

C++编程基础

2020年2月27日 22:24

4.10 格式化控制台输出

这些函数叫做流操作，包含在 `<iomanip>` 头文件中

(1) `setprecision(n)` 操作：设定浮点数精度 eg: `number = 12.2234;`

`cout<<setprecision(3)<<number<<endl;` 则输出为12.2

(2) `fixed`：修改操作：强制数字以非科学计数法的形式，显示小数点后的位数，默认修复六位，和`setprecision`操作联合起来用 eg: `cout << fixed << setprecision(2);` 显示的数字都是两位小数

(3) `showpoint`操作：强制显示小数点 eg: `cout << setprecision(4); cout << showpoint << 1.23; cout << showpoint << 12.3;` 那么就显示1.230 和12.30

(4) `setw (width)` 操作：默认右对齐，指定输出的最小列数：如果本身的字符大于width，则在前面补充空格；反之自动增加width为本身字符数

(5) `left`操作：输出左对齐 eg: `cout<<left; cout<< <<endl;` 那么下面的输出就是左对齐

(6) `right`操作：输出右对齐

注意：精度的优先级比宽度高

4.11 简单的文件输入输出

(1) 写入文件：

首先声明一个 `ofstream` 类型的变量： `ofstream output;`

调用output对象的open函数： `output.open("numbers.txt");` 这样就创建了一个txt文件。

如果可以选择，可以创建一个输出对象并打开这个文件： `ofstream`

`output ("numbers.txt");`

写入文件： `output << 95<< " "<<56<< " "<<34<<endl;` 这样就写入了这三个数字

关闭： `output.close();`

(2) 读取文件：

首先声明一个 `ifstream` 类型的变量： `ifstream input;`

调用input对象的open函数指定一个文件： `input.open("numbers.txt");`

创建一个文件输入对象并打开这个文件： `ifstream input ("numbers.txt");`

读取数据: `input>>something;`

关闭 `input.close () ;`

5.2.6 输入和输出重定向

(1) 输入重定向

命令： `sentinelValue.exe < input.txt`

从文件input.txt获取输入而不是在键盘输入

(2) 输出重定向：把输出发送到一个文件中，而不是在控制台显示

命令： `sentinelValue.exe < output.txt`

5.2.7 从一个文件中读取所有数据

调用input对象的eof () 函数，来检测是不是到了文件的末尾

```
eg: while( !Input.eof() ){           //continue if not end of the file
    Input >> number;                  //read data
    Cout << number << " ";          //display data
    Sum += number;
}
```

浮点数不能比较大小，不能用 ==

定义数组，数组的大小要比里面装的东西要多，因为要加入空格结束

一、Thinking in OOP

2020年2月27日 22:24

- 对象： 相关的数据以及跟这些数据有内在联系的操作（过程 / 功能 / 行为 / 服务）组成的一个单元
- 类 (class)： 用于描述特性相同或相似的一组对象的结构（用于描述数据）和行为。
- 继承 (inherit)： 将类组织成层次，允许共享结构和行为（代码重用）
- 数据抽象： 经过对客观问题的分析，将数据结构及作用于该数据结构上的操作组成一个实体—对象，比如洗衣机对象。这个过程就是数据抽象。
- 抽象数据类型： 对具有相同或相似属性和行为的数据抽象实体的共同描述。创建类就是构造抽象数据类型

程序由一组相互通信的对象组成。

主控 对象 main ()

用面向对象方法解决问题的步骤：

- ◇ 定义类（包含 了 从特殊 到一般 的归纳思维过程）
- ◇ 创建类的实例（包含 了 从一般到特殊 的演绎思维过程）
- ◇ 为类建立类的等级（类层次）**基类（父类）和派生类（子类）** 派生类对基类进行公有继承

发现对象、创造对象和使用对象

面向对象系统的三个特性：

- 封装性： 指将数据和与这些数据相关的操作集合放在一起，形成一个能动的实体的对象
- “对象（类）”的特征：
 - 对象有标识，有一组状态用来描述它的特征，有一组操作来决定该对象有哪些行为或功能。
 - 对象的所有私有数据，操作的实现代码是隐蔽的，外部是不可见的。
 - 具有一组公共接口，其它对象只能通过公共接口来向该对象发消息。
 - 对象是主动的，执行服务的主动权在对象本身。接受消息的对象可以立即响应，也可不立即响应，这样就使程序的并发有了可能。
 - 类是生成实例对象的样板，是实例对象的加工厂。
 - 类兼有模块和类型定义的优点。

类的封装性使程序易维护，稳定性很好，代码重用性好（实例重用）。

- 继承性： 建立类的层次。类的层次可以清楚地表达现实世界中事物的分类问题。
 - 派生类自动继承基类的特性、数据和操作。
 - 允许在派生类中增加新的数据和操作。
 - 如果在派生类中对从基类继承下来的某些特性重新做了描述，则派生类中的这些特性将以新描述为准，低层的特性将屏蔽高层的同名特性。
 - 每 且宗的特性可以一代一代传到最新派生的子类。 -
 - 继
- 多态性： 指一个对象具有多种形态的能力

c++ 中，多态性体现在：
重载 { 函数重载
运算符重载
多态指针：在运行时，可以指向不同类型的对象。

文件结构：

每个c++类由两个文件实现，头文件保存类的声明，定义文件保存类的实现

Head File

- 头文件开头处的版权和版本声明 (to see about_head.dsw)。

- 预处理块

- 所有头文件都应该使用 #define

- 函数、类和结构声明等。

防止头文件被多重包含 (multiple inclusion)。

- 头文件应该只用于 声明对象、函数声明、类定义、结构定义类模板定义、typedef 和宏，而 不应该包含或生成占据存储空间的对象或函数的定义

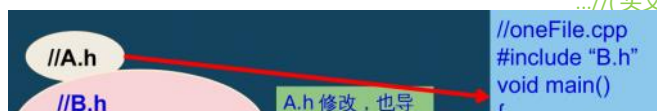
当头文件第一次被包含时，它被正常处

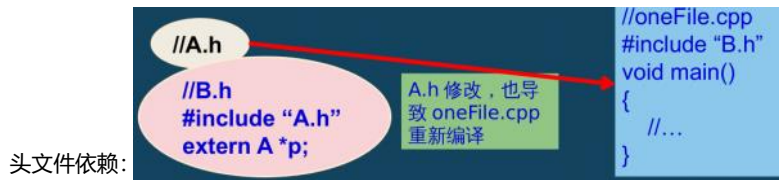
理，符号 _HEADNAME_H 被定义。如果头文件被再次包含，通过条件编译，它的内容被忽略。

#define 保护

- 符号 _HEADNAME_H 按照被包含头文件的文件名进行取名，以避免由于其他头文件使用相同的符号而引起的冲突

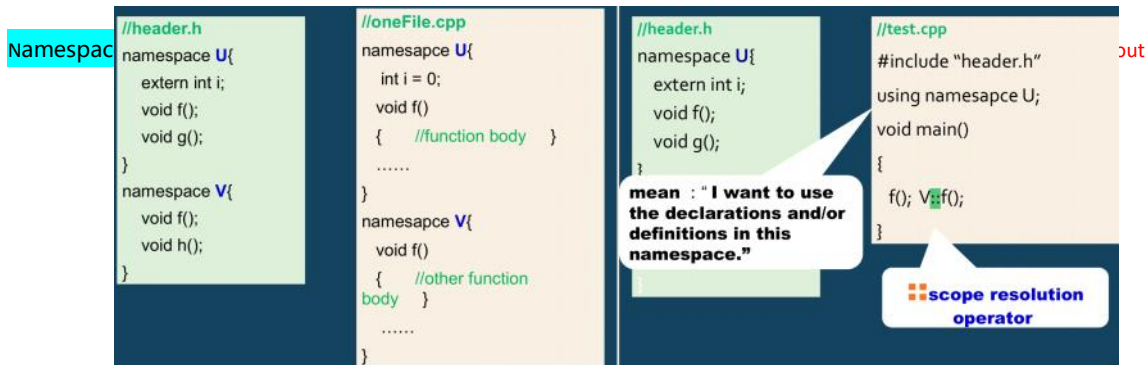
```
#ifndef _HEADNAME_H
#define _HEADNAME_H
...// (头文件内容)
```





应尽量少的包含头文件。可以使用前置声明 (forward declarations) 以尽量减少 .h 文件中 #include 的数量

使用前置声明可以显著减少需要包含的头文件数量。举例说明: 如果头文件中用到类 File, 但不需要访问 File 类的声明, 头文件中只需前置声明 class File; 而无须 #include "file/base/file.h"



Namespace

```
namespace name{
    //variables, functions, classes
}
```

:: 是一个新符号, 称为域解析操作符, 在 C++ 中用来指明要使用的命名空间。

Namespace 解决了名字重复, 同名的变量名或者函数名可以放在不同的名字空间

两种方法使用名字空间的函数与变量, (1) using namespace U; (2) V::f(); 推荐第二种

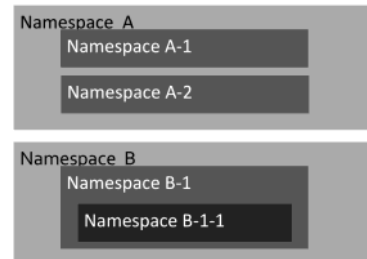
using namespace std; 然后使用 cout, 或者是 std::cout; 意思是某某名词空间的某某对象或者函数

C++ 里面有 "The global namespace" 全局名词空间的含义, 名词空间可以嵌套

甚至可以有 unnamed space 匿名名词空间, #include <stdio.h> 头文件包含即可

全局命名空间就是全局作用域, 访问其中的变量可以直接用变量名或者 :: 变量名访问。

匿名命名空间由于没有名字, 只能直接以变量名的形式访问其中的变量。



实例

```
#include <iostream>
using namespace std;

// 第一个命名空间
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}

// 第二个命名空间
namespace second_space{
    void func(){
        cout << "Inside second_space" << endl;
    }
}

using namespace first_space::second_space;
int main ()
{
    // 调用第二个命名空间中的函数
    func();

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside second_space
```

c++里的I/O流

在C语言中，对字符串的操作容易出错

```
char* p = "I am an student!"; //技术性错误
cout << p << endl;
*p = 'i'; //此处设置断点 I

char* q = (char*)malloc(sizeof(char)*20);
strcpy(q, "I am a student!");
cin.getline(q, 20); I
cout << q << endl;
q++;
*q = 'i';
cout << strcat(name, q) << endl;
```

cin >> lvalue; // lvalue 必须是左值表达式

用法：

cout << Expressions;

- ① 表达式的类型必须是基本数据类型
- ② 不能是 void
- ③ 若是指向 char 的指针，所插入的是一串字符，遇到空格就停止
cin.getline() 可以读取一行字符串

iostream VS stdio.h

C++ 的 流类较之 C 语言的输入 / 输出库函数具有更大的优越性：

- 它是类型安全的，防止用户输入 / 输出数据与其类型不一致的错误。编译器会静态知道 I/O 的对象的型别，而不是动态地由 “%” 查知
- 利用运算符重载，用户定义的类型也可以与内定义类型采用相同的输入 / 输出格式，即 单界面、多方法。

如， **matrix a(2,3);**

std::cin >> a;

std::cout << a;

iostream 库可移植性不如 stdio.h，所以 c++ 写代码的时候也可以用 C 里面的输入输出

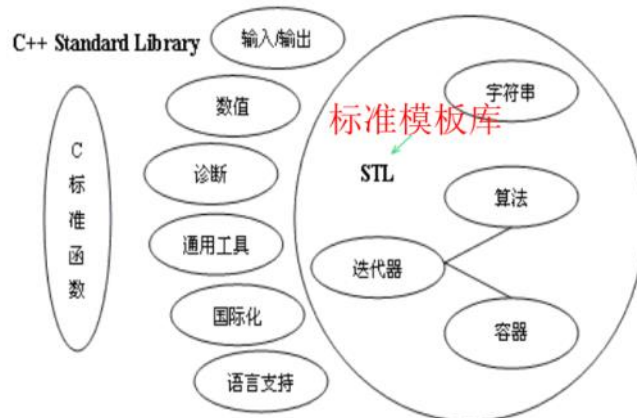
- 书写简单、清晰，程序代码具有很好的阅读性

效率方面，iostream 程序库不如 stdio.h，产生可执行文件尺寸大执行慢

在c++中 #include<string>

c++标准库

c++标准库包含



C 标准函数库基本保持了与原有 C 语言程序库的良好兼容，尽管有些微变化。在 C++ 标准库中存在两套 C 的函数库：

- (1) 标准 C 头文件如 `<stdio.h>` 继续被支持。头文件的内容不在 `std` 中。

`#include <math.h>` // 传统模式或兼容

- (2) 具有 C 库功能的新 C++ 头文件具有如 `<cstdio>` 这样的名字。它们提供的内容和相应的旧 C 头文件相同，只是内容在 `std` 中。

`#include <cmath>` // 标准模式

`using namespace std;`

[网上关于c++头文件的描述链接](#)

STL 主要包含了容器、算法、迭代器（相当于指针）、字符串

- 容器：容器比数组更强大更灵活，可以动态增长（也常是缩减），可以管理属于它们自己的内存，可以跟踪它们拥有的对象数目，可以限制它们支持操作的算法复杂度等等。

标准 STL 容器分成两类：序列容器和关联容器。

序列容器：向量 (vector) 容器、字符串 (string) 容器、双端队列 (deque) 容器和列表 (list) 容器。

- 尽量使用 `vector` 代替数组的使用；用 `string` 来代替字符数组的使用

```
vector<int> a;  
string s("hello");
```

关联容器：集合 (set)、multiset、map 和 multimap

二、The C in C++

2020年3月23日 16:52

2.4 About static elements

2.5 References and The copy-construction

```
enum BOOL{false,true};  
enum BOOL a;
```

顶多就是这样改善一下

```
typedef enum { INT,DOUBLE,CHAR} DATATYPE;  
DATATYPE a;
```

2.6 Memory manage

直接用类型名就可以

2.7 Reading and writing files

2.1 About variable and type

2020年4月19日 23:01

2.1 About variable and type

(1) 在c++中，变量随用随声明，例如： `for (int i = 0; i < 100; i++)`

(2) 变量声明：

在c中：枚举和结构声明的时候

在C++中，

```
enum DATATYPE{INT,DOUBLE,CHAR};  
DATATYPE a;
```

声明：将名称(标识符)引入编译器。它告诉编译器 “这个函数或这个变量存在于某个地方，它应该是这样的。

定义：在这里设定这个变量或者在这里设定这个函数

(3) 范围和可见性

Block scope

File scope

Global scope (全局作用域)

Class scope (类作用域) 类A里面任何函数都可访问A里面的任何数据

(4) 生命周期

本地生命周期 (Local lifetime)：用 `auto` 或者 `register` 定义

全局生命周期 (Global lifetime)：用 `static` 或者 `extern` 定义

(5) 数据类型

built-in data type (内嵌数据类型) 比如 `int double char` 等等

Abstract (抽象数据类型) 比如 `class`

C++里的指针：尽量避免指针互相赋值。指向不同类型的指针是不能赋值的。

相比之下，c里面可以通过中间指针 `void* p` 完成不同类型指针的互相赋值，这很危险

转换类型：在C中，`pd = (double*)pv`编译通过但不安全

C++中，可以用 `pd = static_cast<double*>(pv);` `static_cast<double*>` 是静态转换符

2.2 About functions

2020年4月19日 23:02

2.2 About functions

(1) function prototype 函数原型

C++里面，所有函数必须要定义，必须要包括参数的类型和个数（最好把形参名也写上）

eg: `float vector_sum(float vector[], int size);`

必须：如果没有参数，则应将其声明为参数类型为void

好处：类型安全、方便具体实现的隐藏

(2) Function Signature (函数签名)

签名规则：来进行身份识别的一种手段。在编程语言的设计中，必须定义出识别两个不同实体的规则

C：通过函数名来区别两个函数，不允许在同一个编译/链接单元内出现两个同名的函数

C++：支持函数重载，一个函数名可以被赋予不同语义，提供不同的实现，

签名规则扩展为：函数名 - 参数数量（按照顺序，每一个参数的类型（包括类型修饰符））

eg: `void foo(void) { ... }` 和 `void foo(int a) { ... }` 不一样

(3) 在C++里面调用C的函数

(4) Default arguments 缺省参数值

注意：①缺省值只能出现在函数声明（通常放在头文件），不能出现在函数定义。调用函数的时候，从左到右和参数对应

```
int f(int, int, int = 3, int = 4); //函数声明的时候确定了第三个参数和第四个参数的缺省值
```

```
f(1, 2, 3); //只有第四个的缺省值有效，为4，前面三个用调用函数的时候的参数值
```

```
f(1, 2); //第三个和第四个有缺省值
```

```
f(2, 4, 6, 8); //缺省值都没有用到
```

