

C语言基础

在了解完C语言环境的搭建以后，让我们开始进行编程吧，本节课将涉及到抽象思维，C语言基础操作，以及指针的入门。

希望大家能够在这节课中学到一些有趣的知识。

计算机抽象思维

我们为什么需要抽象？

举个简单的例子

你看得懂这是什么吗？

```
0000: 11 41 e4 06 e0 22 00 00 00 00 07 b7 00 07 85 13
0010: 00 00 00 97 00 07 85 13 00 00 00 97 00 00 80 e7
0020: 47 81 85 3e 60 a2 64 02 01 41 80 82 00 00 00 00
```

你看得懂这个吗

```
.LC0:
    .string "Hello world"
main:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    addi    s0,sp,16
    lui     a5,%hi(.LC0)
    addi    a0,a5,%lo(.LC0)
    call    printf
    li      a5,0
    mv      a0,a5
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```

这个应该能看得懂吧

```
#include <stdio.h>

int main(void) {
    printf("Hello world");
    return 0;
}
```

这三段代码都能完成同一个功能，打印出"Hello World",编写代码时，你想选择哪种呢？想必不用我多说了吧

使用抽象，我们能够在更高层次上与计算机交互，而不是直接操作底层硬件或机器码。通过编程语言、编译器和操作系统等层次，我们可以更快速地开发和维护复杂的软件系统。

当然,在我们使用C语言进行代码开发的时候,也会使用到抽象的思想,比如说你想要实现一个贪吃蛇的功能,你一定不想在main函数里面完成所有功能,因为这样会大大增加开发的难度,没人想要把一段段的代码塞到一个函数里面去,这样不仅开发速度慢,而且出现bug还很难修!

```

/**
 * @param 游戏板: board[BOARD_SIZE][BOARD_SIZE]
 * @param 初始化: initialize_board
 * @param 随机生成: generate_number
 * @param 打印显示: print_board
 * @param 读取移动指令: choose_direction
 * @param 合并并移动: move_and_merge
 * @param 判断游戏终止: is_finished
 */
void run_game() {
    initialize_board();
    generate_number(); generate_number();
    while (true) {
        switch (choose_direction()) {
            case UP: move_and_merge_up(); break;
            case DOWN: move_and_merge_down(); break;
            case LEFT: move_and_merge_left(); break;
            case RIGHT: move_and_merge_right(); break;
        }
        if (is_finished()) break;
        generate_number();
        print_board();
    }
}
}

```

抽象的设计使得我们将复杂的逻辑分解为若干简单的函数模块，各个模块之间相互独立并可以重复使用，这样可以使代码结构更加清晰、易于维护和扩展。

虽然我还没有实现贪吃蛇的功能,但显然我已经把贪吃蛇的思路设计好了,有了思路,我们就能进行进一步的编程,逐个实现各个函数,最后拼在一起就是完整的程序了。这就是抽象，理解了吗。

C语言基础操作

C语言其实是十分基础的一门编程语言，它与计算机底层连接的十分紧密，因此，作为一只计算机系的小猫猫，我们是十分有必要了解这些的。

💡 Tip

假如你不是计算机系的学生，那也必须掌握这项技能。麻省理工学院开设了一个专业，那就是著名的EECS(Electrical Engineering and Computer Science)。电子工程和计算机的界限本身就很模糊。而且，近些年来兴起的物联网技术，无人车等，都离不开计算机和电子信息之间的协作。在交叉领域，我们更需要一些人才，来打破边界，取得突破。

常量

在C语言中，每个变量都有属于它的常量形式
我们可以这样来定义常量

