

# Table of contents

Flask_Tutorial .....	3
Topic title .....	4
C++ Templates .....	5
1. 关于本书 .....	8
第一部分 基础 .....	9
2. 函数模板 .....	10
3. 类模板 .....	11
4. 非类型模板参数 .....	12
5. 技巧性基础知识 .....	13
6. 模板实战 .....	14
7. 模板术语 .....	15
第二部分 深入模板 .....	16
8. 深入模板基础 .....	17
9. 模板中的名称 .....	18
10. 实例化 .....	19
11. 模板实参演绎 .....	20
12. 特化与重载 .....	21
13. 未来的方向 .....	22
第三部分 模板与设计 .....	23
14. 模板的多态威力 .....	24
15. trait与policy类 .....	25
16. 模板与继承 .....	26
17. metaprogram .....	27
18. 表达式模板 .....	28
第四部分 高级应用程序 .....	29
19. 类型区分 .....	30
20. 智能指针 .....	31
21. tuple .....	32
22. 函数对象和回调 .....	33
附录A 一处定义原则 .....	34
附录B 重载解析 .....	35
C++ Conccurrency .....	36
1. 你好， C++的并发世界 .....	37

2. 线程管理 . . . . .	38
3. 线程间数据共享 . . . . .	39
4. 同步并发操作 . . . . .	40
5. C++内存模型和原子类型操作 . . . . .	41
6. 基于锁的并发数据结构设计 . . . . .	42
7. 无锁并发数据结构设计 . . . . .	43
8. 并发代码设计 . . . . .	44
9. 高级线程管理 . . . . .	45
10. 多线程程序的测试和调试 . . . . .	46
C++ Lambda Story . . . . .	47
1. Lambdas in C++98/03 . . . . .	48
2. Lambdas in C++11 . . . . .	55
汇编语言 . . . . .	102
第一章 基础知识 . . . . .	103
第二章 寄存器 . . . . .	118
第三章 寄存器(内存访问) . . . . .	119
现代操作系统 . . . . .	120
操作系统 . . . . .	121
Functional Programming_ . . . . .	122
编译原理 . . . . .	125
内存池(Memory Pool) . . . . .	126
Effective C++ . . . . .	127
第一章 让自己习惯 C++ . . . . .	128
第二章 构造/析构/赋值运算 . . . . .	143
第三章 资源管理 . . . . .	166
第四章 设计与声明 . . . . .	182
STL源码分析 . . . . .	183
The Terms in Book 《STL源码分析》 . . . . .	246
Algorithm_Tutorial . . . . .	251
Topic title . . . . .	281
《TCP/IP详解卷一：协议》 . . . . .	288

# **Flask\_Tutorial**

Start typing here...

# **Topic title**

**ppQwQqq's GitHub**

# C++ Templates

## 1. 函数模板

### 1.1 初探函数模板

#### 1.1.1 定义模板

此处定义了一个返回两数中较大者的函数

```
template<typename T>
inline T const& max(T const& a, T const& b){
    return a > b ? a : b;
}
```

上述代码中的`typename`为模板参数，`T`则为模板参数名。`T`可以换成其他任意的合法字符，如：`MyType`。

模板参数`typename`可以替换为`class`，在当前语义环境下`class`与`typename`的效果是相同的。  
注意：引入类型参数时，`typename`不能替换为`struct`。

```
template<class T>
inline T const& max(T const& a, T const& b){
    return a > b ? a : b;
}
```

#### 1.1.2 使用模板

此处为模板函数的使用：

```
#include <iostream>

template<typename T>
inline T const& max(T const& a, T const& b){
    using Type = decltype(a);
    std::cout << "Type:" << typeid(std::type_identity_t<T>).name() <<
    '\n';
    return a > b ? a : b;
}
```

```

}

int main(){
    printf("Result: %lf\n", ::max(3.0, 4.4));
    printf("Result: %d\n", ::max(3, 5));
    printf("Result: %c\n", ::max('a', 'b'));
    printf("Result: %s\n", ::max("abcf", "abcd"));
    return 0;
}

/* output:
 * Type:d
 * Result: 4.400000
 * Type:i
 * Result: 5
 * Type:c
 * Result: b
 * Type:A5_c
 * Result: abcf
 */

```

注：使用域限定符`::`是为了保证调用的是全局命名空间内的`max()`(即自己定义的`max`函数)，因为在标准库中也存在一个`std::max()`，为避免二义性所以添加域限定符。

由输出结果可知，类型T被转换为了传入参数的类型。因为对于模板函数的每次调用，模板参数T都被实例化(instantiation)为了对应的类型。`operator<`(以及其他运算符)适用于内置类型。如果需要对自定义类进行比较运算，则需要手动重载该运算符。

```

MyClass t1, t2;
std::cout << "Result: " << ::max(t1, t2);
/*
 * Compiling Error
 */

```

由此可知，模板实例化会经历两次编译：

1. 实例化之前，检查模板本身是否存在语法错误。

2. 实例化期间，检查模板函数调用是否有效(对指定的类型是否存在重载函数)。

## 1.2 实参的推导(deduction)

模板参数T的类型由调用函数的实参决定。当实参类型不同时，例如：

```
template<typename T>
inline T const& max(T const& a, T const& b);
max('a', 23);    // Error
max(23, 43);    // Ok
```

这里的T不允许进行自动的类型转换，所以当接受的实参类型不同时就会导致编译器报错。处理上述问题的三种方法：

1. 对实参进行强制类型转换，使其类型相互匹配：

```
max(static_cast<int>('a'), 32);
max<double>(3, 3.4)
```

2. 显示指定或限定T的类型

3. 指定两个参数可以具有不同类型的参数

# 1. 关于本书

Start typing here...

# 第一部分 基础

Start typing here...

## 2. 函数模板

Start typing here...

### 3. 类模板

Start typing here...

# 4. 非类型模板参数

Start typing here...

# 5. 技巧性基础知识

Start typing here...

# 6. 模板实战

Start typing here...

# 7. 模板术语

Start typing here...

# 第二部分 深入模板

Start typing here...

# 8. 深入模板基础

Start typing here...

# 9. 模板中的名称

Start typing here...

# 10. 实例化

Start typing here...

# 11. 模板实参演绎

Start typing here...

# 12. 特化与重载

Start typing here...

# 13. 未来的方向

Start typing here...

# 第三部分 模板与设计

Start typing here...

# 14. 模板的多态威力

Start typing here...

# 15. trait与policy类

Start typing here...

# 16. 模板与继承

Start typing here...

# 17. metaprogram

Start typing here...

# 18. 表达式模板

Start typing here...

# 第四部分 高级应用程序

Start typing here...

# 19. 类型区分

Start typing here...

# 20. 智能指针

Start typing here...

# 21. tuple

Start typing here...

# 22. 函数对象和回调

Start typing here...

# 附录A 一处定义原则

Start typing here...

# 附录B 重载解析

Start typing here...

# C++ Concurrency

Start typing here...

# 1. 你好，C++的并发世界

Start typing here...

## 2. 线程管理

Start typing here...

### 3. 线程间数据共享

Start typing here...

# 4. 同步并发操作

Start typing here...

# 5. C++内存模型和原子类型操作

Start typing here...

# 6. 基于锁的并发数据结构设计

Start typing here...

# 7. 无锁并发数据结构设计

Start typing here...

# 8. 并发代码设计

Start typing here...

# 9. 高级线程管理

Start typing here...

# 10. 多线程程序的测试和调试

Start typing here...

# C++ Lambda Story

Start typing here...

# 1. Lambdas in C++98/03

## 本章的主要内容

1. 如何将仿函数(functor)传给标准库中的算法
2. 仿函数和函数指针(function pointer)的限制
3. 为什么函数辅助工具不够好

### 在 C++98/03 中的可调用对象(callable object)

在C++标准库的设计理念中，像"std::sort"、"std::for\_each"、"std::transform"等算法可以接受任何可调用对象(callable objects)，并将其应用于输入容器的元素，然而，在C++98/03标准中，这些算法只能接受函数指针和仿函数(函数对象)。

此处使用一个函数输出vector中的元素。一般的实现：

```
#include <algorithm>
#include <iostream>
#include <vector>

void PrintFunc(int x){
    std::cout << x << '\n';
}

int main(){
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    for_each(v.begin(), v.end(), PrintFunc);
}
/* output:
 * 1
 * 2
 */
```

上面的代码使用std::for\_each遍历整个vector容器(使用for\_each的原因是C++98/03还不支持范围遍历)， 并且它将PrintFunc作为一个可调用对象进行传递。

接下来，将这个函数转换为一个仿函数(functor)：

```
#include <iostream>
#include <vector>
#include <algorithm>

struct PrintFunctor{
    void operator()(int x) const {
        std::cout << x << '\n';
    }
};

/* or this version
class PrintFunctor{
public:
    void operator()(int x) const {
        std::cout << x << '\n';
    }
};
*/

int main(){
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    for_each(v.begin(), v.end(), PrintFunctor());
}
/* output:
 * 1
 * 2
 */
```

上述例子使用了一个具有operator() 的仿函数。

然而函数指针(function pointer)通常是无状态的，即它们只是指向一个函数，并不能存储任何额外的信息或状态，相比之下仿函数可以拥有成员变量，从而允许它们在多个调用之间存储状

态。

定义一个简单的仿函数，让其记录自己被调用的次数，其关键在于定义仿函数中的状态量：

```
#include <algorithm>
#include <iostream>
#include <vector>

class printFunctor{
public:
    printFunctor(const std::string str): strText(str), numCalls(0) { }
    void operator()(int x){
        std::cout << strText << x << '\n';
    }
    int getNumCalls() const{
        return numCalls;
    }
private:
    std::string strText;
    mutable int numCalls;
};

int main(){
    std::vector v = {1, 2, 3, 4, 5, 6};
    std::string preText = "Elem: ";
    printFunctor visitor =
    std::for_each(v.begin(), v.end(), printFunctor(preText));
    std::cout << "numCalls: " << visitor.getNumCalls() << '\n';
}
/* output:
 * 1
 * 2
 * 3
 * 4
 * 5
 * 6
```

```
* numCalls: 6  
*/
```

## 仿函数(functor)的问题

虽然可以使用一个单独的类去设计仿函数，但是在与算法调用不同的地方编写函数和仿函数，这导致函数代码在源文件中的位置与算法调用的位置相距很远，由此增加了代码的阅读和维护难度。

在C++98/03中，有一个限制是不能使用局部类型(在函数内部定义的类型)作为模板参数，例如：

```
int main(){  
    /* define a type inside a function */  
    struct PrintFunctor{  
        void operator()(int x) const{  
            std::cout << x << '\n';  
        }  
    };  
  
    std::vector<int> v(10, 1);  
    std::for_each(v.begin(), v.end(), PrintFunctor());  
}
```

使用 GCC 的 `-std=C++98` 标准进行编译，会出现如下错误：

```
error: template argument for  
'template<class _IIter, class _Funct> _Funct  
std::for_each(_IIter, _IIter, _Funct)'  
uses local type 'main()::PrintFunctor'
```

## 使用 辅助函数(functional helper) 解决

在标准库中 `<functional>` 头文件，有很多类型和函数可以与标准算法一起使用：  
`std::plus<T>()`: 接受两个参数并返回它们的和。  
`std::minus<T>()`: 接受两个参数并返回它们的差。

```
#include <functional>  
#include <iostream>
```

```

#include <vector>
#include <algorithm>

int main(){
    std::vector<int> vi1 = {1, 2, 3, 4, 5, 6};
    std::vector<int> vi2 = {2, 3, 4, 5, 6, 7};
    std::vector<int> vRes(vi1.size());

    std::transform(vi1.begin(), vi1.end(), vi2.begin(),
                  vRes.begin(), std::plus<int>());
    std::cout << "Res for plus: ";
    for(int n: vRes) std::cout << n << ' ';

    std::transform(vi1.begin(), vi1.end(), vi2.begin(),
                  vRes.begin(), std::minus<int>());
    std::cout << "\nRes for minus: ";
    for(int n: vRes) std::cout << n << ' ';
}

/* output:
 * Res for plus: 3 5 7 9 11 13
 * Res for minus: -1 -1 -1 -1 -1 -1
 */

```

`std::less<T>()`: 接受两个参数并返回第一个参数是否小于第二个参数。 `std::greater_equal<T>()`: 接受两个参数并返回第一个参数是否大于或等于第二个参数。 `std::bind1st`: 创建一个可调用对象，将第一个参数固定为给定值。 `std::bind2nd`: 创建一个可调用对象，将第二个参数固定为给定值。

使用辅助函数的好处：

```

#include <algorithm>
#include <functional>
#include <vector>

int main(){
    std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};

```

```
const size_t smaller5 = std::count_if(
v.begin(), v.end(),
std::bind2nd(std::less<int>(), 5));

return smaller5;
}
```

上面的代码中，使用std::bind2nd将 5 绑定至std::less函数( $x < y$ )的第二个参数( $x < 5$ )，即返回的值为 return  $x < 5$ 。

```
const size_t greater5 = std::count_if(
v.begin(), v.end(),
std::bind1st(std::less<int>(), 5));
```

同理，此处使用std::bind1st将 5 绑定至std::less函数( $x > y$ )的第一个参数( $5 < x$ )，即返回值的为 return  $5 < x$ 。

### ⚠ 补充：std::bind的用法

```
#include <iostream>
#include <functional>

int add(int x, int y){
    std::cout << "1st param: " << x << '\n';
    return x + y;
}

int main(){
    /* 创建一个绑定第一个参数为10的函数对象 */
    auto res = std::bind(add, 10, std::placeholders::_1);

    std::cout << "12 + 10 = " << res(12) << '\n';
}
/* output:
 * 12 + 10 = 1st param: 10
```

```
* 22  
*/
```

然而，在很多情况下，一个功能实现会有很多函数组成，那么此处语法就会显得很复杂：

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8};  
const size_t val = std::count_if(v.begin(), v.end(),  
                                std::bind(std::logical_and<bool>(),  
                                std::bind(std::greater<int>(), _1, 2),  
                                std::bind(std::less_equal<int>(), _1, 6)));
```

上面这么复杂的语法其实实现的结果就是 `return x > 2 && x <= 6`。这些问题在后面的C++11中得到了改善。

## 2. Lambdas in C++11

关于C++11的最终草案N3337 (<https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda>)中Lambda表达式的内容如下： Lambda表达式(Lambda Expressions):

### 1. Lambda 表达式的目的和语法

- Lambda 表达式提供了一种简洁的方式来创建简单的函数对象。
- 示例：

```
• #include <algorithm>
#include <cmath>
void abssort(float* x, unsigned N){
    std::sort(x, x + N, [](float a, float b){
        return std::abs(a) < std::abs(b);
    });
}
```

- Lambda 表达式的语法：

```
• lambda-expression:
    lambda-introducer lambda-declaratoropt compound-statement
lambda-introducer:
    [ lambda-captureopt ]
lambda-capture:
    capture-default
    capture-list
    capture-default, capture-list
lambda-default:
    &
    =
lambda-list:
    capture-opt
    capture-list, capture ...opt
capture:
```

```
identifier  
& identifier  
this  
lambda-declarator:  
  ( parameter-declaration-clause ) mutableopt  
    exception-specificationopt attribute-specifier-seqopt  
  trailing-return-typeopt
```

## 2. Lambda表达式计算的结果

- 计算Lambda 表达式会生成一个prvalue临时对象(closure object)。
- Lambda 表达式不应出现在未计算的操作数中。
- 闭包对象的行为类似于函数对象。

## 3. Lambda 表达式类型

- Lambda 表达式的类型是唯一的、未命名的的非联合类型，成为闭包类型(closure type)。
- 闭包类型在包含相应 Lambda 表达式的最小块作用域、类作用域或命名空间作用域中声明。

## 4. Lambda 表达式的默认行为

- 如果 Lambda 表达式不包含 Lambda 声明符，则默认为'()'。
- 如果 Lambda 表达式不包含尾返回类型，则尾返回类型默认为以下类型：
  - 如果复合语句的形式为"， 则返回表达式的类型。
  - 否则，返回'void'。

## 5. 闭包类型的函数调用运算符

- 闭包类型具有一个公共的内联函数调用运算符，其参数和返回类型由 Lambda 表达式的参数声明子句和尾返回类型描述。
- 如果参数声明子句后没有'mutable'，则函数调用运算符被声明为'const'。
- 该运算符既不是虚拟的也不是声明为'volatile'。

- Lambda 声明符中的任何异常规范适用于相应的函数调用运算符。

## 6. 无捕获的 Lambda 表达式的函数指针转换

- 无捕获的 Lambda 表达式的闭包类型具有一个公共的非虚拟的'const'转换函数，该函数转换为具有相同参数和返回类型的函数指针。
- 该转换函数返回的值是一个函数的地址，调用该函数具有与调用闭包类型的函数调用运算符相同的效果。

## 7. Lambda 表达式的复合语句

- Lambda 表达式的复合语句生成函数调用运算符的函数体，但在名称查找和类型确定方面，复合语句被视为 Lambda 表达式的一部分。

## 8. 捕获规则

- 如果 Lambda 捕获包含默认捕获'&'，则捕获列表中的标识符前不应有'&'。
- 如果 Lambda 捕获包含默认捕获'='，则捕获列表中的标识符前不应有'this'，且每个标识符前应有'&'。
- 捕获列表中的标识符或'this'不能重复。

## 9. 本地 Lambda 表达式

- 如果 Lambda 表达式的最小封闭作用域是块作用域，则称为本地 Lambda 表达式；否则，不应在 Lambda 引入器中有捕获列表。

## 10. 捕获列表中的标识符查找

- 捕获列表中的标识符使用无资格名称查找规则进行查找，每个查找应找到在本地 Lambda 表达式的到达作用域中声明的具有自动存储持续时间的变量。

## 11. 隐式捕获

- 如果 Lambda 表达式具有捕获默认值且复合语句 odr-使用'this'或具有自动存储持续时间的变量，则这些实体被隐式捕获。

## 12. 实体的捕获

- 如果实体被显式或隐式捕获，则称其捕获。捕获的实体在包含 Lambda 表达式的作用域中被 odr-使用。

### 13. 默认参数中的 Lambda 表达式

- 出现在默认参数中的 Lambda 表达式不应显式或隐式捕获任何实体。

### 14. 按值捕获

- 实体按值捕获，如果它被隐式捕获且捕获默认值为'='，或被显式捕获且捕获不包括'&'。

### 15. 按引用捕获

- 实体按引用捕获，如果它被隐式或显式捕获，但不是按值捕获。

### 16. 嵌套 Lambda 表达式的捕获转换

- 如果一个 Lambda 表达式 m2 捕获一个实体，而该实体被其直接封闭的 Lambda 表达式 m1 捕获，则 m2 的捕获根据 m1 的捕获方式进行转换。

### 17. 按值捕获的 id 表达式

- 每个按值捕获的实体的 id 表达式被转换为对闭包类型中相应未命名数据成员的访问。

### 18. decltype 操作符中的捕获

- decltype((x)) 中的 x 被视为对闭包类型中相应数据成员的访问。

### 19. 闭包类型的构造函数和赋值运算符

- 与 Lambda 表达式关联的闭包类型具有被删除的默认构造函数和被删除的复制赋值运算符。它具有隐式声明的复制构造函数，并且可能具有隐式声明的移动构造函数。

### 20. 闭包类型的析构函数

- 与 Lambda 表达式关联的闭包类型具有隐式声明的析构函数。

### 21. 捕获的实体的初始化

- 在评估 Lambda 表达式时，按值捕获的实体用于直接初始化生成的闭包对象的每个相应非静态数据成员。

### 22. 按引用捕获的生命周期

- 如果按引用捕获的实体在其生命周期结束后调用函数调用运算符，可能会导致未定义行为。

## 23. 捕获的包展开

- 捕获后跟随省略号表示包展开。

**⚠ 补充：移动构造函数和移动赋值运算符**

- 移动构造函数的实现示例：

```
class MyClass{
    int* data;

    MyClass(int size): data(new int[size]){ }

    MyClass(MyClass&& other/* 此处接受一个右值 */) noexcept:
        data(other.data){
            // 将其他对象的数据指针置为空，表示资源所有权已经转移
            other.data = nullptr;
    }

    ~MyClass(){
        delete[] data;
    }

    MyClass(const MyClass&) = delete;
    MyClass& operator=(const MyClass&) = delete;
};

int main(){
    MyClass obj1(10);    // 调用普通构造函数
    MyClass obj2 = std::move(obj1); // 调用移动构造函数

    // 此时 obj1.data 为 nullptr，资源所有权已经转移给 obj2
}
```

- 移动构造函数的用途：

- 提高性能：在处理大型数据结构或资源密集型对象时，移动构造函数可以显著减少不必要的深度拷贝操作，从而提高程序性能。
- 资源管理：在实现资源管理类（如智能指针、容器类）时，移动构造函数可以更高效地管理资源转移。
- 什么时候会调用移动构造函数：
  - 当一个临时对象(右值)被用来初始化另一个对象时。
  - 当返回一个局部对象时，如果启用了返回值优化(RVO)，则可能会调用移动构造函数。
  - 当使用标准库函数如'std::move'将一个对象转换为右值引用时。
- 注意事项：
  - 移动构造函数通常与移动赋值运算符一起实现，以确保对象在移动语义下的正确行为。
  - 实现移动构造函数时，通常需要禁用复制构造函数和复制赋值运算符，以避免不必要的拷贝。
- 移动赋值运算符
  - ```
#include <iostream>
#include <utility>

MyClass{
public:
    int* data;

    MyClass(int size): data(new int[size]) { }
    MyClass(MyClass&& other) noexcept: data(other.data) {
        other.data = nullptr;
    }
    MyClass& operator=(MyClass&& other) noexcept{
        if(this != other){
            // 释放当前对象持有的资源
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }
}
```

```

        // 获取源对象的资源
        data = other.data;
        // 将源对象的资源指针置为空
        other.data = nullptr;
    }
    return *this;
}

~MyClass(){
    delete[] data;
}

MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
};

int main(){
    MyClass obj1(10);
    MyClass obj2(20);

    obj2 = std::move(obj1);

    /* 此时obj1.data为nullptr,
       obj2.data为obj1.data的地址
    */
}

```

## 本章的主要内容

- Lambda 的基本语法
- 如何捕获变量
- 如何捕获成员变量
- Lambda 的返回类型

- 什么是闭包对象
- 如何将 Lambda 转换为函数指针并在 C 风格的 API 中使用
- 什么是 IIFE(立即调用函数表达式)
- 如何从 Lambda 表达式继承以及为什么这会有用

## Lambda 表达式的语法

Lambda 表达式的语法结构如下所示：

```
[ ]() specifiers exception attr -> ret { /* code */ }
^ ^ ^
| | |
| | |
| | optional: trailing return type
| |
| optional: mutable, exception specification or noexcept, attributes
|
| parameter list (optional when no specifiers added)
|
Lambda introducer with an optional capture list
```

关于 Lambda 表达式在 C++ 中的明确定义：

### 1. Lambda 表达式计算的结果

- 计算 Lambda 表达式会生成一个 prvalue (pure rvalue, 纯右值) 临时对象 (closure object)。
- Lambda 表达式不应出现在未计算的操作数中。
- 闭包对象的行为类似于函数对象。

### 2. Lambda 表达式类型

- Lambda 表达式的类型是唯一的、未命名的的非联合类型，成为闭包类型 (closure type)。
- 闭包类型在包含相应 Lambda 表达式的最小块作用域、类作用域或命名空间作用域中声明。

## Lambda 表达式的一些示例

### 1. 最简单的 Lambda 表达式:

```
[ ]{ };
```

此 Lambda 表达式只需要'[]'和空的'{}'作为函数体。参数列表'()'是可选的。

### 2. 带有两个参数的 Lambda 表达式:

```
[ ](float f, int a) { return a * f; }
[ ](int a, int b) { return a < b; }
```

这是 Lambda 表达式最常见的类型之一，参数通过'()'部分进行传递，这和常规的函数是一样的，不需要指定返回类型，编译器会自动推断。

### 3. 带尾返回类型的 Lambda 表达式:

```
[ ](MyClass t) -> int { auto a = t.compute(); print(a); return a; };
```

此 Lambda 表达式显式定义了返回值类型，尾返回类型从 C++11 开始也适用于常规函数声明。

#### ▲ 补充：尾返回类型

- 尾返回类型语法：尾返回类型使用关键字'auto'和'->'符号。
- `auto functionName(parameters) -> returnType`

示例：

#### 1. 简单函数的尾返回类型

```
auto add(int a, int b) -> int{
    return a + b;
}
```

## 2. 模板函数的尾返回类型

```
template<typename T, typename U>
auto add(T a, U b) -> decltype(a + b){
    return a + b;
}
```

## 3. Lambda 表达式的尾返回类型

```
auto lambda = [](int a, int b) -> int {
    return a + b;
}
```

## 4. 额外的修饰符：

```
[x](int a, int b) mutable { ++x; return a < b; };
[](float param) noexcept { return param * param; };
[x](int a, int b) mutable noexcept { ++x; return a < b; };
```

在此示例中，Lambda 函数体前添加了修饰符：'mutable'(以便可以更改捕获的变量) 和'noexcept'，第三个 Lambda 表达式中的 'mutable noexcept' 是固定的顺序，若写成 'noexcept mutable' 则不能通过编译，当使用了'mutable'和'noexcept'，则需要在表达式中添加'()'。

## 5. 关于可选的'()'：

```
[x] { std::cout << x; } // 不需要'()'
[x] mutable { ++x; }; // 编译错误，因为mutable存在，故需要'()'
[x]() mutable { ++x; }; // 编译正常
[] noexcept { }; // 编译错误，因为noexcept存在，故需要'()'
[]() noexcept { }; // 编译正常
```

对于后面的C++17和C++20中的'constexpr'和'consteval'也适用。

## 属性

Lambda 表达式的语法还允许使用以'[[attr\_name]]'形式引入的属性。然而，如果将属性应用于Lambda，那么它应用于调用运算符的类型，而不是运算符本身。尝试以下表达式：

```
auto myLambda = [](int a) [[nodiscard]] { return a * a; };
```

Clang会生成如下错误信息：

```
error: 'nodiscard' attribute cannot be applied to types
```

## 编译展开

Lambda 表达式传给std::for\_each的示例：

```
#include <iostream>
#include <algorithm>
#include <vector>

int main(){
    // 定义一个functor与下面一般的 Lambda 表达式做对比
    struct{ /* anonymous */
        void operator()(int x) const{
            std::cout << x << 'x';
        }
    } someInstances;

    const std::vector<int> v{1, 2, 3, 4, 5};
    std::for_each(v.cbegin(), v.cend(), someInstance);
    std::for_each(v.cbegin(), v.cend(), [](int x){
        std::cout << x << '\n';
    });
}
```

在这个示例中，编译器将以下 Lambda 表达式：

```
[](int x) { std::cout << x << '\n'; }
```

转换为一个匿名仿函数，简化形式如下：

```
struct{
    void operator()(int x){
        std::cout << x << '\n';
    }
}someInstances;
```

编译器具体的展开结果如下：

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int, std::allocator<int> > v = std::vector<int,
std::allocator<int> >{std::initializer_list<int>{1, 2, 3, 4, 5},
std::allocator<int>()};

class __lambda_7_39
{
public:
    inline /*constexpr */ void operator()(int x) const
    {
        std::operator<<(std::cout.operator<<(x), '\n');
    }

    using retType_7_39 = void (*)(int);
    inline constexpr operator retType_7_39 () const noexcept
    {
        return __invoke;
    }

private:
    static inline /*constexpr */ void __invoke(int x)
    {
        __lambda_7_39{}.operator()(x);
    }
}
```

```

public:
    // inline /*constexpr */ __lambda_7_39(__lambda_7_39 &&) noexcept =
default;
    // /*constexpr */ __lambda_7_39() = default;

};

std::for_each(v.cbegin(), v.cend(), __lambda_7_39{});
return 0;
}

```

## Lambda 表达式的类型

### 编译器生成闭包类型

- 编译器为每个 Lambda 表达式生成一个唯一的闭包类型(closure type)，无法预测这个类型名。
- 因此，需要使用'auto'(或'decltype')来推断类型

```
auto myLambda = [](int a) -> double { return 2.0 * a; };
```

### 不同的闭包类型

- 即使两个 Lambda 表达式完全相同，它们的类型也是不同的：

```
auto firstLam = [](int x) { return x * 2; };
auto secondLam = [](int x) { return x * 2; };
```

- 编译器必须为每个 Lambda 声明两个独特的未命名类型：

```
#include <type_traits>
int main(){
    const auto firstLam = [](int x) { return x * 2; };
    const auto secondLam = [](int x) { return x * 2; };
    static_assert(!std::is_same(decltype(firstLam),
```

```
        decltype(secondType)>::value,  
        "must be different!");  
}
```

在编译器眼中，可知这样两个 Lambda 表达式生成的是两个不同的闭包类型：

```
#include <type_traits>  
  
int main()  
{  
  
    class __lambda_4_25  
    {  
        public:  
            inline /*constexpr */ int operator()(int x) const  
            {  
                return x * 2;  
            }  
  
            using retType_4_25 = int (*)(int);  
            inline constexpr operator retType_4_25 () const noexcept  
            {  
                return __invoke;  
            };  
  
        private:  
            static inline /*constexpr */ int __invoke(int x)  
            {  
                return __lambda_4_25{}.operator()(x);  
            }  
  
        public:  
            // /*constexpr */ __lambda_4_25() = default;  
    };
```

```

const __lambda_4_25 firstLam = __lambda_4_25{};

class __lambda_5_27
{
public:
    inline /*constexpr */ int operator()(int x) const
    {
        return x * 2;
    }

    using retType_5_27 = int (*)(int);
    inline constexpr operator retType_5_27 () const noexcept
    {
        return __invoke;
    };

private:
    static inline /*constexpr */ int __invoke(int x)
    {
        return __lambda_5_27{}.operator()(x);
    }
};

public:
// /*constexpr */ __lambda_5_27() = default;

};

const __lambda_5_27 secondLam = __lambda_5_27{};
/* PASSED: static_assert(!std::integral_constant<bool,
false>::value, "must be different"); */
return 0;
}

```

**A** C++17中的改进：C++17中可以使用没有消息的'static\_assert'和辅助变量模板'is\_same\_v'：

```
static_assert(std::is_same_v<double, decltype(func(10))>);
```

- 使用'std::function'

- 尽管不能确切知道 Lambda 的类型，但可以指定 Lambda 的签名，并将其存储在'std::function'中：

```
/* std::function<返回值类型(接受参数类型)> */
std::function<double(int)> myFunc = [] (int a) -> double { return
2.0 * a; };
```

- 需要注意的是，'std::function'是一个重量级对象，因为它需要处理所有可调用对象，其内部机制较复杂，涉及类型转换或内存动态分配，现检查其大小：

```
#include <functional>
#include <iostream>

int main(){
    const auto myLambda = [] (int a) noexcept -> double { return
2.0 * a; };

    const std::function<double(int)> myFunc = [] (int a) noexcept
-> double { return 2.0 * a; };

    std::cout << "sizeof(myLambda) is " << sizeof(myLambda) <<
'\n';
    std::cout << "sizeof(myFunc) is " << sizeof(myFunc) << '\n';

    return myLambda(10) == myFunc(10);
}
/* output:
 * sizeof(myLambda) is 1
 * sizeof(myFunc) is 64
 */
```

- 由于'myLambda'只是一个无状态的 Lambda，它也是一个空类，没有任何数据成员字段，所以它的大小只有一个字节。
- 而'std::function'版本要大得多，为64字节(不同的编译器及编译器版本和当前操作系统版本会导致此值不同)，如果可能，依赖'auto'推断以获得最小的闭包对象。

## 构造函数与复制

### 1. Lambda 表达式的闭包类型

- 根据C++规范：
  - Lambda 表达式关联的闭包类型有一个被删除的默认构造函数(default constructor)。
  - 闭包类型还有一个被删除的复制赋值运算符(copy assignment operator)。

### 2. 不能默认构造和赋值，即进行一般的copying操作 由于默认构造函数和复制赋值运算符被禁用，如下代码进行编译会报错：

```
auto foo = [&x, &y]() { ++x; ++y; };
decltype(foo) fooCopy;
```

编译结果：

```
error: no matching constructor for initialization of 'decltype(foo)'
```

### 3. 可以复制 Lambda 虽然不能默认构造和赋值 Lambda，但可以复制 Lambda：

```
#include <type_traits>

int main(){
    const auto firstLam = [](int x) noexcept { return x * 2; };
    const auto secondLam = firstLam;
    static_assert(std::is_same<decltype(firstLam),
        decltype(secondLam)>::value, "must be the same!");
}
/* verify the same type of firstLam and secondLam */
```

4. 捕获变量的复制 当复制 Lambda 时，其状态也会被复制。这在涉及捕获变量时尤为重要。闭包类型将捕获的变量存储为成员字段，复制 Lambda 会复制这些数据成员字段。

5. C++20 的改进 在 C++20 中，无状态的 Lambda 将具有默认构造函数和赋值运算符，使其更加灵活和易用。

## Lambda 表达式的调用运算符

### 1. Lambda 表达式的内部实现

- 在 Lambda 表达式的函数体中编写的代码，会被编译成对应闭包类型的'operator()'函数中的代码。

### 2. 默认行为

- 在 C++11 中，'operator()'默认是一个'const'的内联成员函数。

- Lambda 表达式：

- ```
auto lam = [](double param) { /* do something */ };
```

- 编译展开后：

- ```
struct __anonymousLambda{
    inline void operator()(double param) const { /* do something */ }
};
```

## 重载

### 1. Lambda 表达式不支持重载

- Lambda 表达式无法定义"重载"版本，无法接受不同的参数类型：

- ```
auto lam = [](double param) { /* do something */ };
auto lam = [](int param) { /* do something */ };
```

上述代码无法通过编译，因为编译器无法将这两个 Lambda 转换为单个functor，而且不能重定义相同的变量。

## 2. 使用仿函数实现重载

- 使用functor实现重载

```
• struct MyFunctor{  
    inline void operator()(double param) const { /* do something */ };  
    inline void operator()(int param) const { /* do something */ };  
};
```

- 'MyFunctor'现在可以处理'double'和'int'类型的参数。

## Lambda 表达式的修饰符和捕获

### 修饰符(modifier)

1. 默认声明：在默认情况下，Lambda 表达式生成的调用运算符('operator()')是'const'内联成员函数。
2. 其他修饰符：在C++11中，可以使用'mutable'和异常规范('noexcept')来修饰调用运算符：Lambda 表达式：

```
auto myLambda = [](int a) mutable noexcept { /* do something */ };
```

编译展开后：

```
struct __anonymousLambda{  
    inline void operator()(int a) noexcept { /* do something */ };  
};
```

### 捕获(capture)

- 捕获子句：'[]'不仅引入 Lambda 表达式，还包含捕获的变量列表，称为"捕获子句"。
- 捕获变量：捕获变量会在闭包类型中作为成员变量(非静态static数据成员)存储，可以在 Lambda 体内访问。

### 捕获方式

- '[&]': 按引用捕获所有在作用域中的自动存储变量。
- '[=]': 按值捕获所有在作用域中的自动存储变量。
- '[x, &y]': 显式按值捕获'x'和按引用捕获'y'。
- '[args...]': 按值捕获模板参数包。
- '[&args...]': 按引用捕获模板参数包。

捕获示例：

```
int x = 2, y = 3;
const auto l1 = []() { return 1; }; // 无捕获
const auto l2 = [=]() { return x; }; // 全部按值捕获
const auto l3 = [&]() { return y; }; // 全部按引用捕获
const auto l4 = [x]() { return x; }; // 仅按值捕获 x
const auto l5 = [&y]() { return y; }; // 仅按引用捕获 y
const auto l6 = [x, &y]() { return x * y; }; // x 按值捕获, y 按引用捕获
const auto l7 = [=, &x]() { return x + y; }; // 全部按值捕获, x 按引用捕获
const auto l8 = [&, y]() { return x - y; }; // 全部按引用捕获, y 按值捕获
```

## 捕获变量行为

- 按值捕获：变量在 Lambda 定义时被复制。

- Lambda 表达式：

```
std::string str{"Hello Lambda"};
auto foo = [str]() { std::cout << str << '\n'; }
foo();
```

- 编译展开后：

```
struct _unnamedLambda{
    _unnamedLambda(std::string s): str(s) { }
    void operator()() const {
        std::cout << str << '\n';
    }
}
```

```
    std::string str;
};
```

- 按引用捕获：变量在 Lambda 调用时使用当前值。

- Lambda 表达式

```
int x, y = 1;
const auto foo = [&x, &y]() noexcept { ++x; ++y; };
foo()
```

- 编译展开后：

```
struct _unnamedLambda{
    _unnamedLambda(int& a, int& b): x(a), y(b) { }
    void operator()() const noexcept{
        ++x; ++y;
    }
    int& x;
    int& y;
};
```

- 注意事项：

- 捕获模式：虽然'='或'['捕获所有变量很方便，但显式捕获变量更安全，避免意外副作用。
- 生命周期：C++ 闭包不会延长捕获引用的生命周期，确保在 Lambda 调用时捕获的变量仍然存在。

## mutable 关键字

在默认情况下，Lambda 表达式的闭包类型的'operator()'被标记为'const'，因此不能在 Lambda 体内修改捕获的变量。但如果要改变这种行为，就需要在参数列表后添加'mutable'关键字，这种用法实际上从闭包类型的调用操作符声明中移除了'const'：Lambda 表达式：

```
int x = 1;
auto foo = [x]() mutable { ++x; };
```

编译展开：

```
struct __lambda_x1{
    void operator()(){ ++x; }
    int x;
};
```

## 使用 `mutable` 拷贝捕获两个变量

```
#include <iostream>

int main(){
    const auto print = [] (const char* str, int x, int y){
        std::cout << str << ":" << x << " " << y << '\n';
    };

    int x = 1, y = 1;
    print("in main()", x, y);

    auto foo = [x, y, &print] () mutable {
        ++x;
        ++y;
        print("in foo()", x, y);
    };

    foo();
    print("in main()", x, y);
}

/* output:
 * in main(): 1 1
 * in foo(): 2 2
 * in main(): 1 1
 */
```

上述代码中，Lambda 表达式通过拷贝捕获了'x'和'y'，并通过引用捕获了'print'。在'foo'内部，'x'和'y'的值被修改，但这些修改并不影响外部作用域中的原始变量'x'和'y'。

## 通过引用捕获变量

当通过引用捕获时，Lambda可以在不使用'mutable'的情况下修改引用的值：

```
int x = 1;
std::cout << x << '\n';
const auto foo = [&x]() noexcept { ++x; };
foo();
std::cout << x << '\n';

/* output:
 * 1
 * 2
 */
```

#### 关于 mutable 和 const

使用'mutable'时，不能将生成的闭包对象标记为'const'，因为这会阻止调用 Lambda：

```
int x = 10;
const auto lam = [x]() mutable { ++x; };
// lam(); 将导致编译出错
```

导致编译出错的原因是不能在'const'对象上调用非'const'成员函数。

#### 捕获变量的实例-调用计数器

例子背景：Lambda 表达式在需要使用标准库中的算法并改变其默认行为时很有用。

在'std::sort'中，通常可以自定义比较函数，现在，可以在其中引入一个计数器来增强比较器的功能。代码示例：

```
#include <algorithm>
#include <iostream>
#include <vector>

int main(){
    std::vector<int> vec = {0, 5, 2, 9, 7, 6, 1, 3, 4, 8};
    size_t compCounter = 0;

    std::sort(vec.begin(), vec.end(), [&compCounter](){
```

```

        ++compCounter;
        return a < b;
    });

    std::cout << "Number of comparisons: " << compCounter << '\n';
    for(const auto& v: vec) std::cout << v << ',';
}

/* output:
 * Number of comparisons: 54
 * 0,1,2,3,4,5,6,7,8,9,
 */

```

## 捕获全局变量

在 Lambda 表达式中使用' [=]'按值捕获所有变量，但对于全局变量而言，并不如此：

```

#include <iostream>

int global = 10;

int main(){
    std::cout << global << '\n';

    auto foo [=]() mutable noexcept { ++global; };
    foo();
    std::cout << global << '\n';

    const auto increaseGlobal = []() noexcept { ++global; };
    increaseGlobal();
    std::cout << global << '\n';

    /* compile error
     * const auto moreIncreaseGlobal = [global]() noexcept { ++global; };
     * moreIncreaseGlobal();
     * std::cout << global << '\n';
     */
}

/* output:
 *
 */