②在构造函数中加入缺省参数值，那么cpp文件里面的没有缺省参数的调用构造函数也可以用这个

③缺省值和所需的参数不能交叉使用，缺省值在后面

```
int h(int, int, int = 3, int = 4); // RIGHT!
```

```
int h(int, int = 3, int, int = 4); // WRONG
```

④缺省值不能依赖于其他参数的值，但可以是复杂的表达式或函数调用

```
int h(int i, int = i * 5); // 错
```

```
void delay(int k, int time=f(5)); //对
```

⑤函数的声明、定义和调用都不采用缺省参数。（其实上面讲的一些东西记住就行，但别用）

(5) Function Overloading (函数重载)

因为函数签名的存在，所以C++里面不同函数可以有相同的名字

注意：

①参数列表可以区分函数

```
float abs(float) { //..... }
```

```
real abs(real) { //..... } //这两个函数不同
```

②返回值不能够区分重载函数

```
int process(int) { //..... }
```

```
float process(int) { //..... } //编译出错
```

③const也可以重载函数

```
Class A{  
public:  
    void foo();  
    void foo() const;  
    //...  
}; //函数重载
```

```
void foo(A*);
```

```
void foo(const A*);
```

④类里面的函数重载

构造函数重载

成员函数重载

⑤使用函数重载的情况

一般来说，如果可以选择一个合适的缺省值并且只是用到一种算法，就使用缺省参数。否则，就使用函数重载。

算法取决于给定的输入值。这种情况对于构造函数很常见：

“缺省”构造函数是凭空（没有输入）构造一个对象，而拷贝构造函数是根据一个已存在的对象构造一个对象。

```
//default constructor      //copy constructor
complex::complex()         complex::complex(const complex& obj)
{
    realPart = 0.0;         {
    maginaryPart = 0.0;     realPart = obj. realPart ;
                           maginaryPart = obj. maginaryPart;
                           }
}
```

(2) inline function

怎么设计内联函数？

- (1) **inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。**

关键字inline 必须与函数定义体放在一起才能使函数成为内联，仅将inline 放在函数声明前面不起任何作用。

一般情况下：

```
inline void Foo(int x, int y); //内联函数声明
inline void Foo(int x, int y) //内联函数定义
{
    //.....
}
```

等价于：

```
void Foo(int x, int y); //内联函数声明
inline void Foo(int x, int y) //内联函数定义
{
    //.....
}
```

而如下函数Foo 有可能成为内联函数：

```
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
{
    //.....
}
```

but:

如下函数Foo肯定不能成为内联函数：

```
inline void Foo(int x, int y); // inline 仅与函数声明放在一起
void Foo(int x, int y)
{
    //.....
}
```

- (2) **inline函数源代码对编译器而言必须是可见的，以便它能够在调用点展开该函数。**

(1)

```
//file.h
inline int f(int i);
```

//file.cpp-->file.obj，没有编译错误

```
inline int f(int i)
{
    return i+8;
}
```

```
//main
#include "file.h"
void main()
{
```

cout << f(9); //编译器忽略inline，是普通函数调用（因为想要替换，但是此时内联函数函数体在obj文件里

面，即0101代码，是不可见的)
}

解决方法：把内联函数的定义放在头文件里。

(2)

//main.cpp (都在cpp文件里面 但是还是会有问题)

inline int f(int i); //内联函数声明

int main()

{

cout << f(9); //编译器忽略inline，是普通函数调用

}

inline int f(int i) //内联函数定义

{

return i+8;

}

解决方法：把内联函数的定义放在该函数调用之前 (因为c++是从上往下编译运行)

类的成员函数成为内联函数的方法：

法1：如果在 .h文件类的定义里面直接写了函数的定义，那这个函数就是该类的隐式内联函数

法2：在函数体里面直接给出显式声明，再在类定义外面在后面写出显式定义（就是inline..），这都在.h文件中

以下情况不宜使用内联：

- a)如果函数体内的代码比较长（超过20行），使用内联将导致内存消耗代价较高；
- b)如果函数体内出现循环、switch，那么执行函数体内代码的时间要比函数调用的开销大；
- c)假如在程序中,隐含或显式取该函数的地址,则.....
- d)在程序的Debug版本,不执行内联

2.3 About constant

2020年4月19日 23:03

2.3 About constant

一、C编码规范

- 要避免在代码中使用 诸如 3.1415926 这样的字面常量，而应#define PI 3.1415926该用符号名称和表达式替换它们

#define PI 3.1415926 预编译器创建一个常量

这样做的问题：

宏常量只是一个符号

不占用存储空间

没有类型检查

预处理器值替换带来了一些微妙的问题

宏常量具有全局作用域、外部链接

二、C++中

(1)

```
const int size = 10; //internal linkage
```

```
extern const int L = 100; //external linkage
```

对于extern (外部) 的常量，在其他文件里面也要用extern

(2) 在C和C++中，定义数组的时候，必须是常量

```
int size = 20;
```

```
float a[size]; //error
```

```
const int size = 20;
```

```
float a[size]; //ok
```

(3) 使用const的方法

1、用于指针

```
const int *p;
```

读法：从右往左理解 (p是一个指针，指向整数，而且用const修饰，不能通过该指针改变整数的值)，即只能读不能写

意义：p是一个指向int的指针，但不能通过p改变它(p指向的数据)，否则会编译报错。只能读不能写

```
void f1(const int* p) {  
    *p = 0; // compile-time error  
}
```

p可以是一个常量对象的地址，也可以是普通对象的地址，也就是说，p的值可以被改变

```
const int b = 10;
```

```
int a = 20;
```

```
const int *p = &b; //p points to int or const int
```

```
cout << *p; //only read by p,
```

```
*p = 1; //can't write it. Compile-time error
```

```
p = &a; //p的值是可以变的
```

```
cout << *p; //only read by p,
```

```
*p = 1; //can't write it. Compile-time error unit two/pointers/constpointers.c
```

const pointer (常量指针)

```
int d = 1;
```

```
int* const p = &d; //说明p是一个常量指针，指向一个整数
```

```
*p = *p + 1; //p = new int; compile-time error
```

也就是说，常量指针不能动来动去

```
int * const p = (int*) malloc(10*sizeof(int));
```

```
int * q = p; //通过q 遍历堆中的10个整数 个整数
```

```
p++; // compile-time error
```

```
q++;
```

```
free(p);
```

```
int d = 1;
```

```
const int* const p = &d; //meaning : "p is a pointer, which is const, that points to an int, by p can't change it."
```

例子：

```
int sum(const int *a, int size)  
{
```

```
    const int *p = a;
```

```
    const int * const end = a + size - 1; //end就是指向最后一个元素的指针
```

```

        int total = 0;
        for ( ; p <= end; p++)
        {
            total += *p;
        }
        return total;
    }

```

2、用于函数参数和函数返回值

(1) 用在参数

```

void f1(const int i)  const常值传值容易出问题
{
    i++;    // compile-time error
}

```

```

void g(const object *p) //说明编译器保证：p所指向的对象没有被修改
{
    .....
}

```

(2) 用在返回值

```

Returning by const value
const int g()
{
    return 1;
}

const char* v()
{
    return "result of function v()"; //返回静态字符数组的地址
}

```

对于内置类型，是否按值返回const并不重要。在处理用户定义类型时，按值返回const非常重要。

3、用于数据成员和成员函数

```

class A{
    const int size;    //const data member
public:
    int sum() const;    // const member function,修饰成员函数，因为放在前面代表修饰返回值，放在括号里是修饰参数。所以const只能放在后面来修饰整个函数
};

```

C/C++对数组的处理是非常有效的。数组和指针能非常和谐地在一起工作，使用指针要比使用数组下标快两倍。

- 但是很多编译器(如ANSI/ISO C)没有对使用越界下标的行为作出定义。一个越界下标有可能导致：

- 程序仍能正确运行；
- 程序会异常终止或崩溃；
- 程序仍能运行，但无法得出正确的结果；
- 其它情况；

- 小心使用数组最后一个元素的地址。

- 当把数组作为函数的参数时,必须采取适当的机制告诉函数数组参数的大小。

```
void sort(int b[], int size);
```

函数的数组参数仅相当于指向该数组第一个元素的指针，即与void sort(t int *p, t int size); 等价
数组传递就是地址传递。

- 但是

```
int strlen(char s[]); //这样的就不用告诉数组参数的大小，因为字符数组有结束标志
```

(1) 在数据成员

```

class array{
public:
    array(int z) : size(z) //size(z)是构造函数初始化列表
    { //... }
}

```

```
private:
const int size;    //在一个对象生命周期中，这个数据成员是常量，每个对象有一个不同的常量值
```

构造函数初始化列表：这是为了提醒，列表中的初始化发生在任何主构造函数代码执行之前
必须使用构造函数初始化器列表初始化const常量数据成员。

eg:

```
class C{
private:
    int n;
    const int cint;
    int& rint;
public:
    C(int param) : cint(5), rint(n) //把5给整数常量cint。把n给rint
    { n = param; }
};
public:
C(int param) : n(param), cint(5), rint(n) 引用、常量必须在构造函数初始化列表初始化，其他数据成员尽可能也放在这里
{ }
};
```

先进行构造函数初始化列表的运行，再进行构造函数的运行

```
int size = 20; float a[size]; //语法错误
const int size = 20; float a[size]; //正确
```

(2) 在成员函数

在.h文件中:

```
#ifndef _ARRAY_H
#define _ARRAY_H

class Array{
public:
    Array(int size = 10);
    Array();
    //int getSize() 这行代码可以不要
    int getSize() const; 这里和cpp文件中可以都只保留常量版的，因为是只读的函数
    void print();        //非常量版的成员函数，只有常量版的成员函数才能对常量数据成员操作
    void print() const;  //常量版的成员函数，常量版和常量版的函数是重载关系
    int& operator[] (const unsigned i);
    int operator[] (const unsigned i) const; //这两个是重载函数，但是都要保留。
    //因为第二个是值返回，返回的是数值，第一个返回的是单元本身，为了不至于把所有对象都暴露给客户，所以有时会使用第二个更好一点

    friend istream& operator>>(istream& in, Array& ob);
    friend ostream& operator<<(ostream& out, const Array& ob);

private:
    int* const p; //const pointer
    const int size;
    Array(const Array&);
    Array& operator=(const Array&);
};
#endif

void Array::print();
void Array::print() const;
是重载关系，可以只保留const版的，但具体情况要具体分析。如下面两个都要保留
int& Array::operator[] (const unsigned suffix );
int Array::operator[] (const unsigned suffix) const;
```

在cpp文件中:

```

Array::Array(int size):size(size), p(new int[size]) //构造函数初始化列表
{
    /*
    this->size = size;
    p = new int [size];
    */
    memset(p, 0, size * sizeof(int));
}
delete []p //在析构函数中删除指针p指向的那个数组里面的所有东西
delete p //只删除第一个

```

```

class bob{
private:
const int size = 20;
int array[size]; // 出错，说明动态数组不可以这么构建
};
解决方法：
class bob{
private:
enum {size = 20}; 用枚举
int array[size]; //ok

```

4、用于类的定义

顶多就是这样改善一下

```

typedef enum { INT,DOUBLE,CHAR} DATATYPE;
DATATYPE a;
直接用类型名就可以

```

```

enum BOOL{false,true};
enum BOOL a;

```

```

int main()
{
    //类似常量引用, 指向常量的指针 (pointer to const) 不能用于改变其所指对象的值
    //要想存放常量对象的地址, 只能使用指向常量的指针
    const double pi = 3.14159;
    double *ptr = &pi; //错误, ptr只是一个普通指针
    const double *cptr = &pi; //正确
    *cptr = 42; //错误
    //指针类型必须与其所指对象的类型一致
    //此处一个例外是允许令一个指向常量的指针指向一个非常量对象
    //结果是不能通过指向常量的指针修改它指向的对象的值
    double dval = 99.8; //非常量对象
    cptr = &dval; //cptr是指向常量的指针
    //指针本身和其他对象一样允许被定义为常量
    //常量指针必须在定义时初始化, 之后就不能发生改变, *放在const之前说明指针是一个常量
    //巧辩: const 离指针名近时这个指针是常量指针
    int errNumb = 0;
    int *const curErr = &errNumb; //curErr将一直指向errNumb
    const double *const pip = &pi; //pip是一个指向常量对象的常量指针
    int i2 = 0;

    //练习
    //判断下面的初始化是否合法
    int i = -1, &r1 = 0; //非法, 引用类型的初始化类型必须是一个对象
    int *const px = &i2; //合法, p2将一直指向i2
    const int i = -1, &r2 = 0; //合法, 初始化常量引用可以使用容易表达式作初始值
    const int *const py = &i2; //合法, 指向常量对象的常量指针

```

```

const int &const r3;//非法, 未初始化
const int i2 = i, &r = i;//合法
int i, *const cp;//非法, 定义常量指针, 未初始化
int *p1, *const p2;//非法, p2是常量指针, 未初始化
const int ic, &r = ic;//非法, ic是一个常量对象, 未初始化
const int *const p3;//非法, p3是一个常量指针, 未初始化
const int *p;//合法, p3是一个指向常量的指针, 它无需初始化
i = ic;//合法
p1 = p3;//非法, 不能把const int*的值分配给int*实体
p1 = &ic;//非法, 同上
p3 = &ic;//非法, p3是常量指针, 在定义之后不能再修改
p2 = p1;//非法, 同上
ic = *p3;//非法, 同上
}

```


2.4 About static elements

2020年4月19日 23:37

一、C里面

尽可能少用全局变量，使全局变量模块私有化

static double classFee = 0.0; //班费 这是静态全局变量

double classFee = 0.0; //班费 这样写不好，因为所有文件都会用到这个classFee

```
void oneFun()
{
    static int count = 0; //静态局部变量，静态的不会被释放，第二次执行的时候这一行不会被执行
    count++;
    std::cout << "The " << count << "the times call me." << std::endl;
}
```

按存储区域分：

- 1、全局变量、静态全局变量和静态局部变量都存放在内存的全局数据区
- 2、局部变量存放在内存的栈区

按作用域分：

- 1、全局变量在整个工程文件内都有效；
- 2、静态全局变量只在定义它的文件内有效；
- 3、静态局部变量只在定义它的函数内有效，且程序仅分配一次内存，函数返回后，该变量不会消失；局部变量在定义它的函数内有效，但是函数返回后失效。
- 4、全局变量和静态变量如果没有手工初始化，则由编译器初始化为0。局部变量的值不可知。
- 5、静态局部变量与全局变量共享全局数据区，但静态局部变量只在定义它的函数中可见。静态局部变量与局部变量在存储位置上不同，使得其存在的时限也不同，导致对这两者操作的运行结果也不同

二、C++里面

1、静态数据成员

```
#ifndef STUDENT_H
#define STUDENT_H

class Student{
private://data member,数据成员
    char *name;
    short age;
    char gender[3]; // "男" or "女"
    static double classFee ;//班费--static data member 静态数据成员，是这个类的所有对象共享的，就是所有学生用一个班费
public: //member function ,成员函数
    //constructor,构造函数，负责初始化对象
    Student(const char* name,short age,char gender[]);
    Student();
    //destructor,析构函数，负责销毁对象成员所占用的资源
    ~Student();

    void print() const;
    char* get_name() const;
/*
    if coding
    static double getClassFee() const //查班费 静态成员函数
    {
        return classFee;
    }
    static void setClassFee(double money) //交班费
    {
        classFee += money;
    }
*/
}
```

```

}

每个学生查班费、交班费:
Student zhangsan("张三", 22, "男");
zhangsan.getClassFee();
zhangsan.setClassFee(50);
问题: 班主任怎么查班费?
*/

//if coding
static double getClassFee() //查班费---static member function
{
return classFee;
}

static void setClassFee(double money); //交班费

每个学生查班费、交班费:
Student zhangsan("张三", 22, "男");
zhangsan.getClassFee(); //肯定这样好啊
Student::getClassFee();

zhangsan.setClassFee(50); //肯定这样好啊
Student::setClassFee(50)

班主任查/交班费:
Student::getClassFee(); 老师不是学生类的对象, 所以可以用公有区的静态成员函数这样操作
Student::setClassFee(500)
*/
};
#endif

```

注意: 在cpp文件中对静态成员函数定义的时候不可以写static

静态数据成员是这个类的全局变量

静态成员函数是这个类的全局函数

总结:

- 静态成员是局部于该类的全局变量。为该类所有的对象所共享。不论创建多少个该类的对象, 静态数据成员在内存中只有一个拷贝。
- 静态数据成员只是在类中声明, 必须在类外其它地方初始化, 因为它是该类的全局变量。
- 静态成员与运行的程序有相同的生存期, 即使不创建该类的对象, 它亦存在。
- 通常将静态数据成员声明成私有的。在该类的公有区声明一个静态的成员函数, 让外界通过该静态成员函数来访问该类的静态数据。
- 静态成员函数是局部于该类的一个全局函数, 只能访问类的静态数据成员, 但它不是该类的成员函数(没有this指针), 它同友元函数一样, 在类中只有语法上的作用。

