CS100 Recitation 3

GKxx

目录

预处理指令

预处理指令 (preprocessor directives)

“预处理指令”是那些以 `#` 开头的指令，例如 `#include`, `#define`, `#if`, `#ifdef`, `#endif`, ...

- 有一个可能比较特殊的是 `#pragma`，我们暂时忽略

编译一个 C/C++ 程序时，编译器会首先调用**预处理器** (preprocessor) 处理所有的预处理指令。

- 有时也认为预处理器是编译器的一部分

试一试：`gcc a.c --save-temps` 或者 `gcc a.c -E`

`#include`

`#include` 的含义极其简单：**文本替换**。它会按照某种规则，找到被 `#include` 的那个文件，将其中的内容原封不动地复制粘贴过来。

- `#include` 和其它语言的 `import` 是完全不同的，它远远不如 `import` 聪明。
- 缺乏更智能、更现代的 `import` / `export` /modules 也是 C/C++ 的槽点之一，但是 C++20 有 modules 了！（但是主流编译器支持还不知道要哪年才能完工）

`#include <somefile>` 会让编译器去两个地方找 `somefile`，一是它自己预先设定好的标准库文件的位置，二是你编译的时候通过 `-I` 指定的路径。（GCC）

- `gcc a.c -o a -I/home/gkxx/my_awesome_library/include`

`#include "somefile"` 除了以上两种情况外，还可以将 `somefile` 视为相对路径：

```
#include "../my_library/include/my_header.h"
```

`#define`

`#define` 的含义也是**文本替换**：

```
#define N 1000
#define MAX(A, B) A < B ? B : A
```

在这之后，所有 `N` 都会被替换为 `1000`，所有 `MAX(expr1, expr2)` 都会被替换为 `expr1 < expr2 ? expr2 : expr1`。

* 这样的 `MAX` 真的没问题吗？

`#define`

```
#define MAX(A, B) A < B ? B : A
```

在这之后，所有 `MAX(expr1, expr2)` 都会被替换为 `expr1 < expr2 ? expr2 : expr1`。

```
int i = 10, j = 15;
int k = MAX(i, j) + 1;
```

它被替换为

```
int i = 10, j = 15;
int k = i < j ? j : i + 1; // i < j ? j : (i + 1)
```

#define

```
#define MAX(A, B) (A < B ? B : A)
```

加一对括号就行了？

```
int i = 10, j = 15;
int k = MAX(i, i & j); // 比较 i 和 i & j
```

它被替换为

```
int i = 10, j = 15;
int k = (i < i & j ? i & j : i); // (i < i) & j
```

运算符优先级：比较运算符 > bitwise AND, bitwise XOR, bitwise OR

`#define`

```
#define MAX(A, B) ((A) < (B) ? (B) : (A))
```

全加括号总可以了吧？

```
int i = 10, j = 15;
int k = MAX(i, ++j);
```

它被替换为

```
int i = 10, j = 15;
int k = ((i) < (++j) ? (++j) : (i));
```

`j` 有可能被 `++` 两次！

`#define`

结论：不要用 `#define` 来代替函数。

要定义常量，在 C++ 里也有比 `#define` 更好的办法。

数组

定义并初始化一个数组

以下数组被初始化为何值？

```
int g[100];

int main(void) {
  int a[10] = {1, 2, 3};
  int b[] = {1, 2, 3};
  int c[1000] = {0};
  int d[1000] = {1};
  int e[100];
}
```

定义并初始化一个数组

以下数组被初始化为何值？

```c
int g[100]; // 全部初始化为 0

int main(void) {
  int a[10] = {1, 2, 3}; // 前三个元素 {1, 2, 3} ，后面全是 0
  int b[] = {1, 2, 3}; // b 的类型被推断为 int[3] ，初始化为 {1, 2, 3}
  int c[1000] = {0}; // 全是 0
  int d[1000] = {1}; // d[0] 是 1 ，后面全是 0
  int e[100]; // 未初始化
}
```

