

# 2023-11-6 周一朋辈导师辅导课程

说明：（1）本文档不是教材，不要当作知识点去记忆；也不是课程的全程记录，涉及实践的部分是缺失的。本文档作用在于辅助听讲以及课后回忆、复习，掌握本课程内容的最主要方法是实践；（2）本次课程讲授内容偏抽象、扩展。需要全面复习、预习知识点的同学可以关注其他导师的课程通知。

## 一、编程学习与实践的一般性问题

### 1. 如何提问？如何界定程序的错误？

提问的原则：（1）提供**全部相关方面**的信息；（2）以**最合适的方式**提供信息；（3）信息要**完整**。所有的问题都不是孤立的，而是处于一定语境中的，提问者身处语境中，掌握着一些“理所当然”的信息，但对于被提问者来说，这些信息的缺失容易导致困惑。因此，提问者应该有**语境意识**。不要小看这个意识的重要性，今后参加大型项目、科研工作等，想要和同事合作、汇报进度，没有语境意识的交流是令人困惑的。

对于编写程序而言，相关信息包括：

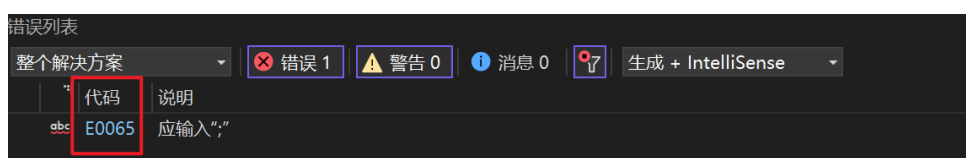
（1）**要解决的问题**，体现在 OJ 题中即为题目。题目最好使用 markdown 语法提供（自行了解 markdown 语法），其次是截图形式，但无论采用什么方式，都一定要完整，不仅包含题目描述，也要包含输入、输出样例；

（2）**源代码**。以文本方式提供，建议直接提供 .c 格式源文件，**不要使用截图!!!**好的程序是调试出来的，不是靠眼睛看出来的，对于复杂的大项目而言更是如此，提问者应该提供文本形式的代码以便调试；

（3）**报错信息**。请将编译器发现的错误和警告一并提供；

组成元素	格式	含义
文件名	filename:	出现错误或警告的文件名
行号	line:	出现错误或警告的行号
列号	column:	出现错误或警告的列号
错误/警告类型	error/warning:	错误或警告类型
错误/警告信息	message	错误或警告信息

行号用于定位错误，是十分重要的（顺便一提，描述某个代码的位置，一般就使用行号），一般表明该行或其上方出现错误，需要提供；列号用于更精确的定位错误位置，如分号缺失就指出分号的位置。有些 IDE 的报错形式做了修正，但实质一样。Visual Studio 还会提供错误号，微软给每种类型的错误编了序号，并在官网给出了修正该错误的指导，请利用好该资源。



