CS100 Recitation 4

GKxx

Contents

- 指针和数组（续）
- 字符串
- 动态内存

Homework 2

Solution 已发。对比一下自己的做法，看看能否改进。

指针和数组

指针类型

对于两个不同的类型 `T` 和 `U`，指针类型 `T *` 和 `U *` 是**不同的类型**（尽管它们有可能指向相同的地址）。

```
int i = 42;
float *fp = &i;
*fp += 1; // undefined behavior
```

尽管在 C 中，不同类型的指针之间可以相互（隐式）转换（在 C++ 中必须显式转换），但如果一个指针指向的对象和它被声明的类型不符，解引用这个指针几乎总是 undefined behavior。

- 除了一些情况（稍后会看到一种）

`void *`：“指向 `void` 的指针”

- 任何指针都可以（隐式）转换成 `void *`
- `void *` 可以（隐式）转换成任何指针
- 可以用 `printf("%p", ptr)` 输出一个 `void *` 类型的指针 `ptr` 的值。
  - 如果 `ptr` 是其它类型的指针，需要先转换：`printf("%p", (void *)ptr)`

`void *`

在没有 C++ 那样强的静态类型系统支持下，`void *` 经常被用来表示“任意类型的指针”、“（未知类型的）内存的地址”等等。

- `malloc` 的返回值类型就是 `void *`，`free` 的参数类型也是 `void *`。
- `pthread_create` 允许给线程函数传任意参数，方法就是用 `void *` 转交。
- 在 C 中，接受 `malloc` 的返回值时**不需要显式转换**。
  - `int *p = malloc(size);` 而非 `int *p = (int *)malloc(size);` 。

`void *` 是 C 类型系统真正意义上的天窗

退化 (decay)

- 数组向指向首元素指针的隐式转换（退化）：
  - `Type [N]` 会退化为 `Type *`
- “二维数组”其实是“数组的数组”：
  - `Type [N][M]` 是一个 `N` 个元素的数组，每个元素都是 `Type [M]`
  - `Type [N][M]` 应该退化为什么类型？

退化 (decay)

- 数组向指向首元素指针的隐式转换（退化）：
  - `Type [N]` 会退化为 `Type *`
- “二维数组”其实是“数组的数组”：
  - `Type [N][M]` 是一个 `N` 个元素的数组，每个元素都是 `Type [M]`
  - `Type [N][M]` 退化为“指向 `Type [M]` 的指针”
- 如何定义一个“指向 `Type [M]` 的指针”？

稍微复杂一点儿的复合类型

指向数组的指针

```
int (*parr)[N];
```

存放指针的数组

```
int *arrp[N];
```

- 首先，记住**这两种写法都有，而且是不同的类型**。
- `int (*parr)[N]` 为何要加一个圆括号？当然是因为 `parr` 和“指针”的关系更近
  - 所以 `parr` **是指针**，
  - 指向的东西是 `int [N]`
- 那么另一种则相反：
  - `arrp` **是数组**，
  - 数组里存放的东西是指针。

向函数传递二维数组

以下声明了**同一个函数**：参数类型为 `int (*)[N]`，即一个指向 `int [N]` 的指针。

```
void fun(int (*a)[N]);
void fun(int a[][N]);
void fun(int a[2][N]);
void fun(int a[10][N]);
```

可以传递 `int [K][N]` 给 `fun`，其中 `K` 可以是任意值。

- 第二维大小必须是 `N`。 `Type [10]` 和 `Type [100]` 是不同的类型，指向它们的指针之间不兼容。

向函数传递二维数组

以下声明中，参数 `a` 分别具有什么类型？哪些可以接受一个二维数组 `int [N][M]`？

1. `void fun(int a[N][M])`
2. `void fun(int (*a)[M])`
3. `void fun(int (*a)[N])`
4. `void fun(int **a)`
5. `void fun(int *a[])`
6. `void fun(int *a[N])`
7. `void fun(int a[100][M])`
8. `void fun(int a[N][100])`

