CSE/CS102

计算机组成与系统

第 1 讲

Introduction to C

- 薛浩
- **stickmind.com**

话题

- C 语言
- GCC 编译器
- make 构建工具

C 语言

C 语言

C 语言是贝尔实验室的 Dennis Ritchie 在 1970 年左右创建的。作为 Unix 系统的程序语言，伴随着 Unix 的发明而出现。

C 语言的成功与其设计哲学密切相关：

- 移植性：和 Unix 的密切关系，方便程序移植到新的设备上
- 简洁性：仅对硬件做了简单的抽象
- 实践性：为实现 Unix 而生，是系统编程的首选

C 语言一直位居 TIOBE 指数的前两名，TIOBE 指数是衡量编程语言受欢迎程度的指标。

语言异同

大部分现代编程语言 C++/Python/Java 都是基于 C 语言发展而来，基本的语法规则大同小异：

- 语法
- 基本数据类型
- 算术、关系、逻辑运算符

和 C++ 相比，C 语言的局限性主要有以下几点：

- 没有高级特性，比如运算符重载、默认参数、引用传参、类和对象、抽象数据类型等
- 没有大量的库，比如图形库、网络库等
- 编译器几乎不提供运行时检查，可能造成严重的安全问题

正因为如此，C 语言是一门不太复杂的语言，只需要一两百页的书就能够讲清楚。这里借用 K&R C 前言中的一句话：“C 是一门越用越顺手的语言”。

FAQ 为什么还要学习 C 语言?

- C 语言仍然是非常流行的编程语言，常年占据 TIOBE 排行榜前三
- 大量的工具，甚至是其他编程语言（Python、Lua）都是由 C 语言编写
- C 语言是开发快速、高效软件的有力竞争者
- C 语言允许你在更低的抽象层操作数据，有利于理解系统的工作原理
- 学好 C 语言，可以更好地学习 CSAPP、OS、Network 等进阶课程

第一个 C 程序

```c
/* File: hello.c
 * -------------
 * This program prints a welcome message to the user.
 */

#include <stdio.h> // for printf

int main(int argc, char* argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

编写、编译、调试、运行

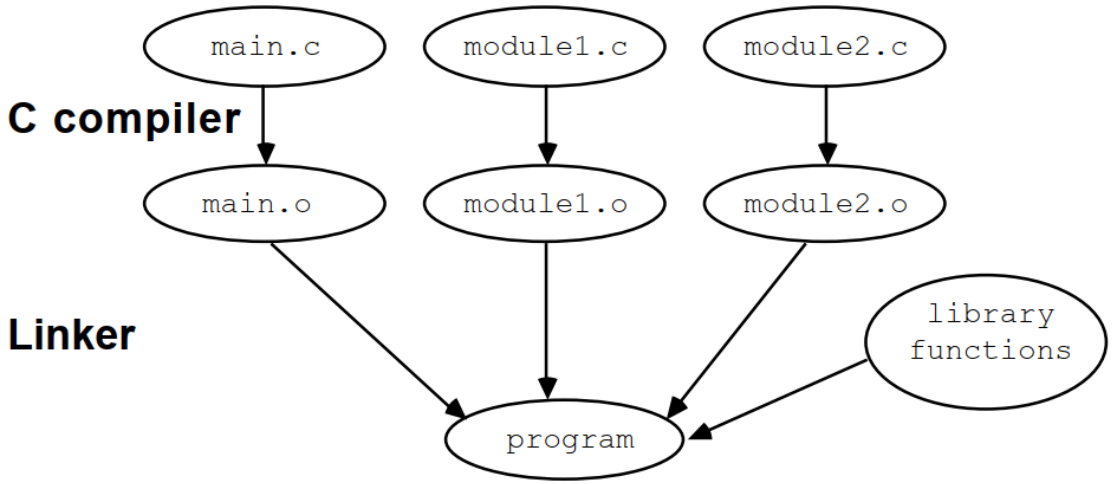
利用第一个 C 程序，演示在 Linux 环境中的工作流程：

- **ssh** - 连接远程服务器
- **vscode** – 尝试学会一款命令行编辑器，Vim/Emacs 任选其一
- **gcc** - 本课程不再使用集成开发环境，而是使用 GCC 编译器，这是使用最广泛的编译器之一
- **gdb** - 本课程最棘手的地方是没有便利的图形化调试工具，所有作业都需要使用纯粹的命令调试方式。不过在实验环节，我们会提供大量的 gdb 调试训练
- **./hello** - 运行程序

GCC

构建过程

- 有了 C 的源码后，需要经过**编译**（compiling）和**链接**（linking）两个步骤才能得到可执行的程序
- **编译**过程将 .c 的源码转成成 .o 的目标文件
  - 目标文件是平台独立的，名称和源码相同
  - 例如 main.c 的目标文件是 main.o
- **链接**过程是将多个目标文件，以及用到的库函数一起打包成一个独立的、可执行的文件



GNU Compiler Collection

- GNU 编译器套装简称 GCC，由 GNU 项目创建，为自由软件的创建提供了有力的支撑
- GCC 包含编译器和链接器，可以编译多种语言，例如 C\C++\Rust 等

小型项目的构建