```

```
* 10  
* 11  
* 12  
*/
```

无论使用什么方式捕获，Lambda 表达式始终引用全局对象，而不会创建局部副本。最后一个 moreIncreaseGlobal() 使用 Clang 会编译失败，说明不能捕获全局变量。

## 捕获静态变量

与捕获全局变量类似，捕获静态对象时也会遇到同样的问题：

```
#include <iostream>

void bar(){
    static int static_int = 10;
    std::cout << static_int << '\n';

    auto foo = [=]() mutable noexcept { ++static_int; };
    foo();
    std::cout << static_int << '\n';

    const auto increase = []() noexcept { ++static_int; };
    increase();
    std::cout << static_int << '\n';

/* compile error
 * const auto moreIncrease = [static_int]() { ++static_int; };
 * moreIncrease();
 * std::cout << static_int << '\n';
 */
}

/* output:
 * 10
 * 11
 * 12
 */
```

与全局变量相同，静态变量不能按值捕获，使用Clang进行编译会报错，因为不能捕获具有非自动存储持续时间的变量。

### 捕获类成员变量和'this'指针

在类成员函数中捕获成员变量会更加复杂，因为所有数据成员都与'this'指针相关联。一个错误示例：

```
#include <iostream>

struct Baz{
    void foo(){
        const auto lam = [s]() { std::cout << s; };
        lam();
    }
    std::string s;
};

int main(){
    Baz b;
    b.foo();
}
```

错误原因：不能捕获'Baz::s'并且'this'指针没有捕获。

```
struct Baz{
    void foo(){
        const auto lam = [this]() { std::cout << s; };
        lam();
    };
    std::string s;
};
```

通过使用'this'指针，可以捕获成员变量。

### 从方法返回 Lambda

```
#include <iostream>
```

```

struct Baz{
    std::function<void()> foo(){
        return [=, this] { std::cout << s << '\n'; }
    }
    std::string s;
};

int main(){
    auto f1 = Baz{"abc"}.foo(); /* temporary object */
    auto f2 = Baz{"xyz"}.foo(); /* temporary object */
    f1();
    f2();
    Baz b("ex");
    auto func = b.foo();
    func();
}
/* output:
 *
 *
 * ex
 */

```

'foo()'方法返回一个 Lambda，该 Lambda 捕获类的成员变量。以下类似：

```

struct Bar{
    std::string const& foo() const { return s; }
    std::string s;
};

auto&& f1 = Bar{"abc"}.foo(); // dangling reference

```

或者：

```

std::function<void()> foo(){
    return[s] { std::cout << s << '\n'; };

```

```
}
```

上面的代码中'f1'和'f2'使用的都是临时对象，可能会出现空悬引用(dangling reference)的问题，导致未定义行为。捕获'this'在 Lambda 的生命周期可能超过对象本事时可能会出现其他问题，特别是在异步调用(async)和多线程(multithreading)中。

### 只能移动对象(moveable-only object)

对于一个只能移动的对象(例如‘unique\_ptr’)，那么不能将其作为捕获变量按值捕获到 Lambda 表达式中，只能够按引用捕获，但是这并不会转移对象的所有权：

```
#include <iostream>
#include <memory>

int main(){
    std::unique_ptr<int> p(new int{10});

    // 按值捕获 - 编译错误
    // auto foo = [p]();

    // 按引用捕获 - 可通过编译，但不转移所有权
    auto foo_ref = [&p]() { std::cout << *p << '\n'; };
    foo_ref();
}

/* output:
 * 10
 */
```

在上面这种情况中，捕获'std::unique\_ptr'的唯一方法是按引用捕获，然后，这种方法不能转移指针的所有权。解决方法：使用 C++14 中的初始化捕获：通过初始化捕获，可以在 Lambda 表达式中捕获一个移动的对象，从而转移其所有权。

```
#include <iostream>
#include <memory>

int main(){
    std::unique_ptr<int> p(new int{10});
```

```

// 使用初始化捕获 - 转移所有权
auto foo = [p = std::move(p)](){
    std::cout << *p << '\n';
};

foo();

if(!p) std::cout << "p is nullptr after being moved" << '\n';
}

/* output:
 * 10
 * p is nullptr after being moved
 */

```

### 保持常量性(const preserving)

如果捕获了一个常量变量，其常量性会被保留：

```

#include <iostream>
#include <type_traits>

int main(){
    const int x = 10;
    auto foo = [x]() mutable{
        std::cout << std::is_const<decltype(x)>::value << '\n';
        // x = 11; 编译错误
    }
    foo();
}
/* output:
 * 1
 */

```

由上面的代码可知，即使在 Lambda 表达式中使用 'mutable' 关键字，'x' 的常量性依然保留，不能被修改。

### 参数包捕获

在捕获子句中，也可以利用可变参数模板(variadic templates)来捕获参数包：

```

#include <iostream>
#include <tuple>

template<class... Args>
void captureTest(Args... args){
    const auto lambda = [args...]{
        const auto tup = std::make_tuple(args...);
        std::cout << "tuple size: " <<
std::tuple_size<decltype(tup)>::value << '\n';
        std::cout << "tuple 1st: " << std::get<0>(tup) << '\n';
    };
    lambda();
}

int main(){
    captureTest(1, 2, 3, 4);
    captureTest("Hello Lambda", 10.0f);
}
/* output:
 * tuple size: 4
 * tuple 1st: 1
 * tuple size: 2
 * tuple 1st: Hello Lambda

```

编译展开:

```

#include <iostream>
#include <tuple>

template<class ... Args>
void captureTest(Args... args)
{

    class __lambda_6_22
    {
        public:
            inline auto operator()() const

```

```

    {
        const auto tup = std::make_tuple(args... );
        (std::operator<<(std::cout, "tuple size: ") <<
std::tuple_size<decltype(tup)>::value) << '\n';
        (std::operator<<(std::cout, "tuple 1st: ") << std::get<0>(tup)) <<
'\n';
    }

private:
Args... args;

public:
__lambda_6_22(const type_parameter_0_0... & _args)
: args{_args...}
{ }

};

const auto lambda = __lambda_6_22{args};
lambda();
}

/* First instantiated from: insights.cpp:15 */
#ifndef INSIGHTS_USE_TEMPLATE
template<>
void captureTest<int, int, int, int>(int __args0, int __args1, int
__args2, int __args3)
{
    class __lambda_6_22
    {
public:
    inline /*constexpr */ void operator()() const
    {
        const std::tuple<int, int, int, int> tup =
std::make_tuple(__args0, __args1, __args2, __args3);
        std::operator<<(std::operator<<(std::cout, "tuple size:
").operator<<(std::integral_constant<unsigned long, 4>::value), '\n');
    }
}

```

```

        std::operator<<(std::operator<<(std::cout, "tuple 1st:
").operator<<(std::get<0>(tup)), '\n');

    }

private:
    int __args0;
    int __args1;
    int __args2;
    int __args3;

public:
    __lambda_6_22(int & __args0, int & __args1, int & __args2, int &
__args3)
        : __args0{__args0}
        , __args1{__args1}
        , __args2{__args2}
        , __args3{__args3}
    {}

};

const __lambda_6_22 lambda = __lambda_6_22{__args0, __args1, __args2,
__args3};
lambda.operator()();
}

#endif

/* First instantiated from: insights.cpp:16 */
#ifndef INSIGHTS_USE_TEMPLATE
template<>
void captureTest<const char *, float>(const char * __args0, float
__args1)
{
    class __lambda_6_22
    {
public:

```

```

    inline /*constexpr */ void operator()() const
    {
        const std::tuple<const char *, float> tup =
        std::make_tuple(__args0, __args1);
        std::operator<<(std::operator<<(std::cout, "tuple size:
").operator<<(std::integral_constant<unsigned long, 2>::value), '\n');
        std::operator<<(std::operator<<(std::operator<<(std::operator<<(std::cout, "tuple
1st: "), std::get<0>(tup)), '\n'));
    }

private:
    const char * __args0;
    float __args1;

public:
    __lambda_6_22(const char * __args0, float & __args1)
    : __args0{__args0}
    , __args1{__args1}
    {}

};

const __lambda_6_22 lambda = __lambda_6_22{__args0, __args1};
lambda.operator()();
}

#endif

int main()
{
    captureTest(1, 2, 3, 4);
    captureTest("Hello Lambda", 10.0F);
    return 0;
}

```

可以通过可变参数模板在 Lambda 表达式中捕获参数包，捕获的参数包可以存储在'tuple'对象中，便于访问和操作。

## 返回类型推断

在很多情况下，可以省略 Lambda 表达式的返回类型，从 C++11 开始，编译器能够推断返回类型，只要所有的 return 语句返回的表达式类型相同：

```
#include <type_traits>

int main(){
    const auto baz = [] (int x) noexcept{
        if(x < 20) return x * 1.1; // return double
        else return x * 2.1;      // return double
    };
    static_assert(std::is_same<double, decltype(baz(10))>::value, "has
to be the same");
}
```

在上面的 Lambda 表达式中，两个返回语句的返回类型都为 double，因此编译器可以推断出返回类型。

## 尾置返回类型语法

使用尾置返回类型语法可以显式地指定返回类型：

```
#include <iostream>

int main(){
    const auto testSpeedString = [] (int speed) noexcept{
        if(speed > 100) return "you're a super fast";
        else return "you're a regular";
    };
    auto str = testSpeedString(100);
    str += " driver";
    std::cout << str;
}
```

上述代码会出现编译错误，因为 const char\* 没有 += 操作符，调整后：

```
auto testSpeedString = [] (int speed) -> std::string {
    if(speed > 100) return "you're a super fast";
```

```

        else return "you're a regular";
    };
    auto str = testSpeedString(100);
    str += " driver";
/* output:
 * you're a regular driver
 */

```

注意，此处在显式设置了返回类型为'std::string'后，需要移除'noexcept'，因为创建了'std::string'可能会抛出异常。或者使用'std::string\_literals'，然后返回""you're a regular"s'来表示'std::string'类型。

此处也可实现一个'std::string'的继承类speedString，并实现operator+=的重载：

```

#include <iostream>

class speedString: public std::string{
public:
    using std::string::string; // 继承 std::string 的构造函数

    speedString& operator+=(const std::string& rhs){
        std::string::operator+=(rhs);
        return *this;
    }

    // 重载 operator+= 以支持 const char* 类型
    speedString& operator+=(const char* rhs){
        std::string::operator+=(rhs);
        return *this;
    }
};

int main(){
    const auto testSpeedString = [] (int speed) noexcept -> speedString{
        if(speed > 100) return "you're a super fast";
        else return "you're a regular";
    };
}

```

```
speedString str = testSpeedString(100);
str += " driver";
std::cout < str;
}
/* output:
 * you're a regular driver
 */
```

## 函数指针转换(Conversion Function Pointer)

如果 Lambda 表达式没有捕获任何变量，编译器可以将其转换为常规函数指针，标准中描述如下：

- ⚠ 对于没有捕获的 Lambda 表达式，其闭包类型具有一个公共的、非虚的、非显式的 `const` 转换函数，该函数转化为具有与闭包类型的函数调用运算符相同参数和返回类型的函数指针。该转换函数返回的值应该是一个函数的地址，当调用该函数时，其效果与调用闭包类型的函数调用运算符相同。

例如：

```
#include <iostream>
void callWith10(void (*bar)(int)){
    bar(10);
}

int main(){
    struct{
        using f_ptr = void()(int);
        void operator()(int s) const { return call(s); }
        operator f_ptr() const { return &call; }
    private:
        static void call(int s) { std::cout << s << '\n'; }
    } baz;

    callWith10(baz);
```

```
callWith10([](int x) { std::cout << x << '\n'; };
```

### ⚠ 解释:

1. 'callWith10()': 'void(\*bar)(int)'是一个函数指针，指向返回为类型为'void',参数类型为'int'的函数，'callWith10()'这个函数接受一个这样的函数指针作为参数，然后调用该函数并传入参数'10'。
2. 'using f\_ptr = void(\*)(int);'等价于 `typedef void(*f_ptr)(int);`; 定义了一个函数指针类型'f\_ptr'。
3. 'void operator()(int s) const { return call(s); }'重载了'operator()',使得对象'baz'对象可以像函数一样被调用，并且会调用私有的静态成员函数'call'。
4. 'operator f\_ptr() const { return & call; }'定义了从结构体类型到函数指针类型的隐式转换操作符，也就是说，这个结构体实例'baz'可以被隐式转换为指向静态成员函数'call'的函数指针。

示例：使用Lambda 调用C库中的'std::qsort'进行反向排序：

```
#include <iostream>
#include <cstdlib>

int main(){
    int values[] = {8, 9, 2, 5, 1, 4, 7, 3, 6};
    constexpr size_t numElements = sizeof(values) / sizeof(values[0]);

    std::qsort(values, numElements, sizeof(int),
               [] (const void* a, const void* b) noexcept {
        return (*(int*)b - *(int*)a);
    });

    for(const auto& val: values) std::cout << val << ", ";
}
```

```
/* output:  
 * 9, 8, 7, 6, 5, 4, 3, 2, 1  
 */
```

上面的代码中，'std::qsort'只接受函数指针作为比较器，编译器可以隐式地将传递地无状态 Lambda 表达式转换为函数指针。

### ⚠ 总结：

#### 1. 无捕获 Lambda 转换为函数指针：

- 无捕获地 Lambda 表达式可以转换为与其函数调用运算符具有相同参数和返回类型地函数指针。
- 这种转换由编译器自动完成，方便在需要C风格回调地情况下使用。

#### 2. 仿函数(functor)显式转换：

- 通过定义一个转换操作符，仿函数可以显式地转换为函数指针。
- 这在需要传递复杂对象(如仿函数)到需要函数指针地接口时非常有用。

## 一个棘手的案例

案例如下：

```
#include <type_traits>  
  
int main(){  
    auto funcPtr = +[]{};  
    static_assert(std::is_same(std::decltype(funcPtr), void(*)()>::value);  
}
```

编译展开：

```
#include <type_traits>  
  
int main()
```

```

{

class __lambda_8_19
{
public:
    inline /*constexpr */ void operator()() const
    {

    }

using retType_8_19 = auto (*)() -> void;
inline constexpr operator retType_8_19 () const noexcept
{
    return __invoke;
}

private:
    static inline /*constexpr */ void __invoke()
    {
        __lambda_8_19{}.operator()();
    }
}

public:
// /*constexpr */ __lambda_8_19() = default;

};

using FuncPtr_8 = auto (*)() -> void;
FuncPtr_8 funcPtr = +__lambda_8_19{}.operator
__lambda_8_19::retType_8_19();
/* PASSED: static_assert(std::integral_constant<bool, true>::value);
*/
return 0;
}

```

源代码使用了'+'，这是一个一元运算符，这个运算符可以用于指针，因此编译器将无状态的 Lambda 转换为函数指针，然后赋值给'funcPtr'，相反，如果没有一元运算符'+'，'funcPtr'就

只是一个常规的闭包对象，同时'static\_assert'也会失效。

在这种情况下，一元操作符'+'和'static\_cast'的作用效果相同，如果不希望编译器创建太多函数实例化时，可以进行如下操作：

```
template<typename F>
void call_function(F f){
    f(10);
}

int main(){
    call_function(static_cast<int(*)(int)>([](int x) {
        return x + 2;
    }));
    call_function(static_cast<int(*)(int)>([](int x) {
        return x * 2;
    }));
}
```

在上面的代码中，编译器只需要创建一个'call\_function'的实例，因为它只接受一个函数指针'int(\*)(int)'，如果去掉了'static\_cast'，那么编译器就会为每个 Lambda 创建两个不同类型的'call\_function'实例。

## IIEF(Immediately Invoked Expression Function) - 立即调用的函数表达式

直接调用 Lambda 表达式示例：

```
#include <iostream>

int main(){
    int x = 1, y = 1;
    [&]() noexcept { ++x; ++y; }();
    std::cout << x << ',' << y;
}
/* output:
 * 2, 2
 */
```

此时，Lambda 表达式创建后没有分配给任何闭包对象，而是直接通过'()'调用。这样的 Lambda 表达式，在初始化一个复杂的'const'对象时比较有用。

```
const auto val = [](){
/* do something */
}();
```

此时，'val'是一个由 Lambda 表达式返回的类型常量值：

```
/* val1 是 int */
const auto val1 = []() { return 10; }();
/* val2 是 std::string */
const auto val2 = []() -> std::string { return "ABC"; }();
```

一个更具体的示例：使用IIFE作为助手 Lambda 来在函数内部创建一个常量值——IIFE 与 HTML 生成示例：

```
#include <iostream>

void Validate(const std::string&) {}

std::string BuildHref(const std::string& link,
                      const std::string& text){
    const std::string html = [&link, &text] {
        const std::string inText = text.empty() ? link : text;
        return "<a href=\"" + link + "\">" + inText + "</a>";
    }();
    Validate(html);
    return html;
}

int main(){
    try{
        const auto ahref = BuildHref("ppqwqqq.space", "ppQwQqq");
        std::cout << ahref;
    }
    catch (...) {
```

```
    std::cout << "bad format...";  
}  
}
```

上面的代码中，'BuildHref'函数，接受两个参数，然后生成一个'<a></a>'HTML标签，基于输入参数，构建'html'变量，如果'text'不为空，则将其用作内部HTML值，否则使用'link'。通过使用 IIEF 可以在对多输入参数的条件下使表达式更加简洁：编写一个独立的 Lambda 表达式，然后将其变量标记为'const'，之后即可将'const'变量传递给'ValidateHTML'。

## 提高 IIEF 代码可读性的方法

### 1. 避免使用'auto'

- 明确地指定类型，以便更清楚地看到变量的类型：

- const bool EnableErrorReporting = [&]() {  
 if(HighLevelWarningEnabled()) return true;  
 if(HighLevelWarningEnabled()) return UsersWantReporting();  
 return false;  
}();

### 2. 添加注释：

- 在'}'后面添加一个注释，指明这是'IIEF'：

- const bool EnableErrorReporting = [&]() {  
 if (HighLevelWarningEnabled()) return true;  
 if (HighLevelWarningEnabled()) return UserWantReporting();  
 return false;  
}(); // call it now

## Lambda 表达式的继承与多态

Lambda 表达式的继承：由于编译器会将 Lambda 表达式展开为带有'operator()'的仿函数对象，因此可以从这种类型继承：

```

#include <iostream>

template<typename Callable>
class ComplexFunctor: public Callable{
public: explicit ComplexFunctor(Callable f): Callable(f) { }
}

template<typename Callable>
ComplexFunctor<Callable> MakeComplexFunctor(Callable&& cal){
    return ComplexFunctor<Callable>(cal);
}

int main(){
    const auto func = MakeComplexFunctor([]() {
        std::cout << "Hello Functor\n";
    });
    func();
}

```

在这个例子中，'ComplexFunctor'类从模板参数'Callable'继承，如果想从 Lambda 继承，必须添加一些额外的操作，因为无法明确知道闭包类型的确切类型(除非将其封装在'std::function'中)，因此需要'MakeComplexFunctor'函数来执行模板参数推导并获取 Lambda 闭包类型。

多重 Lambda 继承：示例：从两个 Lambda 继承并创建一个重载集：

```

#include <iostream>

template<typename TCall, typename UCall>
class SimpleOverLoaded: public TCall, UCall{
public:
    SimpleOverLoaded(TCall tf, UCall uf): TCall(tf), UCall(uf){}
    using TCall::operator();
    using UCall::operator();
};

template<typename TCall, typename UCall>

```

```

SimpleOverLoaded<TCall, UCall> MakeOverloaded(TCall&& tf, UCall&& uf){
    return SimpleOverLoaded<TCall, UCall>(tf, uf);
}

int main(){
    const auto func = MakeOverloaded(
        [] (int) { std::cout << "Int!\\n"; },
        [] (float) { std::cout << "Float!\\n"; }
    );
    func(10);
    func(10.0f);
}
/* output:
 * Int!
 * Float!
 */

```

此处从两个模板进行继承，并显示暴露它们的'operator()'。

## 为什么需要显式暴露

编译器在寻找正确的重载函数时，要求它们得在同一个作用域中：

```

#include<iostream>

struct BaseInt{
    void Func(int) { std::cout << "BaseInt...\\n"; };
};

struct BaseDouble{
    void Func(double) { std::cout << "BaseDouble...\\n"; }
};

struct Derived: public BaseInt, BaseDouble{
    using BaseInt::Func;
    using BaseDouble::Func;
};

```

```
int main(){
    Derived d;
    d.Func(10.0);
}
/* output:
 * BaseDouble...
 */
```

如果没有'using'语句，编译器就会报错，因为'Func()'可以来自'BaseInt'或'BaseDouble'的作用域，编译器无法决定使用哪个。

## 在容器中存储 Lambda 表达式

使用函数指针存储 Lambda: Lambda 表达式不能默认创建和赋值，然而利用无状态 Lambda 表达式转换为函数指针的特性，虽然无法直接存储闭包对象，但可以保存从 Lambda 表达式转换出来的函数指针：

```
#include <iostream>
#include <vector>

int main(){
    using Func = void(*)(int&);
    std::vector<TFunc> ptrFuncVec;

    ptrFuncVec.push_back([](int& x) { std::cout << x << '\n'; });
    ptrFuncVec.push_back([](int& x) { x *= 2; });
    ptrFuncVec.push_back(ptrFuncVec[0]);

    int x = 10;
    for(const auto& entry: ptrFuncVec) entry(x);
}
```

在'ptrFuncVec'中有三个变量：

1. 输出输入参数的值。
2. 修改该值
3. 是第一个的副本，再次输出该值。

这种方法虽然有效，但仅限于无状态的 Lambda 表达式。

使用`std::function`封装Lambda：为了能够在容器中能够使用其他的状态的 Lambda 表达式，可以使用`'std::function'`处理，这样，使其不仅可以处理整数，还可以处理字符串对象：

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>

int main(){
    std::vector<std::function<std::string(const std::string&)>>
vecFilters;

    size_t removedSpaceCounter = 0;
    const auto removeSpaces = [&removedSpaceCounter](const std::string&
str){
        std::string tmp;
        std::copy_if(str.begin(), str.end(), std::back_inserter(tmp),
                    [] (char ch) { return !isspace(ch); });
        removedSpaceCounter += str.length() - tmp.length();
        return tmp;
    }

    const auto makeUpperCase = [] (const std::string& str){
        std::string tmp = str;
        std::transform(tmp.begin(), tmp.end(), tmp.begin(),
                    [] (unsigned char c) { return std::toupper(c);
                });
        return tmp;
    };

    vecFilters.emplace_back(removeSpaces);
}
```

```

vecFilters.emplace_back([](const std::string& x){
    return x + " Amazing";
});
vecFilters.emplace_back([](const std::string& x){
    return x + " Modern";
});
vecFilters.emplace_back([](const std::string& x){
    return x + " C++";
});
vecFilters.emplace_back([](const std::string& x){
    return x + " World!";
});
vecFilters.emplace_back(makeUpperCase);

const std::string str = "    H e l l o      ";
auto temp = str;
for(const auto& entryFunc: vecFilters) temp = entryFunc(temp);
std::cout << temp << '\n';
std::cout << "Removed spaces: " << removedSpaceCounter << '\n';
}

/* output:
 * HELLO AMAZING MODERN C++ WORLD!
 * Removed spaces: 12
 */

```

此代码，在容器中存储'std::function<std::string(const std::string&)>'允许使用任何类型的函数对象，包括捕获变量的 Lambda 表达式。

# 汇编语言

Start typing here...

# 第一章 基础知识

## 1.1 机器语言

机器语言是机器指令的集合。

机器指令就是一台计算机可以正确执行的命令，**机器指令是一列二进制数字**，计算机将其转换为一列高低电平，使计算机的电子器件受到驱动，然后进行运算。

在现代的PC机中，负责执行上述操作的硬件为CPU(*Central Processing Unit*, 中央处理单元)。

**一般来说，一个计算机是由CPU及其它受CPU直接或间接控制的芯片、器件、设备组成的计算机系统。**

不同的CPU，其硬件设计和内部结构都是不相同的，因此就需要用不同的电平脉冲对其进行控制，使其工作，所以**每一种CPU都有自己的机器指令集，也就是机器语言**。

早机器的程序设计使用的机器语言，也就是使用0和1来进行一系列的编程工作，这导致书写和阅读机器码十分困难，并且在后期的调试过程中也会显得十分麻烦，为了解决这样的问题，在后续就引入了汇编语言。

## 1.2 汇编语言

**汇编语言的主体使汇编指令。** 因为汇编语言的出现主要是为了简化机器码的表示，所以汇编指令和机器指令的差别主要在于指令的表示方法上。

例如：

- 操作：寄存器 BX 的内容发送到 AX 中
- 机器指令：1000100111011000
- 汇编指令：mov ax, bx

**▲ 寄存器：**在CPU中可以存储的器件，一个CPU中有多个寄存器。AX是其中一个寄存器的代号，BX是另一个寄存器的代号。

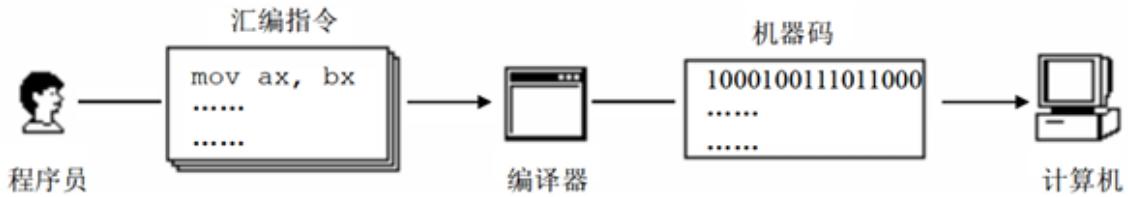


图 1.1 用汇编语言编写程序的工作过程

Assembly0.png

## 1.3 汇编语言组成

汇编语言由以下3类指令组成：

1. **汇编指令**：机器码的助记符，有对应的机器码。
2. **伪指令**：没有对应的机器码，由编译器执行，计算机并不执行。
3. **其他符号**：如+、-、\*、/等，由编译器识别，没有对应的机器码。

汇编语言的核心是汇编指令，它决定了汇编语言的特性。

## 1.4 存储器

CPU 是计算机的核心硬件，它控制整个计算机的运行并进行计算。要让一个CPU工作，就必须要向它提供指令和数据，而**指令和数据会存放在存储器中**，即内存。

## 1.5 指令和数据

指令和数据是应用上的概念，它们都是二进制信息，CPU在工作时会把有的信息看作指令，有的信息看作数据。

例如：内存中的一段二进制信息 1000100111011000，计算机(CPU)可以既可以将其当作数据进行计算，也可以当作指令执行

- 1000100111011000 -> 89D8H(数据)
- 1000100111011000 -> mov ax, bx(程序)

## 1.6 存储单元

存储器被划分为若干个存储单元。每个存储单元从0开始编号。对于一个有128个存储单元的存储器而言，其编号顺序如下图所示：

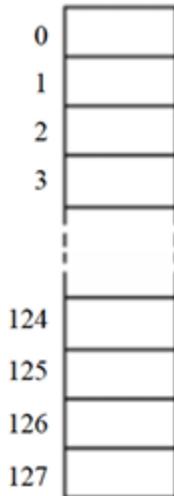


图 1.2 存储单元的编号

Assembly1.png

8bit(比特)组成1Byte(字节)，存储器中的存储单元可以存储1Byte，即8个二进制位(bit)，对于上面的存储器而言，它可以存储128Byte。对于拥有128个存储单元的存储器，其容量为128个字节(Byte)。

存储单位	换算关系
KB	1KB=1024B
MB	1MB=1024KB
GB	1GB=1024MB
TB	1TB=1024GB

## 1.7 CPU对存储器的读写

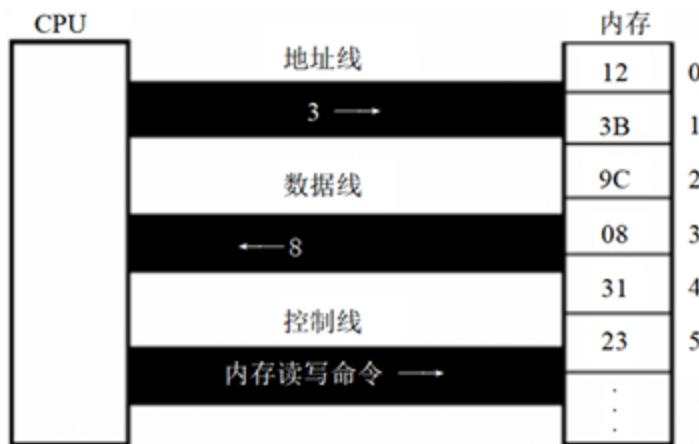
CPU 要从内存中读取数据，首先要确定存储单元的地址，同时也要确定是对哪一个器件进行操作，进行哪种操作。

CPU 想要进行数据的读写，必须和芯片进行下面三类信息的交互：

- 存储的单元的地址(地址信息)
- 器件的选择，读或写的命令(控制信息)
- 读或写的数据(数据信息)

在计算机中，电信号通过导线进行传输，这种连接CPU和其他芯片的导线被称为总线。在物理层面上来讲，就是一根根导线的集合。根据传送信息的不同，总线从逻辑上又分为3类，地址总线、控制总线和数据总线。

CPU 从 3 号单元中读取数据的过程如下：



Assembly2.png

读操作：

1. CPU 通过**地址线**将**地址信息 3**发出
2. CPU 通过**控制线**发出**内存读命令**，选中存储芯片，并通知它，将要从中读取数据。
3. 存储器将 3 号单元中的**数据 8**通过**数据线**送入 CPU。

写操作：

1. CPU 通过地址线将地址信息 3 发出。
2. CPU 通过控制线发出内存写命令，选中存储芯片，并通知它，将要向其中写入数据。
3. CPU 通过数据线将数据 26 送入内存的 3 号单元中。

对于 8086CPU，下面的机器码，可以实现从 3 号单元中读数据：机器码：  
10100001000000011000000000 含义：从 3 号单元读取数据送入寄存器 AX

完成上述操作后，再进行写操作：机器码：101000010000001100000000 对应的汇编指令：MOV AX, [3] 含义：传送 3 号单元的内容至 AX

## 1.8 地址总线

CPU 是通过地址总线来指定存储器单元的，并且地址总线可以传输的不同信息数量决定了 CPU 可以对多少个存储单元进行寻址。

先假设，一个 CPU 有 10 根地址总线，一根导线能够传递的稳定状态有两种，高电平和低电平，用二进制表示就是 1 或 0，10 根导线可以传送 10 位二进制数据，而 10 位二进制数可以表示  $2^{10}$  (0-1023, 即 1024) 个不同的数据。

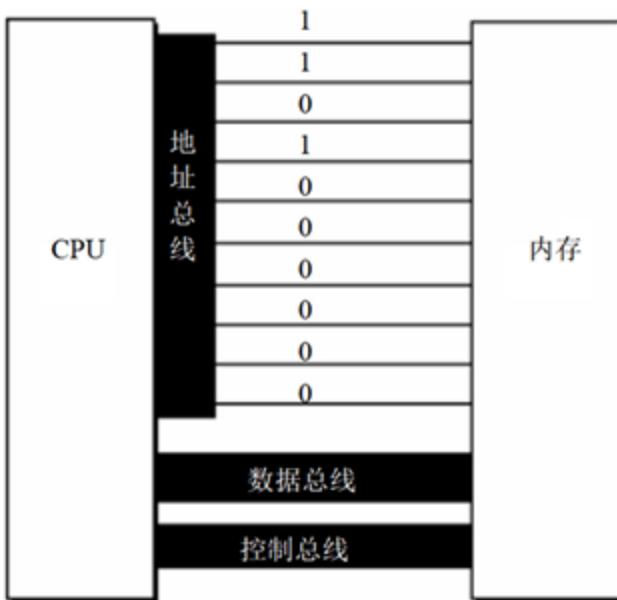


图 1.4 地址总线上发送的地址信息

Assembly3.png

上图展示了一个具有 10 根地址线的 CPU 向内存发出地址信息 11 时 10 根地址线上传送的二进制信息。

地址线0----- $2^0$ -----  
地址线1----- $2^1$ -----  
地址线2----- $2^2$ -----  
地址线3----- $2^3$ -----  
地址线4----- $2^4$ -----

地址线5----- $2^5$ -----  
地址线6----- $2^6$ -----  
地址线7----- $2^7$ -----  
地址线8----- $2^8$ -----  
地址线9----- $2^9$ -----

对应地，当访问地址为 12 时的内存单元时，地址总线上传输的内容为：

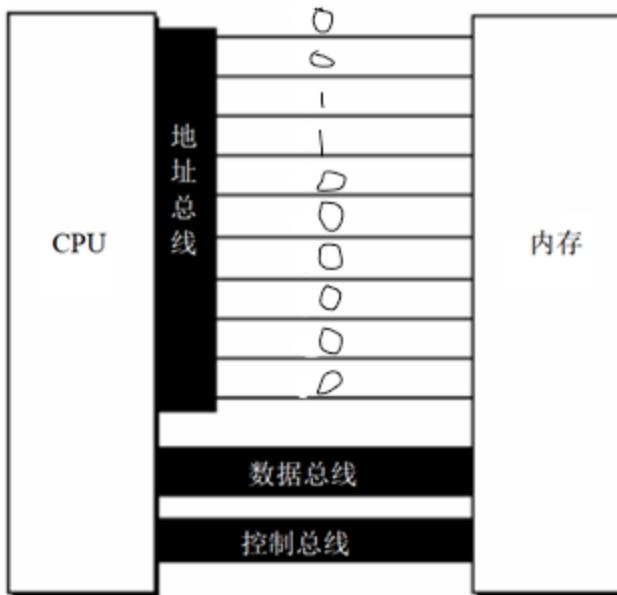


图 1.4 地址总线上发送的地址信息

Assembly4.png

一个 CPU 有 N 根地址线，则可以说这个 CPU 的地址总线的宽度为 N，这样的 CPU 最多可以寻找  $2^N$  个内存单元。

## 1.9 数据总线

CPU 与内存或其他芯片之间的数据传送是通过数据总线来进行的。数据总线的宽度决定了 CPU 和外界的数据传输速度。8 根数据总线一次可以传送一个 8 位二进制数据(即 1 Byte)，16 根数据总线一次可传送 2 Byte，即 1 根数据总线一次可传送 1 bit。

8088CPU 的数据总线宽度为 8，8086CPU 的数据总线宽度为 16，当向内存中写入数据 89D8H 时，数据传输过程如下：

8088CPU 分两次传送 89D8，第一次传送 D8，第二次传送 89。

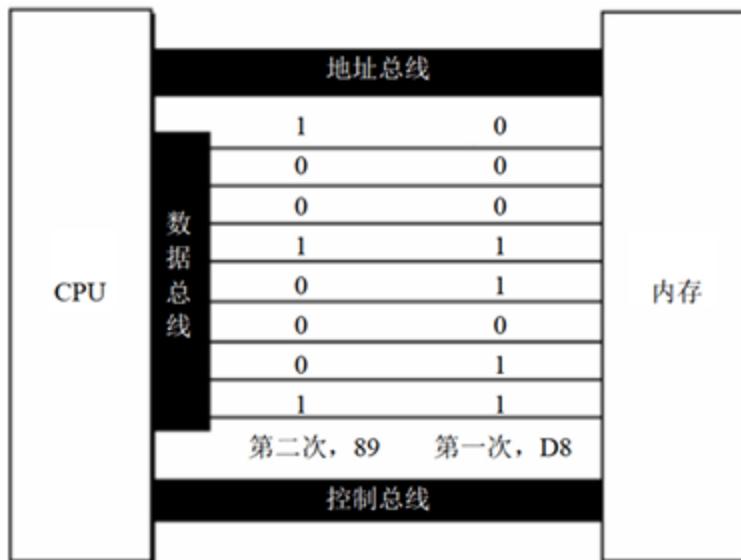


图 1.5 8 位数据总线上传送的信息

Assembly5.png

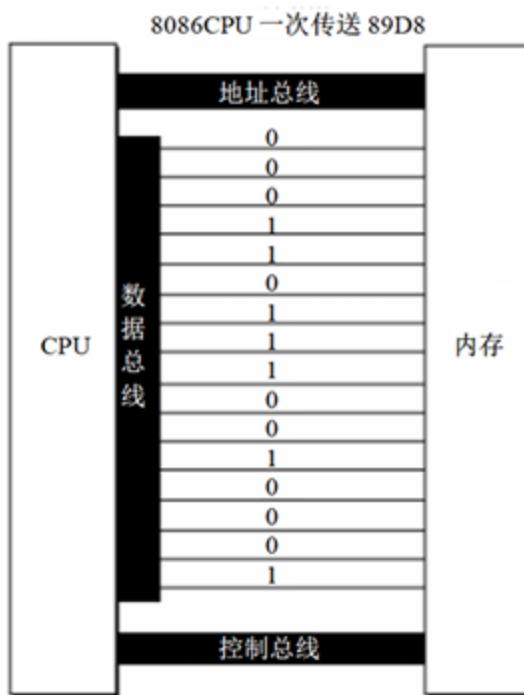


图 1.6 16 位数据总线上传送的信息

Assembly6.png

由上可知，8086 有 16 根数据线，可一次传输16位数据，所以可以一次传输数据 89D8H；而 8088 只有 8 根数据线，一次只能传输 8 位数据，所以向内存写入数据 89D8H 时需要进行两

次数据传输。

⚠ 补充：此处如何对传输的数据进行划分

1. 对于偶数字节的划分，将其划分为高低位，前一半为高位，后一半为低位。
2. 对于奇数字节的划分，将其均等化分为高中低位。
3. 在数据传输的过程中，如果总线宽度不够一次传输所有数据，据需要分高低位多次传输。

