# C++重点知识总结

本文档使用方法：

**正确**方法：利用本文档**回顾梳理**本学期所学知识，对于自己尚不能彻底理解、熟练掌握的知识，及时通过课本查缺补漏。

**错误**方法：只要我背过这些知识点就能考高分！

**注意：一定不要单纯的背诵文字！要在记忆的基础上联想到应用！**

## 基本结构：

C++程序代码由**头文件**与**源文件**组成。

**头文件**中包含：版权与版本信息、宏定义、函数声明和类型的定义。

- ✧ 需要用 ifndef/define/endif 结构来产生预处理的宏定义块，以防止块中的内容被重复包含。
- ✧ 类成员函数可以在类定义中被定义，并且自动生成内联函数。尽管书写起来比较方便，但会造成风格不一致，因此建议头文件中只声明成员函数而不定义。

**源文件**中包含：版权与版本信息、对头文件的包含、程序功能代码的实现（包括类的成员函数的实现）。

- ✧ 用< >包含头文件则直接在系统库中查找头文件；用“ ”包含头文件则先在当前工程目录下查找，若工程目录中不存在然后在系统库中查找。
- ✧ 库分成头文件和实现文件的意义：1.很多时候不便公开源文件代码，因此仅用头文件来支持开发者调用库功能；2.头文件能加强安全类型的检查，如果接口匹配错误会报错，以便开发者调试。

## C++基本要素：

**标识符**：只能由字母、数字、下划线组成，且**不能以数字开头**。标识符在 C++中区分大小写，长度是任意的，一般前 1024 个是有意义的。C 语言中一般前 16 个是有效的。

**常量**：在运行时不能被改变，在定义常量时可以设置初始值。对于常量，编译器将其放置于一个只读的存储区域。

**变量**：在程序中可以被多次赋值。


```
//变量的初始化形式
int min = 0;        //初始化为0

int min = min;      //初始化为自身,这种形式合法，但不明智

int min(10);        //隐式初始化为10

int min = 10, max = min + 20;     //后续变量可以利用前续变量作为初始化值

int min = int();          //利用数据类型构造函数初始化
double max = double();
```

## 变量的存储类型：

变量的**声明**是告知编译器**变量的名称**和**数据类型**。
变量的**定义**是为**变量分配存储区域**。
通常在**同一个语句**中完成变量的声明与定义。

- ✧ **类定义中成员变量的声明，只是声明而不是定义**。
- ✧ **使用 extern 只能声明而不能定义变量**。
  - ■ 若一个文件中声明定义了一个全局变量 int var=0; 那么，在另一个文件中声明 extern int var; 即可访问此变量。
- ✧ static 存储类型表示变量在函数或模块内具有“持久性”，也称为静态变量。静态变量可分为**局部静态变量**和**全局静态变量**。
  - ■ **局部静态变量**：函数内的变量用 static 修饰时，将被分配在持久的存储区域，函数调用结束后并不释放，保留其值以便下次调用。**局部静态变量的作用域为当前函数**，不能被外界函数和文件访问。
  - ■ **全局静态变量**：**作用域仅限于当前定义的文件**，不能被其它文件使用 extern 关键字访问。
- ✧ 使用 register 关键字，表示变量将被放置在 **CPU 寄存器**中。访问 register 变量要比访问普通变量快得多，但 **register 只能用于局部变量或作为函数的形式参数**，不能用来定义全局变量。
- ✧ 变量有静态存储和自动存储，**全局变量和静态变量是静态存储**的，**普通的局部变量是自动存储**的（即进入程序块时分配内存、离开程序块时释放内存）。**auto 关键字表示变量自动存储**，默认情况下，局部变量均属于 auto 变量。


# 数据类型：

- **数值类型：**

  分为**整型**和**实型**，整型又分为有符号型和无符号型。

```
//数值类型的存储字节对比
//      (signed) int                4Bytes
//      unsigned (int)              4Bytes
//      (signed) short              2Bytes
//      unsigned short (int)        2Bytes
//      (signed) long (int)         4Bytes
//      unsigned long (int)         4Bytes
//      float                       4Bytes
//      double                      8Bytes
//      long double                 8Bytes
```