```

const int a = -1;
const float b = 1.2;
const char c = 'C';

```

⚠ Warning

常量一经定义,无法更改!

但凡编译器发现你在程序中更改了这个变量,那会直接报编译失败

💡 Tip

常量有什么用呢

当你和你的好友一起写代码时,你想让你的好友写一段播放音频的代码,并且为你需要为你的好友提供一个程序接口,你一定不想让你的好友更改你传入的音频文件,因为这会导致一些不可预知的bug,这时候,你就能考虑使用 `const` 来保证你的朋友不会更改你的音频信息 `void play_wav(const int*`

`audio, size_t length);`

控制流

顺序, 分支, 循环。这是C语言中最基本的操作, 只要使用这三种操作, 在理论上, 你就能可以编写任何计算机程序

顺序结构

顺序结构就是一条一条执行指令

分支结构

分支结构是程序根据条件的不同执行不同的代码路径,使得程序能够做出决策, 从而根据不同的输入或状态采取不同的行为。

最常见的就是使用if-else语句

```
if(a == 1){
    printf("true");
}else if(a == 0){
    printf("false");
}else{
    printf("Error");
}
```

还有switch,用于处理同一类型数据的各种情况

```
switch(ch){
case 'A':
    printf("you choose \'A\' as your answer!\n");
    break; // 千万别忘加
case 'B':
    printf("you choose \'B\' as your answer!\n");
    break; // 千万别忘加
case 'C':
    printf("you choose \'C\' as your answer!\n");
    break; // 千万别忘加
case 'D':
    printf("you choose \'D\' as your answer!\n");
    break; // 千万别忘加
default:
    printf("you don\'t choose any answer!\n");
    break; // 千万别忘加
}
```

循环结构

常见的循环结构包括while和for两种,并且都需要掌握

```
// 这是while,当表达式为"真"时,会继续执行下面的命令
while(expr){
    ...
}

// 这是do-while结构,程序会先执行一遍,假如表达式为"真"那么就会继续执行
do{
    ...
}while(expr)

// for循环,括号里面的三项分别为(初始化操作;判断条件;结束后需要进行的操作)
for(i=0;i<100;i++){
    ...
}
```

当然不常见的还有 `goto`,不过不利于维护,不建议使用

① Note

在理论计算机科学中,图灵完备性表明,只要掌握了顺序、分支、循环这三种结构,就可以表达任何可计算的问题。这三种结构被称为编程中的基本控制结构,任何复杂的算法都可以通过这三者的组合来实现。

递归是一种特殊的控制结构,通常用于解决分而治之的问题,尤其是在处理树结构和图结构时非常自然。递归可以通过循环模拟,虽然在某些情况下递归代码更加简洁,但需要注意的是递归调用会占用更多的栈空间,如果层次过多,可能导致栈溢出。

函数

函数是抽象的一种体现

- 在编程中,函数是一段可重复使用的代码块,用于执行特定的任务或完成特定的操作。函数可以接受输入参数,并且可以返回一个值或执行一些操作
- 代码重用
- 提高代码可读性
- 提高代码可测试性

如何使用函数

```
int fact(int n); // 这是函数的声明,通常放在代码的开头或者头文件中

int main(){
    int data = fact(5); // 这是函数的使用
    printf("%d\n",data);
    return 0;
}

int fact(int n){ // 这是函数的实现
    if(n<=1){
        return 0;
    }
}
```

```

}
int i = 1, sum = 1;
for(i=1; i<=n; i++){
    sum *= i;
}return sum;
}

```

声明是为了让你的编译器知道存在这个函数

使用返回值时直接使用 '=' 赋值即可

函数分为这么几个部分

对于 `int fact(int n);` 最前面的 `int` 表示函数返回的数据类型是 `int` 类型, `fact` 表示的是函数名称, 括号里的 `int` 表示传入参数的数据类型, `n` 表示传入参数的变量名

通过调用这个函数, 我们就实现了阶乘的计算

数组

- 数组就是一整块变量的集合, 数组名是数组存放的地址, 使用索引确定每个元素的位置
- 定义: `unsigned char temp[10] = {0x11, 0x12, 0x13};`
- 使用: `printf("%02x ", temp[0]);`

