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。

指针和数组

指针 (pointer)

一个指针**指向**一个变量。指针所储存的**值**是它所指向的变量的内存地址。

```c
int i = 42;
int* pi = &i;
printf("%d\n", *pi);
```

- `int* pi` 声明了一个名为 `pi` 的指针，它指向的变量的类型是 `int`。
- `&` 是**取地址运算符**，用来获得一个变量的地址。
- `*pi` 中的 `*` 是**解引用运算符**，用来获得一个指针所指向的对象。

指针 (pointer)

```c
int i = 42;
int* pi = &i;
printf("%d\n", *pi); // 42
*pi = 35;
printf("%d\n", *pi); // 35
printf("%d\n", i);   // 35
```

指针 (pointer)

- **声明语句** `Type* p;` 中，`*` 表示声明的变量是**指针**。
- **表达式** `*p` 中的 `*` 是**解引用运算符**。
- 一个符号只有在**表达式**中才可能是**运算符**。

指针 (pointer)

`Type *p;` 和 `Type* p;` 是一样的： `*` 靠着谁都行，甚至可以 `Type * p;`

- 但是如果要声明多个指针，每个变量名前面必须都有一个 `*`：

```
int *p1, p2, *p3; // p1 和 p3 是 int *，但 p2 是 int
int* q1, q2, q3;  // 只有 q1 是 int *，q2 和 q3 都是 int
```

- `Type* p` 可能更直观（“指针修饰符也是类型的一部分”），但它具有欺骗性。
- 选择一种并坚持。如果你选择 `Type* p`，不要在一条语句里定义多个指针。

指针 (pointer)

如果指针没有被显式初始化：

- 局部非静态：未初始化，拥有未定义的值——不知道指向哪。
- 全局或局部静态：**空初始化**，值是 `NULL` ，即**空指针**。

也可以显式地让一个指针置空： `p = NULL;`

`p = 0;` **也是合法的，但不推荐。**

事实上自 C++11 和 C23 起，标准都引入了更类型安全 (type-safe) 的空指针 `nullptr` 。

在 C++ 中，请使用 `nullptr` 而非 `NULL` 。

参数传递

```c
void fun(int x) {
  x = 42;
}
int main(void) {
  int i = 30;
  fun(i);
  printf("%d\n", i);
}
```

参数传递

```c
void fun(int x) {
  x = 42;
}
int main(void) {
  int i = 30;
  fun(i);
  printf("%d\n", i); // 30
}
```

传参的过程中，相当于发生了 `int x = i;` 这样的初始化。修改 `x` 的值并不会同时修改 `i` 的值。

参数传递

如果想要让函数修改外部变量的值，我们需要传递那个变量的地址。

```c
void fun(int *px) {
  *px = 42;
  // 将 px 指向的变量的值修改为 42，
  // 或者说，将 px 所表示的地址上存放的变量的值修改为 42。
}
int main(void) {
  int i = 30;
  fun(&i); // 传递 i 的地址
  printf("%d\n", i); // 42
}
```

参数传递

交换两个整数的 `swap`

```c
void swap(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
int main(void) {
  int i = 42, j = 35;
  swap(&i, &j);
  // ...
}
```

* **为何 `scanf` 的参数需要取地址，而 `printf` 的参数不需要?**

参数传递

如果要交换的是两个 `int*` 怎么办?

- 如果要交换的是两个 `T` 类型的变量怎么办?

参数传递

如果要交换的是两个 `T` 类型的变量怎么办?

```
void swap(T *pa, T *pb) {
  T tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
int main(void) {
  T x = something(), y = something_else();
  swap(&x, &y);
  // ...
}
```

未定义的行为

如果一个指针没有指向一个实际的对象：

- 它可能是未初始化的（俗称“野指针”）
- 可能是空指针
- 可能是指向的内存刚被释放掉（“空悬指针”）（下次课再说）
- 或者其它无意义的地址：`int *p = 123;`

试图解引用这个指针是 undefined behavior，并且通常是**严重的运行时错误。**

- 还记得逻辑运算符的“短路求值”吗？

```
if (p != NULL && *p == 42) { /* ... */ }
```

数组

一片连续的存储区域

```
Type name[N];
```

`N` 必须是**常量表达式**：它的值能确定在编译时已知。

```
int a1[10];   // 正确，字面值是常量表达式
#define MAXN 10
int a2[MAXN]; // 正确，MAXN 被预处理器替换为 10
int n;
scanf("%d", &n);
int a[n];     // 见下一页
```