- **字符类型：**
  - C++中用**单引号**来确定**字符常量**，用**双引号**来确定**字符串常量**。
  - 字符是以 ASCII 编码的形式存储的，因此**可以直接将整数赋值给字符变量**。
  - 字符存储用一个字节。
  - **数字字符与整型数字之间的转化**可以通过**加‘0’或减‘0’**来实现，例如 3 加‘0’得到‘3’、‘8’减‘0’得到 8，等等。
- **数组类型：**

  数组的**初始化**要注意一些细节，如下图。

```
//数组定义的一些注意细节
int array[10] = { 0 };   //将全部元素初始化为0

int array[10] = { 1 };   //将第一个元素赋值为1，其余元素赋值为0
```

  **定义二维数组并初始化时，可以省略第一维的长度，但不可省略第二维的长度。**
- **布尔类型：**布尔类型和整数类型的变量可以相互赋值。
- **枚举类型：**
  - 枚举类型是用 int 类型实现的，占用 4 个字节。
  - 定义格式为：enum 枚举类型名 {常量 1，常量 2，…};
  - 定义枚举类型时可以为各常量提供一个整数值，**默认情况第一个数为 0**，没有指定的值应为前一个值加 1。
  - 在定义函数时，若将函数参数设置为枚举类型，可以限制调用函数必须提供枚举类型中的某个值，而不能随意输入一个整数。
- **结构体类型与结构体变量：**
  - 定义格式为：struct 结构体类型名 {成员分量声明} 结构体变量名;
  - 若只需要定义一次结构体变量，可以不写结构体类型名称。
  - 访问结构体成员用“.”，两个同类型的结构体变量可以直接相互赋值。


- ✧ **指针**是用来存放变量地址的。**通过变量名访问变量是直接访问，通过指针访问变量是间接访问**。
- ✧ **注意区分指针数组和数组指针**。如下，

```
//指针数组与数组指针的区别
int *a[5];       //这是指针数组，表示有5个指向整形的指针

int(*a)[5];      //这是数组指针，表示指针a指向5个整形元素的数组
```

- ✧ 用 const **关键字**来修饰指针的几种情况：

```
//用const关键字来修饰指针的三种情况
int ivar = 10;
const int *pvar = &ivar;     //const与int可以互换
//*此种情况const是修饰*pvar，故用户不能修改pvar指向的值，但可以修改pvar指向的地址

int ivar = 10;
int *const pvar = &ivar;
//*此种情况const修饰的是pvar，故用户不能修改pvar指向的地址，但可以修改pvar指向的数据

int ivar = 10;
const int*const pvar = &ivar;
//*此种情况下，值和地址均不能修改
```

- ✧ **引用**是目标对象的一个别名，**操作引用与操作实际的目标对象是相同的**。
  - ■ 引用的**定义格式**如下：数据类型 &引用名称 = 目标对象;

## sizeof()运算符

- ✧ sizeof()用于返回变量、对象或数据类型在内存中占用多少个字节。
- ✧ 在 32 位系统中，指针变量总是占用 4 个字节。

## 逗号表达式

- ✧ 逗号运算符的**优先级最低**，逗号表达式的**最终结果**取其中**最右边表达式的值**。

## const 与 define 的比较

- ✧ C++语言可以使用这两者来定义常量，但是使用 const 更好。
- ✧ const 常量有数据类型，而宏常量没有数据类型，编译器可以对前者进行安全检查，而对后者只能进行字符替换，字符替换可能会发生意想不到的错误。


## 栈存储与堆存储的区别

C++程序占用的内存可以分为以下几个部分：

- **栈区**：由编译器**自动分配释放**，存放函数的形式参数、局部变量等
- **堆区**：由**用户分配释放**。**手动分配空间后要手动释放，否则可能会导致内存泄露**

```
//堆的分配方式
int* pvar = new int(10);   //初始化堆数据为10
delete pvar;

int* pvar = new int[5];  //初始化一个数组空间
delete[]pvar;
```

- **全局区**：全局变量和静态变量是存储在一起的；
- **文字常量区（字段区）**：字符串常量存放于此区，程序结束后由系统释放；
- **程序代码区**：存放函数体的二进制代码；

## 类中的常量

- 可以用 const 在类的定义中声明常量数据成员。
- 但**常量数据成员只在某个对象生存周期内是常量，而对整个类而言是可变的**。创建多个对象时，不同对象的常量数据成员的值可以不同。
- 常量数据成员的**初始化只能在类的构造函数的初始化列表中进行，不能在构造函数的函数体中进行**。因为构造函数会在执行函数体之前先对所有数据成员进行初始化，不在初始化列表中的数据成员会获得随机的初始值。
- 可以使用枚举类型在类中建立一个恒定的常量，枚举常量不会占用对象的存储空间，在编译时即确定其对应的数值。