数组类型

一个数组的类型包含两个部分：元素个数、每个元素的类型。

```
ElemType a[Length];
```

`int [10]`, `double [10]`, `int [100]` 是三个不同的类型。不同类型的数组之间不兼容。

C 是静态类型语言：所有表达式的类型都在编译时已知。

既然 `Length` 也是类型的一部分，它的值就必须在编译时可知。

Variable-Length Arrays (VLA)

自 C99 引入：

```
int n;
scanf("%d", &n);
int a[n];
for (int i = 0; i != n; ++i)
  scanf("%d", &a[i]);
// ...
```

C11 起，编译器可以选择是否支持 VLA 。

VLA 带来的影响

- 首先，一些表达式无法保证在编译时求值了：`sizeof(a)`
- 像这样的类型别名声明，也会生成一些代码了：`typedef int ArrayType[n];`
    - 它必须有办法记录这个 `n` 的值。

这在 C++ 中会带来非常多的麻烦，因为 C++ 有很多特性依赖于静态类型系统。

因此 VLA 从来没有加入过 C++ 标准。

VLA 的内存问题

```
int a[n];
```

这个数组是开在栈上的，而你在运行之前并不知道它实际会用多少内存。

如果 `n` 过大，导致**栈溢出** (stack overflow) ，程序就会直接崩溃。

- 你既无法提前知道这件事，也无法从这个错误中恢复出来。

相比之下， `malloc` 在堆上分配内存，内存不足时返回空指针，不会造成灾难。

VLA 也不是一无是处

下周再讲

练习

矩阵乘法，插入排序，二分查找

在练习之前，先复习一下如何向函数传递数组参数

传递数组参数

数组会**退化** (decay) 为指向首元素的指针：`T [N]` → `T *`

你无法声明一个真正的数组参数：以下声明中，`a` 的类型都是 `int *`。

```
void foo(int a[]);
void foo(int *a);
void foo(int a[10]);
void foo(int a[20]);
```

传递二维数组

“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？

传递二维数组

“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。

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。

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])`

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 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

矩阵乘法

定义一个函数，计算两个矩阵的乘积，两个矩阵分别是 $N \times M$ 和 $M \times P$ 的。

简单点，元素都是整数。

如何设计接口？

矩阵乘法

定义一个函数，计算两个矩阵的乘积，两个矩阵分别是 $N \times M$ 和 $M \times P$ 的。

```
void matmul(int a[N][M], int b[M][P], int result[N][P]);
```

插入排序

定义一个函数，给一个整数序列排序。如何设计接口？

插入排序

定义一个函数，给一个整数序列排序。如何设计接口？

```
void insertion_sort(int *a, int n);
```

或者

```
void insertion_sort(int *begin, int *end);
```

插入排序

思想：从左往右处理每一个元素。

在处理第 i 个元素的时候，**假定**前 $i-1$ 个元素是有序的。（画饼）

在处理完第 i 个元素时，**保证**前 i 个元素是有序的。（把画的饼实现）

最终，所有元素就都是有序的。

插入排序

思想：从左往右处理每一个元素。

在处理第 i 个元素的时候，**假定**前 $i-1$ 个元素是有序的。（画饼）

在处理完第 i 个元素时，**保证**前 i 个元素是有序的。（把画的饼实现）

- 在前 $i-1$ 个元素中，找到 a_i 合适的插入位置，把它插入那个位置。
- 如何插入？

最终，所有元素就都是有序的。

插入排序

思想：从左往右处理每一个元素。

在处理第 i 个元素的时候，**假定**前 $i-1$ 个元素是有序的。（画饼）

在处理完第 i 个元素时，**保证**前 i 个元素是有序的。（把画的饼实现）

- 在前 $i-1$ 个元素中，找到 a_i 合适的插入位置，把它插入那个位置。
- 由于序列是连续存储的，我们只能把中间那一段集体往后挪一格。

最终，所有元素就都是有序的。

二分查找

接受一个有序（升序）序列以及一个元素 `target` ，查找 `target` 的位置。

```
int binary_search(int *a, int n, int target);
```

或者

```
int binary_search(int *begin, int *end, int target);
```