实例分析：

```

1104-6.c: In function 'main':
1104-6.c:4:23: error: expected ';' before 'return'
printf("hello world")
                    ^
                    ;
return 0;
~~~~~

```

(4) **运行截图**。输入数据，得到输入输出，截图提供，尽量截取完整的窗口。

常见(OJ)错误:

- (1) **语法错误**，一般无法通过编译，注意库函数不一定通用；
- (2) **运行时错误**：运行过程中产生的错误，编译器无法在编译时期预测运行时会遇到情况，一般无法发现此类错误，比如除以零、数组越界、内存不足、文件不存在等。这种错误通常会导致程序崩溃或终止；
- (3) **逻辑错误**：有时候和语法错误一同存在，如 `a = b + c * c` 忘记加括号等，较难发现；
- (4) **时间超限**：判断复杂度，如果复杂度不高，则大概率是程序没能正常退出，比如循环没结束，正在等待输入等；
- (5) **内存超限、输出超限、格式错误**；

界定错误的方法：调试。最常见的方法就是输出中间结果，并判断输出是否符合预期。

(举例) ...

2. 当我知道一个新的知识但对于其细节不能确定时，如何自己编写程序验证？如何通过自己给自己出题来学习？如何利用报错/警告信息？如何利用好编译器和调试器来学习 C 语言？

C 语言学习的一个方便之处就是有编译器作为辅助。当在课本、PPT 上看到不理解的知识时，可以自行设计一个程序来验证。注意验证时要想办法输出中间变量来验证自己对原理的理解，而不是简单地把各种情况当成特例去记忆。

(举例) ...

3. 如何阅读官方文档/帮助文档？如何从文档中获取信息？如何使用标准库？如何正确记忆函数？

本部分的要点在于不要仅凭感觉记忆函数用法。靠感觉是靠不住的，很可能导致未知的错误。事实上，文档上提供的内容已经包含了你调用函数所需要的一切信息，函数声明中的每一个字母都是有用的，记住这些就是记住了函数用法。

文档里有什么：一般会有函数声明和其他解释性内容。其中最重要的是函数声明，例如 `double atan(double arg)`；其中，**函数声明是最重要的信息**，它包含了调用该函数所需的一切信息，记忆函数其实就是记忆函数的原型。从用户（当我们去调用库函数时，我们是用户的角色）的角度，函数是一个**黑盒子**，它获取一些特定格式的输入，输出特定格式的数据，只要输入符合格式，就能够获取正确的输出（调用标准库的时候，应该对开发者有这样的信任）。因此，我们不能局限于记住参数大概是什么，而是需要记住所有参数和返回值的数据类型。

## filter2D

Convolves an image with the kernel.

C: void **cvFilter2D**(const CvArr\* **src**, CvArr\* **dst**, const CvMat\* **kernel**, CvPoint **anchor**=cvPoint(-1, -1))

**Parameters:** **src** – Source image.

**各个参数的解释**

**函数声明：形参、返回值类型**

**dst** – Destination image of the same size and the same number of channels as **src**.

**ddepth** – Desired depth of the destination image. If it is negative, it will be the same as **src.depth()**.

**kernel** – Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using **split()** and process them individually.

**anchor** – Anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center.

**delta** – Optional value added to the filtered pixels before storing them in **dst**.

**borderType** – Pixel extrapolation method. See **borderInterpolate()** for details.

The function applies an arbitrary linear filter to an image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

## MSELOSS