向函数传递二维数组

以下声明中，参数 `a` 分别具有什么类型？哪些可以接受一个二维数组 `int [N][M]`？

1. `void fun(int a[N][M])`：指向 `int [M]` 的指针，可以
2. `void fun(int (*a)[M])`：同 1
3. `void fun(int (*a)[N])`：指向 `int [N]` 的指针，当且仅当 `N==M` 时可以
4. `void fun(int **a)`：指向 `int *` 的指针，**不可以**
5. `void fun(int *a[])`：同 4
6. `void fun(int *a[N])`：同 4
7. `void fun(int a[100][M])`：同 1
8. `void fun(int a[N][100])`：指向 `int [100]` 的指针，当且仅当 `M==100` 时可以

`const`

`const` 变量：**一经初始化就不能再改变**（“常量”），所以当然**必须初始化**.

- “常量”这个词其实很容易引发歧义，C/C++ 中还有一种真正的“常量”，是指**值在编译期已知的量**。

可以定义“指向常量的指针”：`const Type *ptr` 或 `Type const *ptr`

pointer-to-`const`

一个“指向常量的指针”也可以指向一个 non-`const` variable

- 但它**自以为**自己指向了“常量”，所以不允许你通过它修改它所指向的变量的值。

```
int i = 42;
const int *cip = &i;
int *ip = &i;
++i;    // OK
++*ip;  // OK
++*cip; // Error
```

- “底层 `const`”（low-level `const`）

pointer-to-`const`

不能用 pointer-to-non-`const` 指向一个真正的 `const` 变量，也不能用一个 pointer-to-`const` 给它赋值或初始化（“不能去除底层 `const`”）

- 如果把 `const` 视为一把锁，这就是在试图拆掉锁。
- 你可以用 explicit cast（显式转换）强行拆锁，但由此引发的对于 `const` 变量的修改是 undefined behavior

```c
const int ci = 42;
int *ip = (int *)&ci;
++*ip; // Undefined behavior
```

```c
int i = 42;
const int *cip = &i;
int *ip = cip; // Warning in C, Error in C++
int *ip2 = (int *)cip; // OK
```

顶层 `const`（top-level `const`）

一个指针自己是 `const` 变量：它永远指向它初始化时指向的那个对象

有时称为“常量指针”

```
int ival = 42;
int *const ipc = &ival;
++*ipc; // Correct
int ival2 = 35;
ipc = &ival2; // Error. ipc is not modifiable.
```

当然也可以同时带有底层、顶层 `const`：

```
const int *const cipc = &ival;
```

字符串

“C 风格字符串”（C-style strings）

C 语言没有对应“字符串”的抽象，字符串就是一群字符排在一起。

- 可以是数组，也可以是动态分配的内存
- **末尾必须有一个 `'\0'`，`'\0'` 在哪末尾就在哪。**

```
char s[10] = "abcde";  // s = {'a', 'b', 'c', 'd', 'e', '\0'}
printf("%s\n", s);     // abcde
printf("%s\n", s + 1); // bcde
s[2] = ';';            // s = "ab;de"
printf("%s\n", s);     // ab;de
s[2] = '\0';
printf("%s\n", s);     // ab
```

结束符 `'\0'`

`'\0'` 是所谓的“空字符”，其 ASCII 值为 `0`。

C 风格字符串结束的**唯一**判断标志

所有号称接受、处理字符串的函数都会去找 `'\0'`

- 缺少 `'\0'` 会让他们不停地走下去，（很可能）导致越界访问。

用数组存储字符串时，记得为空字符多开一格。

```
char s[5] = "abcde"; // OK, but no place for '\0'.
puts(s);             // undefined behavior (missing '\0')
```

