CS100 notes of recitation

felomid

C Part

整数类型

- `signed int` 和 `int` 是同一类型
- **例外：`char` 和 `signed char` 不是同一个类型，和 `unsigned char` 也不是。**
- `int` 的大小（表示范围）是多少？`long` 呢？
  - implementation-defined!
  - `short` 和 `int` 至少 16 位；`long` 至少 32 位；`long long` 至少 64 位。
  - `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
  - https://en.cppreference.com/w/c/language/arithmetic_types

算术类型

字面值

- 不存在负字面值：`-42` 是将一元负号 `-` 作用在字面值 `42` 上形成的表达式。
- 整型字面值 (integer literal)：`42`，`100L`，`0ll`，`405ul`
  - 还可以有十六进制字面值：`0xBAADF00D`
  - 以及八进制字面值：`052`
  - 以及 C23 的二进制字面值：`0b101010`
  - 这里所有的字母的大小写都随意。

算术类型

字面值

浮点数字面值：`3.14`，`3.14f`，`3.14l`，`1e8`，`3e-8`

- 不写后缀，默认是 `double`。`f` 是 `float`，`l` 是 `long double`，大小写不敏感
- `'a'` 的类型居然是 `int`？？
- C++ 里它就是 `char` 了。

溢出

一个变量的值超出了这个变量所能表示的范围。

- 这里的“**变量**”有可能是临时量！

```
int ival = 10000000;
long long llval = ival * ival;          // 溢出
long long llval2 = 1ll * ival * ival;   // ok
long long llval3 = 0ll + ival * ival;   // 溢出
```

- 无符号数永远不会溢出：无符号数的运算总是在 $\mod 2^N$ 意义下进行的，其中 N 是这个无符号数的位数。
  - `unsigned uval = -1;` 执行后，`uval` 的值是多少？$(2^N - 1)$
- 带符号整数溢出是 **undefined behavior**：你无法对结果作任何假定。
  - 可能会得到在 2's complement 意义下的一个值，也可能被视为 runtime-error 而崩溃，或者其它任何可能的结果。

逻辑运算符

短路求值 (short-circuited)：先求左边，如果左边的结果能确定表达式的结果，就不再对右边求值。

- `&&`：如果左边是 `false`，则右边不会求值
- `||`：如果左边是 `true`，则右边不会求值

指针

- 如果 `ptr` 是其它类型的指针，需要先转换：`printf("%p", (void *)ptr)`
- 不能认为 `printf("%f", ival)` 等价于 `printf("%f", (float)ival)`（试一试）

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”，甚至是“任意的对象”。

- `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**。

`void *` 是 C 类型系统真正意义上的天窗

参数传递

```
void fun(int *px);
void foo(int x);
int main(void) {
  int i = 42, *p = &i;
  fun(p);               // int *px = p;
```

函数指针

- **一个子程序指针或者一个过程指针**
  - 一个指向函数的指针
  - 指向内存中一段可执行代码

```
double cm_to_inches(double cm) {
  return cm / 2.54;
}

int main() {
  double (*func1)(double) = cm_to_inches;
  char * (*func2)(const char *, int) = strchr;
  printf("%f %s", func1(15.0), fun2("Wikipedia", 'p'));
  // prints "5.905512 pedia"
  return 0;
}
```

函数指针

函数指针有什么用?

- 可以作为参数传入函数

```
double compute_sum(double (*funcp)(double), double lo, double hi) {
  double sum = 0.0;
  int i;
  for(int i = 0; i <= 100; i++) {
    double x = i / 100.0 * (hi - lo) + lo;
    double y = funcp(x);
    sum += y;
  }
  return sum / 101.0;
}
// this function takes a function pointer as an argument, thus it can call the function in its scope
```

退化

- 数组向指向首元素指针的隐式转换（退化）：
  - `Type [N]` 会退化为 `Type *`
  - 但是数组名的值不能改变，不能执行 `array += 5` `array++`
- “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”

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])`：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，**不可以**
4. `void fun(int **a)`：指向 `int *` 的指针，**不可以**
5. `void fun(int *a[])`：同 3
6. `void fun(int *a[N])`：同 3
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

字符串

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";
```

标准库函数

`<ctype.h>` 里有一些识别、操纵单个字符的函数：

- `isdigit(c)` 判断 `c` 是否是数字字符
- `isxdigit(c)` 判断 `c` 是否是十六进制数字字符
- `islower(c)` 判断 `c` 是否是小写字母
- `isspace(c)` 判断 `c` 是否是空白（空格、回车、换行、制表等）
- `toupper(c)`：如果 `c` 是小写字母，返回其大写形式，否则返回它本身

动态内存

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);
```

`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` 分配的内存大小为 `num * each_size`（其实不一定，但暂时不用管）
- `malloc` 分配**未初始化的内存**，每个元素都具有未定义的值
  - 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` 具有**未定义的值**（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`

- `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`
- `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**。

struct

`struct` 初始化

- 全局或局部 `static`：**空初始化**：结构体的所有成员都被空初始化。
- 局部非 `static`：不初始化，所有成员都具有未定义的值。

Initializer list:

```
struct Record r = {p, cnt * each_size, __LINE__, __FILE__};

struct Record r2 = {.ptr = p, .size = cnt * each_size,
                    .line_no = __LINE__, .file = __FILE__};
```

C 允许 designators 以任意顺序出现，C++不允许。

`struct` 初始化

赋值不行：

```
r = {p, cnt * each_size, __LINE__, __FILE__};          // Error
records[i] = {.ptr = p, .size = cnt * each_size,
              .line_no = __LINE__, .file = __FILE__}; // Error
```

但加个类型转换就好了：

```
r = (struct Record){p, cnt * each_size, __LINE__, __FILE__};          // OK
records[i] = (struct Record){.ptr = p, .size = cnt * each_size,
                             .line_no = __LINE__, .file = __FILE__}; // OK
```