```
/* 例如对一个2字节(16位)数据  
使用位运算来提取高低字节  
*/  
unsigned char data = 0x89D8;  
unsigned char high_byte = (data >> 8) & 0xFF;    // 取高字节  
unsigned char low_byte = data & 0XFF;                // 取低字节
```

## 1.10 控制总线

CPU 对外部芯片的控制是通过控制总线来实现的。此处控制总线是一些不同控制线的集合。有多少根控制总线，就意味着 CPU 提供了对多种外部芯片的多种控制，所以控制总线决定了 CPU 对外部芯片的控制能力。

内存的读或写命令是由几根控制线综合发出的，其中就有一根成为“读信号输出”的控制线负责由 CPU 向外传送读信号，CPU 向该控制线上输出低电平表示将要读取数据；有一根称为“写信号输出”的控制线则负责传送写信号。

## 1.1 ~ 1.10 小结

1. 汇编指令是机器指令的助记符。
2. 每一种 CPU 都有自己的汇编指令集。
3. CPU 可以直接使用的信息在存储器中存放。
4. 在存储器中指令和数据没有任何区别，都是二进制信息。

5. 存储单元从0开始按顺序编号。
6. 一个存储单元可以存储8bit，即1Byte、8位二进制数。
7. 1Byte=8bit 1KB=1024B 1MB=1024KB 1GB=1024MB。
8. 每一个CPU芯片都有许多针脚，这些针脚和总线相连，或者说，这些针脚引出总线。一个CPU可以引出3种总线的宽度标志了这个CPU的不同方面的性能：
  - 地址总线的宽度决定了CPU的寻址能力；
  - 数据总线的宽度决定了CPU与其他器件进行数据传送时的一次数据传送量；
  - 控制总线的宽度决定了CPU对系统中其他芯片的控制能力。

## 小测 1.1

1. 一个CPU的寻址能力为8KB，那么它的地址总线的宽度为13。

**⚠** 寻址能力为8KB，即 $2^{13}$ 字节，所以地址总线宽度为13位。

2. 1KB的存储器由1024个存储单元组成，存储单元的编号从0到1023。

**⚠** 1KB = 1024字节(Byte)，每个存储单元存储1字节(Byte)，因此有1024个存储单元。

3. 1KB的存储器可以存储8192bit，1024个Byte。

4. 1GB、1MB、1KB分别是 $1024 \times 1024 \times 1024$ 、 $1024 \times 1024$ 、1024个Byte

5. 8080、8088、80286、80386的地址总线宽度分别为16根、20根、24根、32根，则它们的寻址能力分别为：64KB、1MB、16MB、4GB。

**⚠** 寻址能力为 $2^{\text{地址总线宽度}} \times \text{字节}$ ：

- 8080:  $2^{16} = 64\text{KB}$

- 8088:  $2^{20} = 1\text{MB}$
- 80286:  $2^{24} = 16\text{MB}$
- 80386:  $2^{32} = 4\text{GB}$

6. 8080、8088、8086、80826、80386的数据总线宽度分别为8根、8根、16根、16根、32根，则他们一次可以传送的数据为: 1B、 1B、 2B、 2B、 4B。

▲ 数据总线宽度决定一次可以传送的数据位数，每 8 位为 1 字节

7. 从内存中读取 1024 字节的数据，8086至少要读 512 次， 80386至少要读 256 次。

▲ 8086的数据总线宽度为 16 位(2 字节) 80386的数据总线宽度为 32 位(4 字节)

8. 在存储器中，数据和程序以 二进制 形式存放。

## 1.11 主板

在 PC 机中，主板上集成了一些核心器件和主要器件，这些器件通过总线(地址总线、数据总线、控制总线)相连。这些器件包括但不限于 CPU、存储器、外围芯片组、扩展插槽等。

## 1.12 接口卡

在计算机系统中，CPU 控制所有可用程序控制其工作的设备。CPU 不能对外部设备直接进行控制，直接控制这些设备进行工作的是插在扩展插槽上的接口卡，扩展插槽通过总线和 CPU 相连，所以接口卡也通过总线同 CPU 相连，CPU 可以直接控制这些接口卡，从而实现 CPU 对外部设备的间接控制。

## 1.13 各类存储器芯片

根据存储器的读写属性可以将其分为两类：

- 随机存储器(RAM)：随机存储器可读可写，但必须带电存储，关机后存储内容丢失

- 只读存储器(ROM)：只读存储器只能读取不能写入，关机后其中的内容不丢失。

根据存储器的功能和连接方式上可以分为以下几类：

- 随机存储器

**▲** 用于存放供 CPU 使用的绝大部分程序和数据，主随机存储器一般由两个位置上的 RAM 组成，装在主板上的 RAM 和插在扩展插槽上的 RAM。

- 装有 BIOS (Basic Input/Output System, 基本输入/输出系统) 的 ROM

**▲** BIOS 是由主板和各类接口卡厂商提供的软件系统，可以通过它利用该硬件设备进行最基本的输入输出。

- 接口卡上的 RAM

**▲** 某些接口卡需要大批量输入、输出数据进行暂时存储，在其上装有 RAM。最典型的是显卡上的 RAM，一般称之为显存，显卡随时将显存中的数据向显示器上输出。

PC 系统中各类存储器的逻辑连接情况如下所示：

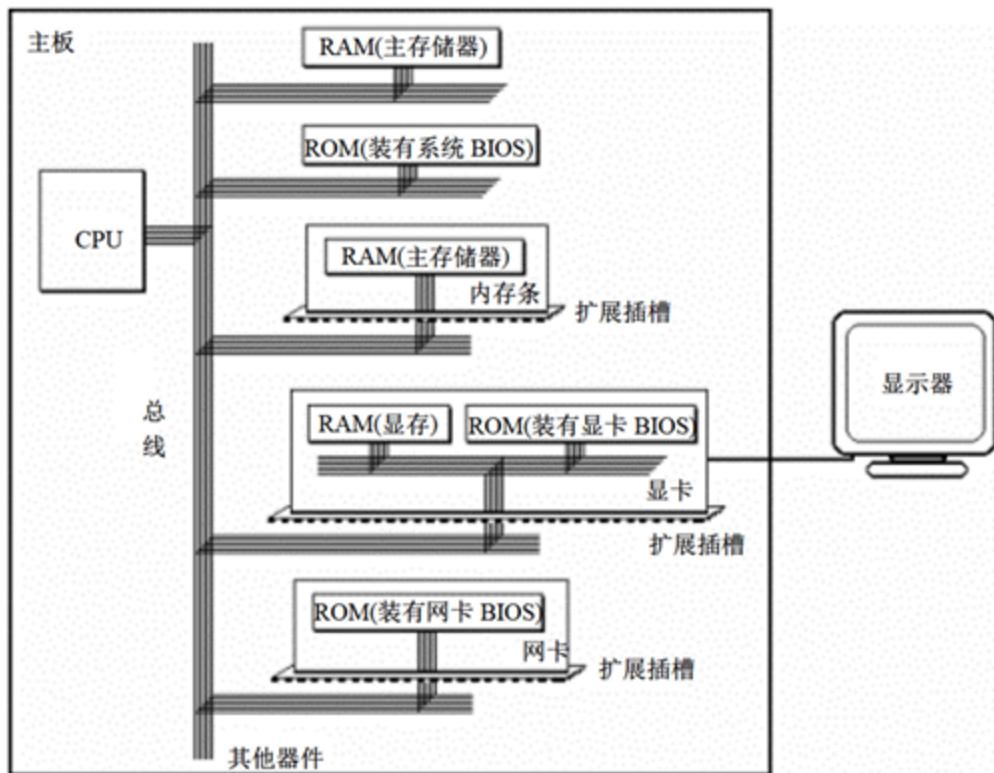


图 1.7 PC 机中各类存储器的逻辑连接

Assembly\_8.png

## 1.14 内存地址空间

内存地址空间(Memory Address Space)指的是计算机系统中内存单元的一个空逻辑排列，用于表示可以被处理器访问的所有存储位置。举个例子，一个 CPU 的地址总线宽度为 10，那么可以寻址 1024 个内存单元，这 1024 个可寻到的内存单元就构成这个 CPU 的内存地址空间。

前面提及的存储器，都有以下两个共同点：

- 都和 CPU 的总线相连
- CPU 对它们进行读或写的时候通过控制线发出内存读写命令

CPU 在控制这些存储器时，把它们当作内存来对待，把它们看作一个由若干存储单元组成的逻辑存储器，这个逻辑存储器就是内存地址空间。

CPU 将系统中各类存储器看作一个逻辑存储器的情况如下图所示：

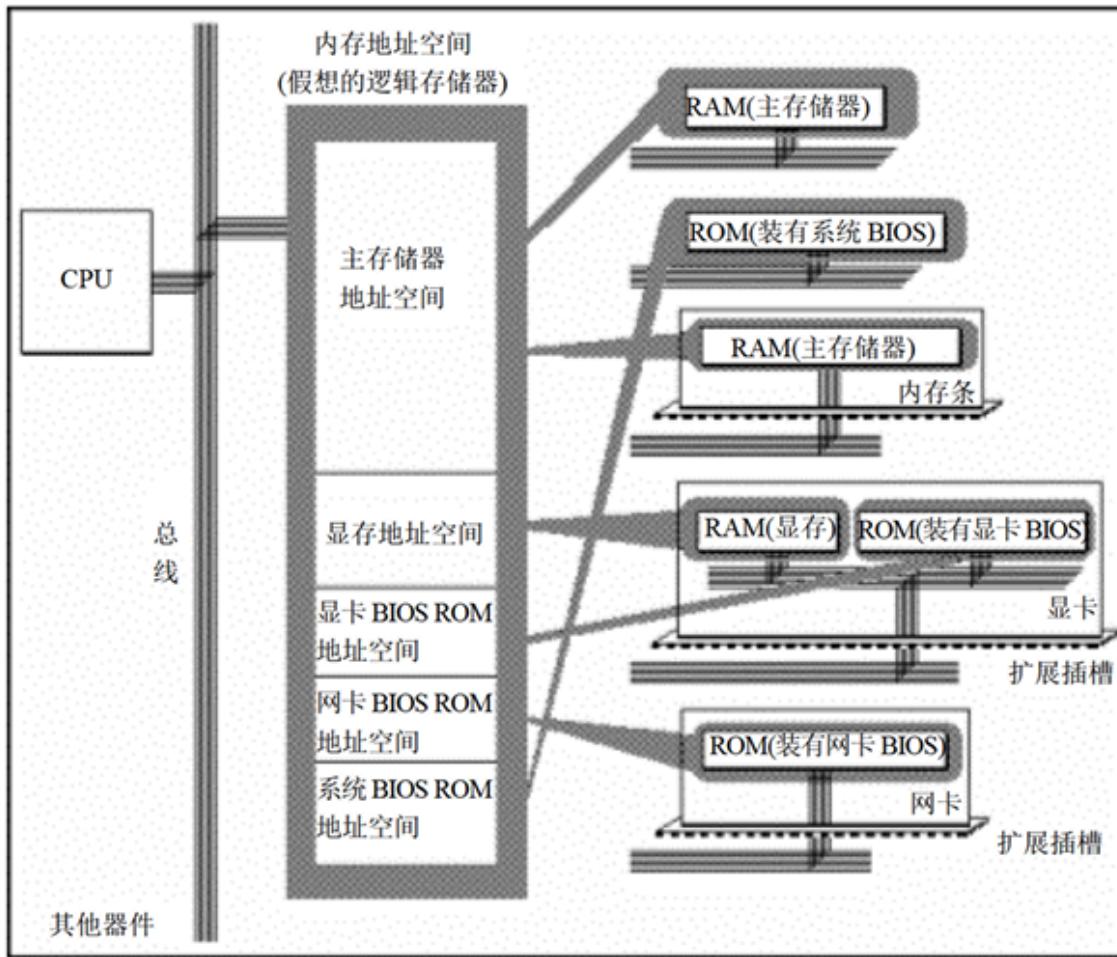


图 1.8 将各类存储器看作一个逻辑存储器

Assmebly\_8.png

在上图中，所有的物理存储器被看作一个由若干个存储单元组成的逻辑存储器，每个物理存储器在这个逻辑存储器中占有一段地址空间。CPU 在这段地址空间中读写数据，实际就是在相对应的物理存储器中读写数据。假设上图中的内存地址空间的地址段分配如下：

- 地址 0 ~ 7FFFH 的 32KB 空间为主随机存储器的地址空间
- 地址 8000H ~ 9FFFH 的 8KB 空间为显存的地址空间
- 地址 A000H ~ FFFFH 的 24KB 空间为各个 ROM 的地址空间

这样，CPU 可以根据不同的地址段对不同的存储器进行操作：

- CPU 向内存地址为 1000H 的内存单元中写入数据，这个数据就被写入主随机存储器中

- CPU 向内存地址为 8000H 的内存单元中写入数据，这个数据就被写入显存中，然后会被显卡输出到显示器上
- CPU 向内存地址为 C000H 的内存单元中写入数据的操作没有结果，C000H 单元中的内容不会被改变，C000H 单元实际上就是 ROM 存储器(只读存储器)中的一个单元

内存地址空间的大小受 CPU 地址总线宽度的限制。8086CPU 的地址总线宽度为 20，可以传送  $2^{20}$  (从 0 至  $2^{20} - 1$ ) 个不同的地址信息。即可以定位  $2^{20}$  个内存单元，则 8086CPU 的内存地址空间大小为 1MB( $1024 * 1024$  Byte)，同理 80386CPU 的地址总线宽度为 32，则内存地址空间最大为 4GB( $1024 * 1024 * 1024 * 4$  Byte)。

在基于一个计算机硬件系统编程时，必须知道这个系统中的内存地址分配情况。因为当对某类存储器进行读写操作时，必须知道它的第一个和最后一个单元的地址，才能保证读写操作是在预期的存储器中进行的。

不同的计算机系统的内存地址空间的分配情况是不同的，下图展示的是 8086PC 机内的地址空间分配的情况：



图 1.9 8086PC 机内存地址空间分配

Assembly\_9.png

从上图中可以看出：

- 从地址 0 ~ 9FFFF 的内存单元中读取数据，实际上就是在读取主随机存储器中的数据

- 向地址A000 ~ BFFF的内存单元中写入数据，就是向显存中写入数据，这些数据会被显卡输出到显示器上
- 向地址C000 ~ FFFF的内存单元中写入数据的操作是无效的，因为这个地址段对应的是ROM存储器(只读存储器)，但是可以向这个地址段进行数据读取的操作

**▲ 内存地址空间** 因为程序最终是运行在 CPU 上的，所以当在使用汇编语言进行编程时，必须要从 CPU 的角度考虑问题。对 CPU 而言，系统中所有存储器中的存储单元都处于一个统一的逻辑存储器中，它的容量受 CPU 的寻址能力(地址总线宽度)的限制。这个逻辑存储器就是内存地址空间。

# 第二章 寄存器

Start typing here...

# 第三章 寄存器(内存访问)

Start typing here...

# 现代操作系统

Start typing here...

# 操作系统

Start typing here...

# Functional Programming\_

## 第一章 Haskell简介

### 1. Haskell简介

- 提供了Haskell起源的历史概述，追溯到Lambda演算和Lisp、ISWIM、Scheme、ML等语言的出现。
- 讨论了在1987年的函数式编程与计算机体系结构会议上创建Haskell的动机，以解决函数式编程语言的泛滥问题。
- 强调了Haskell开发的重要性，将各种函数式编程语言的精髓融入统一的框架中。

### 2. 安装与开发环境设置

- 描述了GHC（格拉斯哥 Haskell 编译器）的安装过程，并推荐使用Haskell平台作为综合开发环境。
- 介绍了使用GHCi进行Haskell程序的交互式测试和调试。
- 建议使用Notepad++、Sublime、Emacs、Vim等编辑器编写Haskell代码，并特别注意保持正确的缩进和对齐。

### 3. 使用GHCi

- 解释了各种GHCi命令，用于导入文件、重新加载模块、更改当前目录、执行系统命令、退出GHCi以及访问帮助文档。
- 演示了如何在GHCi中调用函数，展示了算术运算、逻辑运算符、数学函数和基本列表操作。

### 4. .hs和.lhs文件、注释和库函数

- 区分了.hs和.lhs文件，.lhs文件主要用于文学化的Haskell编程，用于生成格式化的文档。
- 讨论了Haskell文件中的注释约定和编译器指令。
- 强调了Prelude中丰富的Haskell函数库，并鼓励探索各种任务的库函数。

### 5. 创建第一个Haskell程序 - HelloWorld

- 指导如何在Haskell中创建一个简单的HelloWorld程序，强调了main函数作为入口点的作用。
- 提供了使用GHC和runghc命令编译和执行Haskell程序的说明。
- 注意了Haskell和C等语言之间文件大小的差异，将其归因于Haskell对内存和磁盘空间的利用效率。

## 6. 结论

- 总结了本章的内容，并鼓励读者熟悉GHCi并进一步探索Haskell的能力。
- 预见了Haskell等函数式编程语言的光明未来，这是由它们简洁、健壮和安全的代码库驱动的。

# 第二章 类型系统和函数

1. Haskell常用数据类型
  - 1.1 布尔类型：Bool 布尔类型是一个只有 True 与 False 两个值的数据类型。布尔值的运算符号和其他语言相似，&& 表示“逻辑与”运算，|| 表示“逻辑或”运算，not 表示“逻辑非”运算。
  - 1.2 字符型：Char 由单引号包裹的单个字符都是 Char 类型的，与其他语言一致。
  - 1.3 有符号整数：Int Int 几乎是所有的编程语言里都有的数据类型。它的范围与操作系统和 GHC 位数有关。若使用的是 32 位的 GHC，那么整数的范围是  $-2^{31} \sim 2^{31} - 1$ 。对于 64 位 GHC 来说 Int 的范围则是  $-2^{63} \sim 2^{63} - 1$ ，Haskell 里还有另外一个整数类型——任意精度整数，如果不指明类型，Haskell 会将  $2^{32}$  默认为任意精度整数处理。
  - 1.4 无符号整数：Word Word 类型是无符号的整数，它的范围也是系统相关的。在 32 位系统中它的范围是  $0 \sim 2^{32} - 1$  而 64 位系统则为  $0 \sim 2^{64} - 1$ 。Haskell 中的 Word 相当于 C 语言里的 unsigned int 类型。使用 Word 类型需要导入 Data.Word 库，在 GHCi 中可以使用:module (简写为:m) 来控制模块的加载。
  - 1.5 任意精度整数：Integer 与 Int 不同，Integer 类型可以表示任意大小的整数，限制它的大小范围的唯一因素就是计算机的内存。
- 1.6 小数与有理数类型：Float、Double、Rational Haskell 中的单精度浮点数 Float、双精度浮点数 Double 与其他语言没有很大的区别。Haskell 还有有理数类型 Rational，即用两个任意精度的整数来表示一个小数，这在做高精度数学运算时有很多好。

## 2. sd

1.

1. 23

2. 速度

3.

1. 读书

# 编译原理

Start typing here...

# 内存池(Memory Pool)

Start typing here...

# **Effective C++**

# 第一章 让自己习惯 C++

## 条款1：视C++为一个语言联邦

关于C++的四种次语言(*sub-language*):

1. *C*, *C++*以*C*为基础的。
2. *Object-Oriented C++*, 体现了*C++*面向对象的特性。对于内置类型而言 *pass-by-value* 相比较于 *pass-by-reference* 会更高效。但对于面对对象的*C++*而言, *pass-value-by-reference-to-const* 往往更好。
3. *Template C++*, 泛型编程(*generic programming*)是*C++*重要的一部分。
4. *STL*, *C++*中的标准模板库。但在*Template*, *STL*中迭代器(iterator)和函数对象都是在*C*的指针上进行构建的。

总结: 在*C++*中对于不同部分的实现根据其特性采用不同的方法。

## 条款2：尽量以 `const`, `enum`, `inline` 替换 `#define`

在声明常量时尽量使用 `const`, `enum`, 声明函数时尽量使用 `inline`, 而不是 `define` (在头文件中)。对于 `define` 而言, 在定义常量时, 该及其名称会被覆盖, 所以当 `debug` 时, 不易发现这些关于这些常量的错误。简而言之, 在定义常量时: 将 `define _TIME 30` 替换成 `const int _Time = 30;` 以避免上述情况。此外, 在这种情况下, 使用 `define` 也会对性能产生一定影响(可以忽略不计)。在声明常量时替换`#define`的两种情况:

1. 常量指针(*constant pointers*)

定义一个常量的 *char\*-based* 字符串: `const char* const Name = "You Wenfei"`

2. 类(*class*)的成员变量(*member*)

定义一个唯一的成员变量

```
class GamePlayer{
private:
    static const int NumTurns = 5; // 常量声明式(赋值)
```

```
static const int _NumTurns;      // 常量定义式(未赋值)
int scores[NumTurns];          // 使用该常量，此处声明的是一个静态数组
/* static const 类型的变量必须在声明时就被赋值 */
};

const int GamePlayer NumTurns;    // 此处NumTurns已被赋值
const int GamePlayer::_NumTurns = 4; // 此方法也可行
```

By the way, `#define`不能用来定义`class`成员常量，也不具备封装性，例如：`private` `#define`是不可行的。

现在，谈谈`enum`，对于`int scores[NumTurns]`利用`enum`来替代`static const`来获得同样的效果。

```
class GamePlayer{
private:
    enum { NumTurns = 5 };           // enum hack
    int scores[NumTurns];           // 与上面的效果相同
};
```

`enum hack`的特点更像`#define`而不是`const`，`enum hack`中的变量并没有地址，因此无法对这些变量进行取指针和取引用的操作，这是`const`无法实现的，`enum`和`#define`也不会导致额外的内存分配(一般而言，编译器不会为内置类型的`const`对象设定额外的存储空间，除非有一个`pointer`或`reference`指向该对象)。

关于`#define`的其他误用情况：利用`#define`实现宏(macro)，宏看起来像函数，但在使用时不会产生函数调用(function call)的开销，例：

```
/* #define a macro with parameters that call function f() */
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
void f(int a) { std::cout << a << '\n'; }
/* disadvantages list
 * 1. 无法保证传入的参数a, b相同
 * 2. 传入的a, b可能是表达式，增加了不确定性
 * 3. namespace(命名空间)污染
 * 4. 可读性差, debug麻烦
 */
```

关于#define实现macro的奇怪事情：

```
int a = 5, b = 0; // define 核算=_=
CALL_WITH_MAX(++a, b); // 输出: a: 7 b: 0 -> 累加两次
std::cout << "<-max a: " << a << ", b: " << b << '\n';
CALL_WITH_MAX(++a, b + 10); // 输出: a: 8 b: 10 -> 累加一次
std::cout << "<-max a: " << a << ", b: " << b << '\n';
```

为了避免这种情况发生，可以用*template inline*函数进行替代：

```
template<typename T> // 对任意类型作用
inline void max(const T& a, const T& b) { f(a > b ? a : b); }
void f(int a) { std::cout << a << '\n'; }
```

总结：

1. 对于**单纯常量**，用**const**和**enum**替换#define
2. 对于**类似函数形式的宏**，用**inline**函数替换#define

### 条款3：尽可能使用**const**

常见**const**与变量组合：

```
char greeting[] = "hello";
char* p = "world"; // non-const pointer, non-const data
const char* p = "greeting"; // non-const pointer, const data
char* const p = "greeting"; // const pointer, non-const data
const char* const p = "greeting"; // const pointer, const data
```

**const**的声明位置决定了它的作用：

1. **const**声明在\*之前，用来修饰**值(data)**。
2. **const**声明在\*之后，用来修饰**指针(pointer)**。

关于STL和**const**声明**iterator**为**const**等价于声明**pointer**为**const**，使**iterator**指向一个固定的对象。如果想让迭代器指向一个固定的对象，则需要的是**const\_iterator**：

```
std::vector<int> vec;
/* iter的作用: T* const */
const std::vector<int>::iterator iter = vec.begin();
*iter = 10; // 修改iter所指向的对象的值, 可行
++iter; // 修改iter所指向的对象, 不可行
/* cIter的作用: const T* */
std::vector<int>::const_iterator cIter = vec.begin();
*cIter = 10; // 修改cIter所指向的对象的值, 不可行
++cIter; // 修改cIter所指向的对象, 可行
```

## 关于函数的返回值和const

```
class Rational { ... };
const Rational operator* (const Rational& lhs, const Rational& rhs);
```

将一个函数的返回值声明为const，可以避免如下的情况发生：

```
Rational a, b, c;
if(a * b = c);
/* 除非此处重载了操作符 operator=
 * Rational operator(const Rational& a, const Rational& b)
 */
```

让`a*b`的返回值为const，一个`non-const`类型的变量无法赋值给`const`类型的变量，使`a*b = c`这个表达式无法执行。

## const型成员变量

```
class TextBlock{
public:
    const char& operator[] (std::size_t position) const {
        return text[position]; // operator for const 对象
    }
    char& operator[] (std::size_t position){
        return text[position]; // operator for non-const 对象
    }
private:
```

```
    std::string text;  
};
```

示例1：TextBlock类中针对const和non-const类型的对象进行了[]操作的重载。

```
TextBlock tb("Hello");  
std::cout << tb[0]; // 调用 non-const TextBlock::operator[]  
const TextBlock ctb("World"); // 调用 function const  
TextBlock::operator[]  
std::cout << tb[0];
```

示例2：

```
// 调用 const TextBlock::operator[]  
void print(const TextBlock& ctb){ std::cout << ctb[0]; }
```

上述代码中对operator[]进行了重载，因此它可const和non-const的TextBlock进行不同的处理：

```
std::cout << tb[0]; // 可行, 读一个non-const TextBlock  
tb[0] = 'x'; // 可行, 写一个non-const TextBlock  
std::cout << ctb[0]; // 可行, 读一个const TextBlock  
ctb[0] = 'x'; // 不可行, 写一个const TextBlock  
  
/* 因为ctb实例是const类型, 调用[]操作时, 返回的是一个const char&类型  
* 因此对一个const类型的TextBlock实例进行赋值  
* ctb[0] = 'x'; 这个操作不可行  
*/
```

注！！！：

```
/* 上面的operator[]的返回类型是reference to char */  
tb[0] = 'x'; // 这个语句才成立  
/* tb[0]本身是存储的一个地址, 故返回一个引用值(地址)是合理的 */
```

对比一下返回值为reference to char和pointer to char的区别

## 1. 返回值为reference to char

```
class TextBlock {  
public:  
    TextBlock(const std::string& s) :text(s) {}  
    const char& operator[] (std::size_t position) const {  
        return text[position + 1]; // operator for const 对象  
    }  
    char& operator[] (std::size_t position) {  
        return text[position]; // operator for non-const 对象  
    }  
private:  
    std::string text;  
};  
  
TextBlock t1("abcdef");  
const TextBlock t2("abcdef");  
std::cout << t1[1] << '\n';  
std::cout << t2[1] << '\n';
```

输出:

b

c

## 2. 返回值为pointer to char

```
class TextBlock {  
public:  
    TextBlock(const std::string& s) :text(s) {}  
    const char* operator[] (std::size_t position) const {  
        return &text[position + 1]; // operator for const 对象  
    }  
    char* operator[] (std::size_t position) {  
        return &text[position]; // operator for non-const 对象  
    }  
private:
```

```
    std::string text;  
};  
  
TextBlock t1("abcdef");  
const TextBlock t2("abcdef");  
std::cout << t1[1] << '\n';  
std::cout << t2[1] << '\n';
```

输出：

```
bcdef  
cdef
```

综上可知，返回值为*reference to char*时，返回的是字符串text中的一个字符，即*reference*仅指向text中的单一成员；返回值为*pointer to char*时，返回的是text中的一个子串，即以该指针为首的子串。

再看如下示例：

```
class CTextBlock{  
public:  
    char& operator[](std::size_t position) const{  
        return pText[position];  
    }  
private:  
    char* pText;  
};  
  
const CTextBlock cctb("Hello");  
char* pc = &cctb[0];           // 指针pc指向pText字符串的第一个字符  
*pc = 'J';                   // 对pText字符串的第一个字符进行修改  
/* pText = "Jello" */
```

此处，operator[]并不修改成员的值，但是\*pc = 'J' ;间接的修改了成员变量。故此处对operator[]声明的const没有意义。在实际编写代码的过程中也是不现实的。

mutable 与 const 无mutable

```
class CTextBlock
{
public:
    std::size_t length() const;
private:
    char* pText;
    std::size_t textLength;
    bool lengthIsValid;
};

std::size_t CTextBlock::length() const // const 限定
{
    if(!lengthIsValid)
    {
        textLength = std::strlen(pText); // 不可行
        lengthIsValid = true; // 不可行
    }
    return textLength;
}
```

被const限定的函数，无法对成员变量进行修改。

有mutable

```
class CTextBlock
{
public:
    std::size_t length() const;
private:
    char* pText;
    mutable std::size_t textLength;
    mutable bool lengthIsValid;
};

std::size_t CTextBlock::length() const // const 限定
{
    if(!lengthIsValid)
```

```
{  
    textLength = std::strlen(pText);      // 可行  
    lengthIsValid = true;                // 可行  
}  
return textLength;  
}
```

被mutable修饰的成员变量，即使在被const限定的函数内也可被修改。

常量性转除(*casting away constness*) 在某些情况下可能需要对一些被const限定的(成员)变量进行修改，此时就需要对这些变量进行常量性转除操作。示例1：

```
const int i_ = 23;  
const int * pi_ = &i_;  
  
int* non_pi_ = const_cast<int*>(pi_);    // 常量性转除  
*non_pi_ = 233;
```

示例2：

```
class TextBlock  
{  
public:  
    const char& operator[] (std::size_t position) const  
    {  
        return Text[position];  
    }  
  
    char& operator[] (std::size_t position)  
    {  
        return const_cast<char&>           // 要被常量性转除的变量类型为char&  
              (static_cast<const TextBlock&> // 将返回类型强制转换为const  
TextBlock&  
              (*this)[position]);          // 以调用 const operator[]  
    /* 输出：  
        c  
        b
```

```

/*
 * 关于此处常量性转除的另一种写法
 * return const_cast<char&>
 * (static_cast<const TextBlock&>
 * (*this).Text[position]); // 错误写法
 * 直接返回Text中索引为1处的字符
 * 输出:
 *     b
 *     b
 */

/* 错误写法
 * return const_cast<char&>
 * (static_cast<const TextBlock&>
 * Text[position]); // 另一种错误写法
 */

}

private:
    std::string Text;
};

```

在上面代码中，添加了一个`static_cast`的操作，作用是将`operator[]`转成`const operator[]`，这样可以在执行到此处时调用`const operator[]`而不是`operator[]`，即它自身，因为这样会导致它自己调用自己而产生**无限递归**。

所以，想要在`const`限定的成员函数内调用`non-const`成员函数对成员变量进行修改，就需要用到常量性转除`const_cast`方法进行实现。在`non-const`成员函数内调用被`const`限定的成员函数是正确可行的。

总结：

1. 给变量添加`const`修饰符可以让编译器帮你找出错误。
2. 编译器对于`const`使用很严格，所以在使用时也要注意。
3. 当`const`和`non-const`成员函数实现的功能相同时，在`non-const`函数中调用`const`函数可以避免代码重复。

## 条款4：确定对象被使用前已先被初始化

在C++中，变量如果只被声明而未被赋初值，可能会带来很多不必要的麻烦。即使这些未被赋初值的变量可能会被默认赋值，但不一定在所有的情况下都是这样。

```
double x; x = 3.3f;  
double x = 3.3f;  
double x; std::cin >> x;  
/* and so on... */
```

当然，这些都只是对内置类型的初始化，对自定义的类中的变量也需要进行初始化，就是类中常有的构造函数。

关于类成员的初始化的构造函数主要有如下两种：

1. 在构造函数的实现部分进行赋值

2. 利用成员初值列进行初始化

赋值：在构造函数内部进行赋值，这种操作不是初始化

```
class PhoneNumber { ... };  
class ABEntry{  
public:  
    ABEntry(const std::string& name,  
            const std::string& address,  
            const std::list<PhoneNumber>& phones);  
private:  
    std::string theName;  
    std::string theAddress;  
    std::list<PhoneNumber> thePhones;  
    int numTimesConsulted;  
};  
  
ABEntry::ABEntry(const std::string& name,  
                 const std::string& address,  
                 const std::list<PhoneNumber>& phones){  
    theName = name;
```

```
    theAddress = address;
    thePhones = phone;
    numTimesConsulted = 0;
}
```

初始化：利用成员初值列对成员变量进行初始化

```
ABEntry::ABEntry(const std::string& name,
                  const std::string& address,
                  const std::list<PhoneNumber>& phones):
    theName(name),           // 拷贝构造
    theAddress(address),     // 拷贝构造
    thePhones(phones),       // 拷贝构造
    numTimeConsulted(0){}
```

利用成员初值列对成员变量进行初始化的效率较高，此种方法调用的是default(默认)构造函数。然而利用赋值的方法对成员变量进行赋值也会调用default构造函数，但这时的default构造函数没有发挥任何作用。

无参构造：

```
ABEntry::ABEntry()
{
    theName(),           // theName的默认构造
    theAddress(),         // theAddress的默认构造
    thePhones(),          // thePhones的默认构造
    numTimeConsulted(0)// 显示声明numTimeConsulted的值为0
}
```

**注！！！**：在编写成员初值列时，总是列出所有的成员变量，以免给自己带来不必要的麻烦。对于const或reference类型的变量，它们一定得出现在成员初值列中(初始化)，而不允许被赋值。

例如：这种赋值类型的构造就不可行

```
class test
{
public:
    test(const int m, const int* n);
```

```
private:  
    const int t1;  
    const int* b;  
};  
  
test::test(const int m, const int* n)  
{  
    t1 = m;  
    b = n;  
}
```

只能用成员初值列的构造方式对成员变量进行赋值

```
class test  
{  
public:  
    test(const int m, const int* n) :t1(m), b(n) {}  
private:  
    const int t1;  
    const int* b;  
};
```

static对象：

1. global对象
2. 定义与namespace作用域内的对象
3. 在class内
4. 在函数内
5. 在file作用域内被声明为static类型的对象

这些对象通过调用自身的析构函数来自动销毁。

例如对于一个在线文件管理系统：

```

class FileSystem{
public:
    std::size_t numDisks() const;
};

extern FileSystem tfs; // declare a global instance

/* 在另一个文件定义的对象 */
class Directory{
public:
    Directory ( parameter... );
};

Directory::Directory( parameter... ){
    std::size_t disks = tfs.numDisks(); // 使用tfs对象
}

/* 在另一个文件内的对象 */
/* Directory对象的实例化 */
Directory tempDir( parameters... ); // 使用tfs对象

```

为确保上述功能执行顺利，就要在 `tfs` 被调用之前将 `tfs` 初始化，否则会给自己带来很多不必要的麻烦。

解决该方法的另一种思路，将别的文件内的对象复制到需要调用该函数的地方，并声明为 `static`，该函数返回一个指向该对象的一个 `reference`，然后调用这个函数，这样保证了所使用的对象是被初始化了的。例如：

```

class FileSystem ( ... );
FileSystem& tfs(){ // 这个函数用来替换 tfs 对象,
    static FileSystem fs; // 定义并初始化一个local static 对象
    return fs; // 返回一个 reference 指向上述对象
}
class Directory ( ... );
Directory::Directory ( parameters ){
    std::size_t disks = tfs().numDisks(); // 将原来的 reference to tfs
对象
} // 替换为现在的 tfs()
Directory& tempDir(){ // 这个函数用来替换 tempDir 对象

```

```
static Directory td;      // 定义并初始化一个local static 对象
return td;                // 返回一个 reference 指向上述对象
}
```

这种处理方式中的函数简单易用，但也有不足的地方：在多线程环境下，这种操作会带来不确定性，解决方法之一就是在单线程启动阶段(*single-threaded startup portion*)手动调用所有 *reference-returning* 函数，这种做法可以避免与初始化相关的线程竞争(*race conditions*)的情况。

但最重要的一点是，在使用 *local static* 这种解决方法的过程中，应该有一个合理的对象初始化次序，简而言之：

1. 手动初始化内置类型对象
2. 使用成员初值列对各个成员变量进行初始化
3. 仔细考虑对象初始化的先后顺序

总结：

1. 为内置类型对象手动初始化。
2. 使用成员初值列对各个内置类型成员变量进行初始化而不要函数内对各个成员变量进行赋值。
3. 若想跨文件进行初始化，使用 *local static* 替换 *non-local static* 对象这种做法最为妥当。

# 第二章 构造/析构/赋值运算

## 条款5：了解 C++ 默默编写并调用那些函数

当定义了一个没有构造函数 (constructor)、析构函数(destructor)和拷贝构造函数(copy assignment)的类，在编译期间，编译器会为这个类生成默认(default)的构造函数、析构函数和拷贝构造函数，且是public和inline。

例如：定义了一个空类

```
class Empty { };
```

在编译后就成了如下的类：

```
class Empty(){  
public:  
    Empty() { ... }           // default 构造函数  
    Empty (const Empty& rhs) { ... } // default 拷贝构造函数  
    ~Empty() { ... }          // default 析构(non-virtual)函数  
  
    Empty& operator=(const Empty& rhs) { ... } // 重载拷贝赋值操作符  
};
```

只有当这些函数被调用时它们才被编译器实现：

```
Empty e1;        // default 构造函数  
Empty e2(e1);   // 拷贝构造函数  
e2 = e1;         // 拷贝赋值函数  
                 // 析构函数
```

对于拷贝构造和赋值运算而言，编译器为此所实现的功能仅仅是将源对象的每一个non-static成员复制到目标对象。

```
template<typename T>  
class Test{  
public:
```

```
    Test(const char* val_, const T& val_t){}
    Test(const std::string& val_, const T& val_t){}
private:
    std::string val;
    T val_T;
};
```

上面的Test类中已经声明了一个或多个构造函数，因此编译器将不再为它生成default构造函数。然后Test类中只声明了构造函数，还有copy构造没有实现和copy assignment没有重载及析构函数没有实现，因此编译器会自动实现这些函数。

```
Test<int> t("ppqwqqq", 12);
Test<int> t_(t);      // 调用拷贝构造
```

在Test类的对象t(Test<int>)中成员变量有std::string类型的val和int类型的val\_T,在将t复制给t\_时，具体赋值流程如下：

- 对于val(std::string类型)，调用std::string类中的copy构造，来对t\_进行赋值
- 对于val\_t(T->int, int为内置类型)，对t中val\_T的值的每一个bit进行复制，并赋值给t\_的val\_T

编译器自己生成的copy assignment重载函数必须合法才能使用

```
template<typename T>
class T1{
public:
    // 此处的构造函数接受一个reference to non-const std::string类型的参数
    // 和const T&类型的参数
    T1(std::string& str_, const T& val_): str(str_), val(val_){}
private:
    std::string& str;
    const T val;
};

std::string text_1{"Text1"};
std::string text_2{"Text2"};
```

```
T1<int> t1(text_1, 123);
T1<int> t2(text_2, 456);
t1 = t2;
```

关于`t1=t2`, 进行如下讨论:

- 对`t1`赋值之前, `t1.str`和`t2.str`都指向各自的`string`对象
- 然后对于操作`t1.str = t2.str`而言, 它不是合法的, 因为`str`的类型为`reference to string`, 而引用绑定一个对象后, 无法再改指向其他对象。

因此编译器不会为`t1=t2`, 这种含有`reference`的成员生成`copy assignment`重载函数, 因此要实现`t1 = t2`就得需要自己实现`copy assignment`重载函数。不仅是对于`reference`类型的成员变量如此, `const`类型的成员变量也一样。对于这种情况需要自己对`copy assignment`重载函数进行实现:

```
T1& operator=(const T1& other){
    if(this != &other){
        str = other.str;
        /* illegal
         * val = other.val;
         * val is const
         */
        .....
    }
    return *this;
}
```

关于基类和子类: 如果在基类中`copy assignment`被声明为`private`, 那么编译器将不会为其子类生成`copy assignment`, 因为在子类中生成的`copy assignment`可以处理关于基类的成员, 但是又无法访问基类中的`copy assignment`。

## 条款6：若不想使用编译器自己生成的函数，就该明确拒绝

有时候, 可能不想让下面这种情况发生

```
class_ c1;
class_ c2;
class_ c3(c1);
c1 = c2;
```

对于这种拷贝构造、拷贝复制等操作没有实现，且不想让它们被编译器生成，即目标是要阻止copying这种操作被调用。首先要知道的一点是，编译器生成的函数的性质为public，若要实现上述目标可以将其声明为private，来阻止它们被调用，但这么做并不是绝对安全的，因为friend属性的成员函数依旧可以调用private函数。为了解决这个问题，可以将其声明为private属性的函数并不去实现它。

```
class class_{
private:
    class_ (const class_&);
    class_& operator=(const class_);
};
```

一般要对阻止一个类的拷贝构造和拷贝复制函数被调用时，可以让这个类继承一个base class，在这个base class可以进行如下操作：

```
class uncopyable{
protected:
    uncopyable() {}           // 允许继承此类的对象进行构造和析构
    ~uncopyable() {}

private:
    uncopyable (const uncopyable&);          // 阻止拷贝构造
    uncopyable& operator=(const uncopyable&); // 阻止拷贝赋值
};

class derived_ : private uncopyable{           // 此类不需要再声明拷贝构造函数和拷贝赋值函数
};
```

在进行上述声明后，如果进行下面的操作会被拒绝：

```
derived d1;
derived d2;
```

```
derived d3(d1);      // error  
d2 = d1;            // error
```

总结：不想让编译器自己生成的函数被调用，可以对此函数进行private声明并不予实现。

## 条款7：为多态基类声明 virtual 析构函数

在使用C++的过程中，有时可能会实现如下的操作

```
class Person{ }; // 普通人  
class Student:public Person{ }; // 学生  
class Teacher:public Person{ }; // 教师  
class Headmaster:public Person{ }; // 校长
```

现在想要实现一个函数通过基类指针获取派生类的实例，来调用派生类实例的方法，来实现其多态性

```
Person* getPerson();    // 返回一个指针，指向派生类  
// Person类动态分配对象
```

在C/C++中经由 malloc/new 实例化的类往往都被存放在堆上，这些实例的生命周期具有可控性，以至于在使用完这些实例后需要手动将其 free/delete 以避免内存泄漏：

```
Person* prs = getPerson(); // 从Person的继承体系中获取一个动态分配的对象  
...  
delete prs;                // 释放prs，避免内存泄露
```

而在这其中有一个问题，其中getPerson()返回的指针指向一个继承的类对象，然而此对象经由一个基类对象指针调用析构函数(non-virtual)而被删除。在实际执行时会出现这样的结果：基类对象的成员被删除，而继承的类对象的自身特有部分没有被删除，同时继承的类对象里的析构函数也未能执行起来，导致会有多余的成分没有被析构掉从而形成了内存泄露的结果。解决这种问题的一种方法：给基类(base class)定义一个virtual析构函数，这样在调用基类的析构函数时，同时也会调用派生类的析构函数，这样就可以删除整个对象(及所有继承及基类的成员)。

```
class Person{  
public:
```

```

Person() = defualt;
virtual ~Person() = default;
};

class Student: public Person{
    Student() = defualt;
    virtual ~Student() = default;
};