字符串 IO

- `scanf` / `printf` ： `"%s"`
  - `scanf` 读 `"%s"` 有内存安全问题：它并不知道你传给它的数组有多长
  - `scanf` 没被踢出去，（我猜）是因为它还有别的用途
- `gets` ：**自 C11 起被踢出标准**，因为它只有这一个用途
  - 替代品 `gets_s` 在标准的附录里，很遗憾 GCC 没有支持它
- `fgets` ：更通用，更安全

```
char str[100];
fgets(str, 100, stdin);
```

- `puts(str);` 输出字符串 `str` 并换行

拷贝字符串

练习：实现你自己的 `strcpy` 函数，将一个字符串拷贝到另一个地方。

拷贝字符串

大多数人总是按捺不住使用下标的冲动，从而先调用一遍 `strlen` ：

```
char *strcpy(char *dest, const char *src) {
  size_t length = strlen(src);
  for (size_t i = 0; i <= length; ++i)
    dest[i] = src[i];
  return dest;
}
```

但是 `strlen` 也会把 `src` 这个字符串扫一遍。

- 为什么要扫两遍？

拷贝字符串

明明可以只扫一遍：终止条件是**碰到** `'\0'`，而非 `i == strlen(src)`。

```
char *strcpy(char *dest, const char *src) {
  size_t i = 0;
  while (src[i] != '\0') {
    dest[i] = src[i];
    ++i;
  }
  dest[i] = '\0';
  return dest;
}
```

拷贝字符串：更直接的办法

直接移动指针就可以了，不需要下标。

- `dest` 和 `src` 都是你这个函数内部的变量，完全可以随便改！

```
char *strcpy(char *dest, const char *src) {
  char *ret = dest;
  while (*src != '\0')
    *dest++ = *src++;
  *dest = '\0';
  return ret;
}
```

```
char a[100], b[] = "hello";
strcpy(a, b); // 没问题。
```

`strlen` 没有魔法

`strlen(s)` 除了把 `s` 从头到尾扫一遍之外，没有任何更神奇的办法。

```
for (size_t i = 0; i < strlen(s); ++i)
  // ...
```

每次循环体执行完毕时，都要执行一次判断条件 `i < strlen(s)`，而每次算 `strlen(s)` 都需要遍历整个字符串，**非常慢**（时间复杂度为 $O\left(n^2\right)$）

应该改为

```
for (size_t i = 0, n = strlen(s); i < n; ++i)
  // ...
```

一个小问题

```
for (int i = 0; i < strlen(s); ++i)
  // ...
```

编译器在 `i < strlen(s)` 上报了个 warning？

- `strlen` 返回值类型为 `size_t`：无符号整数
- 将 `int` 和 `size_t` 放在一起运算/比较时，**`int` 值会被转换为 `size_t` 类型**
  - `-1 < strlen(s)` 几乎肯定是 `false`

* **不要混用带符号数和无符号数**

字符串字面值 (string literals)

字符串字面值：类似这种 `"abcde"` （**双引号！！！**）

- 类型为 `char [N+1]`，其中 `N` 是它的长度，`+1` 是为了放空字符。
- **它会被放在只读的内存区域**
  - 在 C++ 中，它的类型是 `const char [N+1]`，非常合理。
- 和 `42` 不同，字符串字面值具有长久的生命期，你甚至可以取它的地址 `&"abcd"` 。

用不带底层 `const` 的指针指向一个 string literal 是合法的，但**极易导致 undefined behavior**：

```
char *p = "abcde";
p[3] = 'a'; // undefined behavior, and possibly runtime-error.
```

字符串字面值 (string literals)

用不带底层 `const` 的指针指向一个 string literal 是合法的，但**极易导致 undefined behavior**：

```
char *p = "abcde";
p[3] = 'a'; // undefined behavior, and possibly runtime-error.
```