```c
// hello.h
#include <stdio.h> // for printf
```

```c
// hello.c
#include "hello.h"

int main(int argc, char *argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

- 对于一些比较小的测试项目，可以使用一条命令完成编译和链接的过程
  - `gcc hello.c -o hello`
- 如果头文件不在当前目录，需要使用 `-I` 指定路径
  - `gcc hello.c -o hello -I./include`
- 使用 `-c` 可以单独编译目标文件，再通过 `-o` 链接成独立文件
  - `gcc -c hello.c`
  - `Gcc hello.o -o hello`

大型项目的构建

- 大型项目一般会用多个文件夹管理项目文件
- 此时，如果仍然使用 gcc 命令构建程序，将涉及到多个命令选项
  - -c 生产目标文件
  - -o 指定可执行文件名称
  - -I 指定头文件路径（大写 i）
  - -L 指定库文件路径
  - -l 指定链接库名称（小写 L）

```
piglatin
├── cslib
│   ├── include
│   │   ├── exception.h
│   │   ├── genlib.h
│   │   ├── random.h
│   │   ├── simpio.h
│   │   └── strlib.h
│   └── libSimpleCSLib.a
├── include
│   └── tokenscanner.h
└── src
    ├── piglatin.c
    └── tokenscanner.c

4 directories, 9 files
```

