A Non-Sense C Guideline *rev. 0.1*

for Brief Overview of C lang

by KevinZonda

14 Jul 2024

0x00: Command-style Programming in C

Overview

C 语言是一个编程语言，它是是命令式的。那什么是命令式呢？

考虑你在需要做饭，你需要按顺序执行，洗菜，切菜，炒菜三个步骤。

因此一个做饭程序可以描述为：

做饭：

洗菜()

切菜()

炒菜()

用 C 语言来描述就是

```
void make_dinner() {
  wash_food();
  chop_food();
  cook_food();
}
```

其中 void 表示无返回值类型，也就是说做饭不会返回任何类型的数据。

我们可以简单的认为 C 语言程序就是无数操作序列的组合。

正如上述做饭程序，其中使用了 wash_food() 函数，用于表述洗菜，但是这个函数表示了洗菜这一个动作的一个序列。例如：

```
void wash_food() {
    take_food_out();
    set_tap_status(1);
    wash(food);
    set_tap_status(0);
}
```

通过反复组合，我们可以获得一个大型程序。

0x01: Types in C

什么是类型?

类型是表示一个数据到底是什么样子的术语。例如一个术语可以是一个数字或者是一段话等。

为什么需要类型?

计算机只能存储二进制数据，例如 4 个 1。而如何表示一个数字亦或是一个字符串是困难的。因此我们需要引入一个类型去表示二进制所表示的数据。

数据的大小

我们知道 CPU 的底层二进制，因此我们可以轻松知道任何数据需要用二进制进行表示。目前主流的 CPU 架构是 32 位和 64 位，也就是 32 bit 和 64 bit。CPU 架构中的 32 位和 64 位表示的是一次性 CPU 可以读取和处理多少数据。其对应的正好是 4 byte 和 8 byte。因此目前绝大多数的数据结构是根据 4 byte 和 8 byte 进行对其的。

一些常用的类型

数据类型	Bit 数	类型	占内存大小
`int`	32	整数	4 byte
`long`	64	整数	8 byte
`float`	32	小数/浮点	4 byte
`double`	64	小数/浮点	8 byte
`char`	8	字符，可以用作 8 位整数	1 byte

数据范围

⚠️ 这一节只适用于整数类型。

我们知道 1 位二进制可以表示 0 和 1，也就 2 种情况。对于 2 位，其可以为 01, 00, 10, 11，也就是 4 种情况。以此类推。我们可以知道而对于 N 位二进制数，我们可以认为其可以表示 2^N 种情况。

我们考虑对于整数 int，其有 32 位，因此这个二进制数可以表示2^{32}种情况。但是因为我们存在负数，使用我们可以将 32 位二进制数中的一位考虑为符号位，也就是用来表

示正负数。因此我们用来表述数字的可用二进制位就会变成 31 位。因此我们可以认为当符号位分别为 0 和 1 时，我们有2^{31}种情况。但是因为我们还需要数字 0，因此表示正数的情况我们会从2^{31}变成$2^{31}-1$。因此数据范围就会变成 $[-2^{31}, 2^{31}-1]$ 或者是 $[-2^{31}, 2^{31})$ 。在计算机科学中，我们通常会使用左闭右开表示数据范围。

因此我们可以知道遗下数据范围：

`int`: $[-2^{31}, 2^{32})$

`long`: $[-2^{63}, 2^{63})$

`char`: $[-2^{7}, 2^{7}]$

考虑到上述的范围，我们可能会考虑没有符号位（也就是不考虑负数情况）的数据：

因此我们会有 `unsigned` 关键词。

`unsigned int` 表示无符号 32 位整形，其数据范围为$[0,2^{32})$

`unsigned long, unsigned byte` 同上。

Overflow

溢出是一种特殊情况。在上文我们了解了在 C 语言中不同数据类型的范围，那如果超过这个数据范围会怎么样呢？

```
#include <stdio.h>

int main() {
    // byte's range is [-128, 127]
    char b = 127;
    b = b + 1;
    printf("%d", b);
}

OUTPUT: -128
```

我们惊奇地发现，其回到了 -128。这就是溢出。当我们超过了数据最大值，数据会从最小值开始重新计数。其二进制原理如下

```
Name   | 1st byte | 2nd byte
b           1111       1111
+                         1
--------------------------
b'   1      0000       0000
     ^      +------------+
    丢弃          实际数据
```