2、对类的静态数据成员初始化

直接在cpp文件最前面初始化, 而且不能漏了类型和类名

```

#include "student.h"
#include <string>
#include <malloc.h>
#include <iostream>
double Student::classFee = 0.0;
/*double Student::classFee 与上面效果一样, 因为静态的默认为0*/

```

3、

2.5 References and The copy-constructor

2020年4月19日 23:37

一、References (引用)

引用是已经存在的对象的 别名

```
int count = 0;
int &refcount = count;
refcount = 1; //here, count=1
count++; //here, refcount=2
```

访问某一个变量：直接访问、指针访问、引用变量访问

要当场引用

二、引用符的应用

1、引用用在函数参数里面

当引用用作函数参数时，对函数内部引用的任何修改都会导致函数外部参数的更改。当然，也可以通过传递指针来做同样的事情，但是引用的语法要干净得多。(如果您愿意，可以将引用看作是一种语法上的便利)

```
#include <iostream>
using namespace std;
void swap(int &a, int &b) //sizeof(int)*1 bytes存储空间，因为a和x一样的地址，b和y一样的地址，只有temp

{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
void main()
{
    int x = 10;
    int y = 20;
    swap(x, y); //引用传递 (实参就是形参，形参就是实参)，双向传递
    cout << "x=" << x << " y=" << y << endl;
}
```

向函数传递参数的方法：值传递、地址传递、引用传递

好处很多

function prototyping → type-safe
Default arguments → code-shorter
Function Overloading → Easy to read
inline Function and & → Improve performance
const → Improve program robustness (鲁棒性)

2、引用用在函数的返回

引用返回 (比如 `return int&`) : 返回的是单元本身 (或者是变量本身)

值返回 (比如 `return int`) : 返回的是单元的值

返回引用 和 返回指针 很像，所以要小心变量的存在域和类型等

```
#include<iostream>
char *f();
char *f1();
int main()
{
    std::cout << f() << std::endl;
    std::cout << f1() << std::endl;
    return 0;
}
char* f()
{
    char s[] = "Hello";
    return s;
}
char* f1()
{
    static char s[] = "World" ;
    return s;
}
```

第一个打印的是乱码，因为s是局部变量，离开则释放空间

第二个打印正确，因为s是静态变量，整个文件范围类都存在

(2) 成员函数不能返回类数据的非常量句柄（句柄就是引用或者地址）

```
class Obj
{
public:
    int& getA() //Non-compliant
    {
        return a;
    }
private:
    int a;
};

void fn(Obj& m)
{
    int& a_ref = m.getA();
    a_ref = 0;          //External modification of private m.a这样就会看到私有数据a，这显然不合情理
}
```

3、使用引用的注意关键点

- (1) 在创建引用时必须对其进行初始化（一创建就要初始化）（但是：指针可以在任何时候初始化）

```
int i;
int& j;  //这样是错误的
j = i;
```

- (2) 一旦一个引用被初始化为一个对象，它就不能被更改为引用另一个对象（但是：指针可以在任何时候指向另一个对象）

```
int i, k;
int& j = i;
int& j = k;  //这是错误的
```

- (3) 引用和被引用的对象必须具有相同的类型

```
int a = 1;
long int& la = a;  //error: “初始化”: 无法从“int”转换为“long &&”无法将左值绑定到右值引用
```

- (4) 通过值传递参数时，参数可以是文字、变量(可能是其他类型)、表达式，甚至是另一个函数的返回值

通过引用传递参数时，参数必须是变量

- (5) 不能有空引用。必须始终能够假设一个引用连接到一个合法的存储块。

```
int a=1;
int &refint1 = NULL;  //这是错误的
int &refint2 = 5;     //错的， because 5 is constant
```

- (6) 重载（不要求考试）

```
int process(int) {...}
float process(const int) {...}  //compile-time error
int process(int&) {...}
float process(int) {...}      //compile-time error

void foo(A*);
void foo(const A*);  // Above functions are overloaded relation
void foo(A&);
void foo(const A&);  // Above functions are overloaded relation
```

4、const references

```
const int &refint2 = 5;  //√这是常数引用
const int &refint3 = a;  //通过这个引用，只能读不能写
```

三、拷贝构造函数

- 1、拷贝构造函数的作用：用一个对象初始化另一个新的对象

```
Big(const Big& b)  //拷贝构造函数（位于类的定义中）
{
    i = b.i;
    d = b.d;
    //buf = b.buf;
    strcpy(buf,b.buf);
    cout << "I am an object of Big,i am in Big(const Big& )" << endl;
}
```

用到拷贝构造函数的三个地方：

- (1) `Big b3(b2)` 或者写成 `b3 = b2` 直接调用拷贝构造函数，用已经存在的对象初始化创建的对象
- (2) 函数调用的时候把实参对象传递给形参对象时，调用拷贝初始化构造函数
- (3) 返回一个对象

```
#include <iostream>
#include <string>
using namespace std;
```

```
class Big{
public:
    char buf[100];
    int i;
    double d;
    Big(){
        cout << "I am an object of Big,i am in Big()" << endl;
        //default constructor
    }
    Big(const Big& b) //copy constructor
    {
        i = b.i;
        d = b.d;
        //buf = b.buf;
        strcpy(buf,b.buf);
        cout << "I am an object of Big,i am in Big(const Big& )" << endl;
    }
    ~Big()
    {
        cout << "I am in ~Big()" << endl;
    }
};
```

```
Big fun(Big temp) { //这是随便写的一个函数，实参对象传递给形参对象时，调用拷贝初始化构造函数
    temp.i = 3; // Do something to the argument
    temp.d = 3.3;
    strcpy(temp.buf,"I am temp!");
    return temp; //函数返回对象时，调用拷贝构造函数初始化一个匿名对象放到寄存器，在函数返回后该匿名对象作为函数返回值
                //函数返回前，调用析构函数把temp销毁
}
```

```
int main()
{
    Big b1, b2;
    b1.i = 1;
    b1.d = 1.1;
    strcpy(b1.buf,"I am b1!");
    b2.i = 2;
    b2.d = 2.2;
    strcpy(b2.buf,"I am b2!");
```

```
    Big b3(b2); // call Big(const Big& b)
```

```
    b2 = fun(b1); //函数返回的匿名对象赋值给b2后，调用析构函数把该匿名对象销毁
    cout << "此处观察....." << endl;
    return 0;
}
```

2、自动生成的拷贝构造函数

- 1、没有人工写的时候就会自动生成一个。是逐域拷贝

有时候会出现问题，是因为类里面有指针成员会出现两个指针指向同一个区域，在析构的时候同一片区域释放两次

2、修正方法

```
Array(const Array& com):size(com.size),p(new int[com.size])
{
    memcpy(p, com.p, size*sizeof(int));
}
```

就是重新向堆里面申请空间，进行转移之后分别释放

- 3、有时候并不需要拷贝构造函数，所以要阻止它自动生成

阻止生成拷贝构造函数的方法：在私有区private声明拷贝构造函数（不需要定义）

启发：有时候因为没有函数体或者因为函数在私有区而无法访问

启发：参考Google C++ Coding Style，用来限制类型复制和拷贝等对class类型变量的误操作，也可以提高性能，减少编译器自作主张为我们生成一些无法预料的代码

```
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
    TypeName(const TypeName&); \
    TypeName& operator=(const TypeNam
```

加一个宏

使用时，请在class的private声明区中加入以下的代码：

```
class A
{
private:
    DISALLOW_COPY_AND_ASSIGN(A)
};
```

2.6 Memory manage

2020年4月19日 23:37

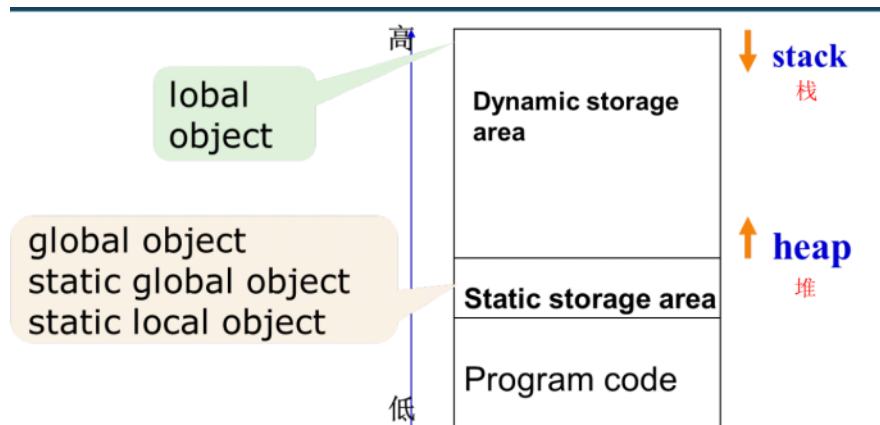
一、内存分配方式有三种:

静态内存分配

自动内存分配

动态内存分配

C始终提供以下函数:malloc()和free(), (以及malloc(): alloc和realloc的变体) 在运行时从堆中分配存储(也称为自由存储)



C's approach to the heap

```
class A{//...};  
A* p = ( A* ) malloc ( sizeof( A ) );  
p->A();          //manager *p ... Sometime later  
p->~A();  
free( p );
```

这都是需要人做的, 人为把对象放在堆里面, 所以不好。

二、对象的创建和对象的析构的顺序是相反的 (先创建的后析构)

2.7 Reading and writing files

2020年4月19日 23:38

三、The basics of C++

2020年3月23日 17:42

3.1 Class

3.2 Composition

3.3 Operator overloading

3.4 Automatic type conversion

3.5 Inheritance

3.6 Polymorphism & Virtual Functions

3.1 Class

2020年5月9日 15:55

3.1 Class

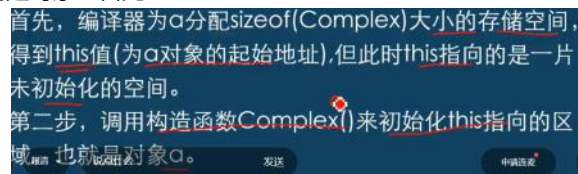
c++里的string class

class

- (1) 安全漏洞：一个类的数据用 private，这个访问权限控制客户端就不能访问（读取和改写），但是客户端通过阅读头文件可以看到什么数据、数据结构，所以还是可以在cpp文件中进行修改。
- (2) 一般来说，成员函数是公有成员，数据是私有成员
- (3) 一个类里面，公有部分和私有部分哪个在前哪个在后一般没什么影响，但是最好把操作写在前面，因为让客户端看到怎么操作的更加重要
- (4) 如果没有写private这个词但是有成员，这是私有的private，这和c语言里面struct正好相反
- (5) 对象名.操作 () 这是用对象名字 p->函数 直接访问 这是通过指向他的指针间接访问
- (6) 一个类的大小只和数据结构所占字节有关，是 \geq 的关系。类定义中的成员函数是全部在一个存储空间的。所有对象共享这些对象函数——

Q: 怎么共享的: c++里面的this指针，这是一个全程变量，c++在运行的时候，任何时候this指针都指向目前激活的对象的地址。只有在有时候需要手动加上this

Q: 怎么创建对象: 首先:



Q: 类的大小 \geq 各个数据成员大小之和

这是因为 1、有对齐的规则，所以成员和成员之间有多余的空间，分别必须是相应对象的字节数整数倍

2、结构体（类）的总体大小必须是占用字节最大的对象的字节的整数倍

3、成员的先后顺序是影响占用内存的（一般来说，按照字节数从大到小，这样就是最优化的）

(7) 构造函数

```
class complex{
public:
    complex(); //default constructor
    complex(double r);
    complex(double real, double imag);

    //complex(double real=0.0, double imag=0.0);

    void print();
    void set_real(double rpart);
    void set_imaginary(double ipart);
    double get_real();
    double get_imaginary();
    complex& operator=(const complex&); //operator overloading
```

需要分别对几个参数进行初始化，就要写相应的构造函数。两个参数、一个参数、无参数的构造函数都要对应写出来

对象创建数组，就要调用默认构造函数

资源回收：堆里面的空间要释放；文件打关闭释放缓冲区——

但是容易内存泄漏（比如有时候free（）命令漏掉一部分释放不完，还有一些其他情况释放不完）

指针赋值要很小心，大部分时候指针不能直接赋值，解决方案是：

```
Student::Student(const char* name, short age, char gender[])
{
    //this->name = name; //使用Student有问题

    this->name = (char*)malloc(strlen(name)+1);
    strcpy(this->name, name);
    this->age = age;
    strcpy(this->gender, gender);
}
```

memcpy:

同一块内存，释放两次，就会出错。

- (1) public 下面的所有成员声明对每个人都是可用的。公共成员就像结构成员。

Private 除了类型的创建者之外，没有人能够访问该类型的函数成员内部的成员。如

果有人试图访问一个私有成员，他们将得到一个编译时错误。

Protected 继承的类可以访问protected的成员，但不能访问private的成员。

[这三者的具体区别请看这个链接](#)

[关于友元函数的介绍链接](#)

完成了信息隐藏

- (2) [define a class](#)

- (1) 声明部分(头文件)

<pre>class Circle { public: Circle(); Circle(double); double getArea(); double getRadius(); void setRadius(double); private: double radius; };</pre> <p>(a)</p>	<pre>class Circle { public: Circle(); Circle(double); private: double radius; public: double getArea(); double getRadius(); void setRadius(double); };</pre> <p>(b)</p>	<pre>class Circle { private: double radius; public: double getArea(); double getRadius(); void setRadius(double); public: Circle(); Circle(double); };</pre> <p>(c)</p>
--	---	---

不做特别说明，类的数据成员和成员函数都被认为是private

如：

```
class lamp
```

```
{
```

```
int number;
```

```
void display();
```

```
public:
```

```
void lower();
```

```
};
```

其中，number 和display()为私有，lower 为公有

来自 <https://zhidao.baidu.com/question/90915361.html>

- (2) 实现部分 (cpp文件)

```
[inline] type className::function_name( parameter list)
{
    function body
}
```

类名
命名空间别漏
成员函数名

- (3) 使用对象

创建对象: `complex c;` //complex是一个类名

操作对象: `c.print();` //执行complex类中的print操作

(4) 对象指针

```
complex a(1.1,2.2);
complex *p = &a;
p->print(); //Make the call using the object pointer
a.print();  // Make the call using the object name
```

(5) this 指针

complex类定义中的成员函数代码存储在某块公用的存储空间中，供该类的所有对象共享——代码共享。

这个指针的值为 此时被调用的对象的地址

工作原理: complex a;

首先，编译器为a分配sizeof(Complex)大小的存储空间，得到this值(为a对象的起始地址),但此时this指向的是一片未初始化的空间。

第二步，调用构造函数Complex()来初始化this指向的区域，也就是对象a。

(5) 对象的大小

对象的大小是其所有成员的大小的总和

可以使用sizeof操作符确定对象的大小。

但是在计算的时候，往往是按照从大到小的类型所占大小来定的，即求和的时候要
求内存对齐

空类的内存为1个字节

1、第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

2、在数据成员完成各自对齐之后，类(结构或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

很明显#pragma pack(n)作为一个预编译指令用来设置多少个字节对齐的。值得注意的是，n的缺省数值是按照编译器自身设置，一般为8，合法的数值分别是

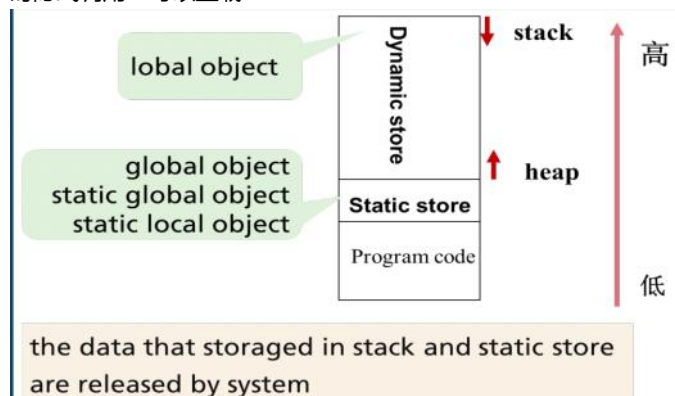
1、2、4、8、16。

即编译器只会按照1、2、4、8、16的方式分割内存。若n为其他值，是无效的。

来自 <<https://www.cnblogs.com/zrtqsk/p/4371773.html>>)

(6) 构造函数

•构造函数是一个特殊的成员函数 •名称与其类相同 •没有返回类型 •在创建实例时隐式调用 •可以重载



(7) 析构函数

(1) 存储在堆中的数据由程序员释放;

存储在栈里的数据由系统自动释放

(2) 出现malloc (字节数目) (必须和free (变量名) 搭配) 或者 new () (与 delete[]) 搭配: 是从堆里面申请空间的, 需要我们自己写析构函数释放空间

(3) 形如 double x; x在栈里面, 是系统自动分配空间的, 是会被默认构造函数自动回收的

(4) 先创建的类后被析构, 后创建的先被析构

(5) 默认析构函数的形式：`~类名() {}`;