数组

```c
int n;
scanf("%d", &n);
int a[n];
```

如果数组 `a` 的大小是运行期确定的，它就是一个 **Variable-Length Array (VLA)**。

- VLA 自 C99 被加入 C 语言标准，但 C11 起编译器有权决定是否支持它。
- 我们**不推荐**也**不允许**使用 VLA 。
  - 等学了动态内存再说。
- C++14 时人们曾经讨论过是否在 C++ 中引入 VLA，但最终还是将它 vote out。
  - 等学 C++ 的时候再解释为何 C++ 难以支持 VLA。

可以看我的博客

数组下标

可以用 `a[i]` 来访问数组 `a` 的第 `i` 个元素，读写都可以。

```
int a[10];

bool find(int value) {
  for (int i = 0; i != 10; ++i)
    if (a[i] == value)
      return true;
  return false;
}
```

```
int main(void) {
  int n; scanf("%d", &n);
  for (int i = 0; i != n; ++i)
    scanf("%d", &a[i]);
  for (int i = 0; i != n; ++i)
    a[i] *= 2;
  // ...
}
```

下标的范围是 $[0, N)$ 的整数。下标访问越界本质上等同于解引用无效的指针，是**未定义的行为**，并且是**严重的运行时错误**。

数组下标

下标的范围是 $[0, N)$ 的整数。下标访问越界本质上等同于解引用无效的指针，是**未定义的行为**，并且通常是**严重的运行时错误**。

还是那句话：编译器可以假定你的程序没有未定义的行为。

```
int a[10];

bool find(int value) {
  for (int i = 0; i <= 10; ++i)
    if (a[i] == value)
      return true;
  return false;
}
```

- 这段代码可能被直接优化为

```
bool find(int value) {
  return true;
}
```

更多例子

数组的初始化

和普通变量类似，如果没有显式初始化：

- 局部非静态的数组：未初始化，数组里的所有元素都具有未定义的值。
- 全局或局部静态的数组：**空初始化**，就是对数组里的所有元素进行**空初始化**。

但数组还有特殊的初始化规则...

数组的初始化

但数组还有特殊的初始化规则：使用大括号初始值列表

- 可以对数组的前几个元素进行初始化：

```
int a[10] = {2, 3, 5, 7}; // Correct: Initializes a[0], a[1], a[2], a[3]
int b[2] = {2, 3, 5};     // Error: Too many initializers
int c[] = {2, 3, 5};      // Correct: 'c' has type int[3].
int d[100] = {};          // Correct in C++ and since C23.
```

- 还可以使用 designators (since C99)：

```
int e[10] = {[0] = 2, 3, 5, [7] = 7, 11, [4] = 13};
```

数组的初始化

如果对数组进行了显式初始化，所有没有指定初始值的元素都被**空初始化**！

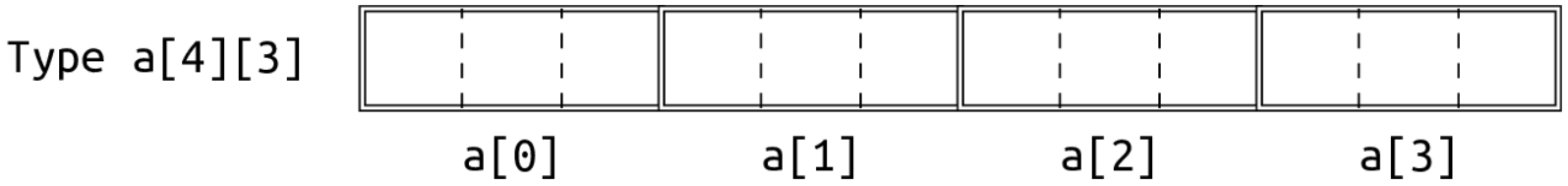
```c
int main(void) {
  int a[10] = {1, 2, 3}; // a[3] 及后续元素都是 0
  int b[100] = {0};      // b 的所有元素都是 0
  int c[100] = {1};      // c[0] 是 1，c[1] 及后续元素都是 0
  int d[100] = {};       // 自 C23 起合法，所有元素都是 0
}
```

