这可能更接近汇编的风格，但也更容易出错。

## 另一个历史遗留问题

在 C99 之前，一个函数如果不写返回值类型，则返回值类型默认为 `int`。

事实上在旧的 C 语言中存在着大量的“默认为 `int`”的规则，在很多场合下这种写法极易引发难以预料的问题。

- 不幸的是，为了兼容已经存在的上亿行旧代码，编译器大概率会允许这种写法，并仅仅给出一个 warning。

你可能见过一些人写这样的 `main` 函数：

```
main() {  
    // ...  
}
```

这不符合现代 C 语言的语法，也是极不推荐的写法。

# 更糟糕的历史遗留问题

如果你调用了一个未经声明的函数

```
#include <stdio.h>

int main(void) {
    int x = my_mysterious_function();
    printf("%d\n", x);
}
```

C++ 编译器会给出一个十分正常的报错: "... was not declared in this scope"

```
a.cpp: In function 'int main()':
a.cpp:4:11: error: 'my_mysterious_function' was not declared in this scope
   4 |     int x = my_mysterious_function();
     |               ^~~~~~
```

# 更糟糕的历史遗留问题

如果你调用了一个未经声明的函数

```
#include <stdio.h>

int main(void) {
    int x = my_mysterious_function();
    printf("%d\n", x);
}
```

而 C 编译器会允许，并且给出一个令人困惑的 warning: "implicit declaration"

```
a.c: In function 'main':
a.c:4:11: warning: implicit declaration of function 'my_mysterious_function'
[-Wimplicit-function-declaration]
    4 |     int x = my_mysterious_function();
      |           ^~~~~~~~~~~~~~~~~~~~~~
```

当然，它是无法运行的，链接器会抱怨“找不到名为 `my_mysterious_function` 的函数”。

## 更糟糕的历史遗留问题

如果你调用了一个未经声明的函数

```
#include <stdio.h>

int main(void) {
    int x = my_mysterious_function();
    printf("%d\n", x);
}
```

C 标准认为你“隐式地声明” (implicitly declare) 了这个函数，于是压力全都给到链接器。

这一规则 C99 起被删除，但为了向后兼容 (backward compatibility)，编译器很可能仍然支持，只是给一个 warning。



## `main` 函数的返回值

函数的**返回值**是传给它的调用者的。

- `main` 的“调用者”是谁？

## main 函数的返回值

函数的**返回值**是传给它的调用者的。

- `main` 的“调用者”是谁？——程序的调用者/“宿主环境” (hosted environment)

`main` 函数返回 `0` ——程序正常退出

`main` 函数返回其它值——代表某种错误

- 一些程序会用不同的返回值表达不同的含义。
- 也有的程序只是在“任何错误”发生时都返回 `1` 。

## scanf 和 printf

虽然文档有点复杂，但也要试着在这里检索你需要的信息。

scanf

printf

## "A+B" problem

```
#include <stdio.h>

int main(void) {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n", a + b);
    return 0;
}
```

## A+B Problem

约定输入格式：两行，每行一个整数，分别表示 `a` 和 `b`。

- `scanf("%d\n%d\n", &a, &b)` 会发生什么？试一试。

# A+B Problem

约定输入格式：两行，每行一个整数，分别表示 `a` 和 `b`。

- `scanf("%d\n%d\n", &a, &b)` 会卡住，直到你输入了一个非空白字符为止。

- 看看标准：

whitespace characters: any single whitespace character in the format string **consumes all available consecutive whitespace characters** from the input (determined as if by calling `isspace` in a loop). Note that **there is no difference** between `"\n"`, `" "`, `"\t\t"`, or other whitespace in the format string.

- whitespace character: “空白字符”，包括空格、换行、回车、制表等。

## A+B Problem

约定输入格式：两行，每行一个整数，分别表示 `a` 和 `b`。

- `scanf("%d%d", &a, &b)` 可以吗？试一试。

# A+B Problem

约定输入格式：两行，每行一个整数，分别表示 `a` 和 `b`。

- `scanf("%d%d", &a, &b)` 可以。
- 中间的换行符不写也可以？看看标准：

`"%d"` matches a decimal integer.

The format of the number is the same as expected by `strtol` with the value 10 for the base argument.

- `strtol` 是什么鬼？[点进去看看](#)：

Interprets an integer value in a byte string pointed to by `str`.

**Discards any whitespace characters** (as identified by calling `isspace`) until the first non-whitespace character is found, then...



## printf

double 需要用 %lf 输出。如果我用 %d 输出，会怎样？

```
#include <stdio.h>

int main(void) {
    double pi = 3.14;
    printf("%d\n", pi);
    return 0;
}
```

它和先将 pi 转换为 int（小数部分截去）再输出的效果一样吗？

```
double pi = 3.14;
int pi_int = pi; // pi_int 值为 3
printf("%d\n", pi_int);
```

## printf

在 `scanf` 和 `printf` 中，conversion specifier 与对应的变量类型不匹配是 **undefined behavior**：你不能对程序的行为做任何假定。