(6) 如果类里面有从堆里面申请的空间，那么在出其作用域的时候必须要使用我们自己写的析构函数来防止内存泄漏

(7) 对象创建，调用构造函数；出其作用域，调用析构函数，如果是静态的，就不会调用析构函数

可以有多个构造函数，但是只能有一个析构函数

对象的声明周期

局部声明周期和全局生命周期

3.2 Composition

2020年5月9日 15:55

一、c++中代码重用：

方法1：实例重用

方法2：成员对象

方法3：继承

1、子对象

(1) 当一个对象被创建时，编译器保证它的所有子对象的构造函数都被调用。即子对象先要被创建，并被其对应的类的构造函数初始化

(2) 关于构造函数初始化列表：

常量，引用，成员对象必须在构造函数初始化列表中初始化，

数据成员也可以在构造函数初始化列表中初始化

```
class C{
private:
    int n;
    const int cint;    //const data member
    int& rint;         //reference data member
    A m;               //object member
public:
    C(int param...) : n(param), cint(5), rint(n), m(...)
    { }
}
```

有时可以在构造函数初始化列表里面省略成员对象的默认构造函数

```
class B{
private: float z;
        A m;    //member subobject
public: B(int r1,int r2,float r): m(r1,r2)
        { z = r; }
        B() {z = 0.0;} //这里其实是省略了 m(), 也可以写成B(): m() {z = 0.0;}。这一步其实是会调用A类的默认构造函数
};
```

注意：应该尽量使用构造函数初始化列表里面初始化，而不要在构造函数里面赋值

```
class people{
private:
    int num;
    string name;    //name是成员对象，因为string是一个类
public:
    people(int number, char *name) //这里其实漏了： name () 注意写的是成员对象的名字
    {
        num = number;
        this->name = name;    //注意这个写法
    }
    //尽可能如下
    people(int number, char *name): name(name), num(number)
    {}
};
people zhang(19, "张三");
```

在构造函数里面赋值：①call string() 构造函数②call people(int number, char *name)构造函数③ call string::operator=() //开销比较大
直接用构造函数初始化列表：①call string(const char *) 构造函数② call people(int number, char *name) 构造函数 //明显更好

要注意：是先执行构造函数初始化列表的

(3) 成员对象的初始化顺序

成员对象初始化的顺序是和其声明顺序一致的，和构造函数列出的顺序无关

```
class A{...};
class B{...};
class C{
private:
    B n;
    A m;
    int i;
public:
    C():i(0) {}    //这里其实就相当于 C():i(0),m(),n() {}
    C(参数列1的类型声明,参数列2的类型声明, int i) : m(参数列1), n(参数列2) //这时先初始化n而不是m
    { this->i = i; }
```

```

    };
    思考: A::A()
    {
        cout << "A()";
    }
    B::B()
    {
        cout << "B()";
    }
    C m; //这时会调用默认构造函数
    输出为: B(A())

```

```

#include <iostream>
using namespace std;
class A{
private:
    int x,y;
public:
    A(): x(0), y(0) //default constructor
    {}
    A(int x1,int y1): x(x1), y(y1)
    {}
};

class B{
private:
    float z;
    A m;      //成员对象
public:
    B(int r1,int r2,float r3):m(r1,r2) //方法1: r1和r2用来初始化成员对象
    {
        this->z = r3;
    }
    /*
    B(int r1,int r2,float r3):m(r1,r2),z(r3) //方法2: 数据成员的初始化也可以挪到构造函数初始化列表
    {}
    //这三个参数排列顺序无所谓, 一般来讲是先初始化成员对象
    */
    /*
    B(int r1,int r2,float r3); //方法3: 实现放在cpp文件中, 并且在cpp中直接可以用构造函数初始化列表
    */
    B(): m(), z(0) //考点: 调用A类的默认构造函数对m对象初始化, 但是其实这里的 m()语法上可以省略, 可以写成B(): z(0)
    {}
    /*
    B(float r):z(r)
    {}*/
};
/*
B::B(int r1,int r2,float r3):m(r2,r3),z(r1) //方法3的实现: 在cpp中直接可以用构造函数初始化列表
{}
*/
/*
class C{
private:
    int n;
    const int cint; //const data member
    int& rint;      //reference data member
    A m;           //object member
public:
    C(int param):n(param), cint(5),m()
    { rint=n;
    }
};
*/

void main()
{
    B b; //b has a subobject: b.m
    B c(4,5,6.6);
}

```


3.3 Operator overloading

2020年5月9日 15:55

3.3 运算符重载

定义重载运算符类似于定义一个函数，但该函数的名称是operator@，在其中@表示正在重载的操作符。重载操作符的参数列表中的参数数量取决于两个因素：

- 要么是一元运算符(一个参数)要么二元运算符(两个参数)。
- 要么操作符被定义为一个全局函数(一元的一个参数，二元的两个参数)要么一个成员函数(一元的零参数，二元的的一个参数——对象变成了左边的参数)。

注意：不能更改运算符的计算优先级，也不能更改运算符所需的参数数量。某些运算符不能重载，主要原因是为了安全。

· (成员运算符) * (指针简洁运算符) :: (作用域) :? (条件运算符) sizeof() 这五个不能重载

注意：在C++中，单目前缀增(减)和后缀增(减)都用++(--)运算符来表示，编译器为了区分它们规定：后缀增(减)函数应带有一整型量，这种参量仅仅用来区分前后缀，调用时，不必显式给出，它的缺省值为0。

```
class counter{
private:
    int second; //60秒计时器
public:
    counter():second(0){}

    counter& operator++(); //prefix
    int operator++(int);    //postfix

    counter& operator--(); //prefix
    int operator--(int);    // postfix

    void reset()
    {
        second = 0;
    }
    operator int() const //类 类型转换函数,隐含调用
    {
        return second;
    }
};
```

在cpp文件中，参数只有类型名而没写对应的形参，不一定错，有时是说明这个参数是系统使用的

哪些运算符重载为全局函数还是成员函数？

Operator	Recommended use
All unary operators	Member
= () [] ->	<u>must be member</u>
+= -= /= *= ^=	
&= = %= >>= <<=	Member
All other binary operators	non-member

注意：[] = 0 -> 必须作为成员函数重载

二目运算符[]，通常用它来定义对象的下标操作，第一个操作数必须是该类的对象，比如m[]。故只能重载为类的成员对象

注意：如果不重载=，就会自动生成，其形式一定是：className& className::operator=(const className&); 赋值操作就是右边覆盖左边，而且是逐域操作。a=b,就会被翻译成a.operator=(b)，如果类中有指针，同一片区域就会释放两次

如果自己写，这样写：比如 c=a，就是先释放c中的内存空间，再根据a的大小，c申请一定的空间，再把a里面的拷到c中。例如：

```
array& operator=(const array& rhs)
{
    if (this == &rhs) return *this; //进行自我赋值处理，比如出现“w=w”，效率高，而且避免出错（因为释放了还怎么赋值）
    delete[] p;
    size = rhs.size;
    p = new int[size*sizeof(int)];
    memcpy(p, rhs.p, size*sizeof(int));
    return *this;
}
```

memberwise 成员化

1、Memberwise Assignment (成员赋值)

如果类包含子对象(或从其他类继承)，则递归地调用这些子对象的运算符=。就是从外向内，依次调用各个类的“=”，顺序和类种声明顺序一样

2、成员初始化：对象里面有子对象，外面的对象初始化的时候，子对象的初始化用他们对应的构造函数进行初始化（自己的事情自己做）

注意：拷贝构造函数和赋值函数

拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用

```
String a("hello");
```

```
String b("world");
```

```
String c = a; // 调用了拷贝构造函数，最好写成 c(a)以区别于第四个语句
```

```
c = b; // 调用了赋值函数
```

【规则】尽可能使用编译器隐式生成的函数

编译器生成缺省函数时，产生优良代码。这种代码通常比用户编写的代码的执行速度快，原因是编译器可以利用汇编级功能的优点，而程序员则不能利用该功能的优点。

为了阻止某些类的对象的赋值，可以在类的声明中将operator= 函数的声明放在私有区

【规则】只要类里有指针时，就要写自己版本的拷贝构造函数和赋值运算符重载

friend：

- C++中的友员相当于为封装隐藏这堵不透明的墙开了一个小孔，任何该类的friend可以通过这个小孔窥视该类的私有数据。
- 注意友员函数与成员函数的区别：友员函数在类中声明，但不是该类的成员函数，不能通过this指针调用。（this指针指向当前激活的对象）
- 使用全局友员函数是不恰当的。它破坏了面向对象程序设计风格的一致性，使数据封装性受到削弱（因为友元可以访问私有数据），导致程序的可维护性变差。在以下情况发生，考虑使用友员函数：
 - 在类的设计中没有为类定义完整的操作集，将友员函数作为对类的操作的一种补充形式。
 - 考虑运行效率。
- 友元函数：
 - 友元成员函数：如A和B是两个类，B中有成员函数f，则class A { friend int B::f(A&obj) } 这行代码把f声明为A的有源成员函数，因为f要访问A
- 友元类。如class A{friend class B} 类B是A的友元类，其中所有成员函数可以访问A内的私有数据
- 友元函数可以放在私有部分和共有部分，但一般情况下放在公有部分
- 友元函数的定义可以放在.h文件,但不属于成员函数

一、运算符重载成类的成员函数

对双目运算符而言：

成员运算符函数的形参表中仅有一个参数,它作为运算符的右操作数,此时当前对象作为运算符的左操作数,它是通过this指针隐含地传递给函数的。

```
a + b ==> a.operator+(b)
```

对单目运算符而言：

成员运算符函数的参数表中没有参数,此时当前对象作为运算符的一个操作数。

```
+a ==>a.operator+()
```

```
c = a + 4;
```

编译器处理这个调用时的情形类似下面这样：

```
const complex temp(4); // 从4产生一个临时complex对象
```

```
c = a + temp; // 同a.operator+(temp);
```

```
//or
```

```
c = 4 + a; //compile-time error
```

解决方法：重载一个全局函数

```
const complex operator+(int, const complex&);
```

```
c = 4 + 4;
```

```
complex& complex::operator=(const complex& r)
```

```
{
```

```
    rpart = r.rpart;
```

```
    ipart = r.ipart;
```

```
    return *this;
```

```
}
```

(1) 双目运算符

const complex operator+(const complex& com) const; //二目加，只有一个参数，因为相当于调用被加数的操作符，a + b 相当于是 a.operator+(b)

第一个const表示返回为const的该类对象，为了避免出现 a+b=c 这样的表达式都能编译通过，所以返回必须为常量，常量不可以做左值

第二个const表示加数不会被修改

第三个表示这个函数只对常量进行操作

在cpp文件中的实现为：

```
const complex complex::operator+(const complex& com) const
```

```
{
```

```

        complex temp(rpart+com.rpart,ipart+com.ipart);
        return temp;
    }

```

(2) 单目运算符

`const complex operator+() const;` //单目加, 没有参数, 用唯一的操作数调用
 第一个const, 返回常量, 不可做左值
 第二个const这是一个常量函数, 因为不能对a重写了

可以有 `a++b`,但是不好, 不可以有 `++a`, 但是可以有 `+(+a)`

(3) 混合类型操作, 比如, complex要和int/float/double混合操作。

①对于 `c=a+4`

a是一个复数类, 但是4是一个int, 怎么样让编译通过呢?

可以对4进行处理, 写一个复数类的构造函数:

```

complex(double d) //具有隐式类型转换功能,but explicit
{
    rpart = d;
    ipart = 0.0;
}

```

所以: `c = a + 4;` 就相当于 `c = a.operator+(complex(double(4)));`

②对于 `c=4+a`

.cpp中写一个全局函数, 将其设置成复数类的友元函数

```

#ifndef COMPLEX_H
#define COMPLEX_H

class complex{
public:
    complex()
    {
        rpart = ipart = 0.0;
    }

    complex(double d) //具有隐式类型转换功能,but explicit
    {
        rpart = d;
        ipart = 0.0;
    }

    complex(double rp,double ip)
    {
        rpart = rp;
        ipart = ip;
    }

    //const complex add(const complex& com);

    //a + b a.operator+(b)
    const complex operator+(const complex& com) const ;//二目加

    //+a a.operator+( )
    const complex operator+( ) const ;//单目加

    void print() const;
    double getImaginary() const
    {
        return ipart;
    }
    double getReal() const
    {
        return rpart;
    }
    void setImaginary(double i)
    {
        ipart = i;
    }
    void setReal(double r)
    {
        rpart = r;
    }
    /*
    operator double() const

```

```

    {
        return sqrt(rpart*rpart+ipart*ipart);
    }*/
    friend const complex operator+(double i, const complex& obj);
private:
    double rpart;
    double ipart;
};

#endif

#include <cmath>
#include <cstdio>
#include "iostream"
#include "complex.h"

const complex complex::operator+(const complex& com) const
{
    complex temp(rpart+com.rpart,ipart+com.ipart);
    return temp;
}

void complex::print() const
{
    if ((-DBL_EPSILON < ipart) && (ipart < DBL_EPSILON))
        std::cout << rpart << std::endl;
    else if (ipart > 0)
        std::cout << rpart << "+" << ipart << 'i' << std::endl;
    else
        std::cout << rpart << "-" << -ipart << 'i' << std::endl;
}

const complex complex::operator+() const
{
    return complex(+rpart,+ipart);
}

//non-member function, global function
const complex operator+(double i, const complex& obj) //这里的参数是什么类型，在.h文件中声明的时候也要写什么类型，否则就是重载关系了
{
    complex temp(i+obj.rpart, obj.ipart);
    return temp;
}

```

二、运算符重载成友元(全局)函数

对双目运算符而言：

当用友元函数重载双目运算符时,两个操作数都要传递给运算符函数。

$a + b \implies \text{operator}+(a, b)$

对单目运算符而言：

用友元函数重载单目运算符时,需要一个显式的操作数。

$+a \implies \text{operator}+(a)$

重载>>和<<

```

int a, b, c;
cin >> a >> b;
c = a + b;
cout << c;

```

```

//hoping:
complex a, b, c;
cin >> a >> b;
c = a + b;
cout << a;

```

重载为成员函数：

```

cin >> a;
cout << a;
extern istream cin;

```

不能实现： `cin.operator>>(a);`

```

extern ostream cout;

```

必须重载成非成员函数，要声明成友元： `operator>>(cin, a);`

`istream& operator>>(istream& in, complex& com)` //istream也是类，是输入流

```

{

```

```

        in >> com.rpart >> com.ipart;
        return in;
    }
}
只有返回istream& 下面的代码才能执行
coding:
cin >> a >> b; ==> operator>>({ operator>>(cin, a), b});

#ifndef COMPLEX_H
#define COMPLEX_H

class complex{
public:
    complex(double rp=0.0, double ip=0.0)
    {
        rpart = rp;
        ipart = ip;
    }
    complex(const complex& obj)
    {
        rpart = obj.rpart;
        ipart = obj.ipart;
    }

    double getImaginary() const
    {
        return ipart;
    }
    double getReal() const
    {
        return rpart;
    }
    void setImaginary(double i)
    {
        ipart = i;
    }
    void setReal(double r)
    {
        rpart = r;
    }

    /*
    double d;
    d = double(12);
    complex m(1.1,2.2);
    d = m; //automatic
    d = double(m); //explicit
    */
    operator double() const
    {
        return sqrt(rpart*rpart+ipart*ipart);
    }

// friend const complex operator+(const complex& add1, const complex& add2); //友员函数版
friend const complex operator+(const complex& x);

    friend const complex operator*( const complex& x, const complex& y) //这个在.h文件中定义，但是不可以用this指针
    {
        complex temp(sqrt(x.rpart + y.rpart), sqrt(x.ipart + y.ipart));
        return temp;
    }

    friend ostream& operator<<(ostream& os, const complex& com); //去掉&调试? ?
    friend istream& operator>>(istream& in, complex& com);
private:
    double rpart;
    double ipart;
};

#endif

#include <iostream>
#include <cmath>
#include <cstdio>
using namespace std;
#include "complex.h"

```

```

/*
//non-member function(global function) x+y ==> operator+(x, y) 用全局函数进行运算符重载，但是调用好多函数，性能太差了
const complex operator+( const complex& add1, const complex& add2) //add2是加数， add1是被加数
{
    complex temp;
    temp.setReal(add1.getReal()+add2.getReal());
    temp.setImaginary(add1.getImaginary()+add2.getImaginary());
    return temp;
}*/

//friend global function a+b ==> operator+(a, b) 用友元全局函数来进行运算符重载，更好。
//在运行的时候先找有没有成员函数operator+，没有的话再看有没有全局函数operator+
const complex operator+( const complex& x, const complex& y) //y是加数不会被修改， x是被加数不会被修改
{
    complex temp(x.rpart + y.rpart, x.ipart + y.ipart);
    return temp;
}
//+a operator+(a)
const complex operator+(const complex& x) //这时不能再后面加const，因为没有常量版的全局函数啊，只有常量版的成员函数
{
    complex temp(+x.rpart, +x.ipart);
    return temp;
}

//cout << a ==> operator<<(cout, a)
//cout << a << b ==> operator<<( operator<<(cout,a),b );
ostream& operator<<(ostream& os, const complex& com) //os是形参，上面两行的cout是实参
{
    os << "(" << com.rpart;
    if (!((-DBL_EPSILON < com.ipart) && (com.ipart < DBL_EPSILON)))
        os << "," << com.ipart << "i";
    os << ")" << endl;
    return os; //返回这个对象本身
}

istream& operator>>(istream& is, complex& com)
{
    is >> com.rpart >> com.ipart;
    return is;
}
/*
A m;
cin >> m;
istream& operator>>(istream& in, A& com)
{
    ??;
    return in;
}
*/

#include <iostream>
#include <cmath>
using namespace std;
#include "complex.h"

int main()
{
    complex a(11.1,22.2),b(33.3,44.4);
    const complex d(0.1,0.2);
    complex c;

    cout << a << b;

    c = a + b; //Compiled: operator+(a,b)
    cout << c;

    c = b + a; //Compiled: operator+(b,a)
    cout << c;

    c = a + b + d; //operator+(operator+(a,b),d)
    cout << c;

    c = +d + b; //operator+(operator+(d),b)

```

```

    cout << c;

    c = a * b;

    c = a + 4; //operator+(a, complex(double(4), 0.0)); OR a.operator double() + 4
    cout << c;

    c = 4 + a;      //operator+(complex(double(4), 0.0),a)
    cout << c;

    a = 4 + 4; //这时要对=重载。但是其实没有什么必要，c++编译器自动就完成了
    cout << a;

    return 0;
}

```

overloading =

the compiler will automatically synthesize.....