```
CLASS torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean') [SOURCE]
```

(举例) ...

### 4. 当我发现输出结果不正确时，如何追踪程序的运行过程？如何通过 debug 检查代码的逻辑错误？

调试、输出中间变量。要设计好输出格式，不要吝啬，多附加一些信息，不要只是打印几个字符，这样阅读难度较高。

(举例) ...

## 二、从细节问题入手深入知识点

### 1. 什么是表达式？什么是表达式的值？表达式如何计算？什么是常量？什么是变量？什么是语句？什么是左值与右值？什么是“左结合”和“右结合”？

在 C 语言中，**表达式**是由**常量**、**变量**、**函数调用**以及**运算符**按照 C 语言的语法规则组合起来的式子。所有的表达式都有值，部分表达式可以完成求值以外的其他功能，如赋值。注意，是否有求值以外的功能有运算符本身决定，但只要是表达式，就有一个值。

**表达式语句：**由表达式组成，以分号结尾

什么是“**副作用**”：在编程语言中，“副作用”是一个术语，用于描述函数或者表达式在计算结果的同时，对系统状态产生的改变。这种改变可能包括修改全局变量、修改输入参数、进行 I/O 操作、调用其他有副作用的函数等等。运算符和函数本质上没有区别（符合广义的函数定义），而函数的天职就是接收输入、输出数据，与这些无关的都是“副作用”。例如函数 **printf**，接受格式化字符串输入，输出正确输出的字符个数，而将字符显示在屏幕上则是“副作用”（需要理解）；赋值符号“=”输入一个左值、一个右值，输出被赋值的值，对左值内存的修改是“副作用”。然而，很多时候我们就是要这个副作用，比如上面两个例子。

**什么是序列点：**

我们都知道 **a++** 会先使用 **a** 的值

语句结束：C 和 C++ 中的每个完整语句后面都是一个序列点。例如，**a = 1; b = a;** 这里 **a = 1;** 语句后面就是一个序列点。

逻辑与(&&)和逻辑或(||)运算符: 这些运算符都是短路的, 这意味着如果左侧的表达式可以决定结果, 那么右侧的表达式将不会被执行。因此, 左侧表达式和右侧表达式之间是一个序列点。

条件运算符(?:): 条件运算符的条件部分和两个可能的结果之间都是序列点。

逗号运算符(,): 逗号运算符左侧和右侧的表达式之间是一个序列点。

函数调用: 函数参数计算完成和函数体开始执行之间是一个序列点。

**左值(lvalue)**是指可以被赋值的对象, 它必须有一个确定的内存地址(有确定地址、可以赋值)。例如, 变量、数组元素、结构体成员、指针解引用等都是左值。左值可以出现在赋值表达式的左边或右边, 当左值出现在右边时, 它表示该对象的值。有确定地址, 意味着它通过声明语句被专门声明, 而不是临时产生的; 同时没有通过 `const` 关键字设置为只读;

**右值(rvalue)**是指可以被读取的数据, 它可以是一个常量、一个表达式、一个函数返回值等。右值只能出现在赋值表达式的右边, 它表示一个临时的数值, 不一定有一个确定的内存地址。右值不能被赋值, 否则会编译错误。

做表达式计算题的两种思路: 从最高优先级开始、从最低优先级开始(举例...)

**例题:** 下面哪些是表达式, 是否合法? 哪些是语句, 是什么语句?

```
(float)a = b + c;
int a = 5;
a + b
printf("Hello, World!");
if (a > b) printf("a is greater");
a = b = 5
a + b = 4
a + (b = 4)
a++
a = (b > c) ? b : c;
int a = "hello";
a = ++b + b++;
printf("first\n"), printf("second\n");
3 = a;
return 0;
```

**例题:** 计算下列表达式, 说出计算过程

```
int main() {
    {
        int a = 5, b = 7;
        a = b = a + b;
        printf("a = %d, b = %d\n", a, b);
    }
    {
        int a = 5, b = 7;
        int result = (a++, b++, a++ + ++b);
        printf("a = %d, b = %d, result = %d\n", a, b, result);
    }
}
```