定义变量

我们可以使用 类型 变量名 进行定义变量;

```
int x;
int y = 11;
```

数组

为了表示一组连续的数据，C 语言支持数组作为数据结构。在 C 语言中，我们可以用 T[N] 的风格表示类型 T 的数组，其中这个数组包含 N 个元素。

定义数组

```
int x[5];           // x 有 5 个 int 类型的元素
int x1[] = {1, 2}; // 等价于 int x1[2] = {1, 2}
                    // 等价于 int x1[2];
                    //        x[0] = 1;
                    //        x[1] = 2;
int z[3] = {9};     // 等价于 int z[3] = {9, 0, 0}
```

0x02: Expressions, Statements & Functions in C

C 语言使用命令式编程。我们可以认为 C 语言程序是由表达式，语句和函数组成的。

定义函数

我们可以使用 类型 函数名(参数 1, 参数 2, ...) 以定义函数：

```
int add(int x, int y) {
  return x + y;
}

void foo() { }
```

if 语句

```
if (condition_expr) { }
if (condition_expr) { } else { }
```

while 语句

```
while (condition_expr) { }
```

equiv. to:

```
start:
if (condition_expr) {
  do_sth();
  goto start;
}
```

for 语句

```
for (init_expr; condition_expr; iter_expr)
```

equiv

```
init_expr();
start:
if (condition_expr) {
```

```
    do_sth();
    iter_expr();
    goto start;
}
```

0x03: pointer

指针，C 语言中最复杂也最离谱的东西。我们可以将 pointer 看作成引用（reference）。

语法

去创建一个变量的指针并不复杂，我们需要一个 & 运算符。

```
int x = 11;
int * x_ptr = &x; // int * 表示 int 的指针
                  // 或者说 int 的引用
(*x)++;           // x++;
                  // x == 12
```

对于构建一个变量的引用，我可以用 &变量 来获得。

而如果我们想获得一个引用所指向的原始产物，则可以用解引用运算符 *。

为什么要用指针

指针看上去很复杂。但是为什么需要指针呢?

C 语言在传递数据时，通常使用一种叫 Copy 的操作，它会在传输时候把数据复制一遍，以保证另一个变量不会影响到这个变量。例如:

```
int x = 11;
int y = x;
x ++; // x = 12, y = 11
y --; // x = 12, y = 10
```

但是有时候当我们处理数据时候，希望原始数据和传入的数据的更改同步，那看上去 Copy 就会让这个操作难以继续。

```
void increase(int a) {
    a ++;
}
int main() {
    int x = 11;
    increase(x); // x 的数据会 copy 到 a 上
```

```
    return 0;
}
```

为了避免这种问题，我们可以用指针：

```
void increase(int* a) {
    (*a) ++;
}
int main() {
    int x = 11;
    increase(&x);
    return 0;
}
```

为什么 Copy 不会影响到指针

这涉及到指针是怎么工作的。变量在内存中都有自己的家，例如如下

```
0x0024 int x = 11;
```

整型 x 位于内存 0x0024

当我们 copy 时，会把内存变成

```
0x0020 int x_copied = x
0x0024 int x = 11
```

因此针对于 x_copy 的操作不会影响到原始变量 x

而对于指针来说其存储的数据类似于

```
0x0020 int* x_ptr = 0x0024
0x0024 int x = 11
```

因此对于指针 x_ptr 的 copy 会变成

```
0x0016 int* x_ptr_copy = 0x0024
0x0020 int* x_ptr      = 0x0024
0x0024 int x = 11
```

由于指针存储的是原始变量的内存地址，所以无论怎么 copy，其永远指向变量 x。

更精确的来说，&操作符，我们认为其是对一个变量取地址，而*操作符，我们认为其是对一个指针，根据其记录的值找到原始的变量。

0x04: malloc() & free()

C 语言变量的生命周期

考虑如果一个变量永远存在于内存中，那一个程序会变得非常非常占用内存。因此 C 语言变量存在其的生命周期。

C 语言将内存分成不同的段，而对于程序执行来说最重要的两个段：一种叫栈（Stack），一种叫堆（Heap）。通常来说，我们定义的变量存在于栈上，其生命周期受限于当前栈帧（Frame）的生命周期。而栈的生命周期依赖于 { 和 }

例如：

```
{                   // set a new Stack Frame A
  int x = 11;
                    // x is valid, y is invalid
  {                 // set a new Stack Frame B
                    // x is valid, y is invalid
    int y = 12
                    // x is valid, y is valid
  }                 // pop the Stack Frame B
                    // x is valid, y is invalid
}                   // pop the Stack Frame A
                    // x is invalid, y is invalid
```

但是考虑到我们很多数据可能是动态的，因此 C 语言使用一个段（Segmentation）以保存这些变量。

内存分配

去在堆中创建变量需要使用 `malloc(size_t)` 函数，我们通常也需要用 `sizeof(type)` 获取一个类型在内存中的大小。

`malloc(size_t)`，是 Memory Allocation 的缩写，表示内存分配，其作用是在堆中创建 size_t 字节的内存。需要注意的是 malloc 不会默认初始化内存，也就是说，其分配的内存可以说是随机内容。