Person* getPerson(){
    return new Student();
}

Person* prs = getPerson();
delete prs;

```

一个基类(base class)将其函数定义为virtual，是为了能让其继承类(derived class)能够对此函数进行不同的实现。任何带有virtual类型函数的类其中也对应存在一个virtual析构函数。如果一个类中没有一个virtual函数，那么说明这个类不将会被作为一个基类。换言之，如果不想要一个类成为基类，在其中声明virtual函数是不适当的。此外，对于有虚函数的类，编译器会为该类创建一个虚函数表(virtual table)，每个对象会存储一个指向虚函数表的指针(virtual pointer)，虚函数表存储了该类的虚函数的地址，因此，对内存的开销也会产生额外的负担：

- vptr指针的内存开销：每个对象存储一个指向虚函数表的指针：4字节(x86), 8字节(x64)
- 虚函数表的内存开销：
  - 虚函数的数量
  - 类的层次结构：
    - 继承关系
    - 虚函数重载
    - 多重继承

总而言之，但类中有一个及以上数量的虚函数时，才为此类声明virtual类型的析构函数。

如果想定义一个抽象基类，但是没有合适的成员函数选择，此时，可以声明一个纯虚析构函数

```
class abst_class{
public:
    virtual ~abst_class() = 0;
};
```

为基类定义一个virtual析构函数主要用于多态性质的基类， 声明一个基类的对象以调用其派生类， 即通过此基类接口来处理派生类对象。 但并非所有的这样的基类(定义了virtual虚构函数的类)都是用来间接处理派生对象的， 而是纯粹作为一个抽象基类来使用， 例如： STL。

总结： 具有多态性质的基类内应该有一个virtual析构函数，在类内部具有virtual属性的成员函数的类内部也应该有virtual属性的析构函数。 如果一个类的设计初衷不是为了作为一个基类或者多态性质的基类使用， 那么就不要在其中声明virtual属性的析构函数。

## 条款8：别让异常逃离析构函数

在C++中，在析构函数中抛出异常不是一个明智之举。

```
class Widget{
    ...
~Widget() { ... } /* 此处可能抛出一个异常 */
};

void func(){
    std::vector<Widget> v;
}                      /* v 在此处被销毁 */
```

此处vector v中的Widget将被释放，若当v[0]释放时有异常抛出，在v[1]释放时又有异常抛出，在这种两个异常同时存在的情况下，将会导致程序调用 std::terminate 从而导致程序崩溃，即使使用其他标准库中的容器也会导致同样的情况，这种情况在C++中是非常糟糕的。

使用场景之一：建立数据库连接和断开数据库连接

```
class DBConnection{
public:
    ...
static DBConnection create();
/* 建立数据库连接，此方法返回一个DBConnection对象 */
```

```
void close();
/* 此处断开数据库连接，断开失败则抛出异常 */
};
```

为了避免用户忘记在DBConnection对象上调用close()方法，我们可以手动实现一个DBConnection的析构函数，在DBConnection对象的生命周期结束时，其析构函数调用close()方法。

```
/* 手动实现一个class来对DBConnection对象进行管理 */
class DBConn{
public:
    ...
~DBConn(){
    db.close();
}      /* 保证在DBConnection对象的生命周期结束后
           数据库的连接总是会断开
    */
private:
    DBConnection db;
};
{
    DBConn dbc(DBConnection::create());
    /* 建立DBConnection对象
       通过DBConn进行管理
       通过DBConne的接口
       使用DBConnection对象
       在DBConnection对象生命周期结束时
       DBConn为DBConnection调用close()
    */
}
```

此处若close()被成功调用，那么程序仍可顺利进行，如果其抛出异常，那么会导致一个棘手的问题，DBConnection对象中的其他成员或许无法被正常释放，导致资源泄露，对此问题的解决办法：

- 如果close抛出异常就结束程序，可以通过abort完成

```
DBConn::~DBConn() {
    try{ db.close(); }
    catch( ... ) {
        /* 将close()抛出异常的记录写入日志文件 */
        std::abort();
    }
}
```

当close抛出异常后，为了阻止这种异常在栈展开的过程中传播出去之前就先结束程序运行。

- 忽视因调用close而引发的异常

```
DBConn::~DBConn(){
    try { db.close(); }
    catch(...){
        /* 生成日志文件，记录调用close抛出异常的记录 */
        /* 此处不对异常进行反馈，程序依然向后运行 */
    }
}
```

以上提到的两种方法属于下策，对于这种情况另有更好的解决办法：对DBConn的设计进行调整

```
class DBConn{
public:
    ...
    void close(){
        db.close();
        closed = true; /* 记录用户是否手动关闭了数据库连接 */
    }
    ~DBConn(){
        if (!closed){ /* 如果用户没有手动关闭数据库的连接 */
            try{ db.close(); }
            catch(...){

```

```

        /* 记录下调用close时的异常信息 */
        /* std::abort(); */
        /* 此处可忽略close抛出的异常
           或者直接结束程序
        */
    }
}

private:
    DBConnection db;
    bool closed;
};

```

此处用户可以手动选择是否关闭数据库连接，如果没有手动关闭，在DBConnection对象的生命周期结束时，其会调用自身的析构函数来断开与数据库的连接。

总结：

- 析构函数绝对不要抛出异常，如果在析构函数中调用了一个函数而导致了异常，此析构函数应该捕捉异常，并阻止它们的在栈的展开的过程中继续传播或直接结束程序。
- 如果用户要对某个函数进行操作的过程中抛出异常做出回应，那么就要在拥有该成员函数的类中提供一个函数来执行该操作。

## 条款9：决不在构造和析构过程中调用 virtual 函数

引入事例：利用class的继承体系来模仿各个行业的交易操作

```

class Transaction{ /* base class */
public:
    Transaction( );
    virtual void logTransaction() const = 0;
};

Transaction::Transaction(){
    ...
    logTransaction();
}

```

```
class BuyTransaction: public Transaction{ /* derived class */
public:
    virtual void logTransaction() const;
}
```

现在将 BuyTransaction 实例化，其顺序为：

1. 先构造基类 Transaction
2. 调用基类内的 logTransaction()
3. 构造继承类 BuyTransaction

由此可见，所调用的 logTransaction 是属于基类 Transaction 的，而不是 BuyTransaction 中的 logTransaction。毕竟，继承类中的 logTransaction 很大概率上会涉及到基类中的成员变量，所以在基类被构造完成之前调用被实现基类的虚函数是一个危险的操作。

换句话说，在继承类(derived class)构造其基类(base class)的过程中，此继承类(derived class)实际为基类(base class)，即当前继承类的仅具有基类的属性。对于上述例子而言，即使是下面这种实现方式也会有一定的潜在问题：在构造继承类时调用的仍是基类中的 logTransaction()

```
class Transaction{
public:
    Transaction( ) { init(); }
    virtual void logTransaction() const = 0;
    void logTransaction() { }

    ...
private:
    void init(){
        ...
        logTransaction();
    }
};
```

解决这种的问题的方法：

- 在 Transaction 类中将 logTransaction() 改为 non-virtual，然后用成员初值列的初始化方式对基类中的成员变量进行初始化，然后继承类的构造函数就可以正常地调用 non-virtual logTransaction。

```

class Transaction{
public:
    explicit Transaction(const std::string& logInfo);
    void logTransaction(const std::string& logInfo) const;
    /* 此时 logTransaction() 是一个non-virtual函数 */
};

Transaction::Transaction(const std::string& logInfo){
    ...
    logTransaction(logInfo);
}

class buyTransaction: public Transaction{
public:
    buyTransaction( params):
        Transaction(createLogString( params )) {
            ... /* 将log信息传给基类构造函数 */
    }
private:
    static std::string createLogString( params );
};

```

简而言之，此处虚函数无法从基类中被调用至继承的类中，但是可以将继承类中需要初始化的变量向基类方向传递至基类构造函数来达到自己想要的目的。

- 避免在构造函数中调用虚函数及其解决方法：
- 使用非虚函数替代虚函数

```

class Transaction{
public:
    explicit Transaction(const std::string& logInfo){
        /* 在构造函数中调用一个非虚函数 */
        logTransaction(logInfo);
    }
    void logTransaction(const std::string& logInfo) const{

```

```

    /* Other components */
}

};

class BuyTransaction: public Transaction{
public:
    BuyTransaction(const std::string& params):
        Transaction(createLogString(params)){
        /* 此处利用基类的构造函数对需要初始化的信息进行初始化 */
        /* 以此为例，其继承类的部分构造过程如下：
         * 1. 调用static createLogString对需要初始化的参数进行传递
         * 2. 继承类在构造函数中调用基类的构造函数
         * 3. 基类在进行构造时调用自身的logTransaction函数
        */
    }
private:
    static std::string createLogString(const std::string& params){
        return "Log info for BuyTransaction" + params;
    }
};

```

- 在构造函数之后进行初始化操作

```

class Transaction{
public:
    Transaction(){}
    virtual void logTransaction() const = 0;
    /* 等待在继承类中进行实现 */

    void init(const std::string& logInfo){
        logTransaction(logInfo);
    }
};

class BuyTransaction: public Transaction{

```

```

public:
    BuyTransaction(const std::string& params): Transaction(){
        init(createLogString(params));
    }

    void logTransaction(const std::string& logInfo) const override{
        /* Other components */
    }

private:
    static std::string createLogString(const std::string& params){
        return "Log info for BuyTransaction" + params;
    }
};

```

- 使用两个阶段的构造模式

```

class Transaction {
public:
    Transaction() {
        // 基本初始化
    }

    virtual void logTransaction() const = 0;
    /* 等待在后面的继承类中实现 */

    void completeConstruction(const std::string& logInfo) {
        logTransaction(logInfo);
    }
};

class BuyTransaction : public Transaction {
public:
    BuyTransaction(const std::string& params) :
        Transaction() {
        completeConstruction(createLogString(params));
    }
};

```

```

void logTransaction(const std::string& logInfo) const override {
    // 记录 BuyTransaction 的日志
}

private:
    static std::string createLogString(const std::string& params) {
        // 创建日志字符串
        return "Log info for BuyTransaction: " + params;
    }
};

```

总结：

- 在构造和析构过程中不要调用virtual函数，因为在继承类(derived class)中无法调用基类的virtual函数（关于相对继承而言）。

## 条款10：令operator=返回一个reference to \*this

关于赋值操作符=

```

int x, y, z;
/* 关于赋值操作符=，可以写为连锁赋值的形式 */
x = y = z = 15;
/* 增强赋值操作的可读性，如下： */
x = (y = (z = 15))

```

实现连锁赋值的操作的前提是赋值操作符必须返回一个指向=左边实参的引用。对于一个自定义类而言，在对其赋值操作符=进行重载时，其返回值类型应该如下：

```

class MyWidget{
public:
    ...
    MyWidget& operator=(const MyWidget& rhs){
        ...
        return * this;
    }
}

```

```

/* 对于其他类型的函数参数也适用 */
MyWidget& operator=(int rhs){
    ...
    return * this;
}

/* 对于其他类型的赋值操作符也适用 */
MyWidget& operator+=(const MyWidget& rhs){
    ...
    return * this;
}

```

对于这样的赋值操作符重载操作在标准库中的一些容器(例如: vector, string, complex等)中也存在。

总结:

- 让赋值(operator=)操作符返回一个 reference to \*this。

## 条款11：在 operator= 中处理“自我赋值”

在代码实现的过程中，可能会出现以下这种情况——“自我赋值”：

```

class MyWidget{ };
MyWidget w;
...
w = w; /* 自己对自己进行赋值 */

/* 一种容易忽视的自我赋值情况 */
a[i] = a[j] /* 当 i == j 时,
              * a[i] = a[j]
              * 这就成了自我赋值 */

/* 另一种容易忽视的自我赋值情况 */
* px = * py;

```

对于 `* px = * py` 这种赋值情况而言，当 `px` 与 `py` 恰好指向同一个实例时，这也就成为了自我赋值。上述情况，都是基于“别名(aliasing)”所产生的结果，因为有一个及以上的方法(pointer or

reference)指向同一个对象。对于继承体系而言，只要派生的对象都继承于同一个基类，就会自然的产生自我赋值的这种情况，因此这一般是一个不可避免的问题：

```
class Base{ ... };
class Derived: public Base { ... };
void action(const Base& rb, Derived* pd);
/* 此处的 rb 和 pd 可能是同一个对象 */
```

当要自行实现资源管理时，很容易掉入“停止使用资源之前提前对此资源进行了释放”这种陷阱。例如，创建一个Widget类保存一个指向动态分配的位图(Bitmap)的指针：

```
class Bitmap { ... };
class Widget {
    ...
private:
    Bitmap* pb; /* * pb 一个指向从 heap 分配得到的对象 */
};
```

对于Widget类的中=赋值的重载实现：

```
Widget& Widget::operator=(const Widget& rhs){
    delete pb; // 删除当前的 pb 指针
    pb = new Bitmap(*rhs.pb); // 使用 rhs 的 Bitmap
    return *this;
}
/* 这种情况并不安全 */
```

这里自我赋值的情况在于，如果 operator= 重载函数中的 \*this(被赋值的对象) 和 rhs 是一个对象，那么 delete pb 不只是删除当前对象的bitmap，也会删除rhs中的bitmap，待到执行结束，\*this 会指向一个已经被删除的对象，这种情况是很危险的。

解决这种问题的方法还是在operator=中添加一个"证同条件"，达到自我赋值检验的目的：

```
Widget& Widget::operator=(const Widget& rhs){
    if(this == &rhs) return * this; // 证同条件
    /* *this 和 &rhs 指向的不是同一个对象 */
    delete pb;
    pb = new Bitmap(*rhs.pb);
```

```
    return *this;  
}
```

前一个 operator= 不仅不具备“自我赋值的安全性”，也不具备“异常安全性”。添加了证同条件的 operator= 也存在着一些问题，如果是在“new Bitmap”导致异常(不论是因为分配时内存不足或因为 Bitmap 的 copy 构造函数抛出异常)，Widget 的 pb 指针最终会指向一个被删除的 Bitmap，这种情况也很危险，对于这种情况，更加需要重视的是如何处理异常安全性，以此再改善之前的“自我赋值”：

```
/* 确保在对pb赋值时，pb所指的对象没有被删除 */  
Widget& Widget::operator=(const Widget& rhs){  
    Bitmap* pOrig = pb;           // 存下原来的 pb  
    pb = new Bitmap(*rhs.pb);    // 可能 throw exception  
    delete pOrig;                // 删除原来的 pb  
    return *this;  
}
```

如果“new Bitmap”抛出异常，pb 及其 Widget 保持原状，在没有证同条件下，依旧能处理自我赋值，因为存在一份事先存下来的 Orig 可以帮助恢复至 pb 被赋值前的状态。

如果想要提高运行效率，可以在 operator= 前加一个证同条件。但是也要知道自我赋值这种情况发生的概率大小，毕竟这个修改给函数增加了一个新的判断分支，因此同时也会影响执行速度，prefetching(预取)、caching(缓存)和 pipelining(流水线)等指令的效率都会因此降低。

**A 补充** **prefetching**: 处理器进行预取时会根据指令流进行预测。如果插入额外的条件判断，可能会改变指令流的顺序，从而影响预取结果，导致预取的数据不是立即需要的，降低预取的效率。**caching**: 额外的判断分支可能导致不同的代码路径访问不同的数据，增加缓存失效(cache miss)的可能性。当缓存失效时，处理器需要从较慢的缓存中加载数据，导致性能下降。**pipelining**: 条件判断会引入分支，如果处理器预测错误的分治，会导致流水线停顿(pipeline stall)，清空错误路径上的指令，重新加载正确的指令。这种停顿会严重影响处理器的指令吞吐量。

要确保 operator= 的“异常安全”和“自我赋值安全”的第一个替代方案是，使用 copy and swap 技术来实现：

```
class Widget{
    ...
    void swap(Widget& rhs); // 交换 *this 和 rhs 的数据
    ...
};

Widget& Widget::operator=(const Widget& rhs){
    Widget temp(rhs); // 对 rhs 进行备份为 temp
    swap(temp); // 将 *this 数据与备份数据 temp 进行交换
    return *this;
}
```

operator=另一种更高效的(但是降低了代码的可读性)写法如下:

```
Widget& Widget::operator=(Widget rhs){ // rhs 是被传递对象的一个备份
    swap(rhs); // 此处利用 pass by value
    return *this; // 将 *this 的数据与备份的数据进行互换
}
```

参考依据:

1. 一个类的 copy assignment 操作符可能被声明为"按值传递(pass by value)的方式接受实参"。
2. 通过按值传递(pass by value)的方式处理会多出一份rhs的备份数据。

总结:

- 确保当前对象在进行自我赋值时operator=有着正确处理行为，例如：
  - 比较"来源对象"和"目标对象"的地址
  - copy and swap 技术
  - 正确且具有可读性的语句
- 确保一个函数在操作多个对象时，对于相同的对象的处理仍然正确。

## 条款12：复制对象时勿忘其每一个成分

良好的面向对象系统会将对象的内部封装起来，只留两个函数负责对象拷贝，即copy构造函数和copy assignment操作符。对于以上两种函数，如果自己不主动实现，编译器会自动生成。对于自己实现的版本，当这些函数出错时，编译器不会给出任何提示信息，因此会增加debug难度。

示例：实现一个顾客类 (Customer Class)，手动实现其中的copying函数，同时实现一个logCall 记录函数的调用

```
void logCall(const std::string& funcName);
class Customer{
public:
    ...
    Customer(const Customer& rhs);
    Customer& operator=(const Customer& rhs);
    ...
private:
    std::string name;
};

/* 拷贝构造 */
Customer::Customer(const Customer& rhs): name(rhs.name) /* 复制rhs的数据 */
{
    logCall("Customer copy constructor");
}

/* =赋值运算符重载 */
Customer& Customer::operator=(const Customer& rhs){
    logCall("Customer copy assignment operator");
    name = rhs.name; // 复制rhs的数据
    return *this;
}
```

以上是对类的copy构造和copy assignment运算符的一般实现。但是对于以下情况就会有所不同：

```
class Date { ... };
class Customer {
public:
    /* 同前 */
private:
    std::string name;
    Date lastTransaction;
};
```

此处若还是使用之前的copying函数，那么执行的就是局部拷贝或部分拷贝(partial copy)，因为它们只复制了顾客的name，但没有复制新添加的lastTransaction，编译器也不会对此做出任何提示或者警告，因为这是手动实现的copying函数，为了实现对所有成员变量的copy，也必须手动修改对应copying函数。倘若这样的问题出现在一个继承体系中：

```
class PriorityCustomer: public Customer{
public:
    ...
    PriorityCustomer(const PriorityCustomer& rhs);
    PriorityCustomer& operator=(const PriorityCustomer& rhs);
    ...
private:
    int priority;
};

PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs):
    priority(rhs.priority){
    logCall("PriorityCustomer copy constructor");
}

PriorityCustomer& PriorityCustomer::
operator=(const PriorityCustomer& rhs){
    logCall("PriorityCustomer copy assignment operator");
    priority = rhs.priority;
    return *this;
}
```

上面的继承类 PriorityCustomer 的copying函数只复制了当前PriorityCustomer中的成员变量，但因为这个PriorityCustomer是继承自Customer的，即其父类(Customer)中的成员变量没有被复制，因此也导致了一个局部复制(partial copy)，而此时因为没有对父类中的成员变量进行赋值，导致父类会调用其default构造函数(无参构造函数)，default构造函数将针对name和lastTransaction执行缺省的初始化操作。

任何时候，为继承类实现copying函数，必须也要复制其父类的所有成员变量，父类中的成员变量往往是private属性的，所以继承类copying函数需要调用父类的相应copying函数对父类中的成员变量进行赋值：

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs):
    Customer(rhs), // 调用父类的构造函数，对父类成员进行赋值
    priority(rhs.priority){
    logCall("PriorityCustomer copy constructor");
}

Priority& Priority::operator=(const PriorityCustomer& rhs){
    logCall("PriorityCustomer copy assignment operator");
    Customer::operator=(rhs); // 调用父类的赋值函数，对父类成员进行赋值
    priority = rhs.priority;
    return *this;
}
```

综上可总结出的经验：

1. 复制所有local成员变量
2. 调用所有 base classes 内合适的copying函数

切记，是调用对应的copying函数，不要让 copy assignment 操作符重载函数去调用 copy 构造函数，因为构造一个已经存在的对象这样的操作是没有意义的，同样，也不要让 copy 构造函数去调用 copy assignment 操作符重载函数。因为构造函数是用来创建新的实例的，而 assignment 操作符只适用于已经创建好的实例上，对于一个没有初始化的对象使用 assignment 操作符是一个没有意义的操作。

如果在 copy 构造函数和 copy assignment 操作符中有相同的部分，建立一个新的成员函数给两者调用。这样的函数一般是private属性且名为init，这个操作可以安全消除copy 构造函数和 copy assignment 操作符之间的代码重复。

```

class PriorityCustomer: public Customer{
public:
    PriorityCustomer(const PriorityCustomer& rhs){
        init(rhs);
        logCall("PriorityCustomer copy constructor");
    }

    PriorityCustomer& operator=(const PriorityCustomer& rhs){
        if(this != rhs){
            Customer::operator=(rhs);
            init(rhs);
        }
        logCall("PriorityCustomer copy assignment operator");
        return *this;
    }

private:
    int priority;
    void init(const PriorityCustomer& rhs){
        priority = rhs.priority;
    }
};

```

总结:

- copying函数应该确保复制“对象内的所有成员变量”及“所有父类(base class)成分”。
- 在继承类中的copying函数应该调用父类中对应的copying函数。
- 将copy构造函数和copy assignment操作符中的具有相同部分功能的代码写入到一个单独的函数中，供这两者调用。

# 第三章 资源管理

使用计算机资源，最重要的一个原则就是有借有还。C++中最常用的资源就是动态分配内存，如果使用了内存后，不将其返还，那么就会导致资源泄露。内存是计算机中必须管理的众多资源之一，其他常见的资源还包括文件描述器(file descriptor)、互斥锁(mutex locks)、图形界面中的字型和笔刷、数据库连接以及网络sockets。

## 条款13：以对象管理资源

先引出示例：实现一个关于投资行为的程序库，其中各种投资类型继承自一个root class Investment。

```
class Investment { ... }; // 继承体系中的一个父类
```

这个程序库通过一个工厂函数来使用特定的Investment对象：

```
Investment* createInvestment(); // 返回指针，指向Investment继承体系内的动态  
分配对象  
// 使用完后，有必要对其进行删除
```

现使用一个函数来实现上述指针资源的回收(删除)：

```
void f(){  
    Investment* pInv = createInvestment(); // 调用 factory 函数  
    ...  
    delete pInv; // 释放pInv所指向的对象  
}
```

如果这个函数提早结束了，使之不能到达delete pInv这一句，pInv就无法被正常删除，以此导致的内存泄露是非常危险的。同样的情况在循环语句中或许因为continue和goto导致控制流无法执行到delete ...语句，导致资源无法被正常释放。

为确保资源能够被正常释放，可以将资源放进对象中，当控制流离开函数(函数执行完)，该对象的析构函数会自动释放那些资源。

**▲ 补充：计算中内存的分配和组成**

## 1. 栈(stack)

```
void f(){
    int localVariable = 42; // 分配在栈上
}
```

内存布局：栈是由一个后进先出(LIFO)结构，栈顶指针随着函数调用和局部变量的创建而增长，随着函数返回和局部变量的销毁而减少。用途：用于存储局部变量、函数参数和函数调用的返回地址 特点：

- 栈内存是由编译器自动管理的。
- 内存分配速度快，但是大小有限(通常由操作系统限制)。
- 变量的生命周期是由其作用域决定的，当离开作用域时，内存自动释放。

## 2. 堆(heap)

```
int* p = new int(42); // 分配在堆上
delete p;           // 手动释放内存
```

内存布局：堆是一个更复杂的区域，由操作系统或运行时库管理，允许自由分配和释放内存。用途：用于动态内存分配，例如通过"new"和"malloc"分配的内存 特点：

- 需要手动管理内存(分配和释放)。
- 可以分配比栈更大的内存块，但速度较慢且容易导致内存泄漏。
- 生命周期由程序员控制，直到显式释放内存(通过"delete"或"free")。

## 3. 全局/静态内存区域(Global/Static Memory Area)

```
int globalVariable = 42;           // 全局变量
void f(){
    static int staticVariable = 42; // 静态变量
}
```

内存布局：这部分内存通常在程序的最底层，由操作系统分配和管理 用途：用于存储全局变量和静态变量，这些变量在程序的整个生命周期内都存在。特点：

- 这些变量在程序启动时分配内存，在程序结束时释放。
- 生命周期贯穿整个程序运行期。

4. 代码段(code segment) 内存布局：代码段包含程序运行的机器指令，是内存中固定的一部分。用途：存储程序可执行代码。特点：

- 通常是只读的，以防止意外修改程序代码。
- 在程序加载到内存时由操作系统分配。

### 整体内存布局

高地址



许多资源被动态分配在heap内，之后被用于单一区块或函数内，它们在控制流离开了所在区块或函数时被释放。为了上面这种清醒，智能指针(auto\_ptr)就应运而生了,auto\_ptr是一个"类指针(pointer-like)对象"，其析构函数自动对其所指对象调用delete以释放资源：

```
void f(){
    std::auto_ptr<Investment> pInv(createInvestment());
    /* 调用 工厂函数(factory function)
```

```
... * 经由auto_ptr的析构函数自动删除pInv  
 */  
}
```

上面这个例子可以体现“以对象管理资源”的两个关键点：

1. 获得资源后立刻放进管理对象(managing object)内，以上代码中createInvestment返回的资源被当作其管理者 auto\_ptr 的初值。使用对象管理资源的观念常被称作RAII(Resource Acquisition Is Initialization)，即在获取资源时就进行初始化，因此所有资源在获得的同时应该被立即放入管理对象中。
2. 管理对象(managing object)运用析构函数确保资源被释放，只要控制流离开了函数，其中的对象离开了作用域，其析构函数就会被自动调用，使其资源被释放。

因为当 auto\_ptr 被释放时，auto\_ptr 指针所指向的对象也会被释放，所以要注意不要让多个 auto\_ptr 指向同一个对象，否则会导致一个对象被释放多次，使其产生未定义行为。针对此种情况，设计者为auto\_ptr添加了一个特性，就是当使用copying函数(copy 构造函数和copy assignment 操作符)对其进行复制时，复制出来的auto\_ptr会变成null，这一点就很好地保证了复制所得指针所取得的资源的唯一性。

```
std::auto_ptr<Investment> pInv1(createInvestment());  
    // pInv1指向createInvestment创建的对象  
std::auto_ptr<Investment> pInv2(pInv1);  
    // pInv2指向pInv1所指向的对象，pInv1现被设置为null  
    // 转移所有权  
pInv1 = pInv2;  
    // 现pInv1指向pInv2所指向的对象，pInv2被设置为null  
    // 转移所有权
```

上面这一过程体现出auto\_ptr对资源管理的局限性，即auto\_ptr指向资源的唯一性。

解决这种问题的一种方案是使用“引用计数智能指针(reference-counting smart pointer-RCSP)”。RCSP也是一种智能指针，其可持续追踪共有多少对象指向某笔资源，并在无指针指向某资源时，自动释放该资源，但是对于环状引用(即两个指针指向彼此，导致其仍然处在被使用的状态)，RCSP无法解决。

```
void f(){
    ...
    std::shared_ptr<Investment> pInv(createInvestment());
    ...
}
```

使用shared\_ptr的复制操作相对于auto\_ptr要正常得多：

```
void f(){
    ...
    std::shared_ptr<Investment> pInv1(createInvestment());
    // pInv1指向createInvestment()返回的对象
    std::shared_ptr<Investment> pInv2(pInv1);
    // pInv1和pInv2指向同一个对象
    pInv1 = pInv2;
    // 同上，不发生改变
    ...
    // pInv1和pInv2被释放，那么它们所指向的对象也会被释放
}
```

auto\_ptr和shared\_ptr两者都在其析构函数执行delete而不是delete[], 这就意味着不应该在动态分配的array上使用auto\_ptr和shared\_ptr，如下：

```
std::auto_ptr<std::string> aps(new std::string[10]);
std::shared_ptr<int> spi(new int[1024]);
```

在C++一般使用vector或string这样边长array来替代动态分配的数组，所以针对动态分配的数组并没有特定指针。

总结：

- 使用RAII对象来防止资源泄露，它们在构造函数中获取资源并在析构函数中释放资源。
- 常使用的两个RAII对象分别为shared\_ptr和auto\_ptr，其中shared\_ptr为更好的选择，因为其copy行为较为直观，auto\_ptr的复制行为会使其指向null。

## 条款14：在资源类管理中小心copying行为

有时候为了更好地管理资源，还是需要自己实现资源管理类。

示例：使用C API函数处理类型为Mutex的互斥器对象(mutex object)，共有lock和unlock两种函数可用：

```
void lock(Mutex* pm); // 锁定pm所指的互斥器  
void unlock(Mutex* pm); // 将互斥器解除锁定
```

为了保证每一个被锁住的互斥器Mutex解锁，可能就需要建立一个类来管理机锁。要设计这样的管理类的管理原则也可以参考RAII对象的设计，即“在获取资源的同时进行初始化，在析构期间释放”：

- 定义一个RAII对象管理mutexPtr

```
class Lock{  
public:  
    explicit Lock(Mutex* pm):  
        mutexPtr(pm){ lock(mutexPtr); } // 获取资源  
    ~Lock() { unlock(mutexPtr); } // 释放资源  
private:  
    Mutex* mutexPtr; // raw pointer  
};
```

- 使用该对象

```
Mutex m; // 定义互斥器  
...  
{ // 定义一个 critical section  
    Lock m1(&m); // 锁定互斥器  
    ...  
} // 自动解除互斥器锁定
```

- 尝试复制Lock对象

```
Lock m11(&m);          // 锁定 m  
Lock m12(m11);        // 尝试将m11复制到m12上
```

此时多个RAII对象拥有着同一个互斥器m，这导致在RAII对象释放时，会使互斥器多次释放，导致未定义行为。此时解决此问题的方案：

- 禁止复制：明确声明禁止对RAII对象的复制，可以将copying操作声明为private，或者删除copying操作：

- 将copying操作声明为private：

```
class Uncopyable{  
public:  
    ...  
protected:  
    Uncopyable();  
    ~Uncopyable();  
private:  
    Uncopyable (const Uncopyable&);  
    Uncopyable& operator(const Uncopyable&);  
};  
class Lock: private Uncopyable{  
};
```

- 删除copying操作：

```
class Lock{  
public:  
    ...  
  
    // 删除copy构造函数和copy assignment操作符  
    Lock(const Lock&) = delete;  
    Lock& operator=(const Lock&) = delete;  
  
    ...  
};
```

- 对底层资源使用"引用计数法(reference-count)"，即当一个资源没有再被使用时将其释放。这种情况下复制RAII对象，该资源的"被引用数"增加。

因为在Lock中的析构函数作用是unlock(解锁)，而不是delete(释放资源)，使用shared\_ptr的缺省行为是删除当前引用次数为0的对象，然而shared\_ptr的deleter(删除器，释放器)是一个函数或函数对象，当引用次数为0时便被调用(这种机制在auto\_ptr中不存在)，因此可以修改传给shared\_ptr的参数：

```
class Lock{
public:
    explicit Lock(Mutex* pm): mutexPtr(pm, unlock){
        // 指定 unlock 为shared_ptr的删除器
        lock(mutexPtr.get());
    }
    /* 此处的析构函数调用非static成员变量的析构函数
     * 即shared_ptr的删除器
     */
};

private:
    std::shared_ptr<Mutex> mutexPtr;
    // 使用 shared_ptr 替代 raw pointer
};
```

- 复制底部资源，当复制资源管理对象时，进行的是"深度拷贝"，即在复制管理资源对象时，也要对其包含的资源进行复制。有些字符串(一般为可变长)类型由"指向heap内存"的指针构成，这种字符串对象内包含一个指针指向一块heap内存，当这样的一个字符串对象被复制时，指针和其所指的内存段都会被复制，这就体现了字符串的深度拷贝行为。
- 转移底部资源的拥有权，某些情况下可能希望永远只有一个RAII对象指向一个未加工资源(raw resource)，即使RAII对象被复制依然如此，此时资源的拥有权会从被复制物转移到目标物，这是auto\_ptr奉行的复制意义，使其在被复制时资源的拥有权在新的RAII手中，以使资源只被一个RAII对象进行管理，避免后续多个对象尝试释放同一资源的问题。

总结：

- 复制RAII对象必须一并复制它们所管理的资源，所以资源的copying行为决定RAII对象的copying行为。

- 普遍而常见RAII class copying行为是：阻止copying、使用引用计数法等。

## 条款15：在资源管理类中提供对原始资源的访问

有些时候，需要实现多个资源管理类对象的不同资源之间的访问，此时就要在资源管理类中提供对原始资源的访问。示例：使用智能指针 auto\_ptr 或 shared\_ptr 保存 factory 函数如 createInvestment 的调用结果：

```
shared_ptr<Investment> pInv(createInvestment());
```

假如需要使用某个函数对 Investment 对象进行处理：

```
int daysHeld(const Investment* pi);
```

如果上面这个函数被这么调用：

```
int days = daysHeld(pInv);
```

很明显会导致错误，因为 daysHeld 接受的是一个 Investment\* 参数，而 pInv 的类型为 shared\_ptr<Investment>。此时就需要一个函数可将 RAII class 对象（此处为 shared\_ptr）转换为它所包含的原始资源（此处为 Investment\*）。实现这一目的的方式有两种：显式转换和隐式转换。

- 显式转换：在 shared\_ptr 和 auto\_ptr 中都提供了一个 get 成员函数，它会返回智能指针内部的原始指针：

```
int days = daysHeld(pInv.get()); // 将 pInv 内的原始指针传给 daysHeld
```

- 隐式转换：在 shared\_ptr 和 auto\_ptr 中也重载了指针解引用(pointer dereferencing)操作符 (operator-> 和 operator\*)，它们允许隐士转换至底部原始指针：

```
class Investment{ // investment 继承体系中的一个父类
public:
    bool isTaxFree() const;
    ...
};
```

```

Investement* createInvestment(); // 工厂函数
std::shared_ptr<Investment> pi1(createInvestment()); // 让
shared_ptr管理资源
bool taxable1 = !(pi1->isTaxFree()); // 通过
operator->访问资源
...
std::auto_ptr<Investment> pi2(createInvestment()); // 让auto_ptr
管理资源
bool taxable2 = ~(*pi2).isTaxFree(); // 通过
operator*访问资源
}

```

如果需要取得RAII对象内的原始资源，就需要提供一个隐式转换函数，示例如下：

```

FontHandle getFont(); // 一个无参 C API

void releaseFont(FontHandle fh); // 与上面一个来自同一个 C API

class Font{ // RAII class
public:
    explicit Font(FontHandle fh) // 获取资源
        : f(fh) // 按值传递 pass by value
    { }
    ~Font() { releaseFont(f); } // 释放资源
private:
    FontHandle f;
};

```

如果有大量与Font相关的C API，它们处理的是FontHandle，那么就需要频繁地将Font对象转换为FontHandle。Font class可以为此提供一个显式转换函数，如：

```

class Font{
public:
    ...
FontHandle get() const { return f; } // 显式转换函数

```

```
...  
};
```

这使得每次调用API时都需要使用get():

```
void changeFontSize(FontHandle f, int newSize); // C API  
Font f(getFont());  
int newFontSize();  
...  
changeFontSize(f.get(), newFontSize); // 显式地将Font转换为FontHandle
```

频繁地要求显式转换，会增加泄露字体风险的可能性，而Font class设计的初衷就是为了防止资源泄露。另一种办法就是令Font提供隐式转换函数，转型为FontHandle：

```
class Font{  
public:  
    ...  
    operator FontHandle() const // 隐式转换函数  
    { return f; }  
    ...  
};  
  
void f(FontHandle fh){  
    ... /* 若接受一个Font类型的参数，其将会被隐式转换为FontHandle */  
}
```

经过这样处理过，使得调用 C API 会比较方便：

```
Font f(getFont());  
int newFontSize;  
...  
changeFontSize(f, newFontSize); // 隐式地将Font转换为FontHandle
```

但是这个隐式转换会增加错误出现的概率：

```
Font f1(getFont());  
...  
...
```

```
FontHandle f2 = f1; // 要将Font类型的f1拷贝给f2，却将f1的类型隐式转换成了  
FontHandle
```

因为FontHandle由Font对象f1管理，但是通过隐式转换，FontHandle也可以通过f2来访问，然而，如果f1被销毁，那么f2会变成一个悬空的对象，因为f2依赖于f1的资源管理。是否提供一个显式转换或隐式转换函数将RAII class转换为其底部资源取决于RAII class被设计于执行在一个什么样的工作场景中。

总结：

- API往往要求访问原始资源(raw resources)，所以每一个RAII class应该提供一个"取得其所管理的资源"的办法。
- 对原始资源的访问可能经由显示转换或隐式转换，其中显式转换可能会比较麻烦但是较安全，隐式转换较方便但是不太安全。

## 条款16：成对使用new和delete时要采取相同形式

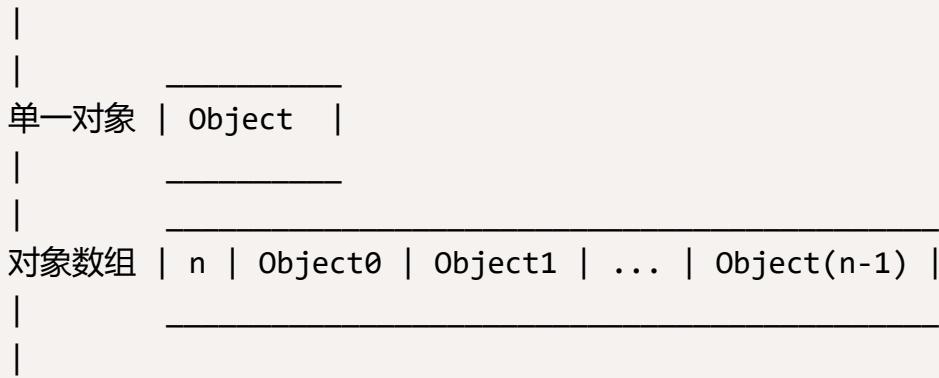
思考下面这段代码有什么问题：

```
std::string* stringArray = new std::string[100];  
...  
delete stringArray;
```

这段代码虽然成对使用了new和delete，但其仍产生了未定义行为，stringArray中包含的100个string对象中的99个string无法被正常处理，因为它们的析构函数很可能没有被调用。仔细分析一下，当使用new和delete时分别会发生什么事情：

- 当使用new动态生成对象时，会发生两件事：
  - 第一，内存通过一个名为operator new的函数被分配出来。
  - 第二，针对此内存段会有一个或多个构造函数被调用。
- 当使用delete删除对象时，会发生两件事：
  - 第一，针对此内存段会有一个或多个析构函数被调用。
  - 第二，此内存段通过一个名为operator delete的函数被释放。

这个问题在于即将被删除的内存段中有多少个对象，这决定了最终需要调用多少个析构函数。首先要考虑的问题：即将被删除的指针所指的是单一对象或对象数组，因为单一对象的内存布局一般而言不同于对象数组的内存布局，更确切的说，数组所用的内存通常还包括“数组大小”的记录，以便`delete`知道需要调用多少次析构函数，而单一对象的内存则不需要记录这点，关于单一对象和对象数组的布局可以抽象如下：



对一个对象数组使用`delete`，能够准确让`delete`知道要删除的数组大小的方法就是直接告诉它：

```
std::string* stringPtr1 = new std::string;
std::string* stringPtr2 = new std::string[100];
...
delete stringPtr1;    // 删除一个对象
delete [] stringPtr2; // 删除一个对象数组
```

对一个对象数组使用`delete[]`进行销毁和对单个对象使用`delete`进行销毁是正确的操作。如果对单个对象使用`delete[]`进行销毁，会让`delete`认为这单个对象是一个对象数组，会使其多次调用析构函数，产生一些未定义行为，所以这是不可取的。当然，如开头所说，对一个对象数组使用`delete`进行销毁，导致对象数组没有被完全销毁，进而产生未定义行为，对于一些内置类型例如int创建的数组没有使用`delete[]`那将会产生更糟的结果。显而易见，当使用`new`动态创建对象时，使用了`[]`那么在使用`delete`进行销毁时也要对应使用`[]`，否则，就不要使用。当一个有着指向动态分配内存的指针的class时，并实现了多个构造函数时，切记在所有构造函数中使用相同的`new`方式动态创建对象，否则，析构函数无法知道使用什么形式的`delete`与其对应。

在使用`typedef`时也要注意上面这点：

```
typedef std::string AddressLines[4];
/* 相当于 std::string* AddressLines = new std::string[4]; */
std::string* pal = new AddressLines;

delete pal; // 对对象数组使用delete, 产生未定义行为
delete[] pal;
```

提示：在实际使用的时候，为避免因为使用typedef导致的问题，可以使用string, vector等templates进行替代，降低出错风险的概率，

总结：

- 使用new动态创建对象时，如果创建的是单个对象，那么就应该使用delete对其进行销毁，如果创建的是多个对象，那么就应该使用delete[]对其进行销毁。

## 条款17：以独立语句将newed对象置入智能指针

假设存在一个函数用来体现程序处理的优先权，另一个函数用来在某动态分配所得的Widget上进行某些带有优先权的处理：

```
int priority();
void processWidget(std::shared_ptr<Widget> pw, int priority);
```

现在调用processWidget，一种错误的写法：

```
processWidget(new Widget, priority());
```

此处的shared\_ptr构造函数需要一个裸指针(raw pointer)，但构造函数是一个explicit构造函数，无法进行隐式转换，现将new Widget的裸指针转换为processWidget所需要的shared\_ptr：

```
processWidget(
    std::shared_ptr<Widget>(new Widget),
    priority()
);
```

这种使用方法仍然不是安全的，对于第一个实参`std::shared_ptr<Widget>(new Widget)`分析，其由两部分组成：

- 执行 `new Widget` 表达式
- 调用 `shared_ptr` 构造函数

在调用`processWidget`之前，编译器需创建代码，其会做以下三件事：

- 调用 `priority`
- 执行 `new Widget`
- 调用 `shared_ptr` 构造函数

分析一下此处编译器执行的顺序，可以确定的是 `new Widget` 会在 `shared_ptr` 构造函数调用之前执行，因为这个表达式的结果还要作为一个实参传给`shared_ptr`的构造函数，一个可能的执行顺序如下：

1. 执行 `new Widget`
2. 调用 `priority`
3. 调用 `shared_ptr` 构造函数

此时，如果`priority()`的调用出现了异常，那么 `new Widget` 返回的指针就会丢失。由于在资源被创建和资源被转换为资源管理对象这个两个过程之间难免会出现异常的干扰，为了避免这样的问题，可以使用分离语句：

1. 创建`Widget`
2. 将它置入一个智能指针内
3. 将智能指针传给`processWidget`：

```
/* 智能指针存储newed所得的对象 */
std::shared_ptr<Widget> pw(new Widget);
```

```
processWidget(pw, priority());  
/* 这样就可以规定编译器的执行的顺序 */
```

总结：

- 以独立语句将newed对象存储入智能指针内，否则，一旦异常抛出，很可能出现难以察觉的资源泄露。

# 第四章 设计与声明

Start typing here...