```

    int a = 5, b = 3, c = 10;
    a *= b += c /= 2;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}
{
    int a = 5, b = 3, c = 10;
    int result = a && b || c++;
    printf("a = %d, b = %d, c = %d, result = %d\n", a, b, c, result);
}
{
    int a = 1, b = 0;
    int result = a && b++ || b;
    printf("a = %d, b = %d, result = %d\n", a, b, result);
}
{
    int a = 6, b = 4, c = 2;
    int result = a = b ? a += 10 : printf("here\n");
    printf("a = %d, b = %d, c = %d, result = %d\n", a, b, c, result);
}
{
    int a = 6, b = 4, c = 2;
    int result = (a = b) ? a += 10 : printf("here\n");
    printf("a = %d, b = %d, c = %d, result = %d\n", a, b, c, result);
}
return 0;
}

```

## 2. 什么是数据类型？为什么要有数据类型？什么是类型转换？如何理解 C 语言中的类型转换规则和类型安全问题？

数据类型决定了数据的存储方式、占用空间大小以及如何进行运算。例如，在 C 语言中，`int`、`float`、`char` 等都是数据类型。在内存中，所有数据都只是一串 `1` 和 `0`，数据类型本质上提供了一种“视角”去看待这些数据，数据类型首先确定以怎样的方式理解这些 `1` 和 `0`，其次确定对多长的 `1` 和 `0` 进行解释。数据类型存在的目的就是解释数据的语义，内存中的一串 `1` 和 `0` 可以是任何事物，但因为有了数据类型，它被解释为一个整数、一个坐标、一张图像。

需要说明的是，数据类型不是天然地能够相互转换的，C 语言的基本数据类型能转换是因为它们对应的现实意义都是数值，都属于实数域，可以定义强行转换方法。但图像类型、矩阵类型、音频类型就不一定能相互转换了。注意，我们目前学习的是基本数据类型，其性质不能等同于广义的数据类型。

然而只定义数据的解释方式还不够，一个数据类型之所以是数据类型，取决于它拥有的运算。数据类型可以不严格的类比为代数系统。代数系统的概念：非空集合  $A$  和  $A$  上  $k$  个一元或二元运算  $f_1, f_2, \dots, f_k$  组成的系统称为一个代数系统，简称代数，记作  $(A, f_1, f_2, \dots, f_k)$ 。由定义可知，一个代数系统需要满足下面 3 个条件：(1) 有一个非空集合  $A$ ；(2) 有一些建立在集合  $A$  上的运算；(3) 这些运算在集合  $A$  上是封闭的（建议理解下这个概念）。所有的运算都定义在一定的域中，`int` 上的加法只能对 `int` 进行操作。



C 语言有时可以对不同数据类型操作，这是因为 C 语言进行了数据类型转换，如：`int a = 3 + 4.3f + 5.1;` 这个语句可以拆分成三部分：（1）先进行左边的加法操作，+号的两个操作数类型不同，需要将 `int` 转换为 `float` 类型，因此实际执行 `(float)3 + 4.3` 得到临时数值 `7.3f`；（2）执行第二个加法，+号两边的操作数类型不同，因此实际执行 `(double)7.3f + 5.1`，得到临时数值 `12.4`；（3）执行赋值操作，左值为 `int` 类型，因此将右值类型转换为 `int` 再赋值，即 `a = (int)12.4`。+号两边的数据一定属于同一个数据类型，如果不是就需要类型转换。（举例说明为什么，比如 `float`、`double` 乘法为什么要转换）

注：`printf` 中占位符不会进行类型转换：`printf("%d\n", 3.4);` // output: 858993459

3. 同为循环，`for`、`while`、`do while` 功能上等价吗？应用上有什么区别？如何选择？如何保证循环次数符合预期？`goto` 呢？

`for`、`while`、`do while` 都是 C 语言中的循环语句，功能上，这三个语句加上 `goto`，四个都是等价的。

循环和分支是想解决流程控制的问题，执行是按顺序一句一句存储的，那么计算机就必须按顺序一行一行执行吗？这是不一定的，有时候，我们希望计算机反复执行一段代码，或根据情况跳过一些代码，而这两种操作就对应了循环结构和分支结构。数学上已经证明，顺序、循环、分支就足够解决问题了。

实现这些流程控制的方式是跳转，在 CPU 内部有一个寄存器用于存储目前正在执行的代码的地址，计算机每次就执行这个地址对应的代码。修改代码的执行顺序本质上就是修改这个寄存器的值，想跳转到哪一行代码就把这一行代码的地址赋值给这个寄存器。从这个意义上，循环和分支的本质就是 `goto`。`goto` 实际上是对机器指令 `JMP` 的封装，具有过高的自由度，使得程序运行逻辑被破坏，不推荐使用。

练习：为掌握 `for`、`while`、`if-else` 等的原理，请大家实现各种循环、分支语句对应的 `goto` 语句实现。

<pre>int i = 0; start: if (i &lt; n) {     // your codes     i++;     goto start; }</pre>	<=等价=>	<pre>for(int i = 0; i &lt; n; i++){     // your codes }</pre>
<pre>if(n == 1){     ; } else if(n == 2){     ; } else {     ; }</pre>	<=等价=>	<pre>if (n != 1) goto L1; // code for n == 1 goto end; L1: if (n != 2) goto L2; // code for n == 2 goto end; L2: // code for else end;</pre>

语法：`for`（表达式 A；表达式 B；表达式 C）语句/语句块 D，执行顺序 ABDCBDCBDC...

`while`（表达式 A）语句/语句块 B，执行顺序 ABAB...

`do` 语句/语句块 A `while`（表达式 B）；执行顺序 ABAB...

上述所有的表达式虽然名义上是初始化、控制、调整表达式等，但其实可以是任何表达式，但编写程序时保证逻辑无误即可。注意理解表达式的含义。其中，`for` 中的表达式 B、`while` 中的表达式 A、`do-while` 中的

表达式 B 又有些特殊，所有表达式都有值，但它们的值有用处，程序得到这些值后会判断是否为真值（不是零就是真值），如果判断为假则退出循环。

练习：请分析输出结果，并说明为什么