Policy: bitcopy (逐域操作)

```

complex a, b, c;
c = a + b;
//自动生成的赋值操作
complex& complex::operator=(const complex& right)
{
    rpart = right.rpart;
    ipart = right.ipart;
    return *this;
}

```

1)为什么不是const member function? a = b; 因为要改写a

2)为什么要返回引用complex& ?

```

if:
complex complex::operator=(const complex& right)
{
    rpart = right.rpart;
    ipart = right.ipart;
    return *this;
}

```

a = b; //ok 注意返回的时候调用的是拷贝构造函数

(a = b) = c; ==> (b改写了a,返回一个内存中的临时对象) = c;==>c改写了内存中的这个临时对象, 其实最后并没有改写a。

3)为什么不返回void

C++程序员经常犯的一个错误是让operator=返回void,这好像没什么不合理的, 但它妨碍了连续(链式)赋值操作。

```

void operator=(const complex& right)

```

```

{
    rpart = right.rpart;
    ipart = right.ipart;
}

```

a = b = c; a = (void)

(a = b) = c; (void) = c;

3.4 Automatic type conversion

2020年5月9日 15:55

一、

- 隐式类型转换
赋值的时候
实参和形参赋值

- 显式类型转换
l-value = (type) expression ;
l-value = type(expression);

- 扩大转换:

从一种类型转换成另一种类型时, 如果后者至少能存储和前者相同范围的值, 发生的就是“扩大转换”。

- 收缩转换: (不安全)

如果后者能存储的值的范围小于前者, 发生的是“收缩转换”

二、

(1) C语言

关于对象的类型转换: C风格的类型转换过于粗鲁, 在程序语句中难以识别, 但你无能为力, 因为它是C语言的内部机制。

```
char ch;  
int i = 65;  
float f = float(2.5); //explicit type conversion 收缩转换  
double db ;
```

```
ch = i; //收缩转换 implicit type conversion
```

```
db = f; //扩大转换
```

```
ch = char(db); //explicit type conversion 收缩转换 强制类型转换不一定安全
```

(2) C++

C++中提倡的写法: 目的是这样的类型转换不论是对人工还是对程序都很容易识别

① static_cast<T>(v)

将表达式 v 的值转换为 T 类型。该表达式可用于任何隐式允许的转换类型。如果类型转换与旧样式一样合法, 则任何隐式转换都可以反向转换。

The static_cast<> operator is more restrictive than the traditional type cast.

- For example

```
enum E { first=1, second=2, third=3 };  
int i = second; // 隐式转换  
E e = static_cast<E>(i); // 反向隐式转换
```

```
ch = static_cast<char>(i); // int to char  
db = static_cast<double>(f); // float to double  
f = static_cast<float>(db); // double to float  
ch = static_cast<char>(db);
```

主要是解决了强制类型转换的危险:

```
//int *pi = (int*)i; // i 转换成16进制地址
```

```
//int *pi = static_cast<int*>(i); //compile-time error! 因为C中不存在这样的隐式转换
```

② const_cast<T>(v)

可用于更改指针或引用的 const 或 volatile 限定符。(在新样式的类型转换中, 只有 const_cast<> 可以去掉 const 限定符。) T 必须是指针、引用的类型。

- For example

```
class A  
{  
public:  
    void f();  
    int i;  
};
```

```
extern const volatile int* cvip;
```



```
extern int* ip;
void use_of_const_cast()
{
    const A a1;
    const_cast<A&>(a1).f();    // 去掉const
    ip = const_cast<int*>(cvip); // 去掉 const 和 volatile
}
```

•volatile 修饰符告诉编译程序不要对该变量所参与的操作进行优化.在两种特殊的情况下需要使用volatile修饰符:

-第一种情况涉及到内存映射硬件(memory-mapped hardware,如图形适配器,这类设备对计算机来说就好像是内存的一部分一样)。

-第二种情况涉及到共享内存(shared memory,既被两个以上同时运行的程序所使用的内存)

③ reinterpret_cast<T>(v)

把数据 v 以二进制存在的形式，重新解释成另一种类型T的值。reinterpret_cast允许任意的类型装换，包括将指针转换为整数，将整数转换为指针，以及将常量转换为非常量等

•For example

```
int *pi = reinterpret_cast<int*>(i); //将整数i的值以二进制(位模式)的方式被解释为整数指针，并赋给pi
int j = reinterpret_cast<int>(pi); //将指针pi的值以二进制（位模式）的方式被解释为整型，
```

(3) c++中类型转换的具体机制

方法1: 用构造函数

```
class One {
public:
    One() {}
    friend class Two;
};

class Two {
public:
    Two(){}
    Two(double d){} //automatic type conversion double->Two explicit（显式），比如 explicit Two(double d){}
    Two(const One&){} //automatic type conversio One->Two
};
```

```
void f(Two) {}
```

```
void main() {
    One one;
    f(one);          // Wants a Two, has a One
    f(1.1);
}
```

/*When the compiler sees f() called with a One object, it looks at the declaration for f() and notices it wants a Two. Then it looks to see if there's any way to get a Two from a One, and it finds the constructor Two::Two(One),which it quietly calls. The resulting Two object is handed to f(Two(one));*/

```
}
```

//通过构造函数自动类型转换，避免了定义两个f()重载版本的麻烦，然而代价是隐藏了构造函数的调用。

//如果我们刻意追求f()函数的调用效率，就应该避免。

显式调用的时候，必须写出a(b) 把括号里面的类型的变量转化成a类型

```
explicit complex(double rp) : rpart(rp),ipart(0)
{
    cout << "I am in complex(double rp)." << endl;
}
c = a + 4; //compile-time error
c = a + complex(4);
```

方法2: 运算符转换

返回类型是正在重载的操作符的名称

```
#include <iostream>
using namespace std;
```

```
class B {
    int i;
public:
```

```

    B(int ii = 0) : i(ii) {}    //把int类型转换成B类型
};

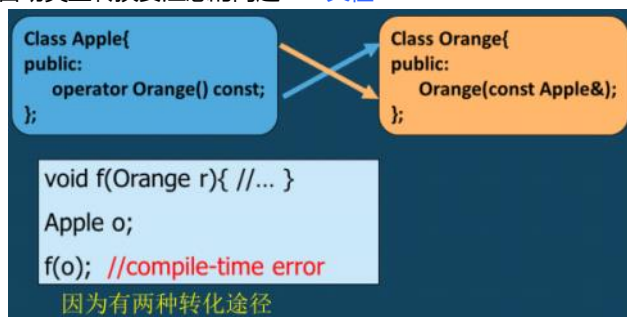
class A{
    int x;
public:
    A(int xx) : x(xx)          //把int类型转换成A类型
    {}
    operator B() const         // 把A类型转换成B类型, 且自动执行, 这里要有空格, 函数名是operator B
    {
        return B(x);           //返回一个B对象
    }
};

void g(B b) {}

void main() {
    A a(1);
    g(a);                       //Calls g(operator B())
    g(1);                       // Calls g( A(1) )
}

```

实现自动类型转换要注意的问题：二义性



解决方法：

看具体情况删除其中一种途径
或者把其中一种方法前面加explicit

提高程序性能：

- 尽量使用引用(&)传递对象——因为值传递的话形参需要空间，需要拷贝构造函数
- 恰当使用inline函数
- 避免 匿名((临时)) 对象——没有名字但在内存中存在
- 使用The return optimization (返回效率)

```

const complex operator+( const complex& x, const complex& y)
{
    complex temp(x.rpart + y.rpart, x.ipart + y.ipart);
    return temp;
}

```

首先，temp对象被创建，与此同时它的构造函数被调用。

然后，拷贝构造函数把temp拷贝到返回值外部存储单元(匿名对象)

最后，当temp在作用域的结尾时调用析构函数

c = a + b; 匿名对象赋值给c (调用 operator=(...))，然后调用析构函数释放匿名对象

说明以上这样写性能太差

返回效率：

```

const complex operator+( const complex& x, const complex& y)
{
    return complex(x.rpart+y.rpart,x.ipart+y.ipart); //返回值优化
}

```

编译器直接地把这个对象创建在返回值外部内存单元。因为不是真正创建一个局部对象，所以仅需要带参构造函数

3.5 Inheritance

2020年5月9日 15:55

继承的作用：可以用继承实现分门别类

一、公有继承

派生类无条件得自动继承基类的数据结构和操作，如果有自己独特的数据和操作就补写上

```
class derivedClass : public baseClass //其实是普通的类，只是多了一个：基类
{
    //difference parts between base and derived; 这里面写基类和派生类不同的：
};
```

①添加数据成员和成员函数：只需要正常的加入

②重新定义在继承现有的成员函数：函数名啊什么的和基类的都一样，只要重新在派生类写了，那就会顶替基类的相应函数

知识点：如果派生类想要访问基类的私有数据：那么基类的那部分数据不用写private，而是写成protected

知识点：派生类的构造函数：

匿名对象：派生类中从基类继承的对象；这部分是由基类的构造函数来初始化，且初始化的顺序和基类中数据成员声明顺序一样

因为是匿名对象没有名字，所以构造函数初始化列表中用基类的名字，否则一般用的是对象的名字

注意派生类的默认构造函数的写法，有时候基类的默认构造函数往往是从构造函数初始化列表省略（一个类有子对象，那么在默认构造函数中子对象的构造函数一般不写）

```
#ifndef MANAGER_H
#define MANAGER_H

class manager : public employee
{
public:
    manager():employee(), position(NULL) //employee(),可以省略
    { }

    manager(char *name,short age,float salary,char *position):employee(name,age,salary)
    {
        this->position = new char[strlen(position)+1];
        strcpy(this->position, position);
    }

    void myDuties() const;
    void Signing() const; // signature
    //redefine 重定义
    //void print() const;
    //float Salary(); //与父类有相同的接口，但不一样的操作

    ~manager()
    {
        if (position) delete[] position; //因为从堆里面分配空间了，所以要自己写析构函数
    }

private:
    char *position;
    manager(const manager&);
    manager& operator=(const manager&);
};
#endif
```

知识点：先创建父类子对象，再创建客户端子对象

```
class C : public A
{
private:
    B m ;
    int i;
};
```



构造函数这么写：

```
C(parameter list1 declarations,
  parameter list2 declarations,
  parameter list3 declarations) : A(parameter list1 ),
                                  m(parameter list2 )
{ //initialize using parameter list3; }
```

有对象名字就不能写类名



```
C(int i): i(i)
{ }
C(int i): i(i), A(), m()
{ }
```

等价, 一般是省略的, 要注意

参数的顺序是无关的，反正初始化的顺序一定是先主后客

注意：如果没有写private还是public等，那缺省的就是private

知识点：构造函数和析构函数调用的顺序

- 构造从类层次结构的最根开始，在每一层，基类构造函数首先被调用，然后是成员对象构造函数。析构函数的调用顺序与构造函数完全相反。
- 成员对象的构造函数调用顺序完全不受构造函数初始化器列表中调用顺序的影响。顺序由成员对象在类中声明的顺序决定

自动析构函数调用

-对于任何类只有一个析构函数，它保证对它们特定的类进行清理。您永远不需要显式地调用析构函数

-编译器确保所有析构函数都被调用，这意味着整个层次结构中的所有析构函数，从最派生的析构函数开始，然后返回到根。

设计一个派生类需特别关注的地方：

- 在继承层次中构造函数和析构函数的行为（尤其是有时候不能缺默认构造函数）
- 默认构造函数（无参构造函数）的重要性
- 析构函数的调用时机
- 析构函数的调用时机

- Public member：对自身和所有客户端可见的公共成员
- Protected member：对自身的成员函数、派生类、友元 可见
- Private member：只对自身的成员函数、友元 可见

知识点：Namehiding

在c语言中，局部变量和全局变量可以同名。在进行操作的时候，一定是本地优先，即局部变量优先

```
int x; //global variable
void someFunc()
{
    double x; //local variable
    x = 100;
    ::x = 200; //这样可以访问全局的那个x
}
```

在c++中：

如果基类有函数重载（根据参数不同而实现重载），如果在派生类重写该函数，根本就不会去调用基类中所有和它同名的函数。在派生类中，**重定义**（redefine）或**重载**（overriding）基类的一个函数，派生类函数会掩盖所有基类同名函数，“本地优先”，在调用该函数名的函数的时候，只能看到派生类的这个函数。这就是**Namehiding**

类同名函数，“本地优先”
//In parent
void parent::f(int a);

```

int parent::f(int a , double d);
void parent::f(char c);

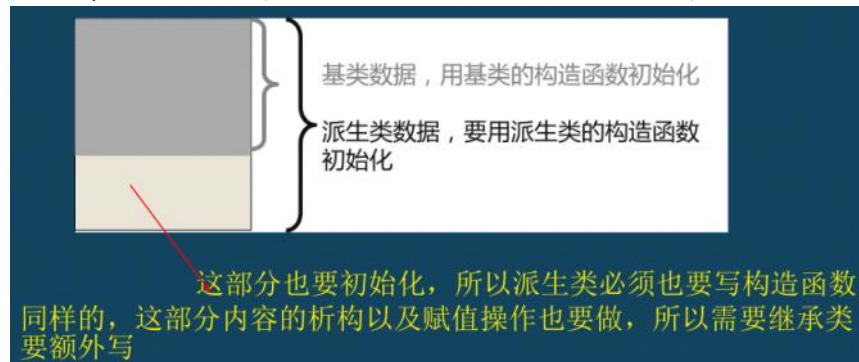
//In child
void child::f(int a );    //redefine,name hiding
void child::f(double a);  //overriding,name hiding

```

应该避免namehiding

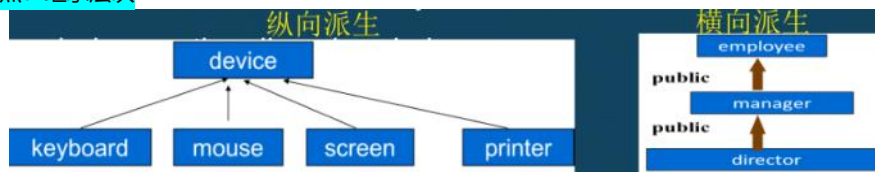
知识点：不会自动继承的函数

- ① 构造函数
- ② 析构函数
- ③ 重载操作符new
- ④ 赋值操作符 '='
- ⑤ 友元关系（因为基类的友元不一定就会成为派生类的友元，父亲的朋友不一定是孩子的朋友）



并不是所有函数都自动从基类继承到派生类。[构造函数和析构函数](#)处理对象的创建和销毁，它们只知道如何处理特定类的对象方面，因此必须调用它们下面层次结构中的所有构造函数和析构函数。因此，构造函数和析构函数不继承，必须专门为每个派生类创建。

知识点：继承层次



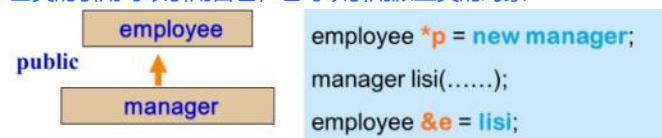
在继承中，派生类几乎继承了基类的所有特征，因而在定义一组类时，将其公共属性抽取出来，放在基类中，就可以避免在每个类中重写基类的代码。在派生类中利用重定义函数修改基类行为进一步增加了重用的灵活性，