```

#include <stdio.h>
int main(){
    // 初始化
    unsigned char temp[10] = {0x11, 0x15, 0x22};
    printf("temp address: %p\n", temp);

    // 修改
    temp[3] = 0xa8;

    // 使用
    for (size_t i=0; i<10; i++) printf("%02x ", temp[i]);
    printf("\n");
    return 0;
}

```

💡 Tip

请注意数组是内存中的一块连续的空间, 加入你定义了一个大小为3的数组 `int tmp[3]`, 千万不要访问第四个元素 `tmp[3]`, 因为这会导致UB(未定义行为), 会导致你的程序出现莫名其妙的bug 并且, 你只能在初始化的时候这么豪爽地将好几个值赋值到数组中, 当完成初始化后, 就不能使用这种方法了, 需要一个一个进行修改

字符串

字符串是一类特殊的数组, 它以 `\0` 结尾

定义字符串的时候需要注意一点, 字符串的大小一定要比数据大至少一个单位, 我们需要用这一个单位的空间存储 `\0` 就比如说

```
char mystr[5] = "abcc";
```

当然,字符串也是只能在复制的时候这么畅快,之后倘若要修改的话,不调用标准库,就只能一个一个改,别忘了,改的时候需要在字符串的末尾添加 `\0`,不然就不是一个完整的字符串

使用标准库 `<string.h>` 就能实现快速赋值

```
#include <string.h>
int main(){
    char my_str[100] = "computer";
    strcpy("science",my_str);    // 字符串复制
}
```

指针的快速入门

指针,作为C语言的精髓,在广受赞誉的同时,有广受鄙夷,以至于 `java` 的创始人 James Gosling 评价指针是一种容易导致程序出错的语言特性

确实,指针的直接操作在 C 和 C++ 等语言中容易导致内存泄漏、缓冲区溢出、指针悬挂（悬空指针）等问题，使得程序不安全且难以调试。

不过,也正是因为指针,C语言可以通过直接访问内存的方式,在函数间传递内存地址，而不是传递数据的副本。特别是在处理大数据结构（如数组、结构体）时，避免了不必要的数据拷贝。

内存

在理解内存时,我们可以将内存理解为一个超大的数组,`地址`可以看作这个数组的索引,通过访问这个"索引"我们就能取到这个数组上的特定元素

值(一个字节)	地址(32位)
0b00110011	0xFFFFFFFF
?	0xFFFFFFFFE
0xA8	0xFFFFFFFFD
...	...
?	0x00000000

变量与垃圾值

```
int a = 1;
float b = 1.2;
char c = 'c';
```

当你定义变量时,你会获得四个参数

变量类型	变量名	值	存储地址
int	a	1	&a
float	b	1.2	&b
char	c	'c'	&c

实际上在内存中已经存好了(使用小端序,64位机)

变量名	数值	内存
a	0x01	0x00007FFEE72F6B34
a	0x00	0x00007FFEE72F6B35
a	0x00	0x00007FFEE72F6B36
a	0x00	0x00007FFEE72F6B37
b	0x9a	0x00007FFEE72F6B30
b	0x99	0x00007FFEE72F6B31
b	0x99	0x00007FFEE72F6B32
b	0x3F	0x00007FFEE72F6B33
c	0x63	0x00007FFEE72F6B2F

Note

每一个地址存储一个字节(8位),在64位机中, `int` 类型占用4个字节, `float` 类型也占用4个字节, `char` 类型占用1个字节

假设我们定义一个变量 `int d;` 不给他传递初始值,那么,编译器会为它分配一片空间,有可能编译器不对他赋初始值,那么,我们倘若直接读取这个变量的时候,有可能读到一个数值,这个数值就是原来内存存储的数值,我们将这种数据称为**垃圾值**