```
for(printf("A "), i = 0; printf("B "), i < 5; printf("C "), i++){
    printf("D ");
}
```

练习：请分析其输出结果

```
int i = 0, a = 1;
for(i = 0; i < 3, i++, i < 4; i++, printf("here\n")){
    printf("here\n");
}
printf("a = %d\n", a);
```

选择：while 循环常常用于不确定循环次数的情况（举例...），for 循环常常用于明确清楚循环次数的情况。OJ 题目中，一般都使用 for，偶尔使用 while，基本不用 do-while，不用 goto。注意，跳出多重循环会使用 goto。

循环 n 次（套路）：

```
int i = 0, n = 5;
while(i++ < n){
    printf("here\n");
}
/*这里要注意 i 的值被修改了，后面要用到 i 的话要注意*/

while(n-- > 0){
    printf("here\n");
}
/*优点是不需要额外变量 i，缺点是 n 的值会被修改，而且 n 必须是变量*/

n = 5; // 重新修改 n 为 5
for(int i = 0; i < n; i++){
    printf("here\n");
}
/*最推荐的方式*/

for(int i = n; i > 0; i--){
    printf("here\n");
}
```

4. switch case、if else 和三目运算符( ? : )功能类似，其原理上有什么区别？语法上到底有什么区别？如何记住这些？

语法：重点在于记忆各个部分是表达式、还是常量等等

```
switch (表达式 A) {
    case 常量 B: 语句/语句块 C
```

default: 语句/语句块 C

}

if (表达式 A) 语句/语句块 B else if (表达式 C) 语句/语句块 D ... else 语句/语句块 E

表达式 A ? 表达式 B : 表达式 C;

前文已经提到，分支的实现本质就是语句的跳转（类似于 `goto`）。但 `if-else` 会从头到尾一个个检查条件是否满足，一旦满足，则进入该条件对应的语句/语句块，执行完后跳出整个 `if-else` 结构；`switch-case` 则不会一个个检查，它用空间换性能，会直接跳转到符合条件的语句，性能更好（当然适用条件也比较少）。`switch-case` 从头开始，只要发现表达式 A 的值和 `case` 后面的常量相等，就执行该标签后的代码，执行完后不会退出整个 `switch-case`，而是会继续往下执行，除非手动使用 `break` 退出。三目运算符的要点在于记住它不是语句，而是表达式，其操作的内容也是表达式，不是语句。

练习：假设 a 为 3，判断正误，分析语句

```
a > 5 ? printf("a > 5\n") : printf("a <= 5\n");
a > 5 ? a++, b-- : a++, b++;
a > 5 ? {a++, b--;} : {a++, b++};
a > 5 ? {a++, b--;} : {a++, b++};
a > 5 ? a++ : if(a > 6) b++;
case a > 3: a++; printf("a == 1\n");
case a: a++; printf("a == 2\n");
```

验证：

```
int a = 3;
if(a > 1){
    printf("a > 1\n");
} else if(a > 2){
    printf("a > 2");
}
// output: a > 1
// 虽然 else if(a > 2)也符合，但是在执行完 if(a > 1)后的语句后就退出了
```