`sizeof(t)`，是 C 语言中的一个标识符，用于查询一个类型在内存中需要多少 byte。

因此如果我们需要一个堆上的整型，可以这么做：

```
int* x = malloc(sizeof(int));
(*x) = 0;                       // 别忘了初始化!
```

需要注意的是！和栈不同，堆中的变量不能自动释放。也就是说我们需要手动还给他们自由。这时候就需要我们使用 `free` 函数了。

```
int* x = malloc(sizeof(int));
// do sth
free(x);
x = NULL; // 释放完后请不要使用 x 变量
          // 将其制 NULL 是一个良好的习惯
```

堆和栈好像啊！为什么要有堆？

是的。但是考虑我们之前学过的内存和磁盘。内存很小但是很快，而磁盘很大但是速度不快。栈和堆也是一个道理。

栈是一个永远存储在内存（DRAM）中的段，但是对于堆，操作系统可以对其进行多种处存储，例如操作系统可以把他们存储在内存，但是当内存紧张时，swap out 到磁盘上。具体的行为依赖于操作系统。

总的来说，堆很大，但是很慢。栈很小，但是很快。他们都在做不同的事情。

内核中的内存分配

在用户态，我们通常使用 malloc 进行分配内存，使用 free 释放内存。但是在内核态中，需要使用 kmalloc 和 kfree 进行内存处理。

0x05: Linux Kernel Driver

Linux 内核驱动是一个长期运行在内核态（Kernelland）的程序。它通过 Linux Kernel Module 进行挂载。

内核态与用户态

一个操作系统包含两个 Lands（态），分别是用户态（Userland）和内核态（Kernelland）。 这两个区域像是两个家，中间有一堵墙隔着。操作系统内核等核心服务运行在内核态，而绝大多数程序运行在用户态。

就像是教室（用户态）和老师办公室（内核态）。操作系统就像是老师，而软件程序就像是学生。老师可以管理程序，可以随意进入和管理教室（用户态）和办公室（内核态）。而学生可以在教室操作，当进入老师办公室（内核态）时需要窃门。

装载与卸载

去查看目前装载的内核模块可以使用

```
$ lsmod # list modules
```

安装模块

```
# insmod $modulePath # install module
```

删除模块

```
# rmmod $moduleName # remove module
```

重要的函数

在教学中，有两个很重要的函数，copy_to_user(buffer, message, length) 和 copy_from_user(target, buff, length)。

copy_to_user 顾名思义，是从内核态复制数据到用户态，也就是相当于把文件从办公室移送到教室。

copy_from_user 而像是学生交作业，把数据从用户态复制到内核态。

驱动的事件

```
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t,
loff_t *);
static ssize_t device_write(struct file *, const char *,
size_t, loff_t *);
static long device_ioctl(struct file *file, unsigned int
ioctl_num, unsigned long);
```

这些是从老师代码的头文件抄来的。它们表示了一个驱动在不同事件时需要执行什么。

```
int init_module(void);
```

这个函数是在整个内核模块被初始化时，被调用。其返回值表示这个模块是否被正确初始化了。

```
void cleanup_module(void);
```

这个函数是在整个内核模块被移除时被调用。

```
static int device_open(struct inode *, struct file *);
```

这个函数是在这个设备被打开时执行。

```
static int device_release(struct inode *, struct file *);
```

这个函数是在这个设备释放时执行。

```
static ssize_t device_read(struct file *, char *, size_t,
loff_t *);
```

这个函数在模块被调用读取数据时被执行。

```
static ssize_t device_write(struct file *, const char *,
size_t, loff_t *);
```

这个函数在模块被调用写入数据时被执行。

我们可以看出，一个驱动模块通常被认为是一个 IO 设备，也就是有读和写两个核心时间。在读和写操作前，都会先执行 device_open，在操作结束后释放后会执行 device_release。

一个简单的驱动的核心生命周期:

初始化模块 init_module

  设备打开 device_open

    读操作 device_read

  设备释放 device_release

  设备打开 device_open

写操作 device_write

设备释放 device_release

卸载设备 cleanup_module

锁

锁结构是用来维护当多个操作同时执行而导致数据不一致的情况。

互斥锁 Mutex Lock

互斥锁表示当这个锁被锁定后，任何其余请求这个锁的操作都需要等待直到锁被释放才能进行操作。

代码

```
DEFINE_MUTEX  (lock);  // 定义互斥锁 .
mutex_lock (&lock);    // 对 lock 上锁
mutex_unlock (&lock);  // 释放 lock 锁
```