# STL源码分析

## 第一章 STL概述

### 可能令你困惑的C++语法

#### 1. 类里面的static成员

当使用模板类初始化多个实例时，这些实例共享这个模板（同类型）类的static成员。

```
#include <iostream>

template<typename T>
class myClass
{
public:
    static int i1_;
    static float f1_;
};

// int type template
int myClass<int>::i1_ = 1;
float myClass<int>::f1_ = 1.1f;

// float type template
int myClass<float>::i1_ = 2;
float myClass<float>::f1_ = 2.2f;

int main()
{
    myClass<int> ic1_, ic2_;
    myClass<float> fc1_, fc2_;

    std::cout
        << ic1_.i1_ << ' '
        << ic1_.f1_ << ' '
```

```

    << ic2_.i1_ << ' '
    << ic2_.f1_ << ' '
    << fc1_.i1_ << ' '
    << fc1_.f1_ << ' '
    << fc2_.i1_ << ' '
    << fc2_.f1_ << ' ';
    return 0;
}

```

综上可知，当对ic1\_实例的i1\_成员进行操作时，i2\_对应的成员也会发生变化，因为它们使用的是同一个模板类myClass<int>，同理，fc1\_和fc2\_也如此。

## 2. 模板类的特殊设计

初始化实例时，编译器会根据提供的指针的类型进行匹配。

```

#include <iostream>

// generalized
template<class T, class O>
struct tc
{
    tc() { std::cout << "I, O" << '\n'; }
};

// specialized
template<class T>
struct tc<T*, T*>
{
    tc() { std::cout << "T*, T*" << '\n'; }
};

// specialized
template<class T>
struct tc<const T*, T*>
{
    tc() { std::cout << "const T*, T*" << '\n'; }
}

```

```

};

int main()
{
    tc<int, char> obj1;
    tc<int*, int*> obj2;
    tc<const int*, int*> obj3;
    tc<const int*, const int*> obj4;
    tc<int*, const int*> obj5;
    return 0;
}

```

输出结果：

```

I, 0
T*, T*
const T*, T*
T*, T*
I, 0

```

image\_10.png

若匹配不到，则会自动选择默认模板类。

### 3. 模板类中可再存在模板成员

```

#include <iostream>

class alloc{};

template<class T, class Alloc = alloc>
class vector{                                // template class
public:
    typedef T value_type;                  // T -> value_type
    typedef value_type* iterator;          // value_type* -> iterator

    template<class I>                    // template member function
    void insert(iterator position, I first, I last){ cout <<

```

```

"insert()" << '\n'; }

};

int main(){
    int ia[5] = {0, 1, 2, 3, 4};

    vector<int> x;
    vector<int>::iterator ite = nullptr;
    x.insert(ite, ia, ia + 5);

    return 0;
}

```

#### 4. template 参数可以根据前一个 template 参数而设定默认值

```

#include <iostream>
#include <cstddef>

class alloc {};

template<class T, class Alloc = alloc, size_t BufSiz = 0>
class deque{
public:
    deque() { std::cout << "deque()" << '\n'; }
};

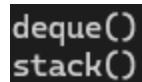
// 根据前一个参数值T， 设定下一个参数Sequence的默认值为deque<T>
template<class T, class Sequence = deque<T>>
class stack {
public: stack() { std::cout << "stack()" << '\n'; }
private: Sequence c;
};

int main() {
    stack<int> x;

```

```
    return 0;  
}
```

输出结果：



```
deque()  
stack()
```

image\_11.png

由输出结果可以看出，在初始化stack实例时，stack内的Sequence会先根据模板内的参数初始化一个deque实例，再实例化stack。

综上可知，当初始化一个模板类时，模板类中的模板中若有其他模板类，会优先初始化其他化模板类对应的实例。

示例1，如下所示：

```
#include <iostream>  
#include <cstddef>  
  
class alloc {};  
  
template<class T, class Alloc = alloc, size_t BufSiz = 0>  
class deque{  
public:  
    deque() { std::cout << "deque()" << '\n'; }  
};  
  
// 先初始化deque的实例  
template<class T, class Sequence = deque<T>>  
class stack {  
public: stack() { std::cout << "stack()" << '\n'; }  
private: Sequence c;  
};  
  
// 先初始化stack的实例  
template<class T, class Test = stack<T>>  
class test  
{
```

```

public:
    test() { std::cout << "test()" << '\n'; }
private:
    Test t;
};

int main() {
    test<int> t_;
    return 0;
}

```

输出结果：



image\_12.png

示例2，如下所示

```

#include <iostream>
#include <cstddef>

class alloc {};

template<class T, class Alloc = alloc, size_t BufSiz = 0>
class deque{
public:
    deque() { std::cout << "deque()" << '\n'; }
};

template<class T, class Test = deque<T>>
class test
{
public:
    test() { std::cout << "test()" << '\n'; }
private:
    Test t;
}

```

```

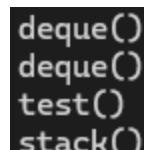
};

template<class T, class Sequence = deque<T>, class test_ = test<T>>
class stack {
public: stack() { std::cout << "stack()" << '\n'; }
private: Sequence c; test_ _t;
};

int main() {
    stack<int> t_;
    return 0;
}

```

结果如下：



deque()  
deque()  
test()  
stack()

image\_13.png

过程推导：

- 实例化deque -> deque()
- 实例化test\_，但实例化test\_前需实例化deque -> deque()
- 实例化test\_完成 -> test()
- 实例化stack完成 -> stack()

## 5. 模板类可拥有Non-type的模板参数

```

#include <iostream>
#include <cstddef>

class alloc {};

```

```

inline size_t __deque_buf_size(size_t n, size_t sz) {
    return n != 0 ? n : (sz < 512) ? size_t(512 / sz) : size_t(1);
}

template<class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator {
    typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
    typedef __deque_iterator<T, const T&, const T*, BufSiz>
const_iterator;
    static size_t buffer_size() { return __deque_buf_size(BufSiz,
sizeof(T)); }
};

template<class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public: typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
};

int main() {
    std::cout << deque<int>::iterator::buffer_size() << '\n';
    std::cout << deque<int, alloc, 64>::iterator::buffer_size() << '\n';
    return 0;
}

```

在类deque中的成员iterator初始化时利用模板中的参数BufSiz是一个value而不是一个type。  
由此可知，模板接受非类型non-type的参数。

## 6. Bound Friend Template

类模板的某个具体实例(instantiation)与其友元函数模板的具体实例(instantiation)一一对应。

```

#include <iostream>
#include <cstddef>

class alloc {};

```

```

template<class T, class Alloc = alloc, size_t BufSiz = 0>
class deque{
public: deque() { std::cout << "deque()" << '\n'; }
};

template<class T, class Sequence>
class stack;

template<class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y);

template<class T, class Sequence>
bool operator<(const stack<T, Sequence>& x, const stack<T, Sequence>& y);

template<class T, class Sequence = deque<T>>
class stack{
    // style 1
    friend bool operator== <T> (const stack<T>&, const stack<T>& );
    friend bool operator< <T> (const stack<T>&, const stack<T>& );
    // style 2
    friend bool operator== <T> (const stack&, const stack&);
    friend bool operator< <T> (const stack&, const stack&);
    // style 3
    friend bool operator== <> (const stack&, const stack&);
    friend bool operator< <> (const stack&, const stack&);
    // invalid style
    //friend bool operator== (const stack&, const stack&);
    //friend bool operator< (const stack&, const stack&);
public: stack() { std::cout << "stack()" << '\n'; }
private: Sequence c;
};

template<class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y){
    std::cout << "operator==( )" << '\t';
}

```

```

    return true;
}

template<class T, class Sequence>
bool operator< (const stack<T, Sequence>& x, const stack<T, Sequence>&
y){
    std::cout << "operator<()" << '\t';
    return true;
}

int main(){
    stack<int> x;
    stack<int> y;

    std::cout << (x == y) << '\n';
    std::cout << (x < y) << '\n';

    //stack<int> y_;
    // invalid
    // std::cout << (x == y_) < '\n';    // no match for this
    // std::cout << (x < y_) << '\n';    // no match for this

    return 0;
}

```

一般步骤：在模板类内用友元函数声明要重载的运算符，（必须）再对声明的重载友元函数进行实现。

输出结果：

deque()	
stack()	
deque()	
stack()	
operator==()	1
operator<()	1

image\_14.png

## 7. 类模板显式声明

```
#include <iostream>

#define __STL_TEMPLATE_NULL template<>

template<class Key>
struct hash{
    void operator() () { std::cout << "hash<T>" << '\n'; }
};

// explicit specialization
__STL_TEMPLATE_NULL struct hash<char>{
    void operator() () { std::cout << "hash<char>" << '\n'; }
};

__STL_TEMPLATE_NULL struct hash<unsigned char>{
    void operator() () { std::cout << "hash<unsigned char>" << '\n'; }
};

int main(){
    hash<long> t1;
    hash<char> t2;
    hash<unsigned char> t3;

    t1();
    t2();
    t3();

    return 0;
}
```

输出结果：

```
hash<T>
hash<char>
hash<unsigned char>
```

image\_15.png

hash<long> -> hash<T> ::hash<long>并没有被显式声明，故调用默认模板

hash<char> -> hash<char> ::hash<char>已经被显式声明，故调用特定模板

hash<unsigned char> -> hash<unsigned char> 原因同上

通过这样的做法，可以自定义不同类型模板类的行为。

## 8. 临时对象的产生与运用

```
#include <vector>
#include <iostream>
#include <algorithm>

template<typename T>
class print{
public: void operator() (const T& elem) { std::cout << elem << ' '; }

int main(){
    int ia[6] = {0, 1, 2, 3, 4, 5, 6};
    std::vector<int> iv(ia, ia + 5);
    // print<int> is an unnamed object but not a invoking operation
    std::for_each(iv.begin(), iv.end(), print<int>());

    return 0;
}
```

临时对象即无名对象(unnamed objects)。 实现方法

对()符号进行重载: void operator() (const T& elem) { std::cout << elem << ' '; }

不需声明指定对象名称，直接通过()进行调用std::for\_each(iv.begin(), iv.end(), print<int>());  
此临时对象的生命周期

状态	动作
开始	for_each()(或其他函数)第一次调用print<int>()(临时对象)
结束	for_each()结束时

## 9. 静态常量成员在class内部直接初始化

```
#include <iostream>

template<typename T>
class testClass{
    public: expedient
    static const int _datai = 5;
    static const long _datal = 3l;
    static const char _datac = 'c';
};

int main(){
    std::cout << testClass<int>::_datai << '\n';
    std::cout << testClass<long>::_datal << '\n';
    std::cout << testClass<char>::_datac << '\n';

    return 0;
}
```

## 10. increment/decrement/dereference 操作符

此类操作符可用于迭代器(iterator)的指针的移动或取值(dereference)。

移动操作可分为前置(prefix)和后置(postfix)。

```
#include <iostream>

class INT{
    friend std::ostream& operator<<(std::ostream& os, const INT& i);
public: INT(int i): m_i(i) {};

    // prefix: increment and then fetch
    INT& operator++(){
        ++(this->m_i);
        return *this;
    }

    // postfix: fetch and then increment
    const INT operator++(int){
        INT temp = *this;
        ++(*this);
        return temp;
    }

    // prefix: minus and then fetch
    INT& operator--(){
        --(this->m_i);
        return *this;
    }

    // postfix: fetch and then increment
    const INT operator--(int){
        INT temp = *this;
        --(*this);
        return temp;
    }

    // deference
    int& operator*() const{
        std::cout << "dereference" << '\n';
        return (int&)m_i;
    }
}
```

```

private: int m_i;
};

std::ostream& operator<<(std::ostream& os, const INT& i){
    os << '[' << i.m_i << ']';
    return os;
}

int main(){
    INT I(5);
    std::cout << I++ << '\n';
    std::cout << ++I << '\n';
    std::cout << I-- << '\n';
    std::cout << --I << '\n';
    std::cout << *I << '\n';
    return 0;
}

```

输出结果：

```

[5]
[5]
[5]
[5]
5

```

image\_16.png

分析：

- **prefix increment**：即前置`++`，类似地像`++i`这样的，优先对*i*进行操作后再使用。

```

// prefix: increment and then fetch
INT& operator++(){
    ++(this->m_i);
}

```

```
    return *this;
}
```

- **postfix increment**：即后置`++`，类似地像`i++`这样的，优先使用`i`后，再对其进行操作。

```
// postfix: fetch and then increment
const INT operator++(int){
    INT temp = *this;    // 声明一个临时对象temp用于存储要操作的目标对象,
即还没被操作的目标对象
    ++(*this);
    return temp;         // 返回没有被操作的目标对象
}
```

- **dereference**：即解引用，返回目标对象的对应类型的地址。

```
// deference
int& operator*() const{
    std::cout << "dereference" << '\n';
    return (int&)m_i;
}
```

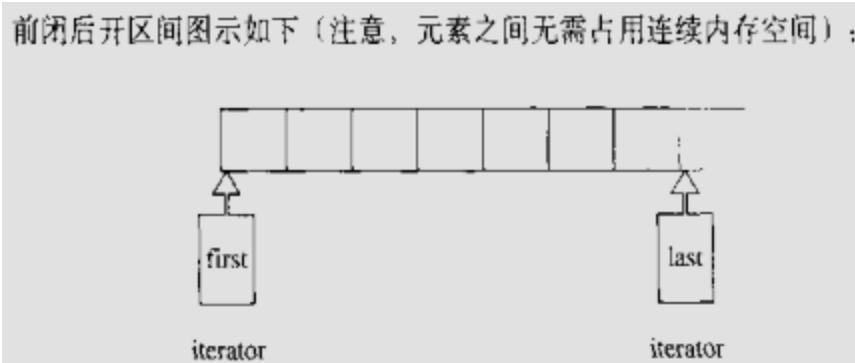
## 11. 前闭后开区间表示法 [ )

```
template<class Iterator, class T>
InputIterator find(Iterator first, Iterator last, const T& value){
    while(first != last & *first != value) ++first;
    return first;
}

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f){
    for(; first != last; ++first) f(*first)
```

```
    return f;  
}
```

前闭后开即 [first, last), 让指针从first开始 而达不到last(last的前一个位置 => last - 1)。



image\_17.png

## 12. function call操作符(operator())

```
#include <iostream>  
  
template<class T>  
struct plus{  
    T operator() (const T& x, const T& y) const { return x + y; }  
};  
  
template<class T>  
struct minus{  
    T operator() (const T& x, const T& y) const { return x - y; }  
};  
  
int main(){  
    plus<int> plusobj;          // functor object of plus  
    minus<int> minusobj;        // functor object of minus  
  
    // invoke the functor by the corresponding instances  
    std::cout << plusobj(5, 10) << '\n';  
    std::cout << minusobj(10, 5) << '\n';
```

```
// create the temporary object as the functor
std::cout << plus<int>()(30, 20) << '\n';
std::cout << minus<int>()(210, 10) << '\n';

return 0;
}
```

此处对各模板类的 () 进行了重载，使之成为了一个仿函数(functor)

## 第二章 空间配置器(allocator)

### 空间配置器的标准接口

一般接口(generalized)

```
allocator::value_type allocator::pointer allocator::const_pointer allocator::reference  
allocator::const_reference allocator::size_type allocator::difference_type
```

特殊化接口(specified)

接口	作用
allocator::rebind	旨在帮助allocator为 <b>不同类型的元素</b> 分配空间，即使allocator已被指定为一个对象分配空间。
allocator::allocator()	默认构造函数
allocator::allocator(const allocator&)	拷贝函数
template<class U>allocator::allocator(const allocator<U>&)	泛型拷贝函数
allocator::~allocator()	默认析构函数
pointer allocator::address(reference x) const	返回指定对象的地址，例如：a.address(x) <=> &x
pointer allocator::allocate(size_type n, const void* = 0)	配置空间，用来存储n个类型为T的对象，第二个参数主要用于优化内存优化和管理
allocator::deallocate(pointer p, size_type n)	释放之前所分配的内存
allocator::allocator::max_size() const	返回所成功分配的最大内存大小的值
allocator::destroy(pointer p)	等价于 p->T()

用一个例子对allocator::rebind进行补充：

```
template<class T>
class alloc;
```

```

template<typename T, typename Alloc = alloc<T>>
class MyContainer{
public:
    template<typename U>
    struct rebind{
        typedef Alloc<U> other; // Allocator type for type U
        // other represents some other objects
    };
};

```

## 2. 设计一个简单的空间配置器(Compiler:G++)

```

#include <new>           // for placement new
#include <cstddef>        // for ptrdiff_t, size_t
#include <cstdlib>         // for exit()
#include <climits>        // for UINT_MAX
#include <iostream>         // for cerr
#include <vector>          // for test

namespace T_ {
    // allocate memory
    template<class T>
    inline T* _allocate(ptrdiff_t size, T*) {
        std::set_new_handler(0);      // invoke when out of memory
        T* tmp = (T*)(::operator new((size_t)(size * sizeof(T)))); // allocate memory
        if (tmp == 0) { // the allocated memory is 0, which means that
            allocating memory fails
            std::cerr << "out of memory" << '\n';
            exit(0); // terminate processs
        }
        return tmp; // return the pointer of the allocated memory
    }
}

// release memory

```

```

template<class T>
inline void _deallocate(T* buffer) {
    ::operator delete(buffer); // release the allocated memory
}

// constructor
template<class T1, class T2>
inline void _construct(T1* p, const T2& value) {
    new(p) T1(value); // assign value at pointer p
}

// destructor
template<class T>
inline void _destroy(T* ptr) {
    ptr->~T(); // invoke the destructor of type
class
}

template<class T>
class allocator {
public:
    typedef T           value_type;
    typedef T*          pointer;
    typedef const T*    const_pointer;
    typedef T&          reference;
    typedef const T&    const_reference;
    typedef size_t       size_type;
    typedef ptrdiff_t   difference_type;

    /*-----implement part of the specific API-----*/

    // allocator::rebind->rebind allocator of type U
    template<class U>
    struct rebind {
        typedef allocator<U> other;
    };

    // pointer allocator::allocate->hint used for locality
}

```

```

pointer allocate(size_type n, const void* hint = 0) {
    return _allocate((difference_type)n, (pointer)0);
}

// allocator::deallocate
void deallocate(pointer p, size_type n) { _deallocate(p); }

// allocator::construct
void construct(pointer p, const T& value) {
    _construct(p, value);
}

// allocator::destroy
void destroy(pointer p) { _destroy(p); }

// pointer allocator::address
pointer address(reference x) { return (pointer)&x; }

// const_pointer allocator::const_address
const_pointer const_address(const_reference x) {
    return (const_pointer)&x;
}

// size_type allocator::max_size()
size_type max_size() const {
    return size_type(UINT_MAX / sizeof(T));
}
};

int main() {
    int ia[5] = { 0, 1, 2, 3, 4 };
    unsigned int i;

    std::vector<int, T_::allocator<int>> iv(ia, ia + 5);
    for (i = 0; i < iv.size(); i++) std::cout << iv[i] << ' ';
    std::cout << '\n';
}

```

```
    return 0;  
}
```

此处的STL设计只是一个简易的allocator。

## SGI空间配置器

SGI配置器与一般的allocator有所不同

	名称	语法	是否接受参数	编译器
GENERAL	allocator	vector<int, std::allocator<int>> i v	是	VC or VB
SGI	alloc	vector<int, std::alloc> iv	否	GCC OR C++

尽管SGI的alloc与标准的allocator有所不同，但是这并不影响使用，因为一般都是用缺省的方式对alloc进行声明。 SGI的每一个容器都使用缺省的空间配置器alloc，如下所示

```
template<class T, class Alloc = alloc> // 缺省使用alloc为配置器  
class vector{ ... };
```

## SGI标准的空间配置器 std::allocator

在SGI内部也有一个名为allocator的配置器，但是相比较于标准的空间配置器，它的效率较低，两者差异在于前者把C++的 ::operator new 和 ::operator delete 进行了简单的包装。以下是SGI\_style的allocator实现。注：SGI\_style的allocator只接受void\*类型的参数。

```
#include <new>  
#include <cstddef>  
#include <cstdlib>  
#include <limits>  
#include <iostream>  
#include <algorithm>  
#include <vector>
```

```

/*-----generalized_style-----*/
template<class T>
inline T* allocate(ptrdiff_t size, T*)
{
    std::set_new_handler(0);
    T* tmp = (T*)(::operator new ((size_t)(size * sizeof(T)))); 
    if(tmp == 0)
    {
        std::cerr << "out of memory" << '\n';
        exit(1);
    }
    return tmp;
}

template<class T>
inline void deallocate(T* buffer)
{
    ::operator delete(buffer);
}

/*-----SGI_style-----*/
template<class T>
class allocator
{
public:
    typedef T           value_type;
    typedef T*          pointer;
    typedef const T*    const_pointer;
    typedef T&          reference;
    typedef const T*    const_reference;
    typedef size_t       size_type;
    typedef ptrdiff_t   difference_type;

    pointer allocate(size_type n)
    {
        return ::allocate((difference_type)n, (pointer)0);
    }
}

```

```

/*----in the book----*/
/*void deallocate(pointer p) { ::deallocate(p); }
   this deallocate is not runnable */
/*----and the below "deallocate" is runnable -----*/
void deallocate(pointer p, size_type n) { ::deallocate(p); }
pointer address(reference x) { return (pointer)&x; }

const_pointer const_address(const_reference x)
{
    return (const_pointer)&x;
}

size_type init_page_size()
{
    return std::max(size_type(1), size_type(4096 / sizeof(T)));
}

size_type max_size() const
{
    return std::max(size_type(1), size_type(UINT_MAX / sizeof(T)));
}

int main() {
    int ia[5] = { 0, 1, 2, 3, 4 };
    unsigned int i;

    std::vector<int, allocator<int>> iv(ia, ia + 5);
    for (i = 0; i < iv.size(); i++) std::cout << iv[i] << ' ';
    std::cout << '\n';

    return 0;
}

```

## SGI特殊的空间配置器 std::alloc

SGI\_style的std::alloc相比较于SGI\_style的std::allocator而言，前者对基层内存配置/释放(operator::new & operator::delete)进行了效率优化。

一般在C++中关于对象对内存的配置和释放的操作

```
class Foo { ... };
Foo* pf = new Foo; // allocate memory, then construct object
delete pf;         // destruct object, and release memory
```

关于内存配置和释放的流程如下

	第一阶段	第二阶段
new	调用::operator new配置内存	调用Foo::Foo()构造对象
delete	调用Foo::~Foo()析构对象	调用::operator delete释放内存

简单来讲就是：

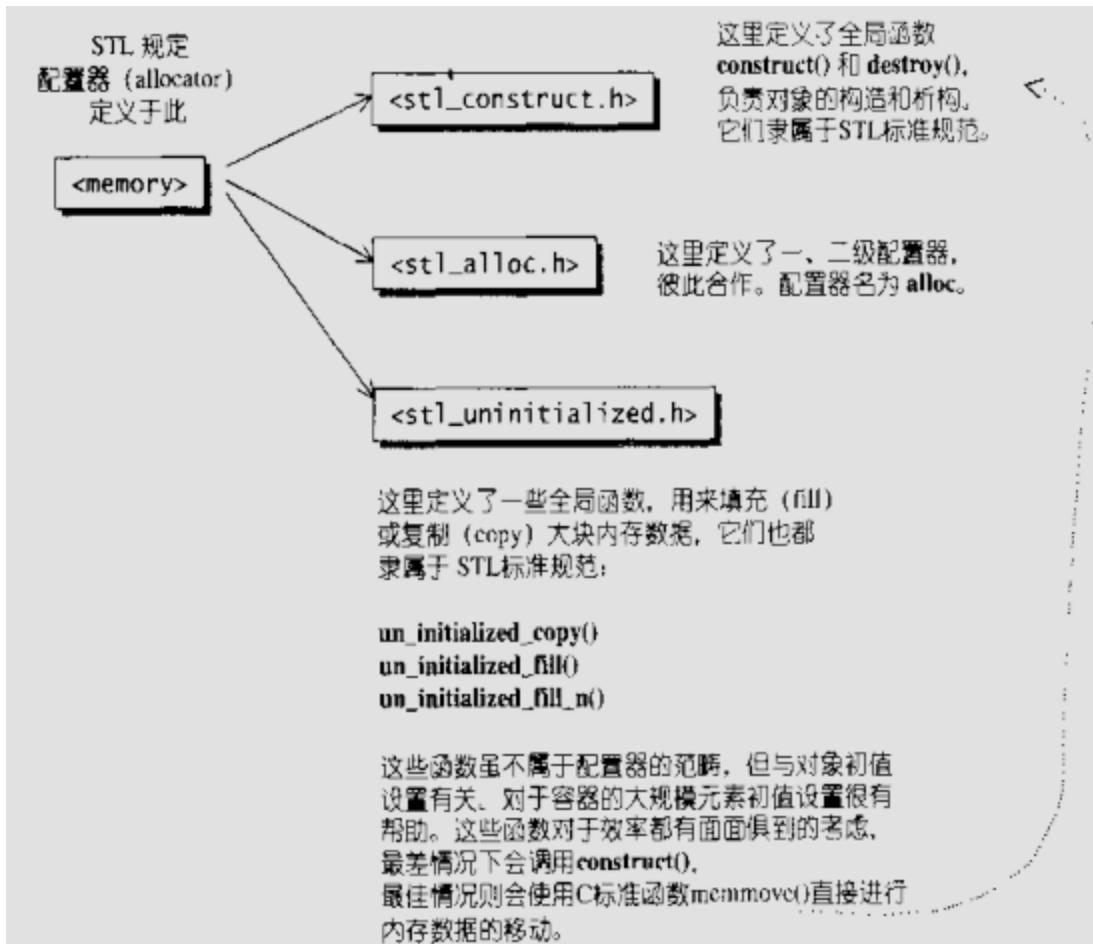
1. 造一个房子 ::operator new
2. 住进去 Foo::Foo()
3. 人走了 Foo::~Foo()
4. 再把房子给拆了 ::operator delete

在STL allocator中对上面的步骤有精细的分工：

1. alloc::allocate() 配置空间
2. alloc::deallocate() 释放空间
3. alloc::construct() 构造对象
4. alloc::destroy() 析构对象

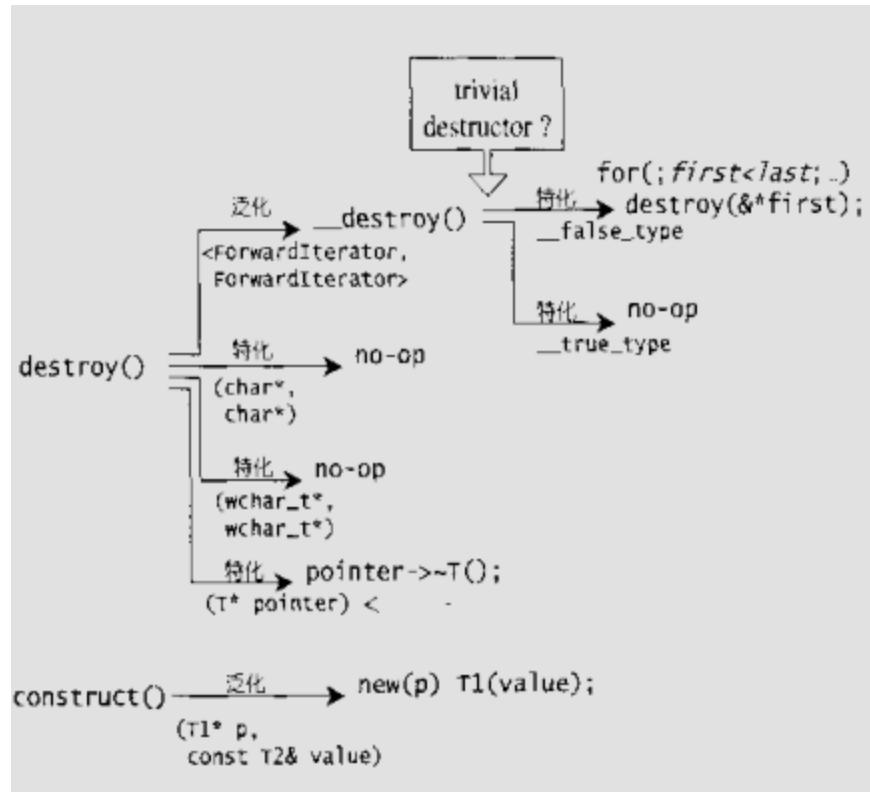
在STL标准中，allocator定义与<memory>中，SGI<memory>中包含如下两个文件

```
#include<stl_alloc.h>      // for allocating and deallocating of memory  
#include<stl_construct.h> // for constructing and destructing of object
```



image\_18.png

## 构造和析构基本工具: **construct()** & **destroy()**



image\_19.png

对上图内容的代码化:

**prefix specified**

```

#include <new>           // for placement new

template<class T1, class T2>
inline void construct(T1* p, const T2& value){
    new(p) T1(value);   // placement new -> invoke T1::T1(value);
}

```

第一版**destroy()****specified**

```

template<class T>
inline void destroy(T* pointer){
    pointer->~T();       // invoke ~T()
}

```

**pointer->~T();**一般析构。

## 第二版destroy()**generalized**

```
template<class T>
inline void destroy(ForwardIterator first, ForwardIterator last){
    __destroy(first, last, value_type(first));
}
```

`__destroy(first, last, value_type(first));`根据元素的类型选择最合适的方式进行析构。

## 第二版destroy()**SE**

- SE1(char\* ver.) -> `inline void destroy(char*, char*) {}`
- SE2(wchar\_t\* ver.) -> `inline void destroy(wchar_t*, wchar_t*) {}`

## 判断元素类型是否有**trivial destructor****generalized**

```
template<class ForwardIterator, class T>
inline void __destroy(ForwardIterator first, ForwardIterator last, T*){
    typedef typename __type_traits<T>::has_trivial_destructor
trivial_destructor;
    __destroy_aux(first, last, trivial_destructor());
}
```

- 元素类型有**non-trivial destructors****specified**

```
template<class ForwardIterator>
inline void __destroy_aux(ForwardIterator first, ForwardIterator last,
__false_type){
    for(; first < last; ++first) destroy(&*first);
}
```

- 元素类型有**trivial destructors****specified**

```
template<class ForwardIterator>
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type)
```

```
{}
```

补充和解释：

1. 用来构造、析构的函数都为全局函数(*STL规范*: 配置器必须拥有名为*construct()* 和 *destroy()* 两个成员函数)
2. 上面的*construct()*接受一个 指针 *p* 和一个 初值 *value->* 将初值设定到指针所指的位置 => *placement new()*
3. 第一版*destroy()*接受一个指针，用于析构掉指针所指的对象。第二版*destroy()*接受 *first* 和 *last* 两个迭代器，用于析构掉 \*[*first*, *last*)\* 范围内的所有对象
4. 在对象析构之前需要考虑该对象所有的析构函数是否 *trivial-> value\_type()* (获取对象类型) -> *\_type\_traits<T>* (判断该类型对象的析构函数是否 *trivial*)
  - 4.1 是(*\_true\_type*)，直接忽略
  - 4.2 否(*\_false\_type*)，对范围内的对象逐个进行*destroy()* (第一版)

## 空间的配置与释放 std::alloc

一般地，在对象构造前，要对其进行空间的配置，在对象析构后，要对配置的内存进行释放。在SGI中，上述过程实现流程如下：

- 向 *system heap* 申请空间
- 考虑多线程(*multi-threads*)状态
- 考虑**内存不足**时的应变措施
- 考虑内存分配时可能导致的**内存碎片**问题

**内存碎片**：不同的对象构造需要申请不同的大小的内存空间，但可能由于分配得不够合理（有大有小），导致会有一些较小的内存区块（这部分内存区块无法满足目标对象构造的内存需求）残留，随着时间的推移这些小的内存区块会越来越多，即使总的剩余内存足够大，但是已无法为大型的对象分配空间，因此造成内存浪费。

内存碎片可能会导致的问题：

1. 碎片化的可用空间

2. 内存使用效率低下

3. 分配失败的概率增加

4. 性能下降

内存碎片问题的可能解决方法：

1. 使用固定大小的内存池：预分配固定大小的内存区块

2. 自定义内存配置器：根据实际需求进行内存分配

3. 内存压缩：手动对内存碎片进行清理

4. 智能内存管理：RAII和智能指针

5. 限制动态内存分配：对小型、生命周期短的对象使用基于堆栈的内存分配，对大型、生命周期长的对象使用动态内存分配

在C++中，对内存进行配置的操作是`::operator new()`，对内存进行释放的操作是`::operator delete()`。可类比C中的`malloc()`和`free()`函数。同时，在SGI中也是通过`malloc()`和`free()`两个函数来实现对内存的配置和释放。

考虑到上面提到的内存碎片问题，SGI设计了双层级的内存配置器：

```
#ifdef  
typedef __malloc_alloc_template<0> malloc_alloc;  
typedef malloc_alloc alloc;      // set alloc as primary allocator  
#else  
// set alloc as secondary allocator  
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;  
#endif
```

- 第一级配置器直接使用`malloc()`和`free()`

- 第二级配置器则根据情况采用不同的分配策略：

- 当需要配置的内存区块**大于128bytes**时，使用一级配置器

- 当需要配置的内存区块**小于128bytes**时，使用二级配置器 -> *memory pool* (内存池)

SGI会为双层及的配置器封装一个如下的接口（为了使接口符合STL的标准）：

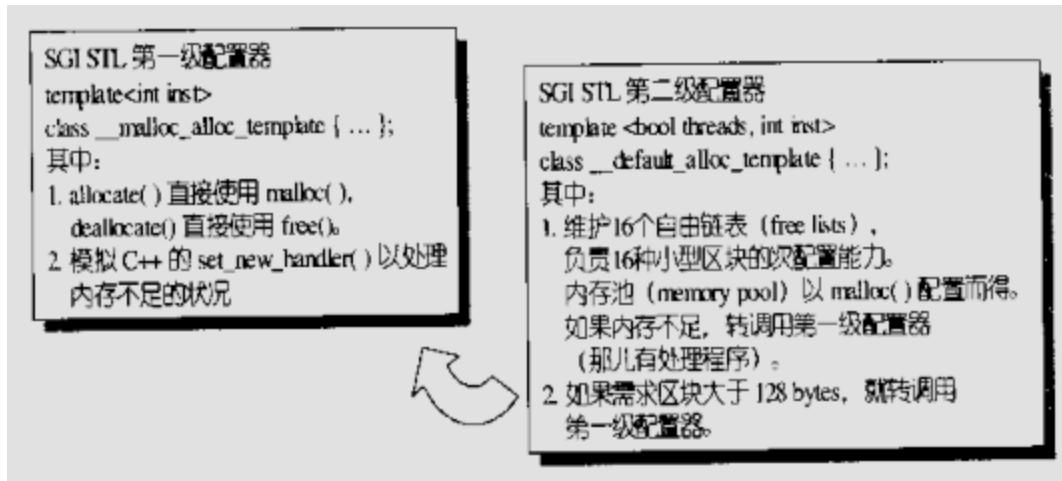
```
template<class T, class Alloc>
class simple_alloc{
public:
    static T* allocate(size_t n){
        return 0 == n ? 0 : (T*) Alloc::allocate(n * sizeof(T));
    }
    static T* allocate(void){
        return (T*) Alloc::allocate(sizeof(T));
    }
    static void deallocate(T* p, size_t n){
        if(0 != n) Alloc::deallocate(p, n * sizeof(T));
    }
    static void deallocate(T* p){
        Alloc::deallocate(p, sizeof(T));
    }
};
```

综上可见，该接口内的成员函数都是调用配置器内的成员函数，其中的配置单位为目标元素的大小 `sizeof(T)`。

SGI的STL容器全都使用 `simple_alloc` 这个接口，如：

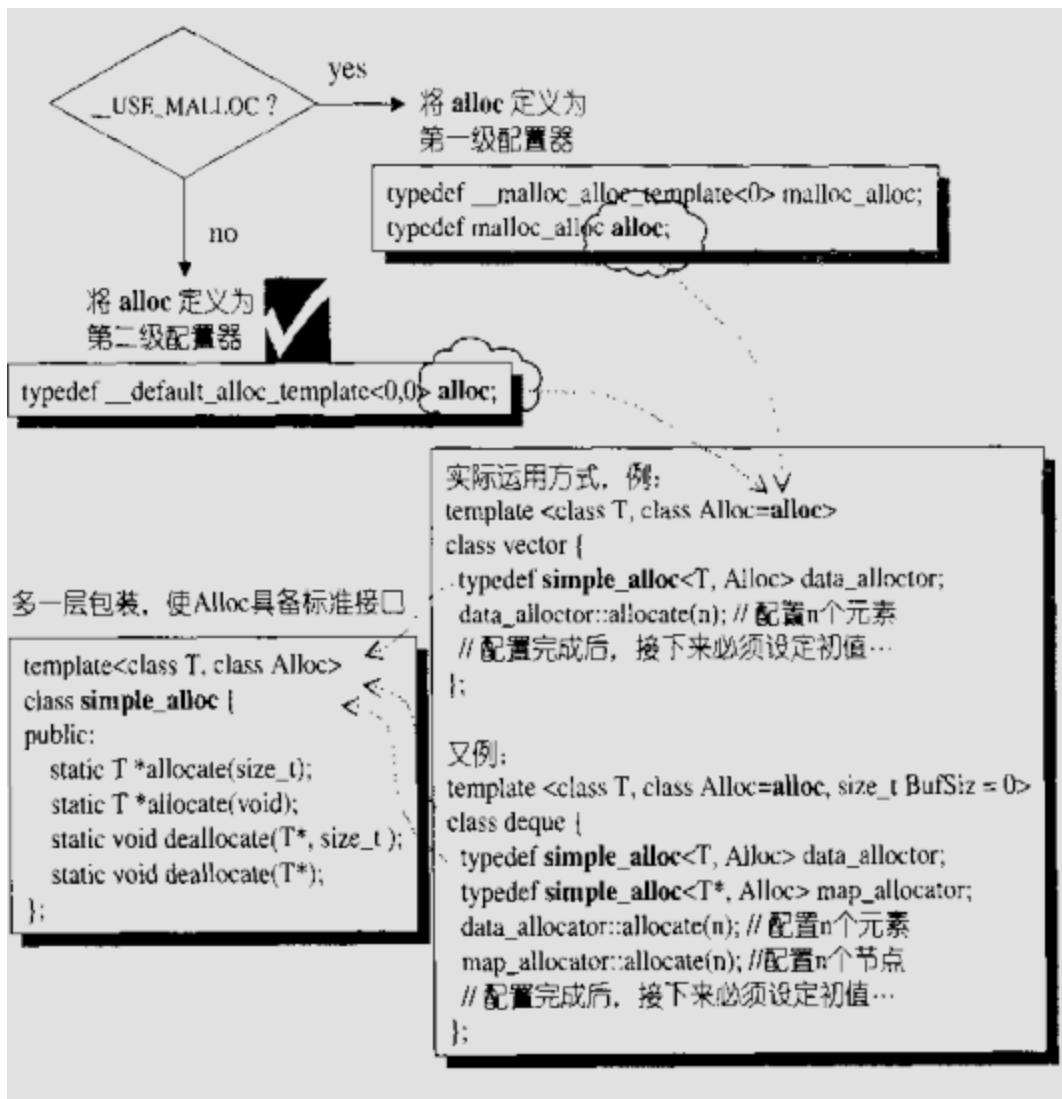
```
template<class T, class Alloc = alloc> // default
class vector{
protected:
    // specified allocator that allocates a size of element per time
    typedef simple_alloc<value_type, Alloc> data_allocator;
    void deallocate(){
        if(...) data_allocator(start, end_of_storage - start);
    }
    ...
};
```