正确的做法：

加上底层 `const` 的保护

```
const char *str = "abcde";
```

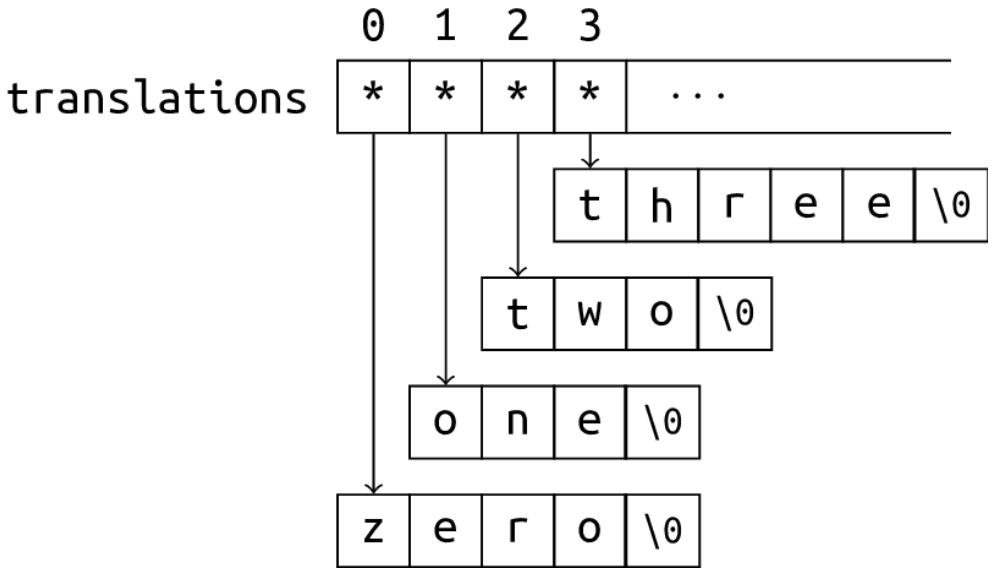
或者将内容拷贝进数组

```
char arr[] = "abcde";
```

字符串数组

```
const char *translations[] = {
  "zero", "one", "two", "three", "four",
  "five", "six", "seven", "eight", "nine"
};
```

- `translations` 是一个数组，存放的元素是指针，每个指针都指向一个 string literal
- `translations` **不是二维数组！**



标准库函数

完整列表，你想要的都在这里

HW3 会有一道造标准库函数的题。

> What I cannot create, I do not understand. - Richard Feynman

动态内存

使用 `malloc` 和 `free`

创建一个“动态数组”：大小在运行时确定

```
Type *ptr = malloc(sizeof(Type) * n);
for (int i = 0; i != n; ++i)
  ptr[i] = /* ... */
// ...
free(ptr);
```

使用 `malloc` 和 `free`

也可以动态创建一个对象

```
int *ptr = malloc(sizeof(int));
*ptr = 42;
// ...
free(ptr);
```