知识点：Upcasting

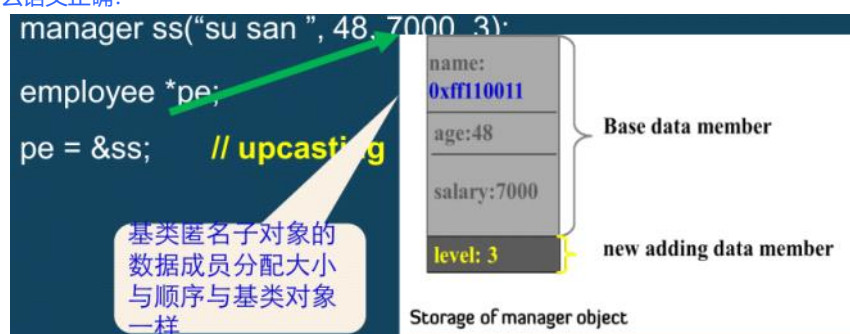
不同类型的指针不可以直接赋值；
 任何类型的指针都可以直接强制类型转换，比如 `pa=(A*)pb`
 c++中，为了安全，强制类型转换的时候加上 `static_cast`

基类的指针可以指向自己的对象也可以直接指向派生类的对象

基类的引用可以引用自己，也可以引用派生类的对象



为什么语义正确？



举例：通过指针阐述upcasting

```
void eat(const Person *pe); // anyone can eat
void study(const Student *ps); // only students study
```

```
Person wang; // wang is a Person
Student chen; // chen is a Student
```

```
eat(&wang); // fine, wang is a Person
eat(&chen); //upcasting, fine, chen is a Student, and a Student is-a Person
```

```
study(&chen); // fine
```

```
study(&wang); // error! wang isn't a Student 派生类对象的指针不能指向基类，即：基类对象的地址不能放到派生类指针中
```

举例：通过引用阐述upcasting

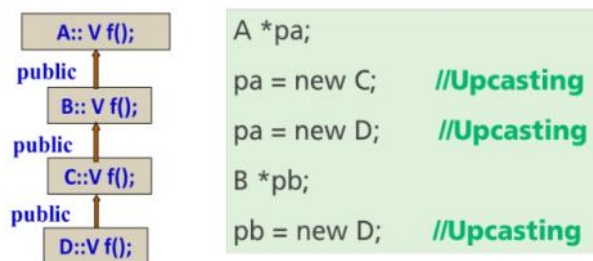
```
void eat(const Person &pe); // anyone can eat
void study(const Student &ps); // only students study
```

```
Person wang; // wang is a Person
Student chen; // chen is a Student
```

```
eat(wang); // fine, wang is a Person
eat(chen); //upcasting, fine, chen is a Student, and a Student is-a Person
```

```
study(chen); // fine
```

```
study(wang); // error! wang isn't a Student
```



upcasting的用途

例如：

一家企业有1000名员工(其中有几名经理)，编程打印个人信息。

```
employee *p[1000]; //p[i] pointed {i} personal 0<=i<1000
```

```
p[i] = new employee("张三",29,7000);
```

or

```
p[i] = new manage("李四",29,10000,"科长"); 这句是用了upcasting
```

```
for (i=0; i<1000; i++)
```

```
p[i]->print();
```

利用upcasting解决了不同类型对象的识别问题

存在的问题：每个人都是调用 employee::print()

解决方法：在要执行的函数前面加关键字 “virtual” 这就是虚函数（只要基类声明了虚函数，那么派生类同名的函数可以不用写 “virtual”）

知识点：公有继承、私有继承、受保护继承

(1) public inheritance

基类的所有私有元素在派生中仍然是私有的，基的所有公共元素在派生中仍然是公共的。基的所有受保护元素仍然在派生中受保护。

(2) Private inheritance

基类的所有元素(公共的、私有的、受保护的)在派生中都是私有的

(3) Protected inheritance

基类的所有私有元素仍然是派生的私有元素。基的所有公共元素和受保护元素在派生类里面都是protected。

	public	protected	private
public inheritance	public	protected	不可见
Private inheritance	private	private	不可见
Protected inheritance	protected	protected	不可见

怎么选择:

公有继承: 特殊和一般的关系

基类是共性, 派生类是在基类的基础上又有特殊的属性

如果类A和类B毫不相关, 不可以为了使B的功能更多些而让B公有继承A的功能和属性

公有派生体现了从普遍到特殊的过程, 子类对象和父类对象之间的(语义)关系是: 普遍跟特殊的关系(IS_A)

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时, 它们都保持原有的状态, 而基类的私有成员仍然是私有的, 不能被这个派生类的子类所访问。

私有继承:

只是为了实现代码重用, 没有任何关系

私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员, 并且不能被这个派生类的子类所访问
私有继承的含义不是“是一个”, 如果使类D私有继承于类B,

这样做是因为你想利用类B中已经存在的某些代码, 而不是因为类型B的对象和类型D的对象之间有什么概念上的关系。因而, 私有继承纯粹是一种实现技术

尽可能地使用分层, 必要时才使用私有继承。

两个类A和B之间的关系:

1、没有关系

2、 is-a关系

```
class B: public A{
    ...
};
```

3、 has-a关系(有一个)

```
class B{
    A m;
    ...
};
```

4、 is-implemented-in-terms-of(根据某物实现出)分层技术

```
class B
{
    A *p;
    ...
};
```


3.6 Polymorphism & Virtual Functions

2020年5月9日 15:55

一、Polymorphism (多态)

a) 虚函数的作用: 使用 **多态指针 (基类引用)** 和 **虚函数**, 实现动态绑定。

两个要素:

b) 虚函数的定义 unit three\Polymorphism\virtual function.cpp

non-virtual member function: 基类提供实现, 派生类继承接口和实现。

Rectangle r;

r.objectID(); 通过派生类对象调用从基类继承的non-virtual member function语法正确, 语义也正确

virtual member function: 基类与派生类有同名操作 (继承接口), **派生类可以有自己的实现, 也可以没有。**

函数重载——静态多态 (编译的时候就确定了)

运算符重载——静态多态

多态指针 (或引用) ——动态多态 (在运行的时候才能确定在公有继承中, 基类的指针可以指向自己也可以指向派生类对象)

```
class employee{
public: virtual void print();
        virtual float salary(); //虚函数    根据语义环境, 派生类manager有自己的print()操作和salary()
};
class manager : public employee{
public: virtual void print();
        virtual float salary(); //基类里面有虚函数, 派生类中与基类同名的函数就自动变成虚函数
};
employee* p = new employee("zhangsan",30,3500);
p->print();    //call employee::print()
p = new manager("lisi",35,4500,3);    //upcasting
p->print();    //call manager::print() P是一个动态多态指针。动态绑定
```

动态绑定Dynamic binding

- 在公共继承中, 基指针(或引用)可以指向派生的对象
- 通过指针(或引用)调用虚函数

什么时候用虚函数: 如果基类中某成员函数被派生类调用的时候达不到效果或者有错误, 那么就要把这个函数设定为虚函数, 然后在派生类写另一种形式但是和他同名的函数

构造函数和析构函数可以是虚函数吗?

c) **构造函数不能是虚的, 但析构函数往往是虚的;**

因为虚函数在编译的时候是先不管的, 就是等会在运行的时候再看, 但构造函数怎么可以等到后面才初始化呢?

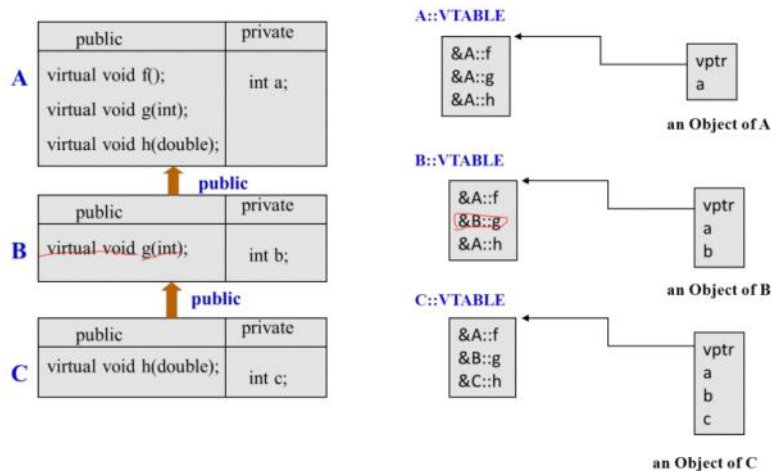
编译器调用构造函数的机制: the constructor initializer list

编译器调用析构函数的机制: Inheritance

d) 编译器自动生成的析构函数不是虚函数

实现动态绑定的机制:

C++编译器为每个包含虚函数的类生成一个虚函数表(VTABLE), 每个虚函数表中存储有相应的虚函数地址。当建立该类的一个对象时, 对象中含有一个隐含的指针(vpointer)数据成员, 该指针指向该类的虚函数表。该指针数据成员由C++编译器自动为这个类定义并在构造函数中初始化



```

• A * p = new C; //....
p->h(5);
//due to h() is a virtual function, when compiling
generate: 编译器生成了这个玩意
(vptr[2])(p, 5);
//等价
When program runtime, if p points an object of B,
call A: *(this->vptr+2)(p,5) call C::h()

```

使用虚函数的利弊

- 1、多态性提高了代码的组织性和可读性。
- 2、它使得程序模块间的独立性加强。
- 3、增加了程序的易维护性。
- 4、它提供了与具体实现相隔离的另一类接口，即把“what”从“how”分离开来，即进一步实现信息隐藏。
- 5、可使得程序具有可生长性，这个生长性不仅指在项目的最初创建期可“长”，而且希望项目具有新的性能时也能“生长”。
- 6、对于许多语言它是可选的，因为它不是相当高效率的，在程序中设置虚函数既需要额外的代码空间，又需要额外的执行时间。

pure virtual function 纯虚函数

问题1：分析张三同学、李四同学、王五同学...抽象出共性：属性和行为

```

class STUDENT{...};
STUDENT zhangsan("张三", 22, 'M');

```

```

class keyboard{...};
class screen{...};
class printer{...};
class mouse{...};

```

结果发现：有相同的属性和操作，为了尽可能代码重用，就在上面几个类的基础上泛化出一个父类device，将公共数据和接口放在基类。

```

class device{//公共数据和接口};

```

```

class keyboard : public device
{...};
class screen : public device
{...};
class printer : public device
{...};
class mouse : public device
{...};

```

```

decice m; //语义错误 no compliant

```

有时，基类只是一种抽象的概念，是人为地虚构出来的。它并不和具体的事物相联系。怎么样从语法角度来刻画这种特性呢？

问题2：

MH航空公司设计的飞机继承体系。该公司开始只有A型和B型两种飞机，两者都以相同方式飞行。因此MH设计出这样的继承体系：

```

class Airport { ... };           // represents airports
class Airplane {
public:
    virtual void fly(const Airport& destination) ;
};

void Airplane::fly(const Airport& destination)
{
    default code for flying an airplane to the given destination
}
class ModelA: public Airplane { ... };
class ModelB: public Airplane { ... };
    很好，代码重用。

```

几年以后，决定购买一种新式C型飞机，C型和A型以及B型的飞行方式不同。

```

class ModelC: public Airplane {
...    // no fly function definition
};

```

由于急着让新飞机上线服务，竟忘了重新定义其fly函数。

吉隆坡机场控制中心：

```

Airport  北京(...);
Airplane *pa = new ModelC ( " MH 3 7 0 " ) ;
pa->fly(北京);    // calls Airplane::fly() 飞机去哪儿了

```

问题不在Airplane::fly有缺省行为，而在于C型在未明白说出“我要”的情况下就继承了该缺省行为。

有纯虚函数的类，只需要在纯虚函数后面加上=0，，这就是个抽象类，则就不能再创建这个类的对象了
一个类的函数是否需要提供函数体（函数实现）：取决于有没有机会调用他，仅此一个判断条件

【实现了的纯虚函数】

```

class Airplane {
public:
    virtual void fly(const Airport& destination) = 0; //纯虚函数
    ...
};

```

```

void Airplane::fly(const Airport& destination) // an implementation of
{
    // a pure virtual function default code for flying an airplane to the given destination
}

```

```

class ModelA: public Airplane {
public:
    virtual void fly(const Airport& destination)
    { Airplane::fly(destination); }
    ...
};

```

```

class ModelB: public Airplane {
public:
    virtual void fly(const Airport& destination)
    { Airplane::fly(destination); }
    ...
};

```

```

class ModelC: public Airplane {
public:
    virtual void fly(const Airport& destination); //如果这里忘记了c类的fly函数的实现，那就会继承基类的fly函数，所以C也就变为抽象类，不能创建对象
    ...
};

```

```

void ModelC::fly(const Airport& destination)
{
    code for flying a ModelC airplane to the given destination
}

```

注意：析构函数不能是纯虚函数

【抽象类】

至少含有一个纯虚函数的类：抽象类

抽象类只能用作其它类的基类。不能创建实例，不能用作函数参数、不能用作函数返回类型以及不能显式转换的类型。

- 可以声明抽象类的指针和引用，此指针指向派生类对象，实现动态束定。（建议这种代码别写）
- 纯虚函数、空函数、不提供函数体。（这三个概念不一样）
- 如果派生类中没有给出基类的全部纯虚函数的实现代码，它继承基类的这些没有实现的纯虚函数，则派生类还是一个抽象类。

```
//Shape is a abstract
Shape m; //complime-time 语义错误，因为它不对应任何一个具体存在的对象

Shape fun(Shape d) //不能创建抽象类的实例
{
    //...
}
Shape* fun(Shape *pd)
{
    screen s;
    return &s; //返回局部对象的地址，错
}

Shape* fun(Shape *pd)
{
    pd = new Circle;
    return pd;
}

Shape& fun() //返回局部对象的引用，错
{
    Rectangle r;
    return r;
}
Shape& fun(Shape& s) //ok
{
    //..
    return s;
}

Shape *pd = NULL;
initialize(pd); //这个pd是实参

void initialize(Shape* pd) //语法正确,but pd是形参，局部的
{
    pd = new Circle; //形参是局部变量，返回的时候这个circle对象没有被释放
}

void initialize(Shape* &pd) //good style 这样改可以
{
    pd = new Circle;
}

initialize(&pd);
void initialize(Shape** pd) //这样改也可以
{
    *pd = new Circle;
}
```

【dynamic_cast】

```
A *pa;
pa = new C; //Upcasting
pa->f(); //fine
pa->f(); //编译出错，因为编译器编译的时候是静态编译，此时认为p是指向A类型的。只有在执行程序的时候才认为p指向C类型对象
(C*)pa->h(); //luckily, security （这里是强制类型转换啊）
(D*)(pa)->e(); //insecurity （不安全）
```

- In downcasting using dynamic_cast（动态转换符）

```
A *pa;
pa = new C; //Upcasting
D *pd = dynamic_cast<D*>(p); （p里面放的就是D类型的对象，这句代码才会编译成功）
if (pd != NULL)
    pd->e();
```

【附加】

•构造函数中的虚函数.cpp

对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，[虚机制在构造函数中不工作](#)。因为构造函数的工作是把对象变成合理的存在物。在任何构造函数中，对象可能只是部分被形成—我们只能知道基类已被初始化了，如果允许在构造函数中实行虚机制，那么所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。

•成员函数中的虚函数.cpp

对于普通成员函数中的虚函数的调用是在运行时决定的，这是因为编译时并不能知道这个对象是属于这个成员函数所在的类，还是属于由它

【虚函数的正确使用】

情况1:

```
class A{
public:
    void f(){...}
    int g(int a) {...}
};
class B: public A{
};
```



```
class C: public B{
};
```

```
B b;
C c;
```

b.f(); 语法正确，语义也正确

c.f(); 语法正确，语义也正确

可以确定: void A::f() is non-virtual member function

b.g(3); 语法正确，语义错误。解决方法: 重定义

```
int B::g(int)
virtual int A::g(int a)    is virtual member function
```

派生类与基类同名（函数原型一样）操作自动就成了虚函数。

```
A *p = new A;
p->g();    // Dynamic binding, call int A::g()
p = new B;
p->g();    //Dynamic binding, call int B::g()
```

or:

b.g(3); 语法正确，语义也正确

c.g(4); 语法正确，语义错误。解决方法: redefine

```
int C::g(int)
virtual int A::g(int a) is virtual member function
```

情况2:

```
class A{
public:
    virtual int g() {return 1;}
};
class B: public A{
public:
    virtual int g(int a) {return a+1;}
};
```

是虚函数，但不会实现动态绑定，编译器忽略virtual。 //只有基类和派生类的函数完全一样，才会实现动态绑定

```
A *p = new B;
p->g(); //call A::g()
p->g(2); //error C2660: 'A::g' : function does not take 1 arguments
```

情况3:

```
class A{
public:
    virtual int g() {return 1;}
};
class B: public A{
public:
    virtual int g() const {return 1;}
};
```

是虚函数，但不会实现动态绑定，编译器忽略virtual。是重载关系，会发生namehiding

```
A *p = new B;
p->g(); //call A::g()
```

```
const A *q = new B;  
q->g(); //error C2662: 'A::g' : cannot convert 'this' pointer from 'const A' to 'A &
```

四、More advanced topics of C++

2020年6月26日 22:24

4.1 Template

2020年6月8日 9:13

4.1、模板与范性编程 (Templates and Generic Programming)

情况1: 函数重载的时候, 不同的代码段代码几乎一样

情况2: 在containers (容器) 中, 如果已经设计了int数组类, 若要设计char数组、complex数组, 设计将是机械的赋值重复工作

解决思路: 类型参数化 (即参数是类型int、char、A、.....)