第一级配置器与第二级配置器关系图



image\_20.png

第一级配置器与第二级配置器的包装接口和运用方式关系图



image\_21.png

## 第一级配置器 \_\_malloc\_alloc\_template 剖析

```

#ifndef _STL_ALLOCATOR_H
#define _STL_ALLOCATOR_H

#if 0
#   include <new>
#   define THROW_BAD_ALLOC throw bad_alloc
#elif #defined(__THROW_BAD_ALLOC)
#   include <iostream>
#   define __THROW_BAD_ALLOC cerr << "out of memory" << endl; exit(1);
#endif

```

特点：

- malloc-base allocator 比 default allocator速度慢
- 一般而言是thread-safe，内存空间利用率较高

注：

### 1. 无参模板用法

`template<>`中定义内置类型与变量名，在实例化时直接给该变量赋值。

```
template<int inst>
class ex{
public:
    void printf() { std::cout << "inst: " << inst; }
};

int main(){
    /* ex<para> e1; here para = inst */
    ex<3> e1;
    e1.printf();
    return 0;
}
```

### 2. static修饰符用法

	<b>static</b>	<b>non-static</b>
with instance	accessible	accessible
without instance	accessible	<b>inaccessible</b>
<code>*this</code>	<b>inaccessible</b>	accessible

```
#include <iostream>
```

```

class MyClass{
public:
    static void staticFunction(){
        std::cout << "Inside staticFunction" << '\n';
    }

    void nonstaticFunction(){
        std::cout << "Inside nonstaticFunction" << '\n';
    }
};

int main(){
    /* call static function without an instance */
    MyClass::staticFunction();

    /* class static function from an instance (this is uncommon but
valid) */
    MyClass obj;
    obj.staticFunction();

    /* cannot call non-static function without an instance */
    MyClass::nonStaticFunction(); /* Error cannot call non-static
function
                                without instance */

    /* call non-static function from an instance */
    obj.nonstaticFunction();

    return 0;
}

```

### 3. 函数指针用法

```

#include <iostream>

int add(int a, int b);

```

```

int subtract(int a, int b);

int main(){
    int(*operation)(int, int); /* declaration of function pointer */

    operation = add; /* assign address of add function to operation */
/*
    printf("Addition result: %d\n", (*operation)(5, 3)); /* call
function through function pointer */

    operation = subtract; /* assign address of subtract function to
operation */
    printf("Subtract result: %d\n", (*operation)(5, 3)); /* call
function through function pointer */

    return 0;
}

int add(int a, int b){ return a + b; }
int subtract(int a, int b) { return a - b; }

```

## 一级配置器

```

template<int inst>
class __malloc_alloc_template {
private:
    /* the below are the function
       pointers which are used for cope with the out-of-memory(oom) */
    static void *oom_malloc(size_t);

    static void *oom_realloc(void *, size_t);

    static void (* __malloc_alloc_oom_handler)();
public:
    /* allocator */
    static void *allocate(size_t n) {

```

```

    void *result = malloc(n); // primary allocator employs malloc()
directly
    /* when out of memory, invoke oom_malloc() */
    if (0 == result) result = oom_malloc(n);
    return result;
}

/* deallocator */
static void deallocate(void *p, size_t /* n */) {
    free(p); /* primary allocator employs free() directly */
}

/* reallocator */
static void *reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
    void *result = realloc(p, new_sz);
    /* primary allocator employs realloc() directly
when out of memory, invoke oom_realloc()*/
    if (0 == result) result = oom_realloc(p, new_sz);
    return result;
}

/* emulate set_new_handler(), which means that
out-of-memory handler is definable */
static void (*set_malloc_handler(void (*f)()))() {
    void (* old)() = __malloc_alloc_oom_handler;
    __malloc_alloc_oom_handler = f;
    return (old);
}
};

/* malloc_alloc out-of-memory banding */
template<int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() =
0;

template<int inst>
void* __malloc_alloc_template<inst>::oom_malloc(size_t n){

```

```

void (* my_malloc_handler)();
void* result;

for(;;){ /* endlessly allocate and deallocate until allocate
successfully */
    my_malloc_handler = __malloc_alloc_oom_handler;
    if(0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
    (* my_malloc_handler)();
    result = malloc(n);
    if(result) return result;
}
}

```

注： `typedef __malloc_alloc_template<0> malloc_alloc;` 此处 `inst` 的值被指定为 0

一级配置器主要采用 C 中的 `malloc()`, `free()`, `realloc()` 的函数实现内存的 配置、释放、重配置操作，并实现仿 C++ 的 `new-handler` 机制(因为此处并没有直接使用 `::operator new` 来配置内存)

在 C++ 中 `new handler` 的实现机制：当系统无法满足内存配置需求时，可调用一个自己指定的函数。换言之，当 `operator::new` 无法完成内存配置的操作时，在抛出异常前，会调用一个自己指定的函数 `new-handler` 来妥协解决内存不足的问题。

注：

1. SGI 中用 `malloc` 而不是 `::operator new` 来进行内存配置，因此需要实现一个类似 C++ 中 `set_new_handler()` 的功能 `set_malloc_handler()`。
2. SGI 的一级配置器的 `allocate()` 和 `realloc()` 都是在调用 `malloc()` 和 `realloc()` 不成功(发生内存不足等情况) 后才调用 `oom_malloc()` 和 `oom_realloc()`，然后一直循环尝试配置和释放内存，直到内存配置成功为止，若内存配置始终不成功那么 `oom_malloc()` 和 `oom_realloc()` 就会抛出 `_THROW_BAD_ALLOC` 的 `bad_alloc` 异常信息，或直接 `exit(1)` 中止程序。
3. 内存不足的问题的处理函数由程序的设计者完成。

## 第二级配置器 `__default_alloc_template` 剖析

二级配置器相较于一级配置器多了一些额外的机制，减少了小型内存分配时所带来的内存浪费(内存碎片)，且分配的内存越小为后续带来的内存空间管理的难度就越大，因为系统总要靠

这多出来的空间进行内存管理。



image\_22.png

SGI二级配置器的内存分配机制：

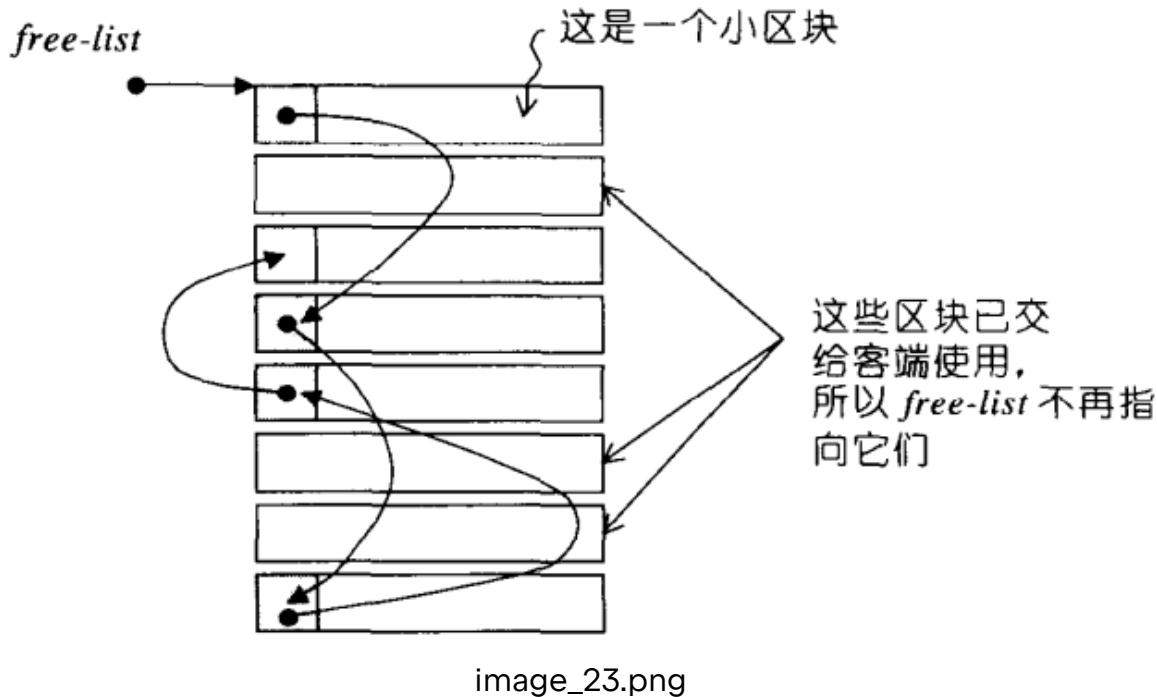
- 当分配的内存区块足够大时(大于128bytes)，就转交给一级配置器处理。
- 当分配的内存区块较小时(小于128bytes)，就通过内存池(memory pool)进行管理——层级管理法(sub-allocation)

**▲ 层级管理法** 每次配置一大块内存，并维护对应的自由链表(free-list)。下次再有相同大小的内存需求，直接从free-lists中进行分配，如果程序释放了小型内存区块，那么就由配置器回收到free-lists中。

为了方便管理，SGI的二级配置器会主动将任何小额的内存需求量上调至 8 的倍数，并维护16个 free-lists，各自管理大小为 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 bytes 的小额内存区块。

free-lists 结构如下：

```
union obj{
    union obj* free_list_link;
    char client_data[1];
};
```



image\_23.png

此处，`union obj* free_list_link` 可类比为链表中的 `LinkList* next`，`char client_data[1]` 则用指向实际的内存区块，此种做法可以节省内存的开销。

补充：

- `struct` 中的各个成员变量都有自己的内存空间
- `union` 中的各个成员变量共享着一块内存空间

二级配置器的部分实现

```
#include <cstddef>

enum {__ALIGN = 8};
enum {__MAX_BYTES = 128};
enum {__NFREELISTS = __MAX_BYTES / __ALIGN };

template<bool threads, int inst>
class __default_alloc_template{
private:
    static size_t ROUND_UP(size_t bytes){ return (((bytes) + __ALIGN - 1) & ~(__ALIGN - 1)); }
```

```

    /* '(bytes + __ALIGN - 1)': This part adds an offset to the 'bytes'
value. The '__ALIGN - 1'
     * is used to ensure that we round up to the next multiple of
'__ALIGN'. For example, if '__ALIGN'
     * is 4, then '__ALIGN - 1' is 3, and adding 3 ensures that the
result will be at least a multiple
     * of 4 greater than 'bytes'.
     * '& !(__ALIGN - 1)': This part performs a bitwise AND operation
with the complement of '(__ALIGN - 4)'.
     * The complement operation '~' flips all the bits of '(__ALIGN -
1)', effectively creating a bitmask where
     * all bits are set to 1 except for the lower bits determined by
'__ALIGN - 1'. By performing a bitwise AND
     * with this bitmask, we effectively set the lower bits to 0, thus
rounding down the result to the nearest
     * multiple of '__ALIGN'.
*/
private:
union obj{
    union obj* free_list_link;
    char client_data[1];
};
private:
/* 16 free-lists */
static obj* volatile free_list[__NFREELISTS];
/* employ Nth free-list according to the size
 * of memory chunk (begin with 1st memory chunk)
*/
static size_t FREELIST_INDEX(size_t bytes){ return (((bytes) +
__ALIGN + 1) / __ALIGN - 1); }
/* return a object in size of n, and add other memory
 * chunks in size of n into free-list if possible
*/
static void* refill(size_t n);
/* allocate a memory space that can hold n objs
 * its capacity is size
 * if inconvenient to allocate for n objs, n could decrease
*/

```

```

    static void* refill(size_t size, int& nobjs);

    // chunk allocation state
    static char* start_free;      // the start position in memory pool,
only changes in chunk_alloc()
    static char* end_free;        // the end position in memory pool, only
changes in chunk_alloc()
    static size_t heap_size;
public:
    static void* allocate(size_t n) { }
    static void deallocate(void* p, size_t n) { }
    static void* realloc(void* p, size_t old_sz, size_t new_sz);
};

/* the original value and definition setting for static data member */
template <bool threads, int inst>
char* __default_alloc_template<threads, inst>::start_free = 0;

template <bool threads, int inst>
char* __default_alloc_template<threads, inst>::end_free = 0;

template <bool threads, int inst>
size_t __default_alloc_template<threads, inst>::heap_size = 0;

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj* volatile
__default_alloc_template<threads, inst>::free_list[___NFREELISTS] =
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

```

## 空间配置函数 allocate()

配置器的标准接口函数allocate(), 其工作方式如下:

- 判断区块大小
  - 大于128bytes调用一级配置器
  - 小于128bytes就检查对应的free list

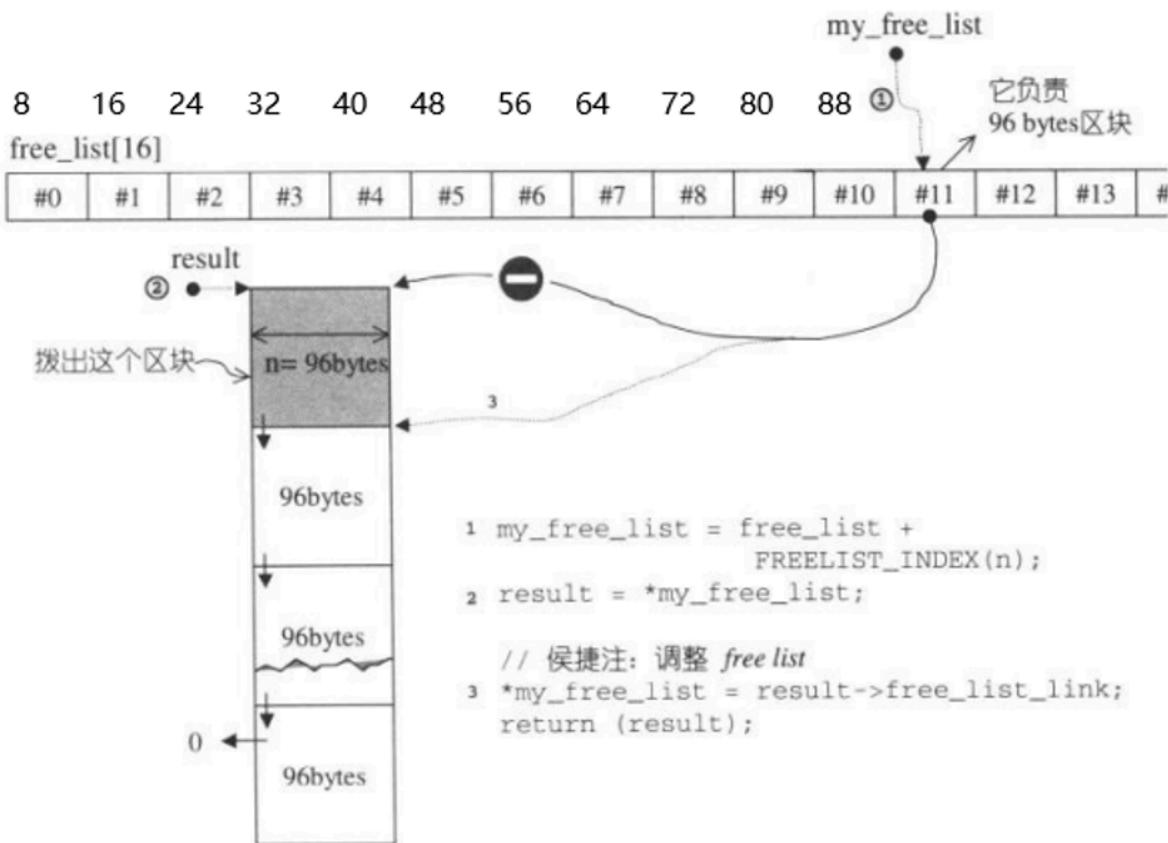
- 如果free list内有可用的区块，就直接用
- 如果free list内没有可用的区块，就将区块大小上调至8倍数bytes，然后调用refill()，准备为free list重新填充空间

```
// n must be > 0, to assure that
// there is the memory allocated
static void* allocate(size_t n){
    obj* volatile* my_free_list;
    obj* result;

    // if allocating memory > 128 bytes
    if(n > (size_t) __MAX_BYTES){
        return (malloc_alloc:allocate(n));
    }

    // select a suitable chunk in 1 of 16 free lists
    my_free_list = free_list + FREELIST_INDEX(n);
    result = *my_free_list;
    if (result == 0){
        // no find a available free list
        // preparing to refill another
        void* r = refill(ROUND_UP(n));
        return r;
    }
    // adjust free list
    *my_free_list = result -> free_list_link;
    return (result);
};
```

## 内存区块从free list调出的操作



image\_25.png

## 空间释放函数 deallocate()

\_default\_alloc\_template拥有一个标准接口函数deallocate(), 其工作方式如下:

- 判断区块大小
  - 大于128bytes, 直接调用一级配置器
  - 小于128bytes, 找出对应的free list, 将区块回收

```

// p can not be 0
static void deallocate(void* p, size_t n){
    obj* q = (obj *)p;
    obj* volatile* my_free_list;

    // if size > 128 bytes, call the primary allocator
    if(n > (size_t) __MAX_BYTES){
        malloc_alloc::deallocate(p, n);
    }
}

```

```

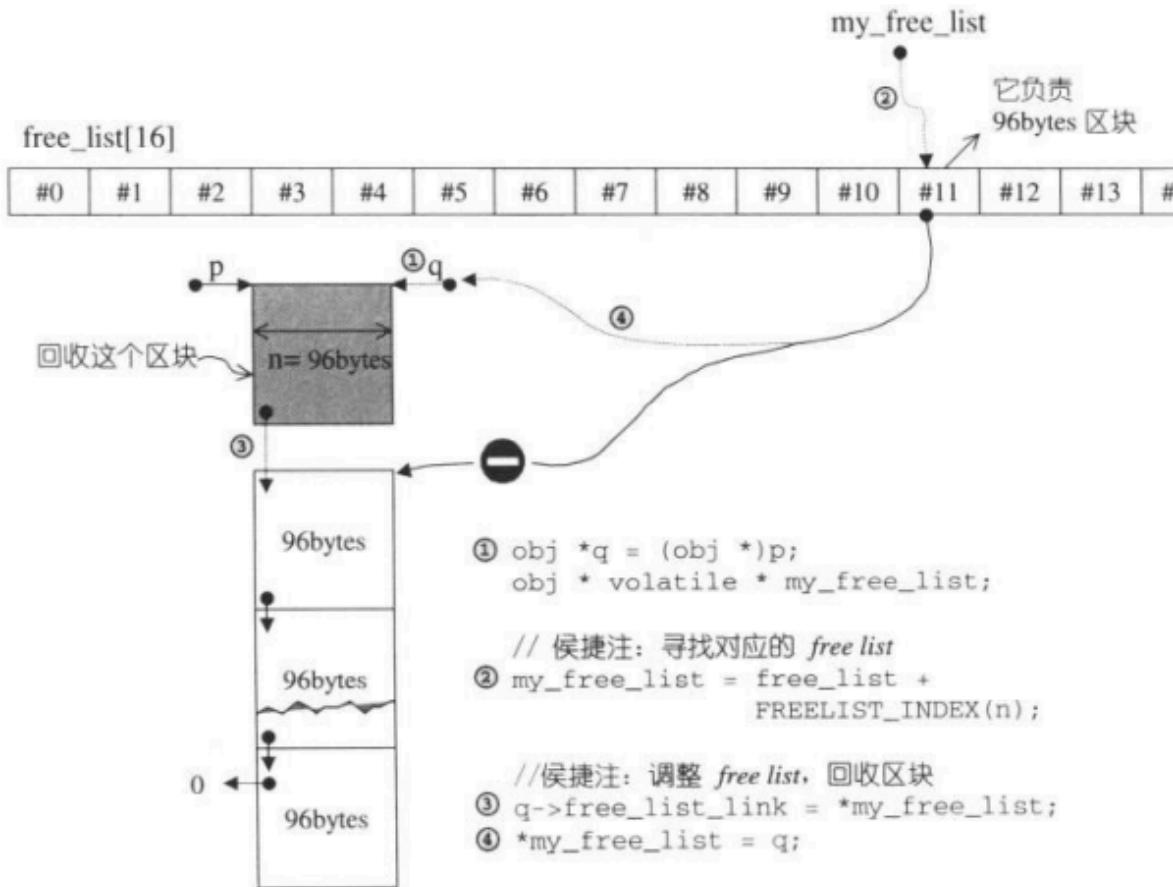
    return;
}

// selecting the corresponding free list
my_free_list = free_list + FREELIST_INDEX(n);

// adjust free list, recycle the chunk
q -> free_list_link = *my_free_list;
*my_free_list = q;
}

```

## 内存区块回收至free list的操作



image\_26.png

## 重新填充free lists

在free lists中没有可用的区块时，调用refill()以为free list重新填充空间，新的空间从内存池

(经由chunk\_alloc()完成)取出：

- 当内存池中剩余的内存足够时，缺省取得20个新区块
- 当内存池中剩余的内存不足时，取得的新区块数可能小于20

```
// return a obj in size of n
// and supply some chunks for free list properly sometimes
// supposing that n is upregulated to a multiple of 8
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n){
    int nobjs = 20;
    // call chunk_alloc(), try to get n(objs) chunks
    // as new nodes for free list
    char* chunk = chunk_alloc(n, nobjs); // nobjs -> pass by reference
    obj* volatile* my_free_list;
    obj* result;
    obj* current_obj, * next_obj;
    int i;

    // if only get one chunk, then allocate this chunk to the target
    // so there is no new node for free list
    if(1 == nobjs) return(chunk);
    // if not, adjust free list and prepare to link a new node
    my_free_list = free_list + FREELIST_INDEX(n);

    // create free list in chunk space
    result = (obj*)chunk; // this piece of chunk will return to user
    // help free list point toward the new allocated space (from memory
    pool)
    *my_free_list = new_obj = (obj*)(chunk + n);
    // connect the each node in free list
    for(i = 1; ; i++){ // i begin with 1, because i[0] will be
    return to user
        current_obj = next_obj;
        if (nobjs - 1 == i){
            current_obj -> free_list_link = 0;
```

```

        break;
    }else{
        current_obj -> free_list_link = next_obj;
    }
}
return(result);
}

```

## 内存池 memory pool

在谈到refill()时，提到了关于chunk\_alloc()这个函数，它是用来从内存池中取空间给free list。

```

template <bool threads, int inst>
char* __default_alloc_template<threads, inst>::chunk_alloc(size_t size,
int& nobjs){
    char* result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free; // the rest of memory
pool

    if(bytes_left >= total_bytes){
        // the rest of memory pool could satisfy the need
        result = start_free;
        start_free += total_bytes;
        return(result);
    }else if( bytes_left >= size){
        // the rest of memory pool could not totally satisfy the need
        // but only one or more chunks
        nobjs = bytes_left / size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    }else{
        // memory pool can not supply the size o only one chunk
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >>

```

```

4);
    // make full use of memory fragments
    if(bytes_left > 0){
        // allocate some surplus memory for free list primarily
        // select the proper free list
        obj* volatile* my_free_list = free_list +
FREELIST_INDEX(bytes_left);
        // adjust free list, integrate the surplus memory in memory
pool
        ((obj*) start_free) -> free_list_link = *my_free_list;
        *my_free_list = (obj*) start_free;
    }

    // configure heap to supply memory pool
    start_free = (char*) malloc(bytes_to_get);
    if(0 == start_free){
        // heap is not enough, malloc() fails
        int i;
        obj* volatile* my_free_list, *p;
        // check the memory that I currently have
        // allocate the small memory space is dangerous under multi-
threads circumstance
        // select a proper free list
        // "proper" means that the unused memory is big enough to
satisfy the need
        for(i = size; i <= __MAX_BYTES; i += __ALIGN){
            my_free_list = free_list + FREELIST_INDEX(i);
            p = *my_free_list;
            if(0 != p){
                // chunk not in free list
                // adjust free list to release the unused memory
                *my_free_list = p -> free_list_link;
                start_free = (char*) p;
                end_free = start_free + i;
                // revoke itself to fix nobjs
                return(chunk_alloc(size, nobjs));
                // any surplus memory will be integrated in free
list as spare
            }
        }
    }
}

```

```

        }
    }

    end_free = 0; // if occur the accident, allocate the
primary allocator
    // turning to out-of-memory
    start_free = (char*)malloc_alloc::allocate(bytes_to_get);
    // if failed, throw exception
    // if succeeded, memory lack will be fixed
}
heap_size += bytes_to_get;
end_free = start_free + bytes_to_get;
// revoke itself to fix nobjs
return(chunk_alloc(size, nobjs));
}

```

`chunk_alloc()`函数通过`end_free - start_free`来判断内存中的内存余量：

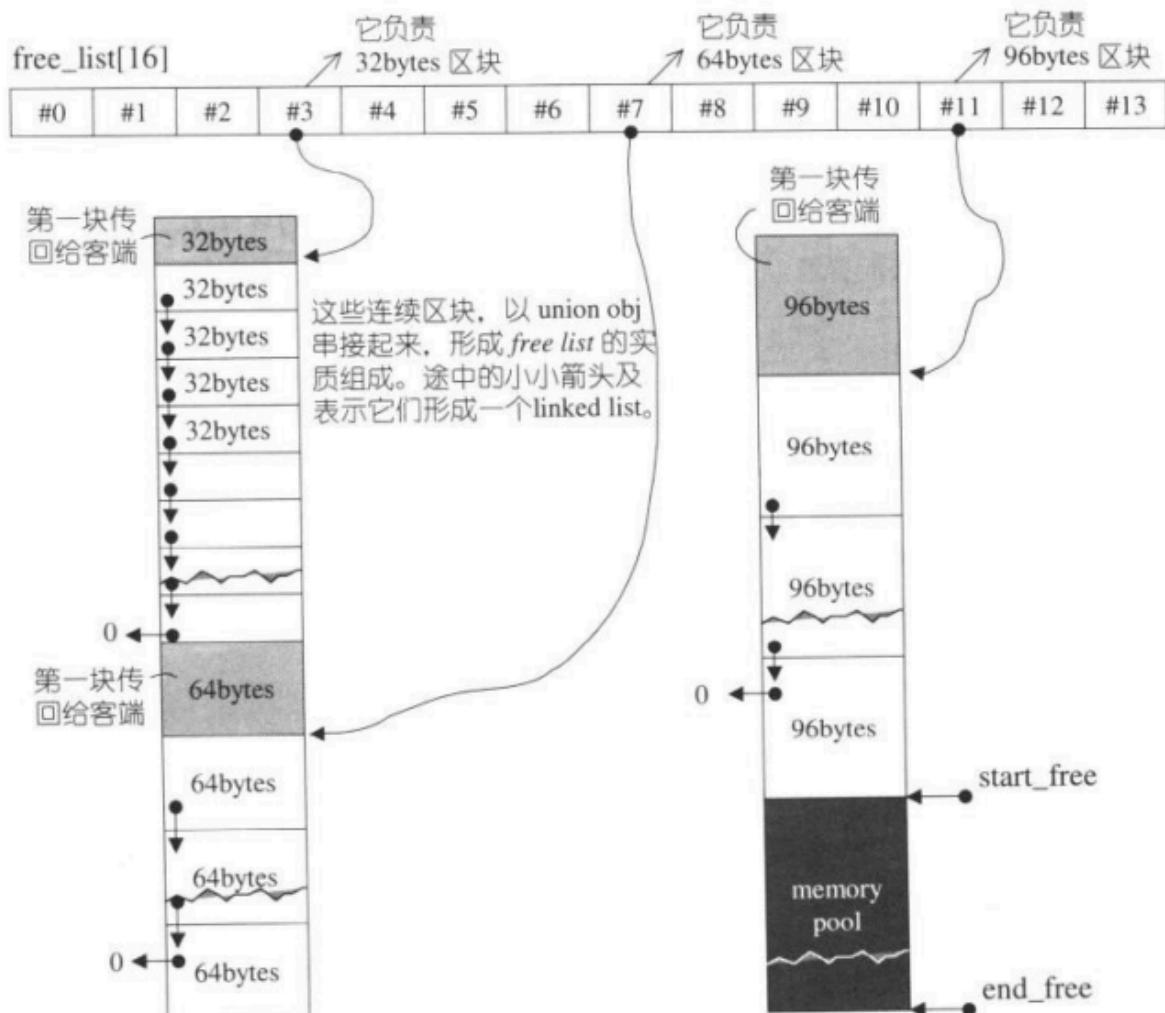
- 如果内存足够，就直接调出20个区块返回给`free list`
- 如果内存不足
  - 但够满足一个及以上区块，就调出(不足20个)剩余的区块
  - 甚至连一个区块都无法调出，此时尝试使用`malloc()`从`heap`中配置内存来增加内存池中的内存余量，配置的内存大小随着配置的次数逐渐增加

举例：

- 当调用`chunk_alloc(32, 20)`时，于是`malloc()`配置40个32bytes区块，取出其中的第一个区块，剩下的19个放入`free_list[3]`进行维护，再将剩余的20个留给内存池
- 当再调用`chunk_alloc(64, 20)`时，此时`free_list[7]`空空如也，而内存池中的内存余量 $20 * 32 \text{ bytes} / 64 \text{ bytes} = 10$ 个，就先把这10个区块返回，取出其中的第一个区块，剩下的9个放入`free_list[7]`中进行维护。此时内存池空空如也
- 再继续调用`chunk_alloc(96, 20)`，此时`free_list[11]`空空如也，此时想要从内存池中找内存，然而内存池中也是空空如也，于是以`malloc()`配置 $40 + n$ (附加量)个96bytes区块，取

出其中的第一个区块，再将19个交给free\_list[11]进行维护，剩下的 $20+n$  (附加量)留给内存池.....

- 当整个system heap空间都不够了，以至于无法再继续向内存池中添加内存，malloc()就无法继续进行，chunk\_alloc()再在free list中寻找是否有可用且足够大的区块，有就用，没有就找一级配置器帮忙，一级配置器使用malloc()进行内存配置，但它有out-of-memory处理机制(类似于new handler):
  - 如果成功，释放出足够的内存以使用
  - 如果失败，抛出bad\_alloc异常



image\_27.png

提供标准配置接口的 simple\_alloc

```

template<class T, class Alloc>
class simple_alloc{
public:
    static T* allocate(size_t n){
        return 0 == n ? 0 : (T*) Alloc::allocate(n * sizeof(T));
    }
    static T* allocate(void){
        return (T*) Alloc::allocate(sizeof(T));
    }
    static void deallocate(T* p, size_t n){
        if(0 != n) Alloc::deallocate(p, n * sizeof(T));
    }
    static void deallocate(T* p){
        Alloc::deallocate(p, sizeof(T));
    }
};

```

SGI通常使用这种方式来使用配置器：

```

template <class T, class Alloc = alloc> // 缺省使用 alloc 为配置器
class vector{
public:
    typedef T value_type;
protected: // 专属配置器，每次配置一个元素大小
    typedef simple_alloc<value_type, Alloc> data_allocator;
};

```

## 内存基本处理工具

STL定义有5个全局函数，作用于未初始化空间上：

1. `construct()`-> **构造**
2. `destroy()`-> **析构**
3. `uninitialized_copy()`-> `copy()`
4. `uninitialized_fill()`-> `fill()`

## 5. uninitialized\_fill\_n() -> fill\_n()

其中 copy(), fill(), fill\_n() 都为 *STL* 算法

### uninitialized\_copy()

```
template<class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
ForwardIterator result);
```

uninitialized\_copy() 将内存的配置与对象的构造分离开来，其工作方式：在输入范围内，利用迭代器 *i*，该迭代器会调用 `construct(&*(result + (i - first)), *i)`，以此来产生 *\*i* 的拷贝对象，并将拷贝得到的对象放置到输出范围的相对位置上。当要实现一个容器时，uninitialized\_copy() 可以帮助容器的全区间构造函数在配置内存区块（以包含范围内的所有元素）后，在该内存区块上构造元素。C++ 对于此还有一个要求——*commit or rollback*：若能对每个元素都成功能进行 uninitialized\_copy()，则 *commit*，否则 *rollback*。

### uninitialized\_fill

```
template<class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last,
const T&x);
```

uninitialized\_fill() 将内存配置与对象的构造分离开来，其工作方式：对于输入范围内的每个迭代器 *i*，调用 `construct(&*i, x)`，在 *i* 所指之处放置 *x* 的拷贝对象。C++ 对此同样也有 *commit or rollback* 的要求：若能对每个元素都能成功进行 uninitialized\_fill()，则 *commit*，如果有一个 *copy constructor* 抛出异常（操作失败），则将所有成功产生的元素全部析构掉。

### uninitialized\_fill\_n

```
template<class ForwardIterator, class Size, class T>
ForwardIterator
uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

uninitialized\_fill\_n() 将内存配置与对象的构造分离开来，它可以为指定范围内的所有元素设定相同的初值。对于 `[first, first + n]` 范围内的每个迭代器 *i*，uninitialized\_fill\_n() 会调用 `construct(&*i, x)`，为其在对应的位置上产生 *x* 的拷贝对象。C++ 对此同样也有 *commit or*

*rollback*的要求：若能对每个元素都能成功*uninitialized\_fill\_n()*，则*commit*，如果有一个*copy constructor*抛出异常(操作失败)，则将所有成功产生的元素全部析构掉。

## **uninitialized\_fill\_n 实现方法**

*uninitialized\_fill\_n()*函数接受3个参数：

- 迭代器 *first* 指向欲初始化空间的起始处
- *n* 表示欲初始化空间的大小
- *x* 表示初值

```
template <class ForwardIterator, class Size, class T>
inline ForwardIterator uninitialized_fill_n(ForwardIterator first,
                                             Size n,
                                             const T& x
                                         ) {
    return __uninitialized_fill_n(first, n, x, value_type(first));
    // for the above, call value_type() fetch out value type of first
}
```

补充：萃取器(Extractor) 萃取器用于取出迭代器中特定信息，如元素类型、迭代器类型等

```
// define a extractor 定义一个萃取器
template <typename Iterator>
struct IteratorTraits{
    using ValueType = typename Iterator::value_type;
};

// polarize pointer 偏特化指针类型
template<typename T>
struct IteratorTraits<T*>{
    using ValueType = T;
};

int main(){
    using Iter = std::vector<int>::iterator;
```

```

        using ValueType = typename IteratorTraits<Iter>::ValueType;
        std::cout << "Value type of iterator: " << typeid(ValueType).name()
<< '\n';
        return 0;
    }

```

输出:

```
Value type of interator: i -> (int)
```

uninitialized\_fill\_n()函数执行逻辑:

- 利用萃取器萃取出 first 的value type
- 判断此 value type 是否为POD型别

```

template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator first,
                                              Size n,
                                              const T& x,
                                              T1*){
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());
}

```

POD(Plain Old Data)指的是标量型别(scalar type)或传统的C struct型别。因此POD型别内必然会有trivial ctor/dtor/copy/assignment函数(类比C++中的class)，因此对POD型别采用效率最高的赋值方法，而对non-POD型别采用最安全的赋值方法：

- 前提条件1：如果 copy assignment 等同于 assignment
- 前提条件2：如果 destructor 等同于 trivial

对于POD型别的处理:

```

template<class ForwardIterator, class Size, class T>
inline ForwardIterator __uninitialized_fill_n_aux(ForwardIterator first,
                                                Size n,

```

```
        const T& x,
        __true_type){

    return fill_n(first, n, x);
}
```

对于non-POD型别的处理：

```
template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first,
                           Size n,
                           const T& x,
                           __false_type){
    ForwardIterator cur = first;
    for(; n > 0; --n, ++cur) construct(&*cur, x);
    return cur;
}
```

补充： **POD型别** 在C++中， Plain Old Data (POD) 是一种数据类型，具有与C语言中结构体和联合体相似的特性。POD类型可以用于在性能和内存布局上保证与C语言兼容的场景。POD类型的主要特征包括：

### 1. 标准布局类型 (Standard Layout Type) :

- 该类型的所有非静态数据成员具有相同的访问权限（全部是public、protected或private）。
- 类/结构体没有虚函数或虚基类。
- 所有非静态数据成员具有相同的访问控制（如全部是public或全部是private）。
- 派生类和基类必须具有相同的访问控制。

### 2. 聚合类型 (Aggregate Type) :

- 没有用户定义的构造函数。
- 没有私有或受保护的非静态数据成员。

- 没有基类。
- 没有虚函数。

POD类型的主要优点包括：

- **二进制兼容性**：POD类型与C语言的数据结构二进制兼容，因此可以直接用于C和C++之间的数据交换。
- **易于序列化**：由于POD类型的内存布局是线性的和固定的，可以直接将其写入文件或通过网络发送。
- **性能优化**：编译器可以对POD类型进行更多的优化，因为它们没有复杂的构造函数、析构函数或虚函数表。

## 示例

以下是一个POD类型的示例：

```
struct Point {  
    int x;  
    int y;  
};  
  
struct Rectangle {  
    Point top_left;  
    Point bottom_right;  
};
```

以上两个结构体都是POD类型，因为它们符合POD类型的所有要求。它们没有用户定义的构造函数、析构函数、虚函数，并且数据成员的访问权限都是public。

## 非POD类型示例

以下是一个非POD类型的示例：

```
struct NonPOD {  
    NonPOD() : x(0), y(0) {} // 用户定义的构造函数  
    int x;
```

```
    int y;  
};
```

由于NonPOD具有用户定义的构造函数，因此它不是POD类型。

## uninitialized\_copy() 实现方法

uninitialized\_copy()函数接受三个参数：

- 迭代器 first 指向输入端的起始位置
- 迭代器 last 指向输入端的结束位置
- 迭代器 result 指向输出端的起始处

输入端区间表示为[first, last)的前闭后开区间

```
template<class InputIterator, class ForwardIterator>  
inline ForwardIterator uninitialized_copy(InputIterator first,  
                                         InputIterator last,  
                                         ForwardIterator result){  
    return __uninitialized_copy(first, last, result,  
                               value_type(result));  
}  
/* value_type 获取 first 的 value type */
```

这个函数的执行逻辑：

1. 先用萃取器萃取出迭代器result的value type
2. 判断该类型是否为POD类型

```
template <class InputIterator, class ForwardIterator, class T>  
inline ForwardIterator __uninitialized_copy(InputIterator first,  
                                         InputIterator last,  
                                         ForwardIterator result,  
                                         T* ){  
    typedef typename __type_traits<T>::is_POD_type is_POD;
```

```
    return __uninitialized_copy_aux(first, last, result, is_POD());
    /* apt to use the result of is_POD() to complement type inference */
}
```

POD(Plain Old Data)指的是标量型别(scalar type)或传统的C struct型别。因此POD型别内必然会有trivial ctor/dtor/copy/assignment函数(类比C++中的class)，因此对POD型别采用效率最高的赋值方法，而对non-POD型别采用最安全的赋值方法：

- 前提条件1：如果 copy assignment 等同于 assignment
- 前提条件2：如果 destructor 等同于 trivial

对于POD型别的处理：

```
template<class InputIterator, class ForwardIterator>
inline ForwardIterator __uninitialized_copy_aux(InputIterator first,
                                                InputIterator last,
                                                ForwardIterator result,
                                                __true_type){
    return copy(first, last, result); // invoke copy() from STL
}
```

对于non-POD型别的处理：