但是为什么需要这样？直接创建 `int ival = 42;` 不香吗？

使用 `malloc` 和 `free`

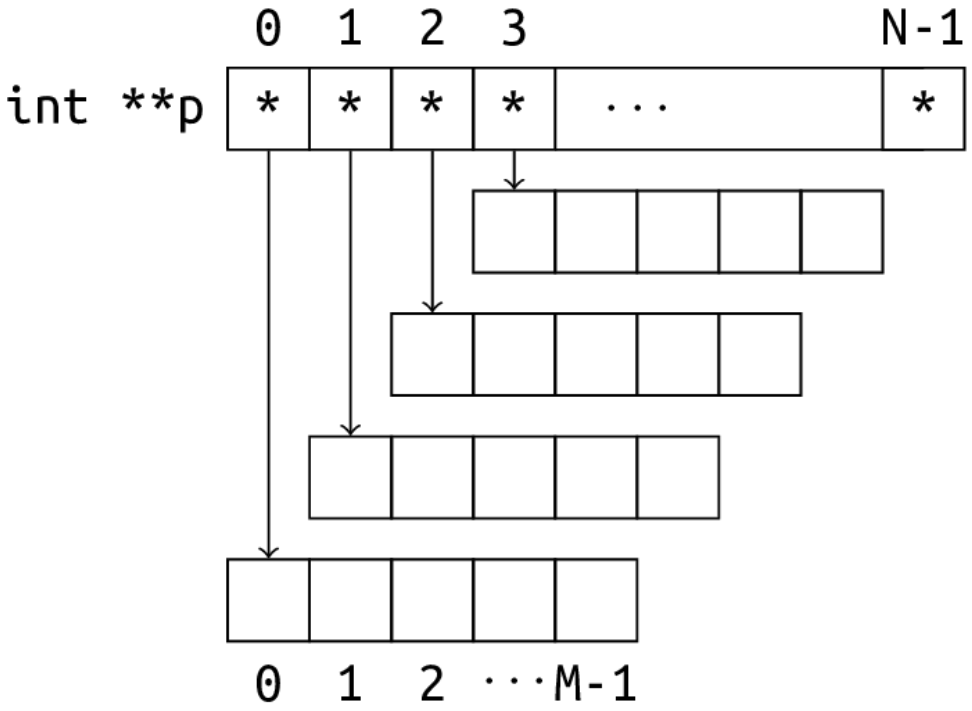
动态创建对象的好处：**生命期跨越作用域**

```
int *create_array(void) {
  int a[N];
  return a; // 返回了一个指向局部变量 a 的指针
            // 当函数返回时，局部变量 a 已经被销毁，该地址无意义
            // 解引用这个指针将导致 undefined behavior
}
int *create_array2(int n) {
  return malloc(sizeof(int) * n); // OK. 动态分配的内存直到被 free 才被销毁
}
```

使用 `malloc` 和 `free`

动态创建一个“二维数组”？

```
int **p = malloc(sizeof(int *) * n);
for (int i = 0; i != n; ++i)
  p[i] = malloc(sizeof(int) * m);
for (int i = 0; i != n; ++i)
  for (int j = 0; j != m; ++j)
    p[i][j] = /* ... */
for (int i = 0; i != n; ++i)
  free(p[i]);
free(p);
```



使用 `malloc` 和 `free`

动态创建一个“二维数组”——另一种方法：创建一维数组

```
int *p = malloc(sizeof(int) * n * m);
for (int i = 0; i != n; ++i)
  for (int j = 0; j != m; ++j)
    p[i * m + j] = /* ... */
// ...
free(p);
```

`malloc`, `calloc` 和 `free`

看标准！ `malloc` `calloc` `free`

```
void *malloc(size_t size);
void *calloc(size_t num, size_t each_size);
void free(void *ptr);
```

- `calloc` 分配的内存大小至少是 `num * each_size` 字节
- `malloc` 分配**未初始化的内存**，每个元素都具有 indeterminant value
  - 但你可能试来试去发现都是 `0` ？这是巧合吗？
- `calloc` 会将分配的内存的每个字节都初始化为零。
- `free` 释放动态分配的内存。在调用 `free(ptr)` 后， `ptr` 指向无效的内存（dangling pointer）

`malloc`, `calloc` 和 `free`

`malloc(0)`, `calloc(0, N)` 和 `calloc(N, 0)` 的行为是 **implementation-defined**

- 有可能不分配内存，返回空指针
- 也有可能分配一定量的内存，返回指向这个内存起始位置的指针
  - 但解引用这个指针是 undefined behavior
  - 这个内存同样需要记得 `free`，否则也构成内存泄漏

正确使用 `free`

- 忘记 `free`：内存泄漏，OJ 上无法通过测试。
- `free(ptr)` 的 `ptr` 必须是之前由 `malloc`, `calloc`, `realloc` 或 C11 的 `aligned_alloc` 返回的一个地址
  - 必须指向动态分配的内存的**开头**，不可以从中间某个位置开始 `free`
  - 会释放由 `ptr` 开头的整片内存，它自有办法知道这片内存有多长
- `free` 一个空指针是**无害的**，不需要额外判断 `ptr != NULL`
- `free` 过后，`ptr` 指向**无效的地址**，不可对其解引用。再次 `free` 这个地址的行为 (double free) 是 undefined behavior。