4.2、函数模板function templates

函数模板:

(函数模板定义的格式)

```
template <typename T1>
T1 functionName( T1 t1, T1 t2 )
{
    //function body
}
```

例如:

```
//function template definition 函数模板定义
template <typename T> //这个地方可以是T或者其他字母, 只要不是关键词就行
T sum(T array[], int size)
{
    T total = 0;
    for (int i=0; i<size; i++)
        total = total + array[i]; //*(array+i)

    return total;
}
```

函数模板是可以重载的, 这个函数和上面那个函数名相同, 但是参数不同

```
template <typename T>
T sum(T *array1, T* array2, int size) //function template overloading 函数模板重载 (这里是两个参数)
{
    T total = 0;
    for (int i=0; i<size; i++)
        total = total + array1[i] + array2[i];
    return total;
}
```

也可以模板函数与非模板函数重载 (见下)

模板函数怎么用

(1) 一般情况:

在.h文件中定义max模板函数

```
template <typename T>
inline T Max(T x, T y)
{
    cout << "In template function Max, ";
    return (x>y)?x:y;
}
```

在.cpp中, 发生静态绑定

```
int i = 90;
char c = 'a';
float f = 91.0;

Max(i, i); //static binding, 模板函数的函数体对预编译器可见
Max(c, c); //过程: 用实参的类型替换模板函数中的T, 然后在内存中生成新的函数体, 把这个函数的入口地址绑定在Max(c,c)
Max(f, f);
Max(1.1, 2.2);
```


(2) 特殊情况

```
Max(c, i);    //错。解决方法：用一个非模板函数重载一个同名的模板函数
Max(1.1, 2); //错
```

错误的原因：模板函数调用时参数传递不会自动类型转换

解决方法：用一个非模板函数重载一个同名的模板函数

```
inline int Max(char x, int y)
{
    cout << "In function Max(char x, int y), ";
    return (x>y)?x:y;
}
```

这段代码放在了.h文件，但是.h文件中不可以有实体函数。不过内联函数（inline）可以放在头文件中

C++编译器静态绑定的时候，遵循的顺序：

先找参数完全匹配的普通函数（即实体函数）

再找参数完全匹配的模板函数（用实参的类型替代模板函数的类型，然后生成一个新的函数）

再找实参经过自动类型转换得到的函数

都找不到的话，编译错误

```
template <typename T>    //这是模板函数
inline T Max(T x, T y)
{
    cout << "In template function Max, ";
    return (x>y)?x:y;
}

inline int Max(char x, int y) //这是内联函数
{
    cout << "In function Max(char x, int y), ";
    return (x>y)?x:y;
}
```

注意：模板函数的声明与定义或在头文件中；或与调用模板函数代码同一个文件（但要保证函数模板的定义要在声明的前面）。
模板函数的源代码编译器可见！！

函数模板的定义也可以在cpp文件中，但是其他cpp不能调用，会发生链接错误，可以在这个cpp文件中再写一下函数模板

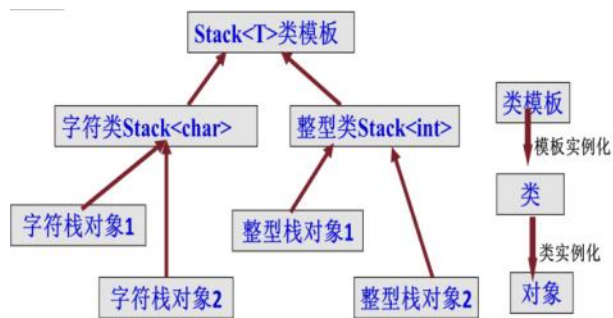
4.3、类模板 class template

常用数据结构：数组、栈、队列、矩阵

（一）栈模板类

- 栈（STACK）又称为堆栈，是一种“特殊”的线性表，这种线性表的插入和删除运算只允许在表的一端进行。
 - （1）允许进行插入和删除运算的这一端称为栈顶（TOP），不允许进行插入和删除运算的另一端则称为栈底（BOTTOM）；
 - （2）向栈中插入一个新元素称为入栈或压栈（push），从栈中删除一个元素称为出栈或退栈（top）；
 - （3）通常，记录栈顶元素位置的变量称为栈顶指针，处于栈顶位置的数据元素称为栈顶元素；
 - （4）而不含有任何数据的栈则称为空栈
 - （5）栈（STACK）的存储结构：顺序存储结构和链表存储结构。
- 栈的存储结构：顺序存储结构和链表存储结构
 - （1）顺序存储结构称为顺序栈。顺序栈可以用连续的一块存储空间和一个记录栈顶位置的变量来实现。比如：数组
 - （2）链表存储结构：链头就是栈顶，链尾就是栈底

类模板是类一级上的更高层次的抽象，而类是对象一级上的抽象



/*inline函数的定义放在源文件的头文件中

模板类的定义与模板类成员函数的定义都放在头文件中*/

#ifndef STACK_H

#define STACK_H

//例：堆栈模板类的定义STACK.h

```

template <typename T>
class STACK{
private:
    T *v;    //用T类型指针v表示栈底
    T *p;    //栈高指示器
    const int size; //栈体大小
public:
    STACK(int z=100); //缺省构造函数
    void push(T a);
    inline T pop();
    inline T top() const; //仅仅是取出，不弹出，即不改变这个栈
    inline int getsize() const;
    inline bool empty() const;
    virtual ~STACK()
    {
        delete[] v;
        cout << "call ~STACK()!" << endl;
    }
};

/*模板类中一些函数的定义，注意，相比以前学的，只是在上面一行加上template<typename T> 以及 类名后面加上<T> */
template <typename T> //这里是模板类的构造函数的定义，注意写法
STACK<T>::STACK(int z):size(z) //缺省参数的数值只能放在声明里面，不能放在定义里面，所以这里z不可以写成z=100
{
    p = v = new T[size];
}

//判断栈是否非空，返回布尔值
template<typename T>
bool STACK<T>::empty() const
{
    return v==p;
}

//压栈操作，注意要进行判断，保证栈不会被挤爆啦
template <typename T>
void STACK<T>::push(T ch)
{
    if (p - v >= size) //栈顶减去栈底，得到栈中元素的实际个数
    {
        cout << "overflow!" << endl;
        exit(1);
    }
    else
        *p++ = ch; //ch压入p指向的空间，然后p加1
}

//弹出操作，注意要进行判断，保证栈里面没有被榨干
template <typename T>
T STACK<T>::pop()
{
    if (p == v)

```

```

    {
        cout << "underflow!" << endl;
        exit(1);
    }
    else
        return *--p;
}

//读取操作，类似于弹出操作，但是这里只是读取，不会弹出，所以不可以改变原来的栈，故这个函数后面加了const
template <typename T>
T STACK<T>::top() const
{
    if (p == v)
    {
        cout << "underflow!" << endl;
        exit(1);
    }
    else
        return *(p - 1);
}

//读取栈的大小
template <typename T>
int STACK<T>::getsize() const
{
    return size;
}

//把栈里面的元素都打印出来
template<typename T>
void printStack(STACK<T>& stack)
{
    while (!stack.empty())
        cout << stack.pop() << " ";
    cout << endl;
}
#endif

```

模板类的用法

STACK<int> intStack; //模板实例化，类实例化（这样就定义了一个元素为int类型的栈，千万不可以写成STACK intStack）

模板的实例化（即作用原理）

用<int>中的int代替类模板的T，然后在内存生成整型堆栈类

给类模板的参数指定具体的类型



在实例化模板时，编译器复制模板的代码，在复制过程中，将模板中的形参替换为具体类型—实参，产生一个具体的类定义。

```

STACK<int> si(10);
STACK<char> sc1(8), sc2(6);

```

- 在编译运行时，有一个整型栈类STACK<int>和一个字符栈类STACK<char>，每个类都有数据结构和成员函数。而同一个类的对象如sc1,sc2有各自的数据结构、共享字符栈类的成员函数。

优劣分析：

- 类模板的实例化是在编译时进行的，故使用模板机制，不会影响程序的运行效率。
- 使用模板机制，也不会节省代码的生成量。
- 增加程序的灵活性和加快编程速度。使用模板，一组类只需描述一次。

（二）矩阵模板类

举例：写出一个矩阵模板类Matrix(m,n)，支持如下写法：

```

Matrix<int> m1(2,3),m2(2,3);    //数据部分,构造函数
m1 << 2 << 5 << 7 << 4 << 3 << 1; // 对m1赋值: 重载插入符<< 2 5 7; 4 3 1
m2 << 6 << 2 << 8 << 5 << 1 << 7;
Matrix m = m1 + m2;    //重载+, 拷贝初始化构造函数
m[1][2] = 3;    //重载第一个[], 第二个 []正常含义a[i] <--> *(a+i) 这里的a是第一行的地址
cout << m << endl;    //m输出, 需要重载插入符<<

```

```

#ifndef CMatrix_H
#define CMatrix_H
template <typename T>
class CMatrix{
public:
    CMatrix(int r,int c);
    ~CMatrix()
    {
        delete[] pv;
    }
    CMatrix(const CMatrix& m); //拷贝构造函数
    CMatrix& input();
    CMatrix operator+(const CMatrix& m); //不能写成:返回值CMatrix&
    int& operator() (int i, int j);
    CMatrix& operator++(); //prefix
    void print() const;
private:
    int row,col;
    T *pv;
    CMatrix& operator=(const CMatrix& m);
};

//构造函数
template<typename T>
CMatrix<T>::CMatrix(int r, int c):row(r),col(c)
{
    pv = new int[row*col];
}

//拷贝构造函数
template<typename T>
CMatrix<T>::CMatrix(const CMatrix& m)
{
    row = m.row;
    col = m.col;
    pv = new int[row*col];
    memcpy( pv, m.pv, row*col*sizeof(T) );
}

template<typename T>
CMatrix<T>& CMatrix<T>::input()
{
    static int i = 0;
    int k;
    if (i >= row*col) i = 0;
    k = i / col;
    *(pv + k*col + i%3) = t;
    i++;
    return *this;
}

//重载+运算符
template <typename T>
CMatrix<T> CMatrix<T>:: operator+(const CMatrix<T>& m)
{
    if(row == m.row && col == m.col)
    {
        CMatrix temp(row,col);
        for (int i=0; i<m.row*m.col; i++)
            *(temp.pv+i) = *(pv+i) + *(m.pv+i);

        return temp;
    }
}

//重载 () 运算符

```

```

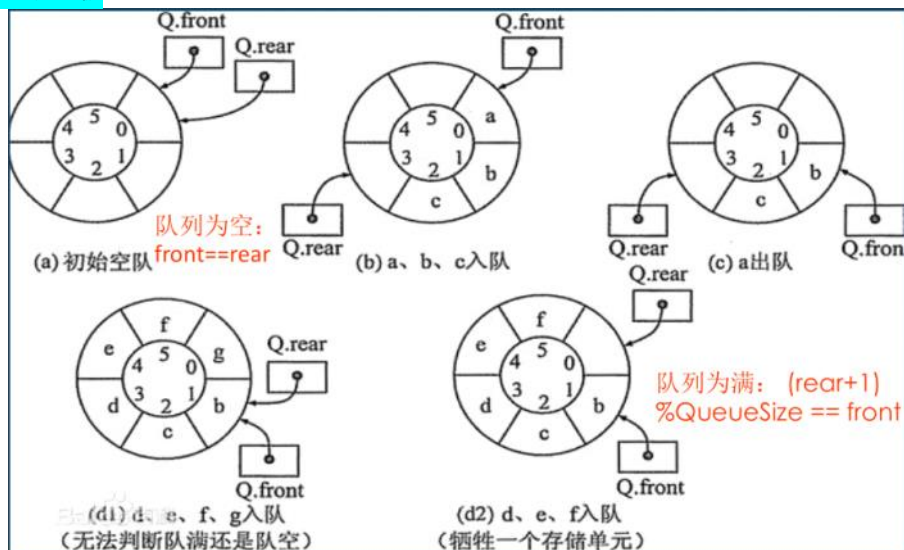
template <typename T>
T& CMatrix<T>::operator() (int i, int j)
{
    if ((i<row && i>=0) (j<col && j>=0))
        return *(pv + i*col + j);
    else
    {
        std::cout << "行、列下标越界!" << std::endl;
        exit(0);
    }
}

//重载++运算符
template<typename T>
CMatrix& CMatrix<T>::operator++()
{
    int i;
    for(i=0;i<row*col)
        ++pv[i];
    return *this;
}

//print函数的定义
template<typename T>
void CMatrix<T>::print() const
{
    for (int i = 0; i < row*col; i++)
        cout << pv[i];
}
#endif

```

(三) 循环队列模板



考试之前要认真复习和准备一下数组、堆栈、队列、矩阵等数据结构的正常写法和模板类写法

准备string类和vector类的用法

4.2 Exception

2020年6月26日 22:25

4.2、异常 Exception

程序在运行中，出现了不寻常的情况。（不包括语法错误）

C的解决方法：

-不采取任何措施，一旦程序出现异常，程序结果不可预知。没有下溢、上溢、除零检查；没有数组越界检查；出现非法月份(/日期)等等。

-**防御性编程**：程序中加入错误检测代码，当判断出错误的数据输入、错误的点击等，从函数返回一个特定的值。加入if语句即可

C的防御性编程的**不好之处**：不便于错误处理与恢复。它可能使相同的错误检测代码在程序的许多地方出现。并且将检测代码与程序进行正常处理的代码混合在一起，不便于程序的理解。

C++

异常处理的机制：

由类的设计者负责检查异常(when/where/what)，当发现异常时，并不在类中处理，而是将异常抛给类的使用者，由类的使用者来处理。



异常发生时所发生的：

```
try{
    A a;
    e = exception();
    throw e;
    A b;
}
catch(exception& m)
{
    m的有效范围;
}
```

刻画异常的方法1：用枚举组织多个异常符号：

- enum Matherr{OverFlow,UnderFlow,Zerodivide//.....};

//不需要定义异常类

- 类的定义：负责检测何时、何地发生什么类型的异常，并抛出该异常类型的对象

```
try{
    //类的使用
}
catch(Matherr m)
{ switch(m){
    case OverFlow:
        //....
    case UnderFlow:
        //...
    case ZeroDivide:
        //...
    }
}
```

这种方法简单。但由于异常对象采用枚举的形式，不能像类异常那样带有异常信息，有时使用起来并不方便。

刻画异常的方法2：定义一个类表示异常

```
//例：堆栈类模板的定义STACK.h
template <typename T>
class STACK{
private:
    T *v;    //用T类型指针v表示栈底
    T *p;    //栈高指示器
    const int size; //栈体大小
public:
    STACK(int z=100);
    void push(T a);
    inline T pop();
    inline T top() const;
```

```

inline int getsize() const;
inline bool empty() const;
virtual ~STACK()
{
    delete[] v;
    cout << "call ~STACK()!" << endl;
}
/*嵌套类的定义*/
class Range{
private:
    int level;
public:
    Range(int i=0):level(i)
    {
        cout << "构造一个异常对象: Range(int ii)" << endl;
    }
    virtual ~Range()
    {
        cout << "销毁一个异常对象: ~Range()" << endl;
    }
    Range(const Range& r)
    {
        level = r.level;
        cout << "复制一个异常对象: Range(const Range* r)" << endl;
    }
    int get() const
    {
        return level;
    }
}; //嵌套类
};

template <typename T>
STACK<T>::STACK(int z):size(z)
{
    p = v = new T[size];
}
template<typename T>
inline bool STACK<T>::empty() const
{
    return v==p;
}
template <typename T>
inline void STACK<T>::push(T ch)
{
    /*
    if (p-v >= size)
    {
        cout << "overflow!" << endl;
        exit(1);
    }
    */
    if (p-v >= size) //cout<<"overflow!"<<endl;    exit(1);
    {
        throw Range(1); //when/where/what The return optimization, 抛出这个对象, 表示栈满了
        //Range e(1); 这两行就是先定义对象, 再抛出, 这和上一行的意思是一样的
        //throw e;
    }
    else
        *p++ = ch;
}
template <typename T>
inline T STACK<T>::pop()
{
    if(p == v) {
        Range r(2); //因为2表示下溢(栈空了), 异常的话创建一个对象然后将其抛出
        throw r;
    }
    else

```



```

        return *--p;
    }
    template <typename T>
    inline T STACK<T>::top() const
    {
        if(p == v)
            throw Range(2);    //2表示栈空，如果异常，就抛出
        else
            return *(p-1);
    }
    template <typename T>
    inline int STACK<T>::getsize() const
    {
        return size;
    }
}