不要认为 `= {x}` 是将所有元素初始化为 `x`！

多维数组

C 没有真正意义上的“多维数组”（但 C++23 有多维下标运算符！！！）

所谓的“多维数组”其实是数组的数组：




```
int a[10][20];

bool find(int value) {
  for (int i = 0; i != 10; ++i)
    for (int j = 0; j != 20; ++j)
      if (a[i][j] == value)
        return true;
  return false;
}
```

多维数组的初始化

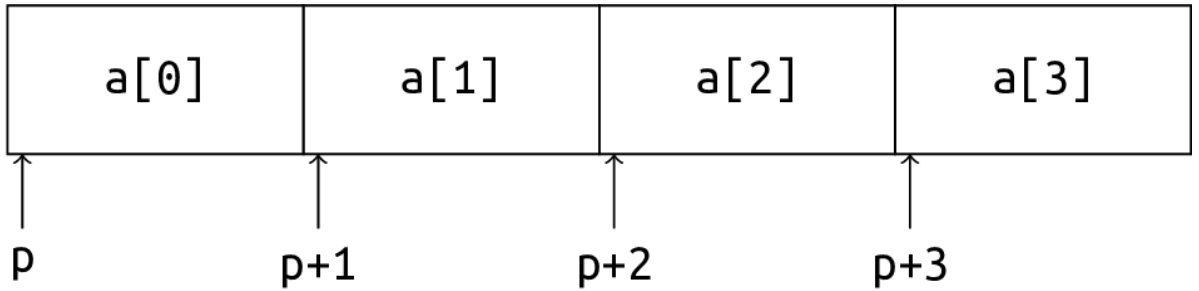
标准几乎已经说得非常清楚了。

```
int a[4][3] = { // array of 4 arrays of 3 ints each (4x3 matrix)
    { 1 },      // row 0 initialized to {1, 0, 0}
    { 0, 1 },   // row 1 initialized to {0, 1, 0}
    { [2]=1 },  // row 2 initialized to {0, 0, 1}
};              // row 3 initialized to {0, 0, 0}
int b[4][3] = {    // array of 4 arrays of 3 ints each (4x3 matrix)
  1, 3, 5, 2, 4, 6, 3, 5, 7 // row 0 initialized to {1, 3, 5}
};                          // row 1 initialized to {2, 4, 6}
                            // row 2 initialized to {3, 5, 7}
                            // row 3 initialized to {0, 0, 0}
int y[4][3] = {[0][0]=1, [1][1]=1, [2][0]=1};  // row 0 initialized to {1, 0, 0}
                                               // row 1 initialized to {0, 1, 0}
                                               // row 2 initialized to {1, 0, 0}
                                               // row 3 initialized to {0, 0, 0}
```

指针的算术运算

对于指向 `Type` 类型的指针 `p` 和一个整数 `i`：

- `p + i` 得到的地址是 `(char *)p + i * sizeof(Type)`，即和 `p` 相距 `i` 个 `Type`。
- 所以如果让 `p = &a[0]`，那么 `p + i` 就等于 `&a[i]`，`*(p + i)` 就等价于 `a[i]`。