```
template<class InputIterator, class ForwardIterator>
inline ForwardIterator __uninitialized_copy_aux(InputIterator first,
                                                InputIterator last,
                                                ForwardIterator result,
                                                __false_type){
    ForwardIterator cur = result;
    for(; first != last; ++first, ++cur) construct(&*cur, last);
    /* 此处只能够一个一个的元素进行构造，不能进行批量操作 */
    return cur;
}
```

对于 char\* 和 wchar\_t\* 这两种类型，采用最有效率的做法 memmove(直接对内存中的内容进行移动)：

- 针对 const char\*

```
/* a specialized version for const char* */
inline char* uninitialized_copy(const char* first,
                                const char* last,
                                char* result){
    memmove(result, first, last - first);
    return result + (last - first);
}
```

- 针对 const wchar\*

```
/* a specialized version for const wchar* */
inline wchar_t* uninitialized_copy(const wchar_t* first,
                                    const wchar_t* last,
                                    wchar_t* result){
    memmove(result, first, last - first);
    return result + (last - first);
}
```

## uninitialized\_fill 实现方法

uninitialized\_fill()函数接受三个参数：

- 迭代器 first 指向输出端的起始位置
- 迭代器 last 指向输出端的结束处(前闭后开区间)
- x 表示初值

```
template <class ForwardIterator, class T>
inline void uninitialized_fill(ForwardIterator first,
                             ForwardIterator last,
                             const T& x){
    __uninitialized_fill(first, last, x, value_type(first));
}
```

这个函数的执行逻辑：

1. 先用萃取器萃取出迭代器result的value type

2. 判断该类型是否为POD类型

```
template<class ForwardIterator, class T, class T1>
inline void __uninitialized_fill(ForwardIterator first,
                                ForwardIterator last,
                                const T& x,
                                T1*){
    typedef typename __type_trait<T>::is_POD_type is_POD;
    __uninitialized_fill_aux(first, last, x, is_POD());
}
```

POD(Plain Old Data)指的是标量型别(scalar type)或传统的C struct型别。因此POD型别内必然会有trivial ctor/dtor/copy/assignment函数(类比C++中的class)，因此对POD型别采用效率最高的赋值方法，而对non-POD型别采用最安全的赋值方法：

- 前提条件1：如果 copy assignment 等同于 assignment
- 前提条件2：如果 destructor 等同于 trivial

对于POD型别的处理：

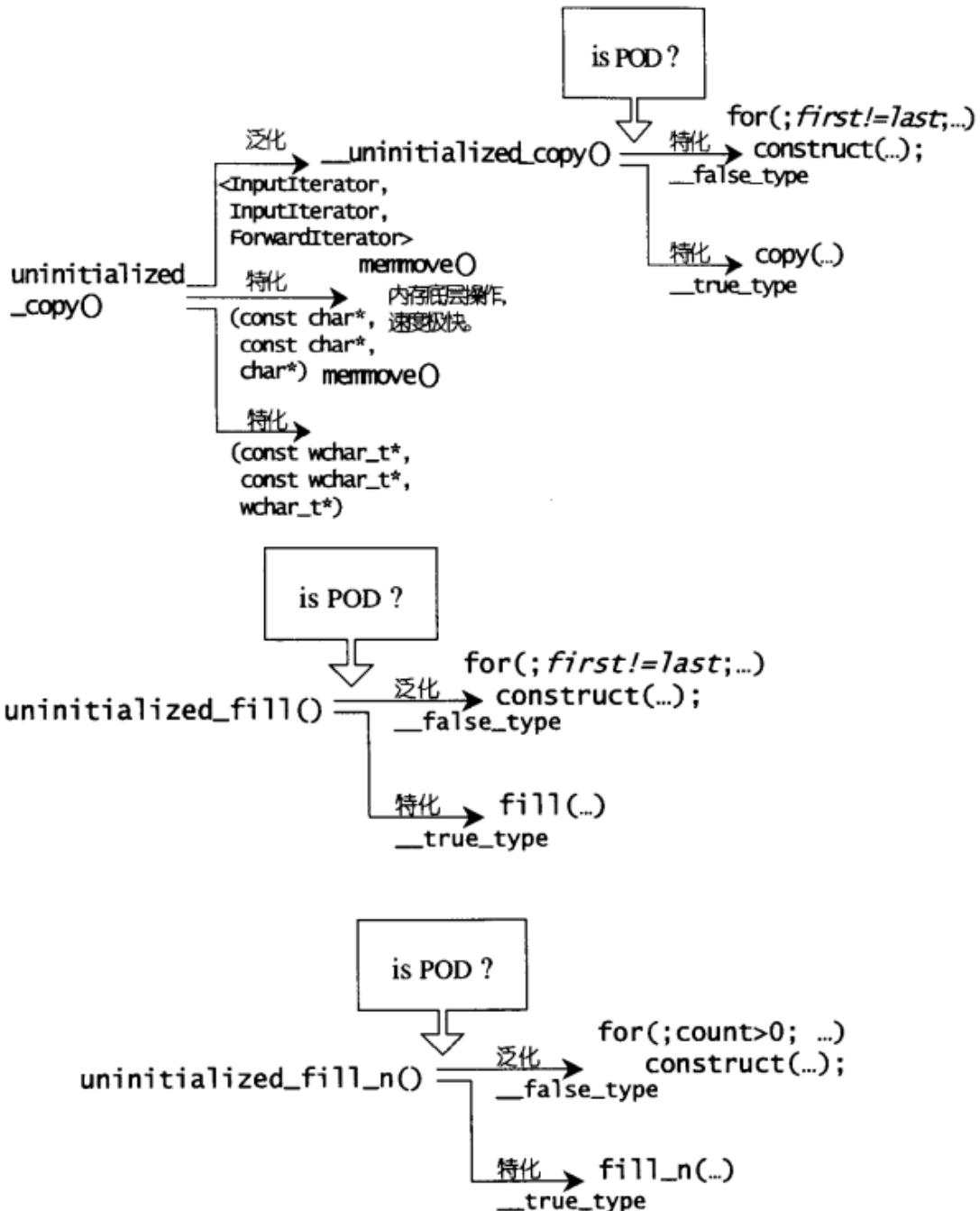
```
template<class ForwardIterator, class T>
inline void __uninitialized_fill_aux(ForwardIterator first,
                                    ForwardIterator last,
                                    const T& x,
                                    __true_type){
    fill(first, last, x);
}
```

对于non-POD型别的处理：

```
template<class ForwardIterator, class T>
void __uninitialized_fill_aux(ForwardIterator first,
                            ForwardIterator last,
                            const T& x,
```

```
    __false_type){  
    ForwardIterator cur = first;  
    for(; cur != last; cur++) construct(&*cur, x);  
}
```

对内存进行操作的函数的泛化版本和特化版本



image\_28.png

# The Terms in Book 《STL源码分析》

中英文对照

英文	中文	中文(台湾)
adapter	适配器	配接器
arguments	实参 (实质参数)	引数
by reference	传引用	传址
by value	传值	传值
dereference	解引用	提领
evaluate	计算	评估
instance	实例	实体
instantiated	实例化	实体化
library	库	程序库
range	范围	区间 (使用于STL)
resolve	解析	决议
parameter	形参 (形式参数)	参数
type	类型	型别
Object Oriented	面对对象	
design patterns	设计模式	

STL (Standard Template Library)	标准模板库
GP(Genetic Programming)	泛型编程
coupling	耦合性
reusability	复用性
Open-Closed	开放性封闭
container	容器
allocator	空间配置器
underscore	下划线
component	组件
iterator	迭代器
sequence container	序列式容器
associated container	关联式容器
algorithm	算法
functor or function object	仿函数或函数对象
polymorphism	多态性
virtual function	虚函数

genericity	泛型	
override	覆盖	
argument deduced	参数推导	
subroutine	子程序	
procedure	程序	
data structure	数据结构	
Genetic Paradigm	泛型思维	
components taxonomy	软件组件分类学	
library of abstract concepts	抽象概念库	
Assignable	可被赋值	
Default constructible	默认构造	
Equality Comparable	等价可判断	
LessThan Comparable	可比较大小	
Regular	常规	
Input Iterator	输入迭代器	
Output Iterator	输出迭代器	

Forward Iterator	单向迭代器	
Bidirectional Iterator	双向迭代器	
Random Access Iterator	随机存取迭代器	
Unary Function	一元函数	
Binary Function	二元函数	
Predicate	一元判断式	
Binary Predicate	二元判断式	
policy	策略	

\begin{公式} \label{

# Algorithm\_Tutorial

## Quick\_Sort

### 快速排序

Luogu P1177 (<https://www.luogu.com.cn/problem/P1177>) 将读入的 N 个数从小到大排序后输出。

### 输入格式

第一行为一个正整数 N。 第二行包含 N 个空格隔开的正整数  $a_i$ 。

### 输出格式

将给定的 N 个数从小到大输出，数之间空格隔开，行未换行且无空格。

对于 100% 的数据，有  $1 \leq N \leq 10^5$ ， $1 \leq a_i \leq 10^9$ 。

样例：

```
输入
8
9 1 7 6 6 3 2 8
```

```
输出
1 2 3 6 6 7 8 9
```

快速排序主要利用**分治思想**，时间复杂度  $O(n \log n)$ 。

```
int n, a[100005];

void quicksort(int l, int r){
    if(l == r) return;
    int i = l - 1, j = r + 1, x = q[(l + r) >> 1];
    while(i < j){
        do i++; while(q[i] < x); // 向右查找 ≥x 的数
        do j--; while(q[j] > x); // 向左查找 ≤x 的数
        swap(q[i], q[j]);
    }
}
```

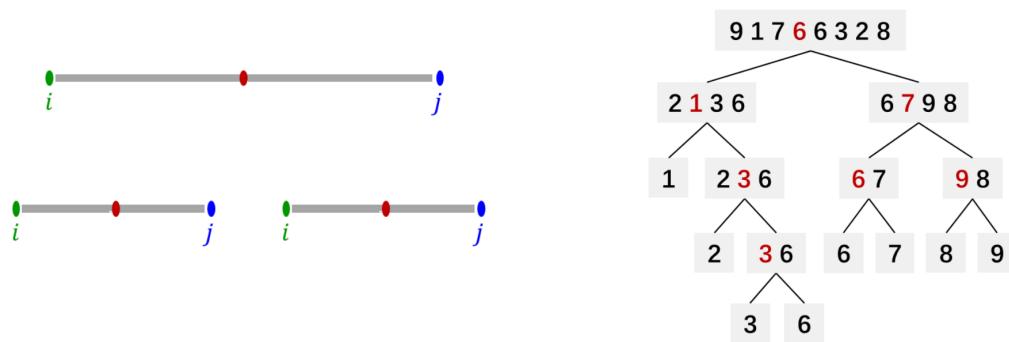
```

    if(i < j) swap(q[i], q[j]);
}
quicksort(l, j), quicksort(j + 1, r);
}

```

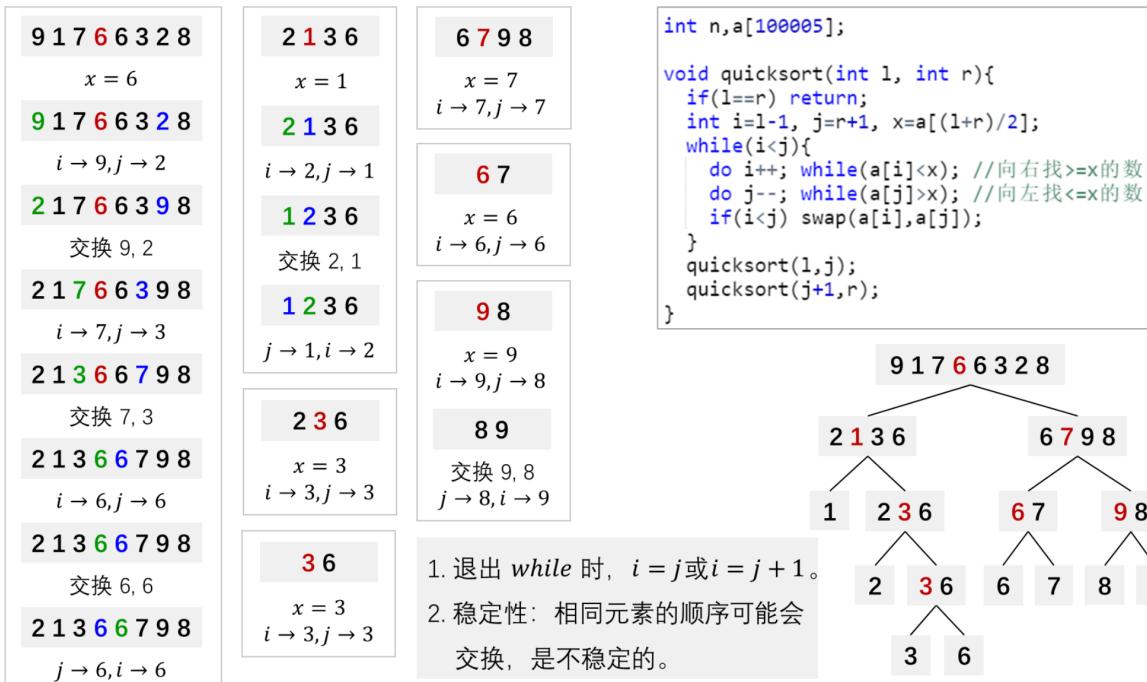
## 思路如下

1. 利用 **i (左指针)**, **j (右指针)** 指向数列的区间外侧, 数列的中值记为 **x**。
2. 将数列中  $\leq x$  的数放左段,  $\geq x$  的数放右段。
3. 对于左右两段, 再递归以上两个过程, 直到每段只有一个数, 即全部有序。



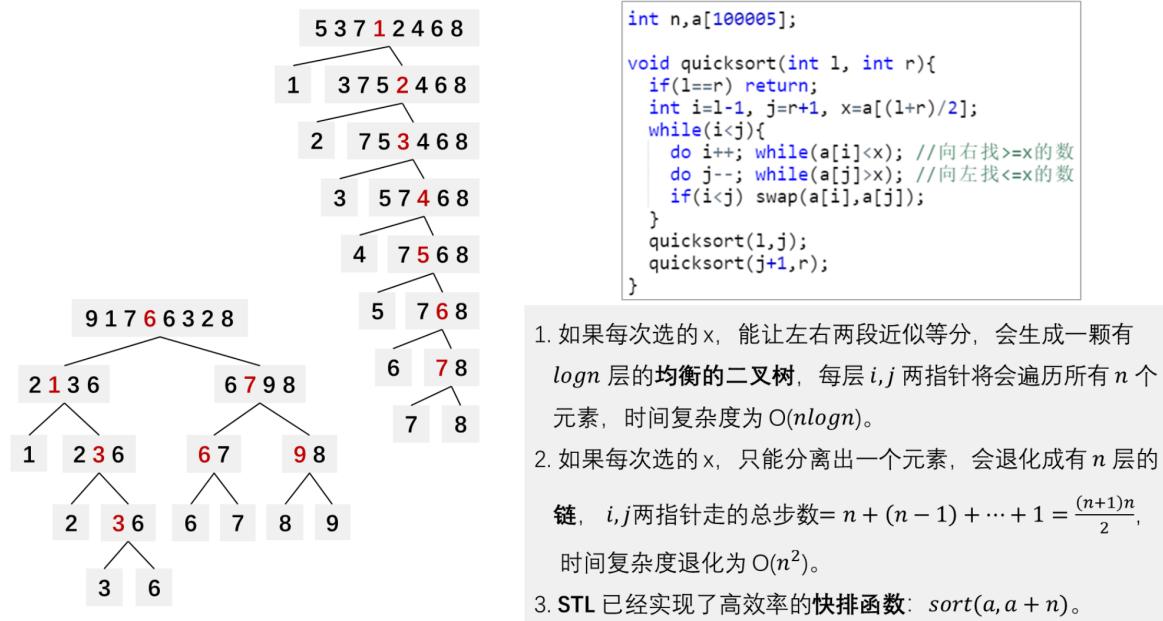
quick\_sort1.png

## 过程模拟



quick\_sort2.png

## 补充



quick\_sort4.png

即在中间值的左边找一个大于中间值的数  $m_1$ , 在中间值的右边找一个小于中间值的数  $m_2$ , 并且当  $m_1 < m_2$  时, 才交换  $m_1$  和  $m_2$ , 交换完成后再进行分割, 再对分割的区域重复上述的操作。

## STL实现

```
#include <iostream>
#include <algorithm>
using namespace std;

int n, a[100005];

int main(){
    cin >> n;
    for(int i = 0; i < n; i++) scanf("%d", &a[i]);
    sort(a, a + n);
    for(int i = 0; i < n; i++) printf("%d", &a[i]);
    return 0;
}
```

## 一般实现

```
#include <iostream>
using namespace std;

int n, a[100005];

void quicksort(int l, int r){
    if(l == r) return;
    int i = l - 1, j = r + 1, x = q[(l + r) >> 1];
    while(i < j){
        do i++; while(q[i] < x); // 向右查找 ≥x 的数
        do j--; while(q[j] > x); // 向左查找 ≤x 的数
        if(i < j) swap(q[i], q[j]);
    }
    quicksort(l, j), quicksort(j + 1, r);
}

int main(){
    cin >> n;
    for(int i = 0; i < n; i++) scanf("%d", &a[i]);
```

```

    quicksort(0, n - 1);
    for(int i = 0; i < n; i++) printf("%d", &a[i]);
    return 0;
}

```

## 题目

Luogu P1923 (<https://www.luogu.com.cn/problem/P1923>) 求第  $k$  小的数

输入  $n$  ( $1 \leq n \leq 5000000$  且  $n$  为奇数) 个数字  $a_i$  ( $1 \leq a_i \leq 10^9$ )，输出这些数字的第  $k$  小的数。最小的数是第 0 小。

请尽量不要使用 `nth_element` 来写本题，因为本题的重点在于练习分治算法。因为 `nth_element` 可以待 `quicksort` 后再输出 `nth_element`，这种做法会消耗多余的时间去处理多余的一半部分。

样例：

输入

```

5 1
4 3 2 1 5

```

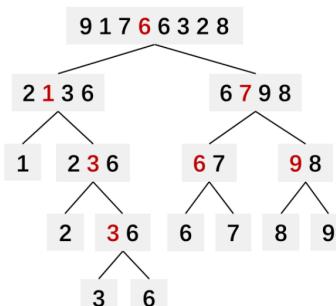
输出

```
2
```

```

int n,k,a[100010];
int qnth_element(int l, int r){
    if(l==r) return a[l];
    int i=l-1, j=r+1, x=a[(l+r)/2];
    while(i<j){
        do i++; while(a[i]<x); //向右找>=x的数
        do j--; while(a[j]>x); //向左找<=x的数
        if(i<j) swap(a[i],a[j]);
    }
    if(k<=j) return qnth_element(l,j);
    else return qnth_element(j+1,r);
}

```



kth\_number.png

每次取一半递归， $i, j$  指针每层走的步数为  $n, \frac{n}{2}, \frac{n}{4}, \dots, 1$ ，  
总步数  $= n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n$ ，  
所以时间复杂度为  $O(n)$ 。

## STL实现

```

#include <iostream>
using namespace std;

```

```

int n, k, a[5000010];

int main(){
    cin >> n >> k;
    for(int i = 0; i < n; i++) scanf("%d", &a[i]);
    nth_element(a, a + k, a + n);
    printf("%d\n", a[k]);
    return 0;
}

```

## 一般实现

```

#include <iostream>
using namespace std;

int n, k, q[5000010];

template<typename T>
void Swap(T& t1_, T& t2_){
    T t_ = t1_;
    t1_ = t2_;
    t2_ = t_;
}

void kth_number(int q[], int l, int r, int k){
    if(l >= r) return q[l];

    int i = l - 1, j = r + 1, x = q[(l + r) >> 1];
    while(i < j){
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);
        if(i < j) Swap(q[i], q[j]);
    }
    if(k <= j) return kth_number(q, l, j, k);
    else return kth_number(q, j + 1, r, k);
}

```

```
int main(){
    cin >> n >> k;
    for(int i = 0; i < n; i++) scanf("%d", q[i]);
    cout << kth_number(q, 0, n - 1, k);
    return 0;
}
```

## Merge\_Sort

### 归并排序

Luogu P1177 (<https://www.luogu.com.cn/problem/P1177>) 将读入的  $N$  个数从小到大排序后输出

### 输入格式

第一行为一个正整数  $N$ 。 第二行包含  $N$  个空格隔开的正整数  $a_i$ ，为你需要进行排序的数。

### 输出格式

将给定的  $N$  个数从小到大输出，数之间空格隔开，行末换行且无空格。

对于 20% 的数据，有  $1 \leq N \leq 10^3$ ；

对于 100% 的数据，有  $1 \leq N \leq 10^5$ ， $1 \leq a_i \leq 10^9$ 。

### 样例

#### 输入

```
5
5 2 4 5 1
```

#### 输出

```
1 2 4 4 5
```

归并排序主要利用分治思想，时间复杂度为  $O(n \log n)$ 。

```

int n, a[100010], b[100010]; /* 此处a, b为辅助数组 */

void merge_sort(int q[], int l, int r){
    if(l >= r) return;
    int mid = (l + r) >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r); /* 递归拆分 */

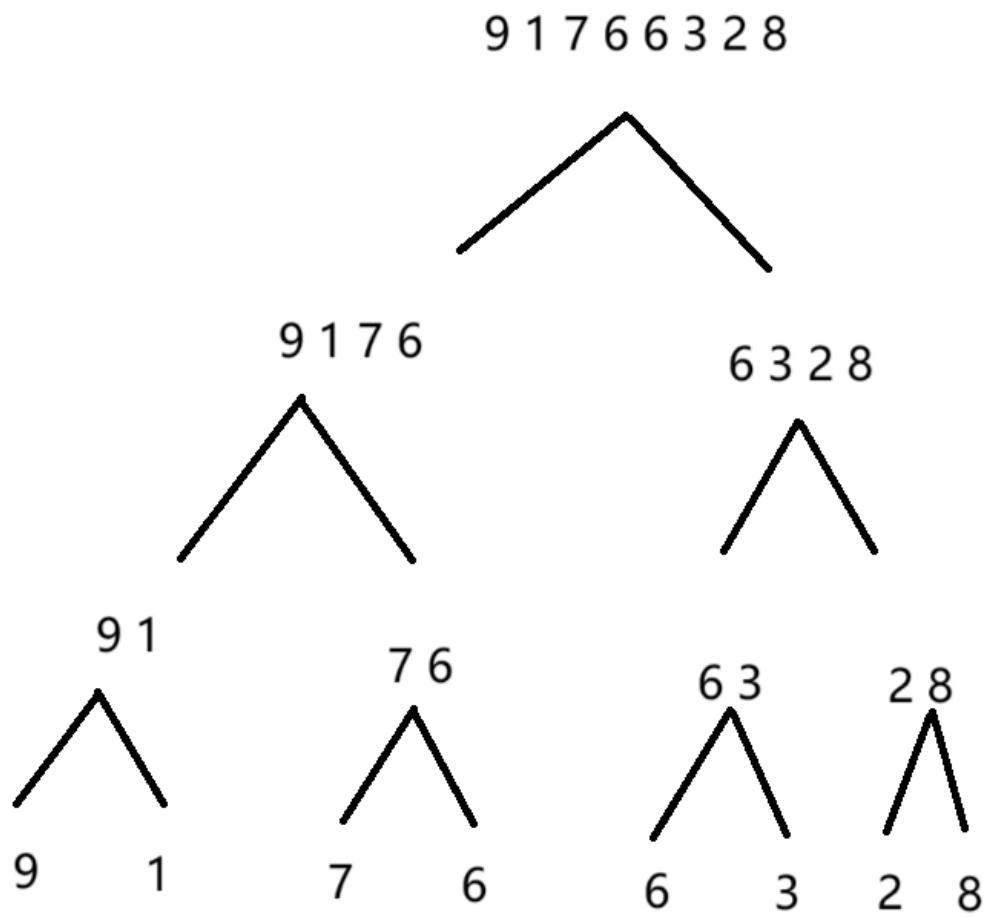
    int i = l, j = mid + 1, k = l; /* 合并 */
    while(i <= mid && j <= r){
        if(a[i] <= a[j]) b[k++] = a[i++];
        else b[k++] = a[j++];
    }

    while(i <= mid) b[k++] = a[i++];
    while(j <= r) b[k++] = a[j++];
    for(i = l; i <= r; i++) a[i] = b[i];
}
### 思路如下 ####

```

1. 对数列不断进行等长**拆分**，直到为一个数的长度；
2. 回溯时，按升序**合并**左右两段。
3. 重复以上两个过程，直到递归结束。

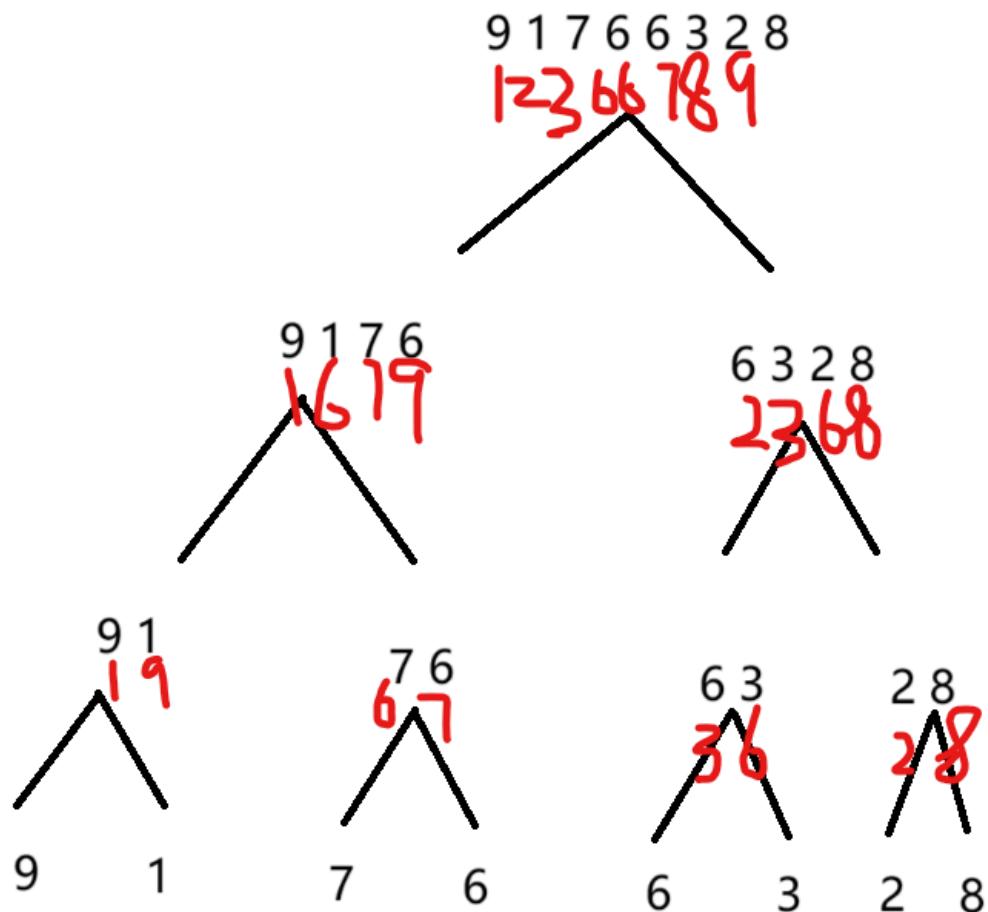
## 划分



divide.png

合并:

1. i, j 分别指向a的左右段起点, k指向b的起点。
2. 枚举a数组, 如果左数≤右数, 把左数放入b数组, 否则, 把右数放入b数组。
3. 把左段或右端剩余的数放入b数组。
4. 把b数组的当前段复制回a数组。



conjunct.png

	快速排序	归并排序
分治	先交换后拆分	先拆分后交换
稳定性	不稳定	稳定

完整实现如下

```
#include <iostream>
using namespace std;
int n, a[100010], b[100010];

void merge_sort(int l, int r){
```

```

if(l >= r) return;
int mid = (l + r) >> 1;
merge_sort(l, mid);
merge_sort(mid + 1, r);

int i = l, j = mid + 1, k = l;
while(i <= mid && j <= r)
    if(a[i] <= a[j]) b[k++] = a[i++];
    else b[k++] = a[j++];

while(i <= mid) b[k++] = a[i++];
while(j <= r) b[k++] = a[j++];
for(i = l; i <= r; i++) a[i] = b[i];
}

int main(){
    cin >> n;
    for(int i = 0; i < n; i++) scanf("%d", &a[i]);
    merge_sort(0, n - 1);
    for(int i = 0; i < n; i++) printf("%d ", a[i]);
    return 0;
}

```

Luogu P1908 (<https://www.luogu.com.cn/problem/P1908>)逆序对

对于给定的一段正整数序列，逆序对就是序列中 $a_i > a_j$ 且 $i < j$ 的有序对。注意序列中可能有重复数字。

## 输入格式

第一行，一个数n，表示序列中有n个数。

第二行n个数，表示给定的序列。序列中每个数字不超过 $10^9$ 。

对于25%的数据， $n \leq 2500$ 。

对于50%的数据， $n \leq 4 \times 10^4$ 。

对于所有数据， $n \leq 5 \times 10^5$ 。

## 输出格式

输出序列中逆序对的数目。

输入

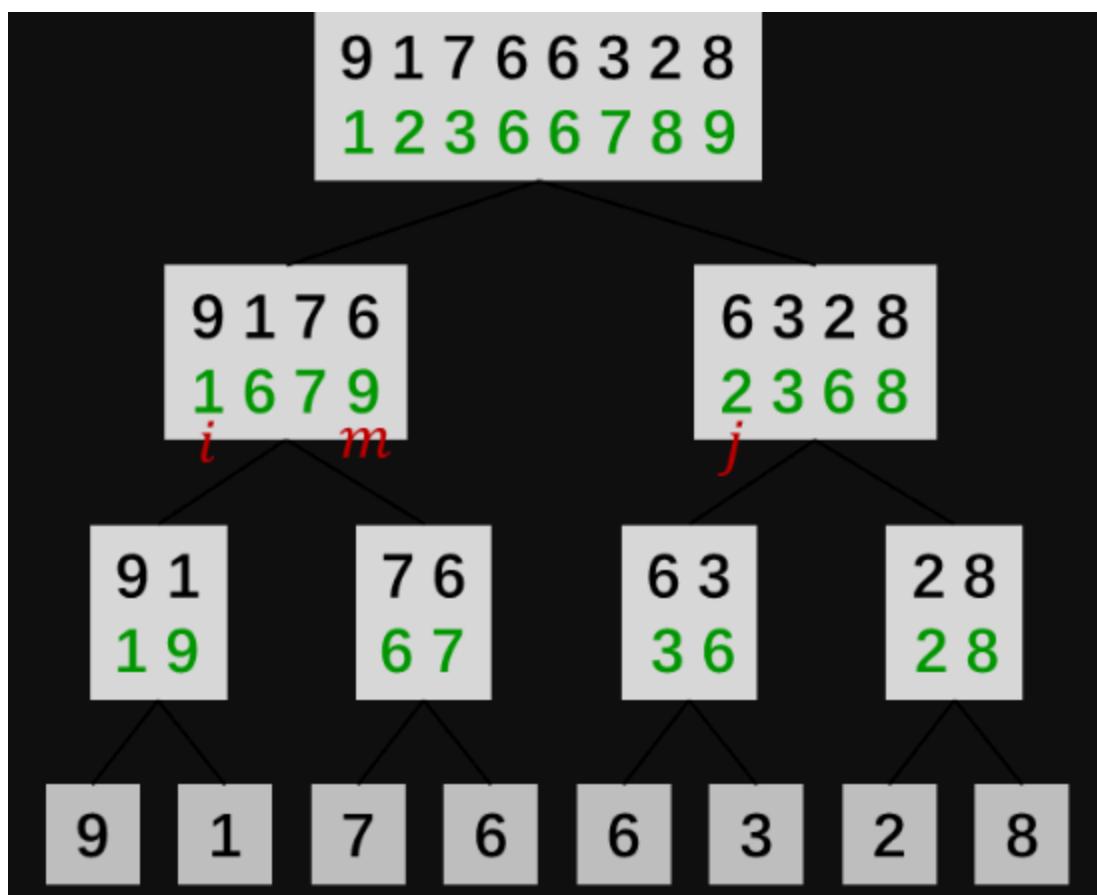
6

5 4 2 6 3 1

输出

11

## 示例



reverse\_pair.png

## 实现方法

```

#include <iostream>
using namespace std;

typedef long long LL;

int n, a[5000010], b[5000010];
LL res = 0;

void reverse_pair(int l, int r){
    if(l >= r) return;
    int mid = (l + r) >> 1;
    reverse_pair(l, mid);
    reverse_pair(mid + 1, r);

    int i = l, j = mid + 1, k = l;
    while(i <= mid && j <= r)
        if(a[i] <= a[j]) b[k++] = a[i++];
        else b[k++] = a[j++], res += mid - i + 1;

    while(i <= mid) b[k++] = a[i++];
    while(j <= r) b[k++] = a[j++];
    for(i = l; i <= r; i++) a[i] = b[i];
}

int main(){
    cin >> n;
    for(int i = 0; i < n; i++) scanf("%d", &a[i]);
    reverse_pair(0, n - 1);
    cout << res;
    return 0;
}

```

解释 of 'res += mid - i + 1': 当合并两个有序数组时，如果  $a[i] > a[j]$ ，说明  $a[i]$  及其后面的元素都比  $a[i]$  大，因为数组是有序的。所以，对于当前的  $a[j]$  来说，它与前半部分数组中剩余的元素构成逆序对。

Luogu P1966 (<https://www.luogu.com.cn/problem/P1966>) 火柴排队

## 题目描述

涵涵有两盒火柴，每盒装有  $n$  根火柴，每根火柴都有一个高度。现在将每盒中的火柴各自排成一列，同一列火柴的高度互不相同，两列火柴之间的距离定义为： $\sum(a_i - b_i)^2$

其中  $a_i$  表示第一列火柴中第  $i$  个火柴的高度， $b_i$  表示第二列火柴中第  $i$  个火柴的高度。

每列火柴中相邻两根火柴的位置都可以交换，请你通过交换使得两列火柴之间的距离最小。请问得到这个最小的距离，最少需要交换多少次？如果这个数字太大，请输出这个最小交换次数对  $10^8 - 3$  取模的结果。

## 输入格式

共三行，第一行包含一个整数  $n$ ，表示每盒中火柴的数目。

第二行有  $n$  个整数，每两个整数之间用一个空格隔开，表示第一列火柴的高度。

第三行有  $n$  个整数，每两个整数之间用一个空格隔开，表示第二列火柴的高度。

## 输出格式

一个整数，表示最少交换次数对  $10^8 - 3$  取模的结果。

## 样例1

输入

4

2 3 1 4

3 2 1 4

输出

1

## 样例2

输入

4

1 3 4 2

1 7 2 4

输出

2

### 输入输出样例说明一

最小距离是 0，最少需要交换 1 次，比如：交换第 1 列的前 2 根火柴或者交换第 2 列的前 2 根火柴。

### 输入输出样例说明二

最小距离是 10，最少需要交换 2 次，比如：交换第 1 列的中间 2 根火柴的位置，再交换第 2 列中后 2 根火柴的位置。

### 数据范围

对于 10% 的数据， $1 \leq n \leq 10^1$ ；

对于 30% 的数据， $1 \leq n \leq 10^2$ ；

对于 60% 的数据， $1 \leq n \leq 10^3$ ；

对于 100% 的数据， $1 \leq n \leq 10^5$ ， $0 \leq \text{火柴高度} \leq 2^{31}$ 。

### 整体思路

1. 根据每盒火柴的高度，用快速排序 quick\_sort 对其从小到大进行排序
2. 利用第 2 盒火柴中的位置对第 1 盒火柴中的位置进行映射，使之与排序之前的位置相同
3. 再利用归并排序 reverse\_pair 计算逆序数(即要交换的次数，题目中指出相邻的两根火柴可交换)。

### 实现方法

```
#include <cstdio>

struct node{
    int num, ord;
    bool operator<(node b1_){ return num < b1_.num; }
    bool operator>(node b2_){ return num > b2_.num; }
}f[100010], s[100010];
```

```

template<typename T>
void Swap(T& t1_, T& t2_){
    T t_ = t1_;
    t1_ = t2_;
    t2_ = t_;
}

int a[100010], b[100010], n, ans = 0;

// 类比归并排序求逆序数
void reverse_pair(int l, int r){
    if(l >= r) return;
    int mid = (l + r) >> 1;
    reverse_pair(l, mid);
    reverse_pair(mid + 1, r);

    int i = l, j = mid + 1, k = l;
    while(i <= mid && j <= r)
        if(a[i] <= a[j]) b[k++] = a[i++];
        else{
            b[k++] = a[j++];
            ans += mid - i + 1; // 求逆序数
            ans %= 99999997; // 处理结果
        }

    while(i <= mid) b[k++] = a[i++];
    while(j <= r) b[k++] = a[j++];
    for(i = l; i <= r; i++) a[i] = b[i];
}

// 一般快速排序
void quick_sort(node q[], int l, int r){
    if(l >= r) return;
    int i = l - 1, j = r + 1;
    node x = q[(l + r) >> 1];
    while(i < j){
        do i++; while(q[i] < x);

```

```

        do j--; while(q[j] > x);
        if(i < j) Swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}

int main(){
    // 读入数据
    scanf("%d", &n);
    for(int i = 1; i <= n; i++){
        scanf("%d", &f[i].num);
        f[i].ord = i;
    }

    for(int i = 1; i <= n; i++){
        scanf("%d", &s[i].num);
        s[i].ord = i;
    }

    // 根据高度对两盒火柴进行排序
    quick_sort(f, 1, n);
    quick_sort(s, 1, n);

    // 映射火柴的顺序
    for(int i = 1; i <= n; i++) a[f[i].ord] = s[i].ord;

    reverse_pair(1, n);
    printf("%d", ans);

    return 0;
}

```

## Binary\_Search

Luogu P2249 (<https://www.luogu.com.cn/problem/P2249>) 查找

### 题目描述

输入  $n$  个不超过  $10^9$  的单调不减的（就是后面的数字不小于前面的数字）非负整数  $a_1, a_2, \dots, a_n$ ，然后进行  $m$  次询问。对于每次询问，给出一个整数  $q$ ，要求输出这个数字在序列中第一次出现的编号，如果没有找的话输出 -1。

## 输入格式

第一行 2 个整数  $n$  和  $m$ ，表示数字个数和询问次数。

第二行  $n$  个整数，表示这些待查询的数字。

第三行  $m$  个整数，表示询问这些数字的编号，从 1 开始编号。

## 输出格式

输出一行， $m$  个整数，以空格隔开，表示答案。

## 样例

输入

```
11 3
1 3 3 3 5 7 9 11 13 15 15
1 3 6
```

输出

```
1 2 -1
```

数据保证， $1 \leq n \leq 10^6$ ， $0 \leq a_i, q \leq 10^9$ ， $1 \leq m \leq 10^5$

## 思路

1. 向存在目标值的区间缩小

2. 同时注意区间问题

## 模板一

```
#include <iostream>
using namespace std;
```

```

int n, m, q, a[1000005];

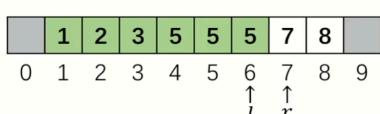
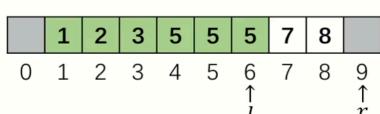
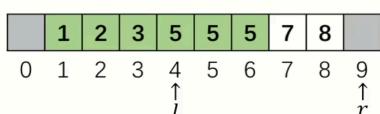
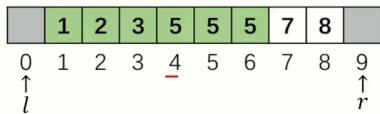
int find(int q){
    int l = 0, r = n + 1;
    while(l + 1 < r){ // 开区间
        int mid = (l + r) >> 1;
        if(a[mid] >= q) r = mid;
        else l = mid;
    }
    return a[r] == q ? r : -1;
}

int main(){
    scanf("%d %d, &n, &m);
    for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for(int i = 1; i <= m; i++) scanf("%d", &q), printf("%d ", find(q));
    return 0;
}

```

当要查找的值不在左右边界时

满足条件的区域称可行区 (绿区)



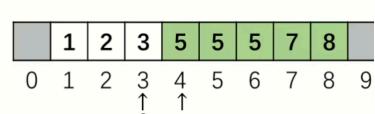
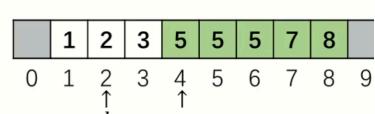
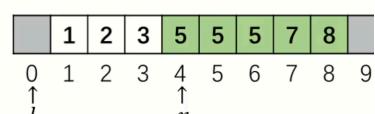
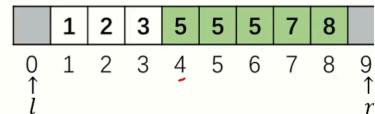
查找最后一个  $\leq 5$  的数的下标