```
int a;
printf("a = %d\n",a);
```

-8455

数组的越界行为

```
int array[5] = { 1, 2, 3, 4, 5 };
```

array 这个标识符是什么呢?

值 (四个字节)	地址 (32位)	访问
0x00000005	0xFFFFFFFF	array[4]
0x00000004	0xFFFFFFF8	array[3]
0x00000003	0xFFFFFFF7	array[2]
0x00000002	0xFFFFFFF3	array[1]
0x00000001	0xFFFFFEEF	array[0]

上面提到array这个数组实际上是一片连续的内存空间,所以array存储的是第一个值的地址 0xFFFFFFFF 通过访问这个地址,我们就能快速定位这个数组,并进行访问

那么访问array[5]就会导致访问到不属于数组的空间,就会引发bug,这就是数组越界

多维数组

```
int array[2][2] = {{1,2},{3,4}};
```

值	地址	访问	访问	访问
1	0xAFFFFFF0	matrix[0][0]	matrix[0]	matrix
2	0xAFFFFFF4	matrix[0][1]		
3	0xAFFFFFF8	matrix[1][0]	matrix[1]	
4	0xAFFFFFF12	matrix[1][1]		

我们可以发现,不管是多维数组,还是一维数组,存储结构都是线性的,多维数组是按行进行展开的

⚠ Warning

请先跳过这段,等到学完指针之后,再来看下面这个程序,你会发现什么呢?

```
#include <stdio.h>

int main(){
    int matrix[2][2] = {{1,2},{3,4}};
    printf("matrix = %p\n", (void*)matrix);
    printf("matrix[0] = %p\n", (void*)matrix[0]);
    printf("**matrix = %p\n", (void*)(*matrix));
    printf("***matrix = %d\n", (int)(**matrix));
    printf("**matrix[0] = %d\n", (int)(*matrix[0]));
}
```

```
matrix = 0x7ffc624f4cd0
matrix[0] = 0x7ffc624f4cd0
*matrix = 0x7ffc624f4cd0
**matrix = 1
*matrix[0] = 1
```

指针

指针也可以看作一个变量,我们对其的定义是这样的

```
int a = 1;
int* p = &a;
printf("%d", *p);
```


`int*` : `int`表示指针指向的元素类型 `*`代表的是我要定义一个指针
`p` : 标识符
`&a` : 指针的值 (指针指向的地址)
`&p` : 存放指针的地址

值 (四个字节)	地址 (32位)	指针访问	标识符
0x00000001	0xAFFFFFFF	*p	a
0xAFFFFFFF	0xAFFFFFB		p

可以看到指针存放的是a的地址我们可以通过操作符`*`对指针进行解引用来提取出指针指向地址的值,然后使用指针提取a处的值

指针也是有类型的,指针的类型决定了指针该对指向的地址进行什么操作,假设指针的类型为`int*`,那么在解引用时,指针就会提取出指向的地址以及指向地址的后面三位地址的值,将其拼接为一个

`int`

指针的初始化及使用

```
int* p;  
*p = 1; // 危险, p是野指针
```

应该这么做

```
int* p = NULL;
```

⚠ Warning

还记得之前讲过的垃圾值吗,正是因为有垃圾值的存在,我们对指针进行初始化的时候,假如不对指针进行赋空操作,那么垃圾值就会成为指针指向的位置,这非常危险,相当于有人把枪口对准你,即使不开枪,我们也绝对不会允许这种事情发生!

我们将指针赋值为空,也就是让枪口指向地面,即使开枪,也会被操作系统强制终止,因为地址0是不被允许访问的

指针的作用

指针是C语言的精髓, 广泛运用于传参等方面

```
#include <stdio.h>  
  
void swap1(int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
  
void swap2(int *x, int *y) {  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```

```
int main(void) {
    int a = 1, b = 2;
    printf("a = %d, b = %d\n", a, b);
    swap1(a, b);
    printf("a = %d, b = %d after swap1\n", a, b);
    swap2(&a, &b);
    printf("a = %d, b = %d after swap2\n", a, b);
    return 0;
}
```

```
a = 1, b = 2
a = 1, b = 2 after swap1
a = 2, b = 1 after swap2
```