```
int a = 1;
switch(a) {
    case 1: a++; printf("a == 1\n");
    case 2: a++; printf("a == 2\n");
    default: a++; printf("Into default\n"); break;
    case 3: printf("a == 3\n");
}
```

Output: `a == 1`  
`a == 2`  
`Into default`

使用建议：做 OJ 用 `if-else` 就可以了，偶尔可以用三目运算符简化代码。`switch-case` 适合用于菜单选择等情况，现阶段可以暂时不用。

5. C 语言的数组本质上、物理上是什么？声明时指定的数组长度是形同虚设吗？如何选择数组长度？定义后就万



事大吉了吗？为什么要以及如何防止数组越界？有没有更高级的手段？

物理上，数组就是内存中一段连续的空间。想要操作一个数组，需要知道：数组的起始地址、每个元素的长度、如何解释每个元素对应的 0 和 1。后两个条件由数组元素的数据类型决定，而数组名本身则存储了数组的起始地址。声明数组时规定的数组长度仅用于确定预留空间，程序在运行时（runtime）阶段是不知道数组长度的，sizeof 之所以能获取长度是因为它是在编译的时候执行的，而不是在运行时求值。

C 语言不会判断是否越界，因为判断是否越界性能损耗过大（解释...）。

数组长度的选择：目前阶段我们只实现静态长度的数组，因此建议尽可能为数组留足充足的空间，即使有大量的空间浪费。可以使用 `#define SIZE 1024` 定义数组长度。

高级手段：手动实现数组的越界检查（请运行验证此程序并分析流程）：

```
#include <stdio.h>
#include <stdlib.h>

void set_value(int a[], int i, int value){
    if(i < a[0]){
        a[i + 1] = value;
    } else {
        printf("error\n");
        exit(0);
    }
}

int get_value(int a[], int i){
    if(i < a[0]){
        return a[i+1];
    } else{
        printf("error\n");
        exit(0);
    }
}

int main(){
    int a[100] = {0};
    a[0] = 10;
    set_value(a, 3, 1);
    printf("The 4th element of a is %d\n", get_value(a, 3));
    set_value(a, 11, 1);
    return 0;
}
```

哨兵法是一种常用的算法设计技术，它主要用于简化某些算法的实现，特别是在循环结构中。哨兵是一个特定的值，它被添加到数据结构中以标示数据的结束或边界，从而避免在循环中进行额外的边界检查。一个应用：字符串确定边界。