- `i + p`，`p += i`，`p - i`，`p -= i`，`++p`，`p++`，`--p`，`p--` 等运算也是类似。

数组向指针的隐式转换

如果 `p = &a[0]`，那么 `p + i` 就等于 `&a[i]`，`*(p + i)` 就等价于 `a[i]`。

鉴于数组和指针有如此紧密的联系，C/C++ 允许数组**隐式转换**为指向其首元素的指针：

- `a` ⟶ `&a[0]`
- `T [N]` ⟶ `T *`

因此，

- `p = &a[0]` 可以直接写成 `p = a`。
- `*a` 就是 `a[0]`。

数组向指针的隐式转换

如果 `p = &a[0]`，那么 `p + i` 就等于 `&a[i]`，`*(p + i)` 就等价于 `a[i]`。

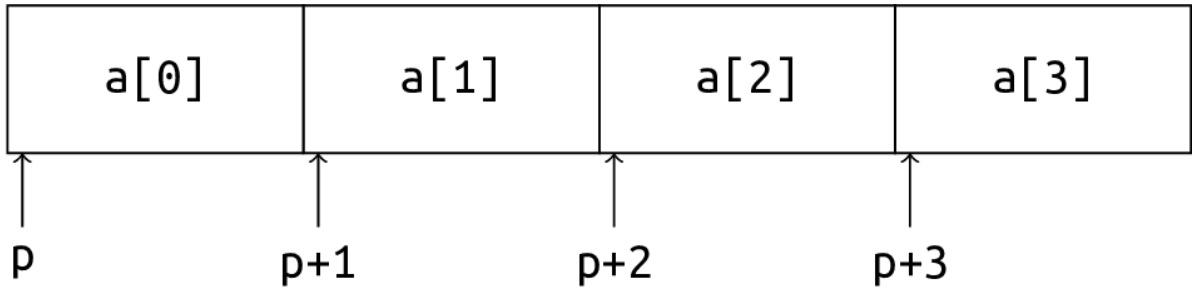
鉴于数组和指针有如此紧密的联系，C/C++ 允许数组**隐式转换**为指向其首元素的指针。

- 可以用指针的方式遍历数组：

```
int a[10];

bool find(int value) {
  for (int *p = a; p != a + 10; ++p)
    if (*p == value)
      return true;
  return false;
}
```

指针相减



如果指针 `p1` 和 `p2` 分别指向某一个数组的下标为 `i` 和 `j` 的位置，则 `p1 - p2`

- 的结果是 `i - j`，
- 类型为 `ptrdiff_t`：一个定义在 `<stddef.h>` 里的类型，是一种带符号整数，其具体大小是 implementation-defined。
  - 例如，在 64 位系统上 `ptrdiff_t` 可能是 64 位。
- 特别地，这里的 `i` 和 `j` 可以等于 `N`，即 `p1` 或 `p2` 指向数组的最后一个元素的**下一个**位置（“尾后”，past-the-end）

指针的算术运算

简单来说，指针的算术运算只能在一个数组内部以及“尾后”位置的范围内发生。

以下统统是 undefined behavior:

- `p1 - p2`，其中 `p1` 和 `p2` 指向两个不同的数组中的位置
- `p + 2 * N`，其中 `p` 指向某个长度为 `N` 的数组
- `p - 1`，其中 `p` 指向某个数组的首元素 `a[0]`

向函数传递数组

C 语言**没有办法**声明一个数组参数（为什么？），所以传递数组的唯一方式是传递**指向数组首元素的指针**。

以下四种声明是**完全等价的**：参数 `a` 的类型是 `int *`。（如何验证？）

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

向函数传递数组

C 语言**没有办法**声明一个数组参数（为什么？），所以传递数组的唯一方式是传递**指向数组首元素的指针**。

以下四种声明是**完全等价的**：参数 `a` 的类型是 `int *`。（如何验证？）

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

验证：使用 `sizeof` 获得一个变量或类型所占的字节数。

向函数传递数组

就算以数组的形式声明一个参数，它的类型也是一个普通的指针：传递任何一个地址给它都是合法的，传递任何大小的数组给它也是合法的。

> 为何使用 `scanf`、`gets` 之类的函数读取字符串被视为不安全的?

为了让函数知道这个数组究竟有多长，常见的方法是**显式地传递一个参数** `n`：

```
void print_array(int *a, int n) {
  for (int i = 0; i != n; ++i)
    printf("%d ", a[i]);
}
```

*** 指针也能用下标?**

向函数传递数组

练习：设计并编写一个函数，将一个整数数组里的奇数倒序拷贝给另一个数组。

向函数传递数组

练习：设计并编写一个函数，将一个整数数组里的奇数倒序拷贝给另一个数组。

```
void copy_odd_reversed(int *from, int n, int *to) {
  for (int i = n - 1, j = 0; i >= 0; --i)
    if (from[i] % 2 == 1)
      to[j++] = from[i];
}
```

指针也能用下标?

看看标准，你会发现下标运算符的语法是

- `pointer[integer]`
- `integer[pointer]`

也就是说：

- 下标运算符在语法上是为指针提供的，取数组的下标 `a[i]` 时实际上也发生了数组向首元素指针的隐式转换
- 不仅可以 `p[i]`，还可以 `i[p]` ~~是不是很奇怪~~
- 你只要知道"数组和指针都可以用下标"就行了，不是非得记住谁向谁转换。

数组向指针的隐式转换

事实上这种隐式转换无处不在：

> Any lvalue expression of array type, when used in any context **other than**
>
> - as the operand of the address-of operator (`&`)
> - as the operand of `sizeof`
> - as the string literal used for array initialization
>
> undergoes a conversion to the non-lvalue pointer to its first element.

以至于某些人会告诉你“数组名就是指针”，**但这是不对的！**