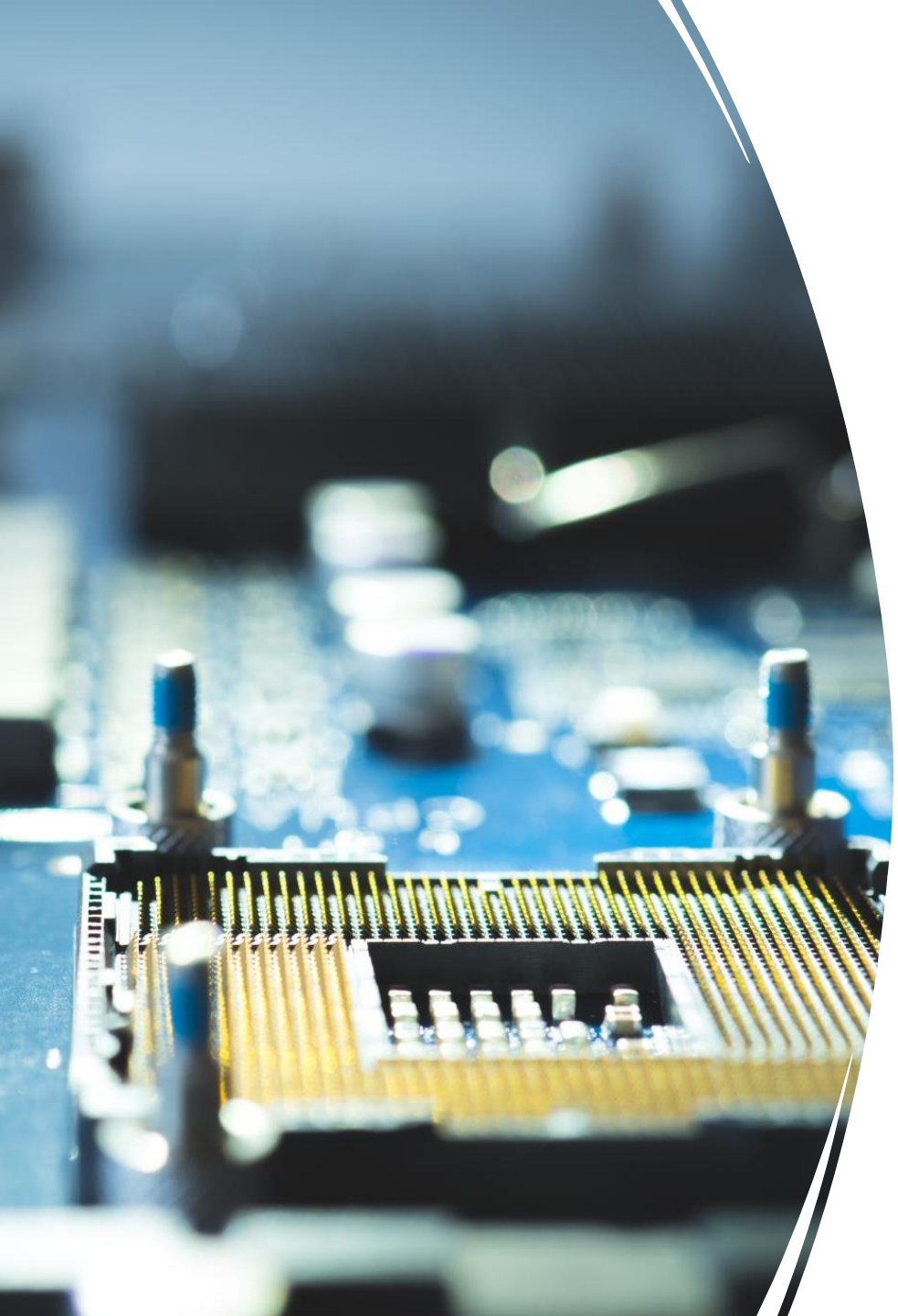
其他编译选项

- `-g` 将调试信息添加到可执行文件，方便 `gdb` 调试
- `-Wall` 编译器会检查源码中的错误，并给出警告信息；有些情况下，忽略警告信息，不影响程序运行
- `-std=c17` 指定 C 的版本，其他版本比如 `c89`、`c99`、`c11` 等
- `-O` 指定优化等级，其他等级比如 `-Og`、`-O0`、`-O1`、`-O2` 等

```
gcc -g -Wall -std=c17 -Og hello.c -o hello
```

更多选项参考 https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html

make



Make

- GNU Make 是一个命令行工具，可以自动化编译和链接的过程。
- 可以用于构建 C、C++、Java 等多种编程语言
- 思想：通过编写一些编译**规则**（recipe），自动为我们生成**目标**（target）

make

- make 通过定义项目中的文件，以及文件间的关系，自动生成合适的编译和链接命令
- make 可以提高构建的速度，比如多个 .c 文件，只修改某一个的情况下，只需要编译修改后的文件
- make 相比 cmake 同样非常复杂，但一些常见的需求，只需要积累几个模板即可

Makefile

- Makefile 或 makefile 包含一系列变量和依赖规则，供 make 创建构建命令。
- 通常将该文件置于项目当前目录，执行 make 命令将自动读取。
- 变量名一般大写，使用 = 进行赋值，
  - 例如创建变量 CC，值为 gcc

```
CC = gcc
```

三个重要的变量

- CC　　指定编译器的名称，一般默认为 cc 或 gcc
- CFLAGS　指定编译过程的选项，例如设置头文件路径（-I）、添加调试信息（-g）
- LDFLAGS 指定连接过程的选项，例如设置库的路径（-L）、链接第三方库（-l）

规则

```
target: dependencies...
\tab    commands
\tab    ...
```

- Makefile 的规则由命令（commands）、目标（target）以及多个依赖（dependencies）构成。
- ***重要！***命令 commands 前面必须要使用 tab 制表符缩进，否则会出现如下错误：

  ```
  Makefile:6: *** missing separator.
  Stop.
  ```

- VS Code 建议安装 **Makefile Tools** 避免上述问题
- 如果依赖（dependencies）文件没有变化，则目标（target）不会重复构建

小型项目的构建

```
CC = gcc
CFLAGS = -g -Wall -std=c17 -Og -Iinclude
LDFLAGS =

hello : hello.o
	$(CC) $(CFLAGS) -o hello hello.o $(LDFLAGS)

hello.o : hello.c include/hello.h
	$(CC) -c hello.c -Iinclude
```

- 一份比较简单的 `Makefile` 案例

创建一份通用模板

```
CC = gcc

CFLAGS = -g -Wall -Og -std=c17 -I./cslib/include -I./include
LDFLAGS = -L./cslib -lSimpleCSLib

SOURCES = $(wildcard src/*.c)
OBJECTS = $(SOURCES:.c=.o)

TARGET = piglatin

$(TARGET) : $(OBJECTS)
	$(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

.PHONY: clean
clean:
	rm -f $(TARGET)
```

- 对于一些简单的项目，可以省略依赖文件的编写，从而创建一份通用的模板

伪目标

```
.PHONY: clean
clean:
	rm -f $(TARGET)
```

- 如果一个目标没有真实的文件名，则称为伪目标（phony target）。
- 常用的伪目标是 clean 用于清除构建过程中的中间文件，其中 .PHONY 表示伪目标不会发生真实的构建