## C++语句

- C++语句一般是由表达式和分号构成的。
- 只有分号的语句为空语句。程序中允许有多条空语句，空语句不执行任何功能。
- { }是复合语句，也可用复合语句来代替空语句，但是括号后面没有分号。
- 多条 if 语句，程序会依次执行各条语句，因此条件判断较为复杂。if/else 语句，程序只选择一个分支按条件执行。
- return 语句用来退出当前函数的执行。若函数没有返回值（返回类型为 void），则只使用 return，不加任何表达式。使用 return 语句时，要注意，**如果函数在堆中分配了内存，则在 return 语句前要考虑释放内存，以防止内存泄漏（除非函数返回后仍能通过外部的指针或引用访问该堆空间）**。


- ✧ exit 语句用于终止当前进程，通常用于结束当前的应用程序。exit 包含一个整型参数，用于标识退出的代码。与 exit 不同的是，return 语句只退出当前调用的函数。除非当前函数是主函数，否则 return 不会结束当前进程，而 exit 会直接结束当前进程，无论当前函数是否是应用程序的主函数。

## C++函数

- ✧ 在定义函数时，**若函数返回类型不是 void，一定要在函数中加入 return 语句**；若调用函数处于被调用函数之后，则要对被调用函数进行前置声明。声明语句可以放置于 main 函数内部。
- ✧ 若函数有**多个参数**，应保证**带默认值的参数出现在参数列表的最右方**。
- ✧ 在**数组作为函数参数**时，不指定数组的大小，调用函数时 C++编译器不对数组长度进行检查，只是**将数组首地址传给函数**。
- ✧ 参数的传递方式：
  - ■ 值传递：在函数调用时为形式参数分配内存空间，再用实际参数为初值对形式参数进行初始化，函数中对形式参数的修改不影响实际参数的值；
  - ■ 地址传递：如果函数修改了形式参数的值，实际参数的值也会发生改变。
  - ■ 通常在定义函数时，如果参数为数组、指针或引用类型，则用地址传递方式，否则用值传递方式（**指针其实是已值传递+他址传递**）。
- ✧ 内联函数：对于程序中的函数调用表达式，如果是内联函数，编译器直接将函数体代码代替函数调用表达式，省去了程序跳转的过程。用 inline 关键字标识内联函数。**函数内联的目的是提高程序的执行效率，但代价是代码膨胀。**
- ✧ 函数的重载：指多个函数具有**相同的函数名**，而**参数类型或参数个数不同**。在调用时，编译器正是通过参数类型和参数个数来区分调用哪个函数。
  - ■ **返回类型不作为区分重载的依据；对于参数来说，如果参数是指针或引用类型，const 作为区分依据，否则不作为区分依据；参数的默认值不作为区分依据。**
  - ■ 在局部作用域（如类类型）中声明函数，将使该域外的同名函数隐藏而不是重载。要访问其他域的函数，要使用::（作用域限定符）。
- ✧ 局部作用域和全局作用域：
  - ■ 局部作用域描述的是函数体中变量、常量等对象的作用范围。**处于同一个作用域的对象不允许重名**。当编译器发现变量名时，它在当前的局部作用域中搜索变量的定义，如果未定义则向外搜索，直到找到该变量的定义。
  - ■ **全局变量的作用域**是**从其定义的位置开始到其所在的文件结束为止**。全局变量若未初始化，其初始值为 0。局部变量若未初始化，则其值是不可预见的。


- ✧ 函数模板：函数模板提供了一种机制，使函数的返回类型、参数类型能够被参数化，从而支持用同一逻辑处理不同类型的数据，而不需要修改函数体。这极大的增强了函数的灵活性。函数模板在函数头的前部作如下定义：

```
//函数模板
template<class Type>  //Type是类型参数，可以用class、typedef、typename修饰

template<class Type, int len>;    //len是非类型参数
```