以下说法统统是错的：

- 它会进行某种类型转换再输出
- 它会输出一个值，只是我不知道它是多少罢了

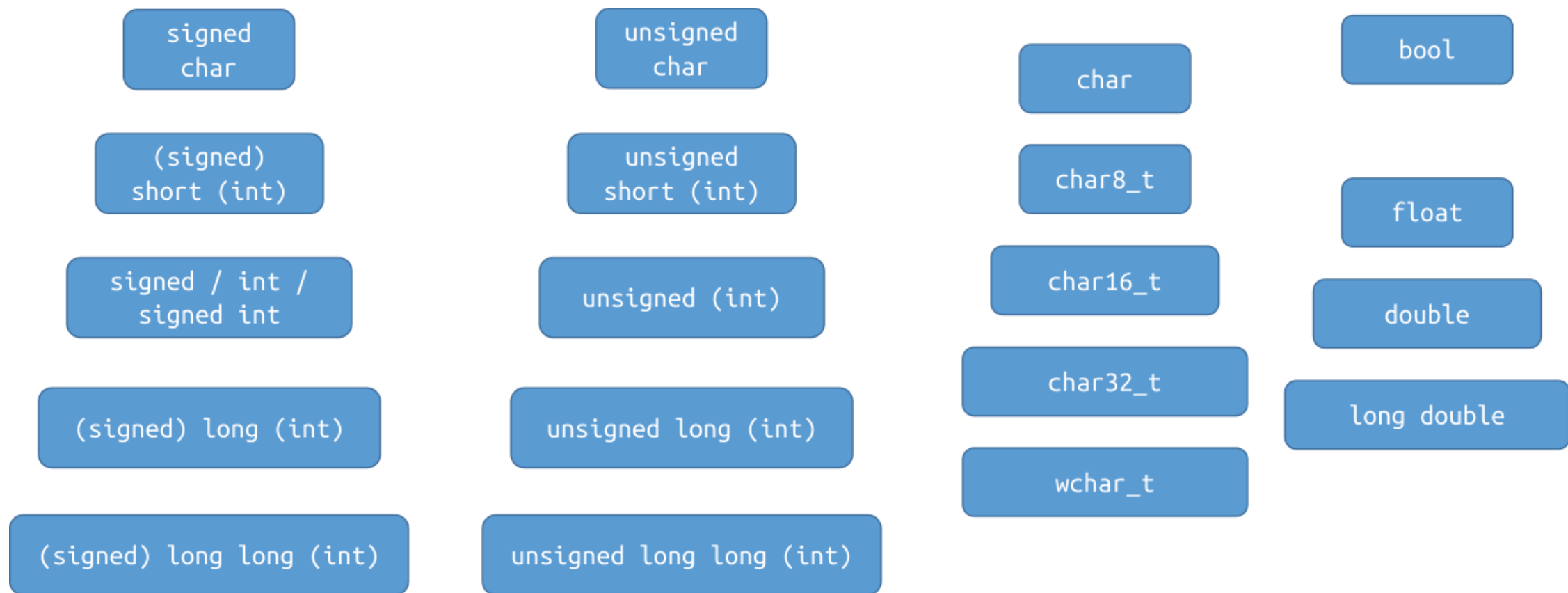
程序完全有可能在这里直接崩溃。此外，编译器可以假定你的程序不含 undefined behavior，所以它如果发现这里有 undefined behavior，就可以直接把这行代码删掉，这完全符合语言标准。

# 算术类型

正课 slides 已经足够完整，这里仅作为回顾。

# 算术类型：整数，字符，布尔，浮点数

有时候字符和布尔也认为是整数。



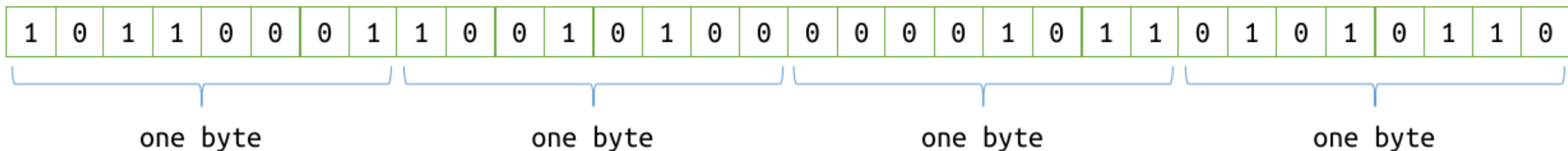
# 整数类型的大小

type	width (at least)	width (usually)
<code>short</code>	16 bits	16 bits
<code>int</code>	16 bits	32 bits
<code>long</code>	32 bits	32 or 64 bits
<code>long long</code>	64 bits	64 bits

- `sizeof(signed T) == sizeof(unsigned T)`
- `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

# 位 (bit) 和字节 (byte)

一个 32 位整数长这样：



通常一个字节是 8 位。 `sizeof(T)` 可以获得 `T` 所占的字节数量。

```
printf("%zu\n", sizeof(double));
```

- 据说 MinGW 对 `%zu` 的支持有问题，不行就把结果转成 `int` 再用 `%d`

```
printf("%d\n", (int)sizeof(double));
```

## 浮点数不是万能的

定义一个函数，接受两个 64 位整数，返回它们的平均值。你会使用什么返回值类型？

## 浮点数不是万能的

定义一个函数，接受两个 64 位整数，返回它们的平均值。你会使用什么返回值类型？

看起来唯一的选择是 `double`

```
double midpoint(long long a, long long b) {  
    return (a + b) / 2.0;  
}
```

它对吗？试试看 `9007199254740993 9007199254740993`



## 浮点数误差是难免的

判断一个浮点数是否等于零，必须使用  $|x| < \epsilon$ ，其中  $\epsilon$  可以是  $10^{-8}$  这样的一个小数。

判断两个浮点数是否相等，必须使用  $|a - b| < \epsilon$ 。