在C语言中,函数的返回值只能是一个值,并且,假设你传a,b两个变量的话,无论在函数中进行了什么操作,都不会影响原来的变量

但我们传入一个指针之后就完全不一样了,我们可以通过指针,直接修改a,b所在地址的值,这就能达到交换数据的效果

指针数组和数组指针

```
int *p[10];
int (*p)[10];
// []运算符的优先级>*运算符的优先级
```

```
int *p[10];
// 化为int* array[] -> 存放着 10个指向int类型的指针 的数组
int (*p)[10];
// 化为int pointer[] -> 指向 存放了10个int类型的数组 的指针
```

加大难度

```
int* (*p)[20][10];
```

问: p是什么?
p是指针还是数组?
p的类型是什么?

想明白之后点击链接查看答案

<https://godbolt.org/z/hv6rnTj5Y>

数组和指针的关系

```
// p1的类型是int* , array的类型是int[10]
int array[10];
int *p1 = array;

// p2的类型是int*[2] , matrix的类型是int[][]
int matrix[2][2];
int (*p2)[2] = matrix;
```

数组与其元素类型的指针存在隐式转换, 指向数组首元素的指针可通过数组类型表达式初始化。

问: 二维数组可以和二维指针相互转化吗

```
int array[2][2];
int** ptr = array;
//可以这么做吗
```

恭喜你,喜提报错一个

```
main.c: In function 'main':
main.c:10:19: warning: initialization of 'int **' from incompatible pointer type
'int (*)[2]' [-wincompatible-pointer-types]
```

```
10 |         int** ptr = matrix;
    |         ^~~~~~
```

二维数组和二维指针是两个完全不一样的存在,对二维数组matrix进行解引用,(void)matrix和(void)(matrix)的值是一样的

他们都是指向数组的首地址,区别在于他们的类型,matrix的类型是int()[2],而*matrix的类型是int[2]而对二维指针进行解引用,那么,它就会提取出matrix指向位置的**值**来作为一位指针*ptr的指向位置,这等同于将垃圾值赋值给指针,会产生严重的后果

Note

C语言对于指针比较"宽容",即使检查到你错误的使用了指针,编译器也只会报一个警告,而不是错误,所以,当出现关于指针的警告是,请将它看作报错,并想办法去解决它

或者你也可以在使用gcc进行编译时,加入 `-Werror` 将警告变为错误处理

指针的加减法

加、减

加、减整数n -> 指向后/前第n个元素

```
int array[5] = {1,2,3,4,5};
for(i=0;i<5;i++){
    printf("%d",*(p+i));
}
```

指针相减

表示指针之间的偏移量

仅当原指针和结果指针都指向同一数组中的元素, 或该数组的尾后一位置, 行为才有定义。

Warning

警惕越界行为!!!

void*

使用void类型指针的时候无法进行解引用, 也无法进行加减运算

使用void*类型的指针的时候一定要对其进行强制类型转化

```
int a = 1;
void *p = &a;
printf("%d\n",*((int*)p));
```

警惕UB(未定义行为)

我们很多时候需要警惕未定义行为，比如说

```
a[i] = ++i + 1;
```

i在这里多次使用且数值发生了改变，没人知道a[i]先执行还是++i先执行

还有就是

```
// i=0
int a = f(i++)+f(i++)-f(i++);
```

虽然在最后的结果上编译器会将其翻译为int a = (f(i++)+f(i++))-f(i++);
但没人知道哪个f(i++)最先执行，可能是第一个，也可能是第三个

问题

经过学习,我相信你也已经掌握了C语言的一些基本技能了,那下面几个问题可以去尝试一下哦

1. 定义一个常量 `const int a=0;` 能不能使用一个指针访问它来更改它的值?
2. switch语句中,不使用break会发生什么?为什么有时候我们需要使用switch而不是一直使用if-else语句。