  - ■ 函数模板也可以重载，要注意区分。

## C++的类

面向对象的基本任务是**描述**对象并对对象进行**归类总结**。

类的**属性**和**对外接口**是类定义的重点和难点，原则是尽量让内部操作私有化，提供简单易用的接口函数。

- ✧ 在类定义中声明数据成员和成员函数时，若未指明访问权限，默认为 private。
- ✧ 若在类定义中完成了某个成员函数的定义，则该函数前即使没有使用 inline，也被认为是内联函数。
- ✧ 在类定义中声明成员函数时，若函数中不用修改当前对象的数据成员，则最好在成员函数声明的最后使用 const 关键字（如果是内联函数，则 const 在函数头之后、函数体之前），使之成为**常量成员函数**。
  - ■ 若类中包含指针成员，在常量成员函数中不可以重新为该指针赋值，但可以对指针所指向的对象进行赋值。
  - ■ 类的常量对象只能调用类中的常量成员函数或静态成员函数（因为静态成员函数没有 this 指针，所以不会修改调用它的对象）。
- ✧ 在类中，除静态成员可以通过类名直接访问外，**其它成员是通过对象来实现访问的（静态成员也可以通过对象来访问）**。
- ✧ 在定义类时并没有分配存储空间，只有当定义对象时，才分配存储空间（静态数据成员除外，它需要专门的定义语句）。可以定义一个类的指针，并使用 new 运算符为该类的对象分配内存，然后将这块内存的起始地址赋值给该指针。

## 构造函数与析构函数

- ✧ **每个类都有构造函数与析构函数**，构造函数在**创建对象**时被调用，析构函数在**撤销对象**时被调用。
- ✧ **构造函数负责对象刚生成时的初始化，析构函数负责对象即将消亡前的处理。**


- ✧ **构造函数没有返回类型和返回值**。若程序员没有提供构造函数和析构函数，则编译系统自动生成默认的构造函数和析构函数。一个类可以包含多个构造函数，各函数通过重载来区分。
- ✧ 用 new 分配内存的动态对象在创建时也调用构造函数，实际参数写在小括号中。
- ✧ 数据成员（包括对象成员）的初始化在**构造函数的初始化列表**中实现。
- ✧ **析构函数不仅没有返回类型和返回值，也没有参数，故不能重载**。

## 拷贝构造函数（复制构造函数）

- ✧ 复制构造函数与类的其它构造函数类似，以类名作为函数的名称。第一个参数为该类的常量引用类型（即：const 类名 & 参数名）。
- ✧ **下面的三种情况要用到复制构造函数：**
  ① 对象**以值传递**的方式**传入函数参数**；
  ② 对象**以值传递**的方式**从函数返回**（**即返回的不是指针或引用**）；
  ③ 对象创建时需要**用另外一个对象进行初始化**。
- ✧ 若程序员没有提供拷贝构造函数，编译器会生成默认的拷贝构造函数，这个函数仅仅使用老对象的数据成员的值对新对象的数据成员一一进行赋值。这称为**浅拷贝**，即只是给对象中的数据成员进行**简单的赋值**。
  - ■ 当对象中存在指针成员时，浅拷贝可能会出现问题。因为浅拷贝只是使两个指针有相同的值，即指向同一块空间，而不是通常需要的两块不同的空间。
- ✧ **深拷贝**，对于对象中指向堆空间的指针成员，**将重新动态分配空间，然后再赋值**，从而使得指针指向两块不同的空间，但这两块空间中的值相同。
- ✧ 在编写函数时，**尽量按引用方式传递参数**，这样可以避免调用复制构造函数，可以极大地提高程序效率。

## 友元

友元即朋友，**私有成员只有自己和朋友可以访问**。

- ✧ 当类的创建者希望**另一个类**可以访问**当前类的私有成员**时，可以在当前类中将另一个类声明为自己的**友元类**，即在开头加上 friend 关键字。
- ✧ 若只想让另一个类的**某个成员函数**访问**当前类的私有成员**，则可以将此**成员函数**声明为当前类的**友元成员**。
- ✧ 全局函数也可以被声明为一个类的友元，即**友元函数**。

## 静态类成员

- ✧ 普通类成员只能通过实例化的对象来访问，**静态类成员还可以通过类名来直接访问**，访问时使用作用域限定符::。**对于静态数据成员，要在类定义外部对静态数据成员进行定义初始化。**静态数据成员是被类的所有对象共享的。