```
int numbers[10] = {1, 2, 3, 4, 5};
```

```

int target = 3;
/*A: fill you code*/
int i = 0;
while (/*B: fill you code*/) {
    i++;
}
if (/*C: fill you code*/) {
    printf("Finded. %d is at %d\n", target, i);
} else {
    printf("Can't find.\n");
}

```

答案: A: numbers[5] = target; B: numbers[i] != target C: i < 5

如何实现动态长度的数组：学习了指针后讲解。基本原理为，当数组长度为  $n$  时，就声明长度为  $n$  的数组；当数组长度变成  $m$  时，就新建一个长度为  $m$  的数组，并将之前的数组的所有元素复制过去。其实动态数组、越界判断的实现并没有什么神秘的，就是基础的 C 语法，C++ 的部分容器就是这样实现的。

6. C 语言中，printf、scanf 的底层原理是什么？大致的工作流程是什么？（已有导师讲过，可能从略）

孟子杰学长讲过该专题，此处从略。

7. 什么是缓冲区、输入流、输出流、标准输入流、标准输出流？什么是标准输入/出流的重定向？它们本质上/硬件上做了什么操作？

缓冲区是一块内存区域，主要用于临时存放数据，让我们能够更加自如地控制数据的流动。标准输入流、标准输出流通常连接到终端（命令行界面），标准输入从终端接收数据（例如用户的键盘输入），标准输出将数据发送到终端（例如显示在屏幕上）。

输入、输出的过程并不神秘，其实就是将一个个的 0 或 1 读取、输出。那么为什么一定要从键盘读取数据呢？我们可以从文件中读取数据，然后输出到文件中。同时需要注意，输入、输出的数据都会被复制到显存中以显示出来，这使得它们显示的时候混在一起，但其实输入流、输出流的数据是分开的。

### 三、抽象内容：编程思维的一般性问题、编程的本质、编程语言的本质

#### 1. 程序=数据+算法

程序由数据（它需要操作的信息）和算法（它如何操作这些信息）组成。数据是静态的部分，算法是对数据的操作。这是由图灵机的性质决定的。（举例说明为什么所有程序都是数据+算法...）

#### 2. 函数、模块、设计者/用户的相对论

计算机的一大优势：易于模块化，调用已有模块成本低。

设计者向上层提供服务，用户使用下层提供的服务。设计者/用户是相对的，一般的程序员相对 C 库的开发者是用户，对于使用他们开发的程序的人员而言又是设计者（解释）。

#### 3. 什么是程序设计？什么是软件工程？它们的本质是什么？

程序设计（Programming）是将算法转换为特定编程语言的代码的过程。这包括在特定编程语言中寻找解决问题和执行任务的最优方式，以及编写、测试、调试和维护代码。程序设计的本质是解决问题和实现功能。

软件工程（Software Engineering）是一种系统性、规范化和量化的方法，用于开发、操作和维护软件。要点在于工程思维。

4. 什么是高内聚、低耦合？什么是解耦？如何解耦？解耦就一定好吗？

"高内聚"和"低耦合"是软件工程中两个重要的设计原则，高内聚是指模块或组件内部元素之间关联程度高。高内聚意味着一个模块或组件只负责一个功能或任务，它的所有元素（如类、方法、属性等）都服务于同一目标。低耦合是指模块或组件的独立程度高。低耦合意味着一个模块或组件的改变尽可能少地影响其他模块或组件。耦合度低的模块之间通过特定设计的通道进行交流。（图例说明）

解耦的方法：模块化、分层化、设计模式

耦合性低不一定好，耦合性低带来的最大的问题就是性能损失，操作系统内部部分耦合度就比较高，因此虽然原理不太难，但难以维护。

四、资源推荐、学习路线指导、复习方法指导

课程：浙江大学 翁恺 <https://www.bilibili.com/video/BV1dr4y1n7vA>

算法课程：北京大学 郭炜 <https://www.bilibili.com/video/BV1Zb411q7iY>




编程思维科普：<https://www.bilibili.com/video/BV123411p7rf>

教材：C Primer Plus

刷题（适合基础实在需要加强的同学）：各类计算机二级 C 语言教程，见群文件

代码阅读：目前阅读大型项目代码还较为吃力，建议阅读《C Programming Language》（被誉为程序员圣经）当中的代码，也可以一起跟着做例题。不过这本书本身不是那么适合学习。见群文件。

算法：如果有同学对算法有特别兴趣可以自行扩展。首先推荐《算法图解》，想进一步学习的推荐《算法》，建议结合北京大学郭炜老师课程使用。

	[图灵原创].算法的乐趣.pdf	10-25
	8.4 MB 19次	来自：21-数媒-廖...
	[图灵程序设计丛书].算法 第4版.pdf	10-25
	35.8 MB 16次	来自：21-数媒-廖...
	算法图解(绝对推荐!!!!!!).pdf	10-25
	17.1 MB 50次	来自：21-数媒-廖...

竞赛：如果想专门竞赛道路可以考虑预习《数据结构与算法》课程，并购买专门的 ACM 教材。

不少同学由于刚上大一，对自己定位不明确，想学太多，其实对于大多数同学来说能够弄懂本学期的 OJ 题目就很难了，因此对一般学生只推荐《C Primer Plus》、北京大学郭炜老师算法课程，其他的除非是大佬、决定走 ACM 道路、有基础，否则没必要。