```

int find(int q){
    int l=0,r=n+1;//开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>>1;
        if(a[mid]<=q) l=mid;
        else r=mid;
    }
    return l;
}

```

满足条件的区域称可行区 (绿区)



- 1. 指针的跳跃次数  **$\log n$**
- 2. 可行区的指针最后一定指向**答案**

binary\_search.png

当要查找的值在左右边界时

1	2	3	5	5	5	7	8
1	2	3	4	5	6	7	8

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
1	2	3	4	5	6	7	8

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
1	2	3	4	5	6	7	8

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
1	2	3	4	5	6	7	8

$\uparrow \quad \uparrow$   
 $l \quad r$

查找最后一个  $\leq 8$  的数的下标

```
int find(int q){
    int l=0,r=n+1;//开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>>1;
        if(a[mid]<=q) l=mid;
        else r=mid;
    }
    return l;
}
```

1	2	3	5	5	5	7	8
0	1	2	3	4	5	6	7

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
0	1	2	3	4	5	6	7

$\uparrow \quad \uparrow$   
 $l \quad r$

查找第一个  $\geq 1$  的数的下标

```
int find(int q){
    int l=0,r=n+1;//开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>>1;
        if(a[mid]>=q) r=mid;
        else l=mid;
    }
    return r;
}
```

1	2	3	5	5	5	7	8
0	1	2	3	4	5	6	7

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
0	1	2	3	4	5	6	7

$\uparrow \quad \uparrow$   
 $l \quad r$

1	2	3	5	5	5	7	8
0	1	2	3	4	5	6	7

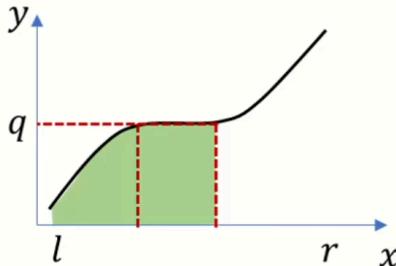
$\uparrow \quad \uparrow$   
 $l \quad r$

3. **开区间**可以正确处理**边界**

binary\_search\_.png

最大化和最小化查找 最大化即找最大下标 最小化即找最小下标

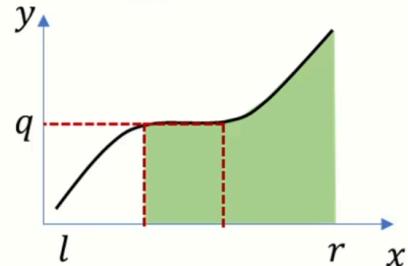
### 1. 最大化查找 (可行区在左侧)



查找最后一个  $\leq q$  的数的下标

```
int find(int q){
    int l=0, r=n+1; //开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>>1;
        if(a[mid]<=q) l=mid;
        else r=mid;
    }
    return l;
}
```

### 2. 最小化查找 (可行区在右侧)



查找第一个  $\geq q$  的数的下标

```
int find(int q){
    int l=0, r=n+1; //开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>>1;
        if(a[mid]>=q) r=mid;
        else l=mid;
    }
    return r;
}
```

1. 指针的跳跃次数  $\log n$
2.  $l + 1 = r$  时结束
2. 可行区的指针最后一定指向 答案
3. 开区间可以正确处理边界

binary\_search\_.png

## 模板二

```
/* 闭区间 */
#include <cstdio>

int n, m, q, a[1000005];

int find(int k){
    int l = 1, r = n; /* 闭区间 */
    while(l < r){
        int mid = (l + r) >> 1;
        if(a[mid] >= k) r = mid;
        else l = mid + 1; /* *** */
    }
    return a[r] == k ? r : -1;
}
```

```
}

int main(){
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for(int i = 1; i <= m; i++) scanf("%d", &q), printf("%d ", find(q));
    return 0;
}
```

## 模板三

```
#include <cstdio>

int n, m, q, a[1000005];

int find(int k){
    int ans = 0;
    int l = 1, r = n; /* 闭区间 */
    while(l <= r){ // l == r + 1 时结束
        int mid = (l + r) >> 1;
        if(a[mid] >= q) ans = mid, r = mid - 1; /* 利用 ans 记录 mid
        else l = mid + 1;
    }
    return a[ans] == q ? ans : -1;
}

int main(){
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for(int i = 1; i <= m; i++) scanf("%d", &q), printf("%d ", find(q));
    return 0;
}
```

## 模板四

```

/* 算法库实现 */
#include <cstdio>
#include <algorithm>

int n, m, q, a[1000005];

int main(){
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for(int i = 1; i <= m; i++){
        scanf("%d", &q);
        int ans = lower_bound(a + 1, a + n + 1, q) - a; // 缩小区间
        if(a[ans] == q) printf("%d ", ans);
        else printf("-1 ");
    }
    return 0;
}

```

Luogu P1024 (<https://www.luogu.com.cn/problem/P1024>) 一元三次方程求解

## 题目描述

有形如： $ax^3 + bx^2 + cx + d = 0$  这样的一个一元三次方程。给出该方程中各项的系数（ $a,b,c,d$  均为实数），并约定该方程存在三个不同的实根（根的范围在 -100 至 100 之间），且根于根之差的绝对值  $\geq 1$ 。要求由小到大依次在同一行输出这三个实根（根于根之间留有空格），并精确到小数点后2位。

## 输入格式

一行， 4个实数 $a,b,c,d$ 。

## 输出格式

一行， 3个实根，从小到大输出，并精确到小数点后2位。

## 样例

输入

1 -5 -4 20

输出

-2.00 2.00 5.00

### 整数二分：最大化查找

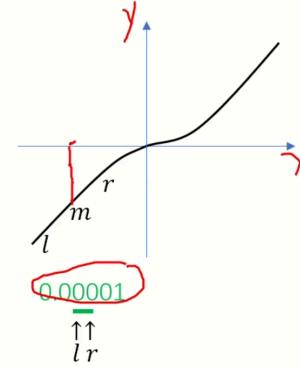
查找最后一个  $\leq q$  的数的下标

```
int find(int q){
    int l=0, r=n+1; //开区间
    while(l+1<r){ //l+1=r时结束
        int mid=l+r>1;
        if(a[mid]<=q) l=mid;
        else r=mid;
    }
    return l;
}
```

### 浮点数二分：最大化查找

求一个浮点数 ( $-10000 \leq y \leq 10000$ ) 的三次方根

```
double find(double y){
    double l=-100, r=100;
    while(r-l>1e-5){
        double mid=(l+r)/2;
        if(mid*mid*mid==y) l=mid;
        else r=mid;
    }
    return l;
}
int main(){
    double y; scanf("%lf", &y);
    printf("%.3lf\n", find(y));
    return 0;
}
```



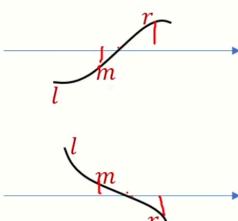
root1\_.png

有形如:  $ax^3 + bx^2 + cx + d = 0$  这样的一个一元三次方程。给出该方程中各项的系数 ( $a, b, c, d$  均为实数), 并约定该方程存在三个不同实根 (根的范围在  $-100$  至  $100$  之间), 且根与根之差的绝对值  $\geq 1$ 。要求由小到大依次在同一行输出这三个实根(根与根之间留有空格), 并精确到小数点后 2 位。

提示: 记方程  $f(x) = 0$ , 若存在 2 个数  $x_1$  和  $x_2$ , 且  $x_1 < x_2$ , 之间一定有一个根。

#### 输入格式

一行, 4 个实数  $a, b, c, d$ 。



#### 输出格式

一行, 3 个实根, 从小到大输出, 并精确到小数点后 2 位。

#### 输入输出样例



#### 输入 #1

1 -5 -4 20

#### 输出 #1

-2.00 2.00 5.00

```
double fun(double x){
    return a*x*x*x+b*x*x+c*x+d;
}
double find(double l,double r){
    while(r-l>0.0001){
        double mid=(l+r)/2;
        if(fun(mid)*fun(r)<0) l=mid; //最大化
        else r=mid;
    }
    return l;
}
int main(){
    scanf("%lf%lf%lf%lf", &a, &b, &c, &d);
    for(int i=-100;i<100;i++){
        double y1=fun(i), y2=fun(i+1);
        if(!y1) printf("%.2lf ", 1.0*i);
        if(y1*y2<0)printf("%.2lf ", find(i,i+1));
    }
    return 0;
}
```

root2\_.png

## 实现

```

#include <stdio.h>

double a, b, c, d;

double fun(double x) { return a * x * x * x + b * x * x + c * x + d; }

double root(double l, double r){
    while(r - l > 0.0001){ // 精度控制
        double mid = (l + r) / 2;
        if(fun(mid) * fun(l) < 0) r = mid;
        else l = mid;
    }
    return r;
}

int main(){
    scanf("%lf %lf %lf %lf", &a, &b, &c, &d);
    for(int i = -100; i < 100; i++){
        double y1 = fun(i), y2 = fun(i + 1);
        if(!y1) printf("%.2lf ", i * 1.0); // 若 i 本身就是根的情况
        if(y1 * y2 < 0) printf("%.2lf ", root(i, i + 1));
    }
    return 0;
}

```

## High\_Precision

### 高精度加法

#### 题目描述

高精度加法，相当于  $a + b$  problem, 需用考虑负数

#### 输入格式

分两行输入。  $a, b \leq 100^{500}$ 。

#### 输出格式

输出只有一行，代表  $a + b$  的值。

样例

输入

1001

9099

输出

10100

## 实现一

```
#include <iostream>
using namespace std;

const int N = 505;
int a[N], b[N], c[N];
int la, lb, lc;

void add(int a[], int b[], int c[]){
    for(int i = 1; i <= lc; i++){
        c[i] += a[i] + b[i];
        c[i + 1] = c[i] / 10;
        c[i] %= 10;
    }
    if(c[lc + 1]) lc++; // 若 a = 900, b = 300, c(lc = 3) = 1300
                    // 此时 lc = 3, 故需要特判 lc 位置的下一位, 检测是否还有数字
}

int main(){
    string sa, sb; cin >> sa >> sb;
    la = sa.size(), lb = sb.size(), lc = max(la, lb);
    // 因为读入时 高位在后, 所以要逆序读入
    for(int i = 1; i <= la; i++) a[i] = sa[la - i] - '0'; // 将 sa 中的数字逆序存入 a 中
```

```

    for(int i = 1; i <= lb; i++) b[i] = sb[lb - i] - '0'; //同理
    add(a, b, c);
    for(int i = lc; i; i--) printf("%d", c[i]); //逆序输出
    return 0;
}

```

## 实现二

```

#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> VI;
VI a, b, c;
int la, lb, lc;

void add(VI& a, VI& b, VI& c){
    int t = 0;
    for(int i = 0; i < lc; i++){
        if(i < la) t += a[i];
        if(i < lb) t += b[i];
        c.push_back(t % 10);
        t /= 10;
    }
    if(t) c.push_back(t);
}

int main(){
    string sa, sb; cin >> sa >> sb;
    la = sa.size(), lb = sb.size(), lc = max(la, lb);
    for(int i = la - 1; ~i; i--) a.push_back(sa[i] - '0');
    for(int i = lb - 1; ~i; i--) b.push_back(sb[i] - '0');
    add(a, b, c);
    for(int i = c.size() - 1; ~i; i--) printf("%d", c[i]);
}

```

```
    return 0;  
}
```

## 高精度减法

### 题目描述

高精度减法

### 输入格式

两个整数  $a, b$  (第二个可能比第一个大)

### 输出格式

结果 (是负数要输出负号)

### 样例

输入

```
2  
1
```

输出

```
1
```

## 实现一

```
#include <iostream>  
using namespace std;  
  
const int N = 100050;  
int a[N], b[N], c[N];  
int la, lb, lc;  
  
bool cmp(int a[], int b[]){  
    if(la != lb) return la < lb;  
    for(int i = la; i-->0) if(a[i] < b[i]) return a[i] < b[i];  
    return false;
```

```

}

void sub(int a[], int b[], int c[]){
    for(int i = 1; i <= lc; i++){
        if(a[i] < b[i]) a[i + 1]--, a[i] += 10;
        c[i] = a[i] + b[i];
    }
    while(c[lc] == 0 && lc > 1) lc--;
}

int main(){
    string sa, sb; cin >> sa >> sb;
    for(int i = 1; i < la; i++) a[i] = sa[la - i] - '0';
    for(int i = 1; i < lb; i++) b[i] = sb[lb - i] - '0';
    if(cmp(a, b)) cout << "-";
    sub(a, b, c);
    for(int i = lc; i; i--) printf("%d", c[i]);

    return 0;
}

```

## 实现二

```

#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> VI;
VI a, b, c;

bool cmp(VI a, VI b){
    if(a.size() != b.size()) return a.size() < b.size();
    for(int i = a.size(); ~i; i--) if(a[i] != b[i]) return a[i] < b[i];
    return false;
}

void sub(VI a, VI b, VI& c){

```

```

int t = 0;
for(int i = 0; i < a.size(); i++){
    t = a[i];
    if(i < b.size()) t -= b[i];
    if(t < 0) a[i + 1]--, t += 10;
    c.push_back(t);
}
while(c.size() > 1 && !c.back()) c.size()--;
}

int main(){
    string sa, sb; cin >> sa >> sb;
    for(int i = sa.size() - 1; ~i; i--) a.push_back(sa[i] - '0');
    for(int i = sb.size() - 1; ~i; i--) b.push_back(sb[i] - '0');
    if(cmp(a, b)) swap(a, b), cout << '-';
    sub(a, b, c);
    for(int i = c.size() - 1; ~i; i--) printf("%d", c[i]);
    return 0;
}

```

## **Prefix\_Sum & Difference**

## **Dual\_Pointers**

## **Bit\_Operation**

## **Discretization**

## **Merge\_Parts**

**Edited by ppQwQqq**

# Topic title

为了在MySQL中进行相应的实验，我们需要将每个实验的内容从SQL Server转换为MySQL。以下是详细的替代实验步骤和内容：

## 实验一 MySQL的安装及管理工具的使用

### 实验要求与目的

1. 了解MySQL安装对软、硬件的要求，掌握安装方法。
2. 了解MySQL的配置方法。
3. 了解MySQL包含的主要组件及其功能。
4. 熟悉MySQL管理平台的界面及基本使用方法。
5. 了解在MySQL管理平台中执行SQL语句的方法。

### 实验步骤

#### 1. 安装MySQL

- 下载MySQL Community Server安装包并进行安装。
- 安装过程中，注意选择需要的组件，如MySQL Server、MySQL Workbench等。

#### 2. 配置MySQL

- 使用mysql\_secure\_installation进行初始配置。
- 配置my.cnf文件以满足特定需求（如端口、字符集等）。

#### 3. MySQL主要组件

- 了解MySQL Server、MySQL Workbench、MySQL Shell等组件及其功能。

#### 4. MySQL Workbench使用

- 打开MySQL Workbench，连接到MySQL服务器。

- 了解Workbench界面的基本布局，如导航面板、SQL编辑器等。

## 5. 执行SQL语句

- 在MySQL Workbench的SQL编辑器中输入并执行简单的SQL查询，例如：

```
SELECT VERSION();
```

# 实验二 MySQL数据库的管理

## 实验要求与目的

1. 了解MySQL数据库的逻辑结构和物理结构的特点。
2. 掌握使用MySQL Workbench对数据库进行管理的方法。

## 实验步骤

### 1. 逻辑结构

- 了解数据库、表、视图、存储过程等逻辑结构。

### 2. 物理结构

- 了解MySQL的存储引擎（如InnoDB、MyISAM等）的特点。

### 3. 数据库管理

- 使用MySQL Workbench创建数据库和表：

```
CREATE DATABASE testdb;
USE testdb;
CREATE TABLE test_table (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

# 实验三 SQL定义语言（DDL）操作

## 实验要求与目的

- 熟悉SQL定义语言（DDL）的基本概念和作用。
- 掌握使用SQL DDL语句进行基本表结构的定义、修改、删除的方法。
- 了解常见的数据类型及其在MySQL中的应用。

## 实验步骤

### 1. 创建表

```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    enrollment_date DATE,
    PRIMARY KEY (student_id)
);
```

### 2. 修改表

- 添加列：

```
ALTER TABLE students ADD COLUMN email VARCHAR(100);
```

- 删除列：

```
ALTER TABLE students DROP COLUMN email;
```

- 修改列数据类型：

```
ALTER TABLE students MODIFY COLUMN first_name VARCHAR(100);
```

### 3. 删除表

```
DROP TABLE students;
```

## 实验四 MySQL数据表的增、删、改操作

### 实验要求与目的

- 熟悉MySQL中数据的基本操作，包括插入（INSERT）、修改（UPDATE）、删除（DELETE）。
- 理解MySQL的常用数据类型，并能在实际的数据操作中正确应用。

### 实验步骤

#### 1. 插入数据

```
INSERT INTO students (first_name, last_name, enrollment_date)  
VALUES ('John', 'Doe', '2023-01-01');
```

#### 2. 修改数据

```
UPDATE students  
SET last_name = 'Smith'  
WHERE student_id = 1;
```

#### 3. 删除数据

```
DELETE FROM students  
WHERE student_id = 1;
```

## 实验五 使用SQL语言进行简单查询

### 实验要求与目的

- 熟练掌握SQL的SELECT语句，包括投影、选择、排序、分组等操作。

2. 熟练使用SQL的比较运算符、逻辑运算符、字符匹配运算符、算术运算符等。
3. 利用分组函数进行数据的汇总和统计。

## 实验步骤

### 1. 简单查询

```
SELECT first_name, last_name FROM students;
```

### 2. 使用WHERE子句

```
SELECT * FROM students WHERE enrollment_date > '2023-01-01';
```

### 3. 排序

```
SELECT * FROM students ORDER BY last_name ASC;
```

### 4. 分组和聚合函数

```
SELECT COUNT(*), AVG(student_id) FROM students;
```

## 实验六 使用SQL语言进行复杂查询

### 实验要求与目的

1. 熟练掌握SQL的连接查询和嵌套查询的语法和用法。
2. 理解并应用连接条件和子查询的逻辑，确保查询结果的准确性和完整性。

## 实验步骤

### 1. 连接查询

```
SELECT a.first_name, b.course_name  
FROM students a
```

```
INNER JOIN courses b ON a.student_id = b.student_id;
```

## 2. 嵌套查询

```
SELECT first_name, last_name  
FROM students  
WHERE student_id IN (SELECT student_id FROM courses WHERE course_name  
= 'Math');
```

# 实验七 索引的创建与管理

## 实验要求与目的

1. 掌握在MySQL中创建、修改和删除索引的方法。
2. 了解主键、外键和唯一约束的概念和作用。
3. 学会在MySQL中创建、修改和删除主键、外键和唯一约束。
4. 理解这些数据库对象如何影响数据的完整性和查询性能。

## 实验步骤

### 1. 创建索引

```
CREATE INDEX idx_lastname ON students (last_name);
```

### 2. 删除索引

```
DROP INDEX idx_lastname ON students;
```

### 3. 主键、外键和唯一约束

- 创建主键：

```
ALTER TABLE students ADD PRIMARY KEY (student_id);
```

- 创建外键：

```
ALTER TABLE courses ADD CONSTRAINT fk_student  
FOREIGN KEY (student_id) REFERENCES students(student_id);
```

- 创建唯一约束：

```
ALTER TABLE students ADD CONSTRAINT unique_email UNIQUE (email);
```

以上是将SQL Server实验内容转换为MySQL实验内容的详细步骤和说明。希望这些内容能帮助你顺利完成实验任务。

# 《TCP/IP详解卷一：协议》

## 第一章 概述

### 分层

层次	功能
应用层	Telnet、FTP和e-mail
运输层	TCP和UDP
网络层	IP、ICMP和IGMP
链路层	设备驱动程序及接口卡

1. 链路层：一般指设备驱动程序及接口卡，处理一些物理传输媒介的细节。
2. 网络层：一般用于处理分组在网络中的活动。
3. 运输层：一般用于两台主机上应用程序间的端到端的通信。

协议	特点	处理方式
TCP	可靠性高	数据分组->发送到网络层->确认收到->再进行相关的时钟设置等->发送到下一层
UDP	可靠性低	数据分组->发送到下一层

**任意的可靠性必须由应用层提供**

4. 应用层：一般用于处理特定的应用程序细节。

此处，顶层（应用层）与下三层的不同在于，**顶层只需要关心应用程序的细节，下三层处理通信细节**。

局域网上运行FTP的两台主机

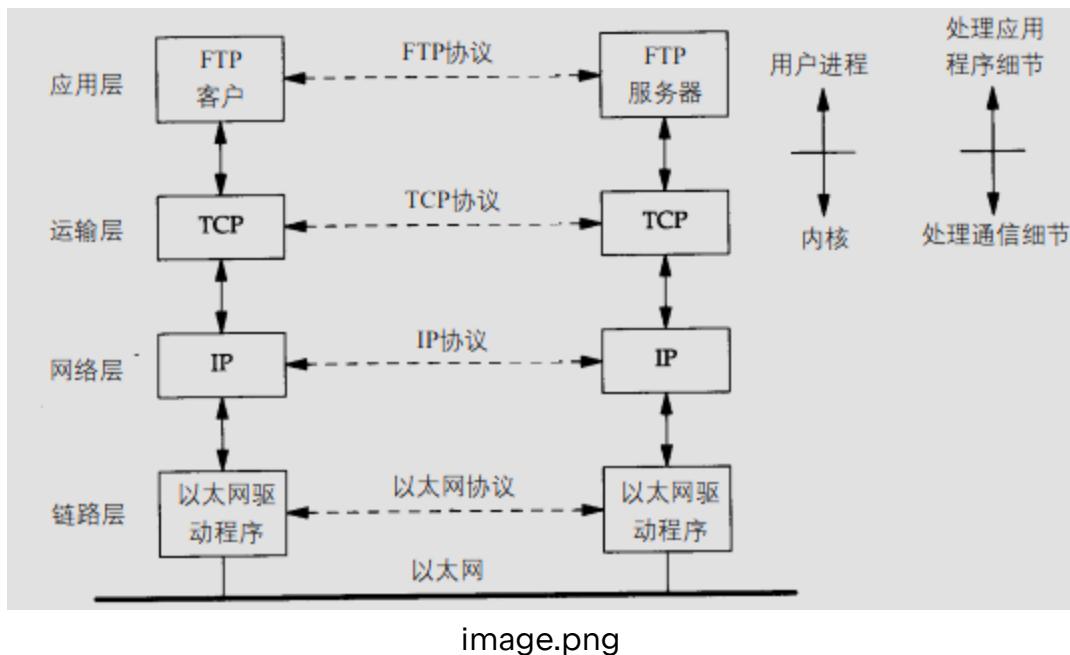


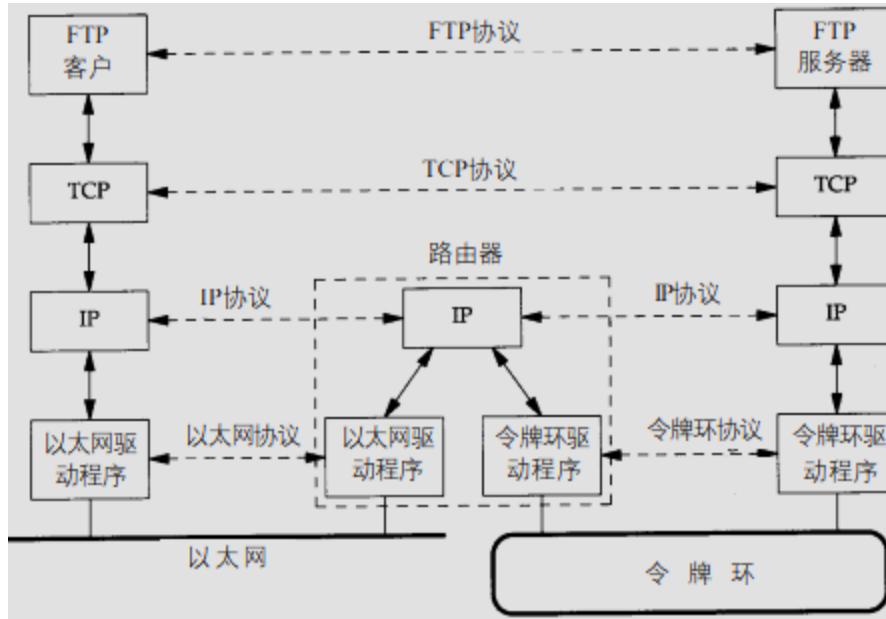
image.png

服务模式：Client - Server: Client - 请求-> Server, Server - 响应-> Client

协议	层次
FTP	应用层
TCP	运输层
IP	网络层
以太网协议	链路层

TCP/IP为一个协议族 (Internet Protocol Suite)

链接互联网的方法	层次
路由器(最简单)	网路层
网桥	链路层

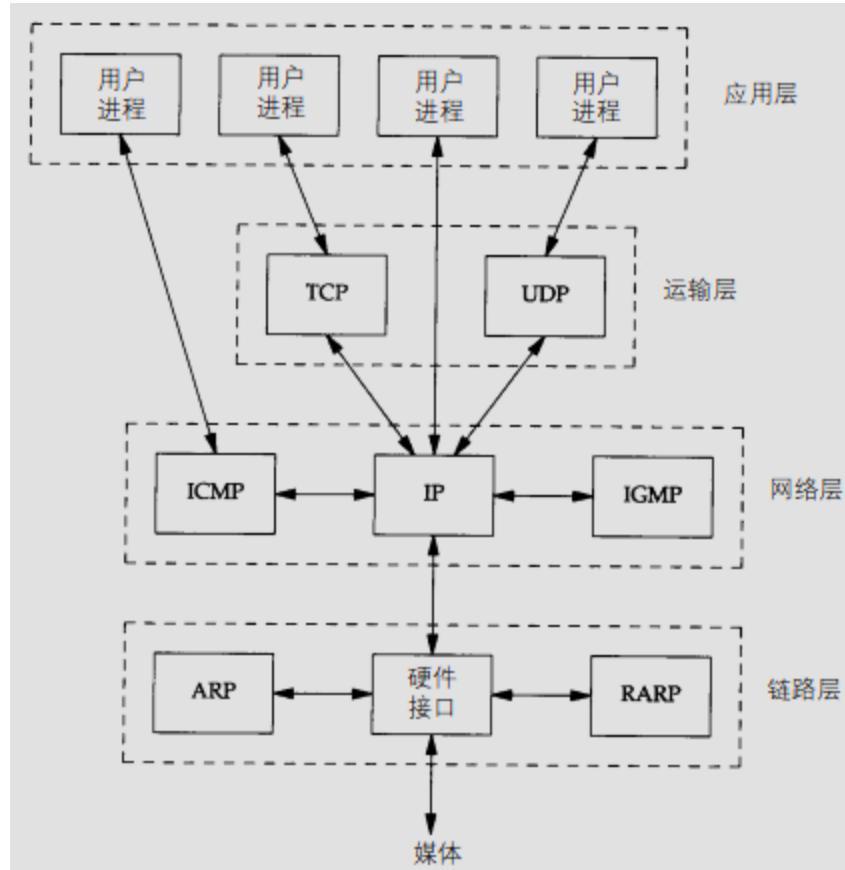


image\_1.png

通过路由器(Router)连接的两个网络

在TCP/IP协议族中，网络层IP提供的是一种不可靠的服务。

## TCP/IP的分层



image\_2.png

### TCP/IP协议族中不同层次的协议

虽然TCP使用不可靠的IP服务，但是它却提供一种可靠的运输层服务。

UDP为应用程序发送和接收数据报。

IP是网络层上的主要协议，同时被TCP和UDP使用，TCP和UDP的每组数据都通过端系统和每个中间路由器中的IP层在互联网中进行传输

ICMP是IP协议的附属协议，IP层用它来与其他主机或路由器交换错误报文和其他重要信息。

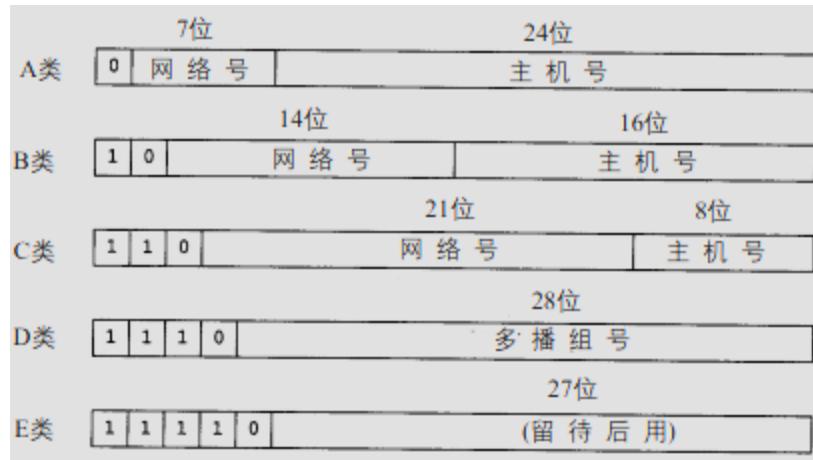
IGMP是Internet组管理协议，它用来把一个UDP数据报多播到多个主机。

ARP (地址解析协议)和RARP (逆地址解析协议)是某些网络接口使用的特殊协议，用来转换IP层和网络接口层的地址。

## 互联网的地址

互联网上的每个接口必须有一个唯一的Internet地址（也称作IP地址，其长度为32bit）。

IP地址的结构



image\_3.png

类型	范围
A	0 .0.0.0 到 127 .255.255.255
B	128 .0.0.0 到 191 .255.255.255
C	192 .0.0.0 到 223 .255.255.255
D	224 .0.0.0 到 239 .255.255.255
E	240 .0.0.0 到 247 .255.255.255
结论	可以通过观察这些地址的第一个十进制整数来区分各个类型的地址

这些32位的地址通常写成四个十进制的数，其中每个整数对应一个字节——**点分十进制表示法 (Dotted decimal notation)**

**重点：**多接口主机具有多个IP地址，其中**每个接口都对应一个IP地址**。

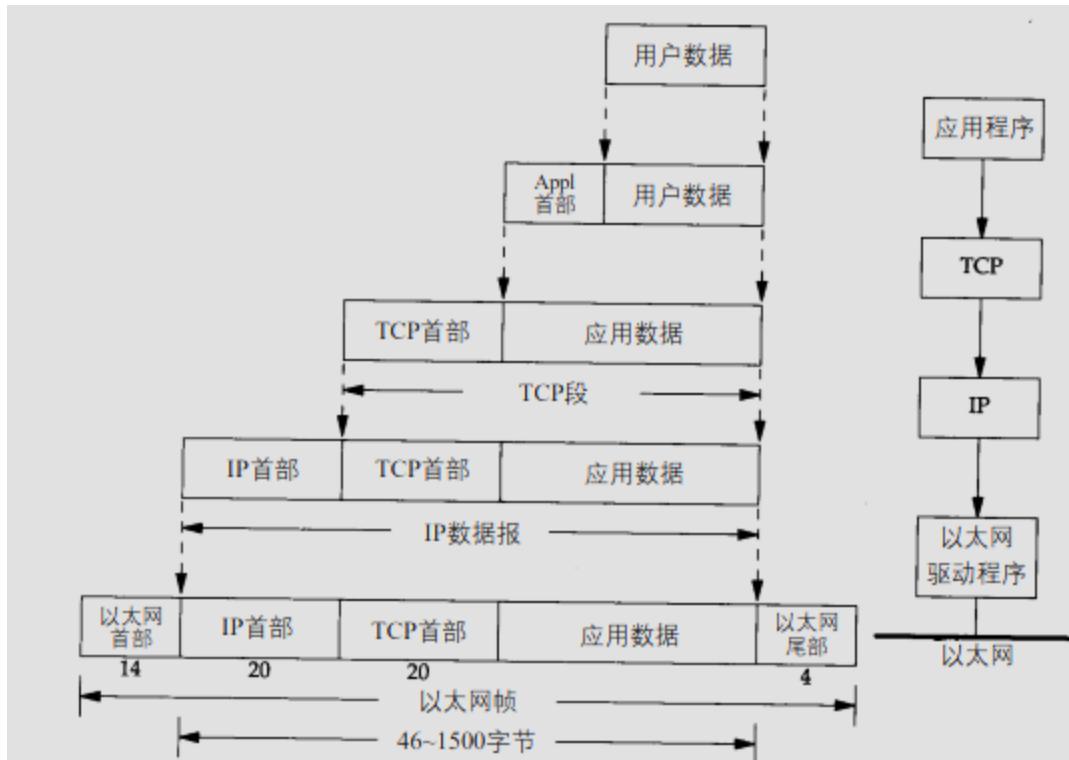
IP地址类型	目的端
单播地址	单个主机
广播地址	目标网络上的所有主机
多播地址	同一组内的所有主机

## 域名系统

在TCP/IP领域中，**域名系统(DNS)**是一个分布的数据库，由它来提供IP地址和主机名之间的映射信息。

## 封装

当应用程序用TCP传送数据时，数据被送入**协议栈**中，然后逐个通过每一层直到被当作一串比特流送入网络。



image\_6.png

TCP传给IP的数据单元称作TCP报文段 (TCP段, TCP segment)。

IP传给网络接口层的数据单元称作IP数据报（IP datagram）。

通过以太网传输的比特流叫做帧（Frame）。

以太网数据帧的**物理特性**是其长度必须在46~1500字节之间。

UDP数据与TCP数据基本一致。唯一不同在于UDP传给IP的信息单元称作**UDP数据报（UDP datagram）**，而且**UDP的首部长为8字节**。

由于TCP、UDP、ICMP、IGMP都要向IP传送数据，因此IP必须在生成的IP首部中加入**某种标识**，用于表明数据**属于哪一层**。为此，IP在首部中存入一个长度为8bit的数值，称作**协议域**。

bit值	协议
1	ICMP协议
2	IGMP协议
6	TCP协议
17	UDP协议

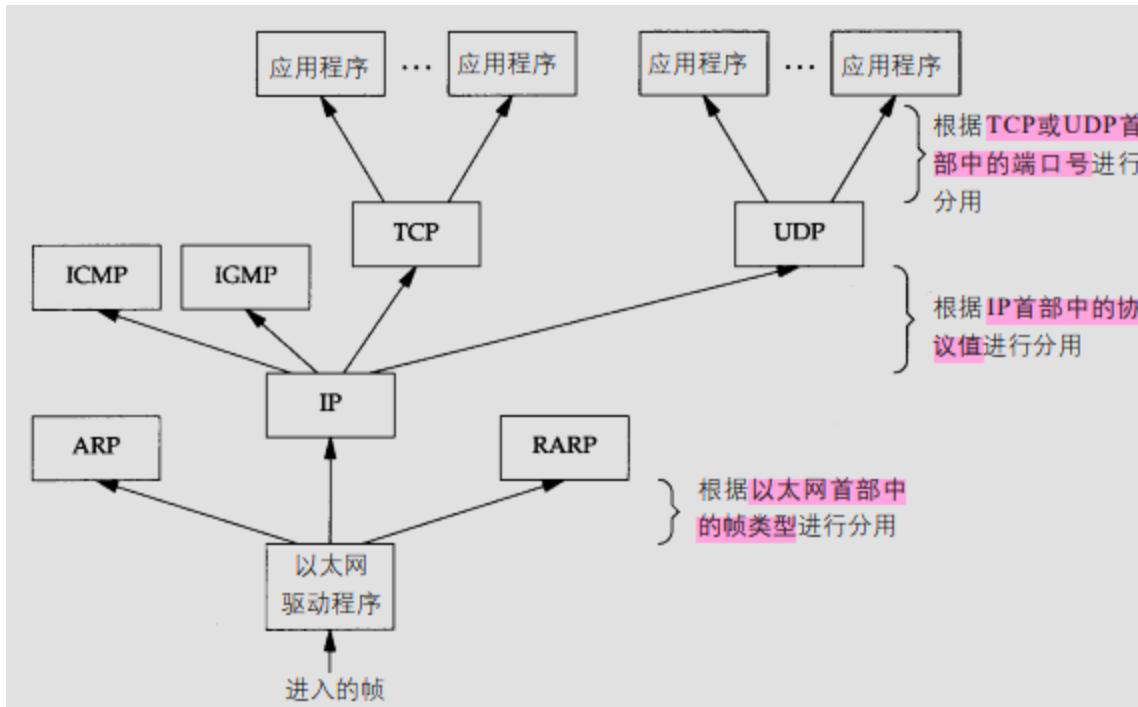
类似地，许多应用程序在使用TCP或UDP来传送数据时，运输层协议在生成报文首部时要存入一个应用程序的标识符。TCP和UDP都用一个16bit的端口号来表示不同的应用程序。TCP和UDP把源端口号和目的端口号分别存入报文首部中。

网络接口在发送和接收IP、ARP和RARP数据时，也必须在以太网的帧首部加入某种形式的标识，以指明生成数据的网络层协议。因此，以太网的帧首部也有一个16bit的帧类型域。

## 分用

当目标主机收到一个以太网数据帧时，数据就开始从协议栈中有底向上升，同时去掉各层协议加上的报文首部。

每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。



image\_7.png

## 客户-服务器模型

模型设计目的：让服务器为客户提供一些**特定的**服务。

服务器类型	工作流程	优缺点
重复型 f or UDP	等待request -> 处理request -> 发送response -> 重复上述流程	缺点：当在服务器处理reques t时，不能为其他用户提供服务
并发型 f or TCP	等待request -> 启动一个新的服务器处理requ est (其余操作由操作系统完成) -> 重复上述流程	优点：可以根据需求生成其他服务其为用户提供服务

## 端口号

服务器一般都是通过知名**端口号**来识别的。

名字	TCP端口号	UDP端口号	RFC	描述
echo	7	7	862	服务器返回客户发送的所有内容
discard	9	9	863	服务器丢弃客户发送的所有内容
daytime	13	13	867	服务器以可读形式返回时间和日期
chargen	19	19	864	当客户发送一个数据报时，TCP服务器发送一串连续的字符串流，直到客户中断连接，UDP服务器发送一个随机长度的数据报
time	37	37	868	服务器返回一个二进制形式的32bit数，表示从UTC时间1900年1月1日午夜至今的秒数

从上表可以看出，当使用TCP和UDP提供相同的服务时，一般选择相同的端口号。

注：之所以端口号都是奇数，是因为这些端口号都是从NCP端口号派生出来的（NCP，即网络控制协议，是ARPANET的运输层协议，是TCP的前身）。NCP是单工的，不是全双工的，因此每个应用程序需要两个连接，需要预留一对奇数和偶数端口号，当TCP和UDP成为标准的运输层协议时。每个应用程序只需要一个端口号，因此就使用了NCP中的奇数。

## 互联网

internet的意思是用一个共同的协议族把多个网络连接在一起。而Internet指的是世界范围内通过TCP/IP互相通信的所有主机集合。

Internet是一个internet，但internet不等于Internet。

## 小结

1. TCP/IP协议族分为四层：链路层、网络层、运输层和应用层，每一层各司其职。

2. 在TCP/IP中，网络层和运输层之间的区别是最关键的：网络层（IP）提供点到点的服务，而运输层（TCP和UDP）提供端到端服务。
3. 一个互联网是网络的网络。构造互联网共同的基石是路由器，它们在IP层把网络连在一起。
4. 在一个互联网上，每个接口都用IP地址来标识，尽管用户习惯使用主机名而不是IP地址。
5. 域名系统为主机名和IP地址之间提供动态的映射。端口号用来标识互相通信的应用程序。服务器使用知名端口号，而客户使用特定的端口号。

## 第二章 链路层

链路层主要的三个作用：

1. 为IP模块发送和接受IP数据报
2. 为ARP模块发送ARP请求和接收ARP应答
3. 为ARP发送RARP请求和接收RARP应答

TCP/IP支持多种不同的链路层协议，这取决于网络所使用的硬件。

### 以太网和IEEE 802封装

以太网采用一种称作CSMA/CD的接入方法，即带冲突检测的载波侦听多路接入，其速率为10Mb/s，地址为48bit。

### 尾部封装

#### SLIP: 串行线路IP