✧ **静态数据成员可以是其所属类型的对象，而其他数据成员只能是其所属类型的指针或引用**，如：

```
class CBook
{
public:
    static unsigned int price;
    CBook mbook;       //这是错误的
    static CBook mbook;   //下面三种都是正确的
    CBook* pbook;
    CBook& mbook;
};
```

✧ 针对静态成员有如下几点：

① 静态数据成员可以作为成员函数的默认参数，但是普通成员不可以；

② 类的静态成员函数只能访问类的静态成员，不能访问普通数据成员；

③ 静态成员函数末尾不能用 const 关键字修饰；

④ **静态数据成员和静态成员函数在类外定义时，不加 static。**

## 引用和指针的区别

✧ **引用**是一个变量的别名，引用被创建的同时**必须被初始化**（相比之下，指针可以在定义的时候不初始化，之后再赋值）。

✧ 不能有空引用，引用必须对应合法的存储单元。指针则可以是空指针（NULL）。

✧ **引用一旦被初始化，就不能改变引用关系**，指针则可以随时改变所指的对象。

## 类的继承

继承是面向对象的主要特征之一，它使一个类可以从现有类中派生，而不必重新定义一个新类。

✧ 类继承时使用“:”运算符。类继承与访问权限如下表：

| 访问指示符 | 自己所属类可访问 | 派生类可访问 | 类之外对象可访问 |
|---|---|---|---|
| **Public** | 是 | 是 | 是 |
| **Protected** | 是 | 是 | 否 |
| **Private** | 是 | 否 | 否 |

- 用 public 继承时，基类的 public 和 protected 成员在派生类中权限不变；
- 用 protected 继承时，基类的 public 和 protected 成员在派生类中都变成 protected 成员；
- 用 private 继承时，基类的 public 和 protected 成员在派生类中都变为 private 成员。
- 基类的 private 成员在派生类中是隐藏的，不可直接访问。


- ✧ 在派生时可能存在一种情况，即在子类中定义了一个与父类同名的成员函数，此时称**子类重定义父类成员函数**，这样父类中所有的同名成员函数（包括重载函数）均被隐藏。若要访问父类中的同名成员函数，需用**域访问（即::）**的方式。
- ✧ 在派生完一个子类后，可以定义一个父类指针，并用子类对象的地址为其赋值，但通过该指针只能调用父类的成员函数，而不调用子类的同名成员函数。这是因为**在不涉及虚函数的情况下**，编译器对同名成员函数是静态绑定的，即**根据表达式中出现的类型来确定调用哪个类的成员函数**。
- ✧ 子类对象的创建和撤销：当定义一个子类对象时，它将依次调用父类构造函数、子类构造函数。在撤销对象时，先调用子类析构函数，再调用父类析构函数。

## 内存的分配方式

- ✧ 从**静态存储区**分配。这种方式在程序编译的时候已经分配好，在程序的整个运行期间均存在，如**全局变量、静态变量**；
- ✧ 从**栈**上创建。在执行函数时，**函数内部局部变量**的存储单元都可以在栈上创建，函数结束时这些单元被自动释放。栈内存分配操作内置于处理器指令集中，效率较高，但容量有限；
- ✧ 从**堆**上分配。程序运用 new 申请内存，**之后用 delete 手动释放**。动态内存的生存周期由程序员决定，使用灵活，但也容易出问题。

## 常见的内存错误及处理方法

- ✧ **内存未分配成功就去使用它。**故在 new 申请内存后，**立即检查指针是否为 NULL**，防止使用地址为 NULL 的内存。
- ✧ **内存分配成功但未初始化。**故不要忘记为**数组**或**动态内存赋初值**，防止未被初始化的内存作为右值使用。
- ✧ **分配成功也初始化但越界。**故要注意不要让数组或指针下标**越界**；
- ✧ **忘记释放内存，造成泄漏。**故动态内存**申请与释放必须要配对**；
- ✧ **释放了内存却继续使用它。**这有三个方面：
  - ■ 首先，程序中对象调用关系复杂，此时应重新设计数据结构，从根本上解决对象管理方面的问题；
  - ■ 其次，return 语句写错了。不能返回指向栈内存的指针或引用，因为该内存在函数体结束时被自动释放；
  - ■ 最后，使用 delete 后，没有将指针设置为 NULL，导致产生“**野指针**”。

## 指针和数组的对比

- ✧ 非动态数组要么在**静态区**被创建（全局数组），要么在**栈**上被创建。一个数组对应着一块内存，其**地址和容量**在生命周期内保持**不变**，只有内容可以改变。
- ✧ 指针可以随时指向任意的内存块，特征是可变，故常用**指针来操作动态内存**。
- ✧ **对数组名进行直接赋值与比较是错误的**，无法赋值或比较数组的内容。
- ✧ 在 32 位系统中，一个指针变量本身在内存中总是占用 4 个字节。
- ✧ **当数组作为函数参数进行传递时，形参 arr[]为同类型的指针，故 sizeof(arr)=4 字节。**

## delete 操作

- ✧ delete p 只是将指针 p 所指的内存空间释放掉，并没有消灭指针 p。
- ✧ delete 后面不一定非得是指针变量，只要是能表示堆区地址的值即符合语法。
- ✧ 用 delete p 释放内存后，最好要将指针 p 置空。
- ✧ 如果指针消亡了，并不表示所指的内存会被自动释放；内存释放了，并不表示指针会消亡或成为空指针，而会保留其原有值。
- ✧ 野指针不是空指针，而是指向内容不明的内存区域的指针。**注意：任何指针变量刚被创建时不会自动成为空指针，而是随意指的。**因此，在创建指针变量时，要么初始化为 NULL，要么让其指向合法的内存。

## 类的基本函数

- ✧ **构造函数**（以及**拷贝构造函数**）、**析构函数**和**赋值函数**是每个类最基本的函数。每个类只有一个析构函数和一个赋值函数，可以有多个构造函数。
- ✧ 对于任意一个类，若未编写各个函数，编译器会默认产生四个函数，如下所示：

```
class Art
{

    //对于任意一个类Art，如果未编写各函数，编译器会默认产生四个缺省函数
    Art(void);          //无参构造函数
    Art(const Art& art);     //拷贝构造函数
    ~Art(void);              //析构函数
    Art& operator=(const Art& art);     //缺省赋值函数

};
```

注意：拷贝构造函数是在对象被创建时调用，赋值函数只能被已经存在的对象调用。

## 构造函数的初始化列表

- ✧ 初始化列表位于构造函数参数表后，函数体前，表示列表的初始化工作发生在函数体内代码之前。
- ✧ 列表中各数据成员的初始化顺序与它们在类定义中声明的顺序相同，与在初始化列表中的排列顺序无关。初始化列表的使用规则如下：
  - ■ 若类存在继承关系，派生类需在初始化列表中调用基类的构造函数；
  - ■ 类的常量数据成员只能在初始化列表中初始化，不能在函数体内赋值；
  - ■ **类的非内置类型的对象成员应采用初始化列表方式，以调用其构造函数；**
  - ■ 类的其它数据成员可以采用初始化列表和函数体赋值两种方式，但初始化列表效率较高。



## 如何在派生类中实现类的基本函数

基类的构造函数、析构函数不能被派生类继承，派生类实现基本函数的原则是：

① 派生类的构造函数应在初始化列表中调用基类的构造函数；

② 当基类构造函数要传递参数才能初始化时，派生类必须显式定义构造函数，为基类构造函数传递参数。

## 用 const 来提高函数的健壮性

const 不仅定义常量，更大的作用是可以修饰函数的参数、返回类型和函数的定义体。被修饰的东西受强制保护，可以预防意外的变动，能提高程序的健壮性。

- ✧ **const 修饰输入参数：**
  - ■ 若采用值传递，没必要使用 const；
  - ■ 若采用指针传递，可以起到保护指针的作用；
  - ■ 对于占用较大内存空间的自定义类型，可以用 const 修饰引用的方式，高效传递。
- ✧ **const 修饰函数的返回类型：**
  - ■ 若返回类型是 const 修饰的指针，则指针内容不能改变，返回值只能赋给加 const 修饰的同类型指针；
  - ■ 若返回类型是值传递，不使用 const。
- ✧ **const 修饰成员函数（常量成员函数）：**任何不修改数据成员的函数都应该声明为 const 类型。

## 提升程序效率的规则

程序的时间效率是指运行速度，空间效率是指程序占用的内存或外存状况。提高效率的规则如下：

① 应在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高效率；

② 提高程序的全局效率为主，局部效率为辅；

③ 先优化数据结构和算法，再处理程序代码；

③ 不要追求紧凑的代码，因为这并不能产生高效的机器码。