```

有异常，就要抛出。这里的Range类可以作为模板类的嵌套类，也可以在外面

在cpp中

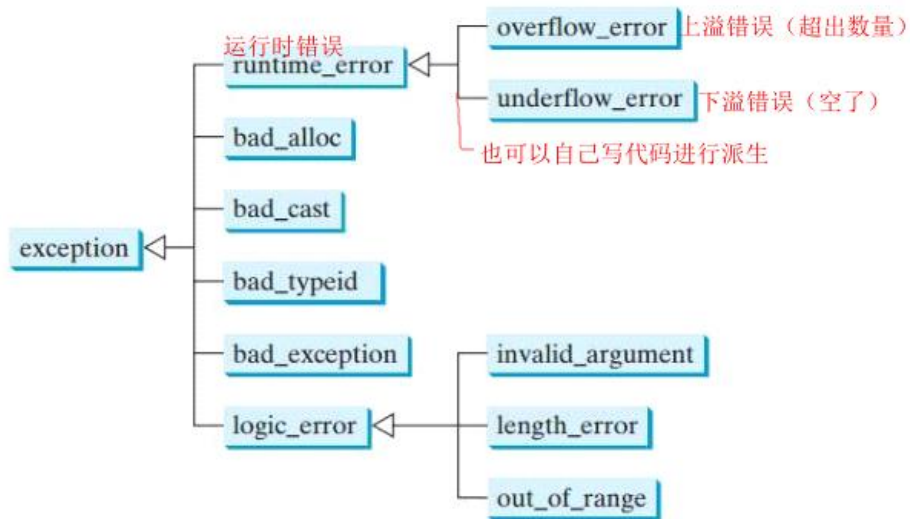
```

#include <iostream>
using namespace std;
#include "GenericSTACK.h"
void main()
{
    try{ //有异常抛出的代码放在try块中
        STACK<int> s(10);
        s.push(1);
        s.push(2);
        s.push(3);
        for (int i = 0; i < 10; i++)
            s.push(i);
        int a = s.pop();
        int b = s.pop();
        int c = s.pop();
        cout<<s.getsize()<<" "<<a<<b<<c<<endl;
    } //出现异常，就从try块中退出，并且try中后面的代码就不执行了，并且析构掉已经创建的对象，再执行catch语句

    catch(STACK<int>::Range& m) //STACK<int>::Range m; 因为定义的表示异常的类是嵌套类，所以这里不能漏了作用域
    {
        STACK<int>; 当然，也不能写成STACK<T>，因为是具体的类
        {
            if (1==m.get())
                cout<<"overflow! "<<endl;
            else if (2==m.get())
                cout << "underflow! " << endl;
            //char k = getchar();
        }
        catch(...) //这里必须写3个点，
        {
            //...
        }
        getchar();
    }
}

```

C++标准库提供的异常类层次：



C++标准异常类的使用:

标准类 `exception` 是由所选语言结构或 C++ 标准库抛出的所有异常的基类。类 `exception` 的对象可以被构造、复制，销毁。虚成员函数 `what()` 返回了描述异常的字符串。

```

namespace std {
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};

#include <iostream>
#include <stdexcept>
using namespace std;
int quotient(int number1, int number2)
{
    if ( 0 == number2 )
        throw runtime_error("Divisor cannot be zero");//相当于这个类有的带参构造函数，参数是字符指针
    return number1 / number2;
}

int main()
{
    // Read two integers
    cout << "Enter two integers: ";
    int number1, number2;
    cin >> number1 >> number2;
    try
    {
        int result = quotient(number1, number2);
        cout << number1 << " / " << number2 << " is " << result << endl;
    }
    catch (runtime_error& ex)
    {
        cout << ex.what() << endl;
    }
    cout << "Execution continues ..." << endl;
    return 0;
}

try{ //code }
catch(overflow_error &)
{
    //处理代码
}
catch(underflow_error &)
{
    //处理代码
}
  
```

```
}
catch(runtime_error &)
{
    //处理除上述以外的从runtime_error派生的任何异常的通用处理
}
catch(exception &)
{
    //处理除上述以外的从exception派生的任何异常的通用处理
}
catch(...)
{
    //处理除上述以外的任何错误的通用处理
}
```

应按照先派生类，后基类，最后省略参数的形式来排列catch语句。

五、Learn by myself

2020年7月1日 16:00

5.1 string类

2020年7月1日 12:05

一、C++中string常用函数用法总结

string (s小写) 是C++标准库中的类, 纯C中没有, 使用时需要包含头文件`#include<string>`, 注意不是`<string.h>`, 下面记录一下string中比较常用的用法。

string的定义及初始化

```
string s1 = "hello"; //初始化字符串
string s2 ("world"); //另一种初始化
string s3; //初始化字符串, 空字符串
string s4(5, 'a'); //s4由连续5个a组成, 即s4="aaaaa";
string s5(s1, 2, 3); //从s1的2位置的字符开始, 连续3个字符赋值给s5, 即s5="llo";
string s6(s1, 1); //从s1的2位置的字符开始, 将后续的所有字符赋值给s6, 即s6="ello";
```

string的读入

当使用C++的cin读入字符串时, 程序遇到空白字符就停止读取了。比如程序输入是:

```
"  hello  world"
```

那么当我们使用如下代码时, s1得到的只是"hello"。

```
string s1;
cin>>s1;
```

如果我们想读取一整行输入, 包括空格及空格后面的字符, 我们可以使用getline。

```
string str;
getline(cin, str);
cout << str << endl;
```

```
hello world !
hello world !
请按任意键继续. . .
```

重载的运算符

此处列举一下被重载的运算符, 基本意思一目了然。其中**注意“+”操作**

```
s1 = s2;
s1 += s2;
s1 = s2 + s3;
s1 == s2;
s1 = "s" + s2; //正确
s1 = "s" + "s"; //错误, 加号两边至少要有有一个string类型的对象
s1 = "s" + s2 + "s" + "s"; //正确
```

“+”的两边要保证至少有一个string类型, 所以5正确, 6错误。由于在C/C++中, +的返回值还是原类型, 所以第7行中, “s”+s2返回一个string类型, 因此string+ “s” 也是正确的。以此类推

遍历string (迭代器)

遍历string中的元素时, 我们可以使用类似C中的数组形式访问, 如s1[1], 也可以使用STL特有的迭代器访问:

表 3.6: 标准容器迭代器的运算符	
*iter	返回迭代器 iter 所指元素的引用
iter->mem	解引用 iter 并获取该元素的名为 mem 的成员, 等价于 (*iter).mem
++iter	令 iter 指示容器中的下一个元素
--iter	令 iter 指示容器中的上一个元素
iter1 == iter2	判断两个迭代器是否相等 (不相等), 如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器, 则相等; 反之, 不相等
iter1 != iter2	

```
string::iterator it;
for (it = s1.begin(); it != s1.end(); it++){
    cout << *it << endl;
}
cout << *(s1.begin()); //正确, 即访问s1[0]
cout << *(s1.end()); //错误, s1.end()指向了空
```

```
cout << *(s1.begin()); //正确，即访问s1[0]
cout << *(s1.end()); //错误，s1.end()指向了空
```

若想要从后向前遍历string时，可以用到rbegin()和rend()函数。

```
const_iterator begin() const;
iterator begin(); //返回string的起始位置
const_iterator end() const;
iterator end(); //返回string的最后一个字符后面的位置
const_iterator rbegin() const;
iterator rbegin(); //返回string的最后一个字符的位置
const_iterator rend() const;
iterator rend(); //返回string第一个字符位置的前面
```

插入insert ()

```
string s1 = "hello";
s1.insert(1, "ins"); //从s1的1位置开始，插入"ins"字符串，即s1="hinsello";
s1.insert(1, "ins", 2); //从s1的1位置开始，插入"ins"字符串的前2个字符，即s1="hinello";
s1.insert(1, "ins", 1, 2); //从s1的1位置开始，插入"ins"字符串的从1位置开始的2个字符，即s1="hnsello";
```

删除erase ()

```
iterator erase(iterator first, iterator last); //删除[first, last) 之间的所有字符，返回删除后迭代器的位置
iterator erase(iterator it); //删除it指向的字符，返回删除后迭代器的位置
string &erase(int pos = 0, int n = npos); //删除pos开始的n个字符，返回修改后的字符串
```

查找 find()

```
cout << s.find("aa", 0) << endl; //返回的是子串位置。第二个参数是查找的起始位置，如果找不到，就返回string::npos
if (s.find("aal", 0) == string::npos)
{
    cout << "找不到该子串!" << endl;
}
```

C++中string.find()函数与string::npos

查找字符串a是否包含子串b,

不是用strA.find(strB) > 0而是strA.find(strB) != string::npos

```
string::size_type pos = strA.find(strB);
if(pos != string::npos){
```

```
-----
int idx = str.find("abc");
if (idx == string::npos)
...
```

上述代码中，idx的类型被定义为int，这是错误的，即使定义为 unsigned int 也是错的，它必须定义为 string::size_type。

npos 是这样定义的：

```
static const size_type npos = -1;
```

因为 string::size_type (由字符串配置器 allocator 定义) 描述的是 size，故需为无符号整数型别。因为缺省配置器以型别 size_t 作为 size_type，于是 -1 被转换为无符号整数型别，npos 也就成了该型别的最大无符号值。不过实际数值还是取决于型别 size_type 的实际定义。不幸的是这些最大值都不相同。事实上，(unsigned long)-1 和 (unsigned short)-1 不同(前提是两者型别大小不同)。因此，比较式 idx == string::npos 中，如果 idx 的值为-1，由于 idx 和字符串string::npos 型别不同，比较结果可能得到 false。

要想判断 find() 的结果是否为npos，最好的办法是直接比较：

```
if (str.find("abc") == string::npos) { ... }
```

错误：if(str.find("abc"))

注：找不到abc会返回-1，不为0为True。0为False

```
    ///find函数返回类型 size_type
    string s("1a2b3c4d5e6f7g8h9i1a2b3c4d5e6f7g8ha9i");
    string flag;
    string::size_type position;
    //find 函数 返回jk 在s 中的下标位置
    position = s.find("jk");
    if (position != s.npos) //如果没找到，返回一个特别的标志c++中用npos表示，我这里npos取值是4294967295，
    {
```

```

    cout << "position is : " << position << endl;
}
else
{
    cout << "Not found the flag" + flag;
}

//find 函数 返回flag 中任意字符 在s 中第一次出现的下标位置
flag = "c";
position = s.find_first_of(flag);
cout << "s.find_first_of(flag) is : " << position << endl;
//从字符串s 下标5开始, 查找字符串b, 返回b 在s 中的下标
position=s.find("b", 5);
cout<<"s.find(b,5) is : "<<position<<endl;
//查找s 中flag 出现的所有位置。
flag="a";
position=0;
int i=1;
while((position=s.find_first_of(flag, position))!=string::npos)
{
    //position=s.find_first_of(flag, position);
    cout<<"position  "<<i<<" : "<<position<<endl;
    position++;
    i++;
}
//查找flag 中与s 第一个不匹配的位置
flag="acb12389efgxyz789";
position=flag.find_first_not_of (s);
cout<<"flag.find_first_not_of (s) : "<<position<<endl;
//反向查找, flag 在s 中最后出现的位置
flag="3";
position=s.rfind (flag);
cout<<"s.rfind (flag) : "<<position<<endl;
}

```

说明:

1. 如果string sub = " abc ";

```
string s = " cdeabcigld ";
```

s.find(sub) , s.rfind(sub) 这两个函数, 如果完全匹配, 才返回匹配的索引, 即: 当s中含有abc三个连续的字母时, 才返回当前索引。

s.find_first_of(sub), s.find_first_not_of(sub), s.find_last_of(sub), s.find_last_not_of(sub) 这四个函数, 查找s中含有sub中任意字母的索引。

2. 如果没有查询到, 则返回string::npos, 这是一个很大的数, 其值不需要知道

来自 <<https://www.cnblogs.com/web100/archive/2012/12/02/cpp-string-find-npos.html>>

string特性描述

可以用 ==、>、<、>=、<=、和!=比较字符串, 可以用+或者+=操作符连接两个字符串, 并且可以用[]获取特定的字符。

来自 <<https://www.cnblogs.com/X-Do-Better/p/8628492.html>>

可用下列函数来获得string的一些特性:

```

int capacity() const;    //返回当前容量 (即string中不必增加内存即可存放的元素个数)
int max_size() const;    //返回string对象中可存放的最大字符串的长度
int size() const;        //返回当前字符串的大小
int length() const;      //返回当前字符串的长度
bool empty() const;      //当前字符串是否为空
void resize(int len, char c); //把字符串当前大小置为len, 多去少补, 多出的字符c填充不足的部分

```

举例:

```

if (str.empty())
    cout<<"str is NULL."<<endl;
cout<<"str is "<<str<<endl;
cout<<"str's size is "<<str.size()<<endl;
cout<<"str's max size is "<<str.max_size()<<endl;
cout<<"str's length is "<<str.length()<<endl;
cout<<"str's capacity is "<<str.capacity()<<endl;

```

```
str.resize(20, 'c');cout<<"str is "<<str<<endl;
str.resize(5);cout<<"str is "<<str<<endl;
```

来自 <<https://www.cnblogs.com/X-Do-Better/p/8628492.html>>

其他常用函数

```
string &insert(int p,const string &s);    //在p位置插入字符串s
string &replace(int p, int n,const char *s); //删除从p开始的n个字符, 然后在p处插入串s
string &erase(int p, int n);    //删除p开始的n个字符, 返回修改后的字符串
string substr(int pos = 0,int n = npos) const;    //返回pos开始的n个字符组成的字符串
void swap(string &s2);    //交换当前字符串与s2的值
string &append(const char *s);    //把字符串s连接到当前字符串结尾
void push_back(char c)    //当前字符串尾部加一个字符c
const char *data()const;    //返回一个非null终止的c字符数组, data():与c_str()类似, 用于string转const char*其中它返回的数组是不以空字符终止.
const char *c_str()const;    //返回一个以null终止的c字符串, 即c_str()函数返回一个指向正规C字符串的指针, 内容与本string串相同, 用于string转const
```

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str1 = "abc123defg";
    string str2 = "swap!";
    cout<<str1<<endl;
    cout<<str1.erase(3,3)<<endl;    //从索引3开始的3个字符, 即删除掉了"123"
    cout<<str1.insert(0,"123")<<endl;    //在头部插入
    cout<<str1.append("123")<<endl;    //append()方法可以添加字符串
    str1.push_back('A');    //push_back()方法只能添加一个字符
    cout<<str1<<endl;
    cout<<str1.replace(0,3,"hello")<<endl;    //即将索引0开始的3个字符替换成"hello"
    cout<<str1.substr(5,7)<<endl;    //从索引5开始7个字节
    str1.swap(str2);    //交换str1和str2
    cout<<str1<<endl;
    const char* p = str1.c_str();    //返回一个指向正规c字符串的指针, 内容与str1一样
    printf("%s\n",p);
    system("pause");
    return 0;
}
```

总结:

C++——String类超详细介绍

原创 置顶 Xdut 2019-06-01 17:32:56 29880 收藏 156
分类专栏: C++

版权

(文中错误已更正, 欢迎及时指正错误! 谢谢)

STL的含义: 标准模板库

STL的内容:

- 容器: 数据的仓库
- 算法: 与数据结构相关的算法、通用的算法 (和数据结构无关)

注: 熟悉常用的算法 sort reverse

- 迭代器: 算法和容器的连接
- 适配器: 类似于转接线, 苹果线要连接安卓线

容器:

序列式容器 (线性结构)

string:

序列式容器（线性结构）

string:

array: C11静态顺序表

vector: 动态顺序表

list: 带头节点的双向循环链表

deque: 动态二维数组

forward_list: 带头结点的循环单链表

stack: 栈

queue: 队列

String类：按照类的方式进行动态管理字符串

底层：是一种顺序表的结构，元素是char类型的字符

string类的常用构造函数：

- string str——构造空的string类对象，即空字符串
- string str(str1)——str1 和 str 一样
- string str("ABC")——等价于 str="ABC"
- string str("ABC",strlen)——等价于 "ABC" 存入 str 中，最多存储 strlen 个字节
- string str("ABC",stridx,strlen)——等价于 "ABC" 的stridx 位置，作为字符串开头，存到str中，最多存储 strlen 个字节
- string str(srlen,'A')——存储 strlen 个 'A' 到 str 中

```
1 //用法小实例
2 #include<iostream>
3 using namespace std;
4
5 int main() {
6     string s1; //空字符串
7     string s2(10,'$'); //十个$
8     string s3("hello world"); //hello world
9
10    cout << s3.size << endl;
11    cout << s3.length << endl; //都是求字符串长度 11
12    cout << s3.capacity << endl; //求s3的容量
13
14    s3.clear(); //清空
15    string s4(s3); //s4 和 s3 一样
16 }
```

注：使用string类时，必须包含头文件以及using namespace std。

string常用成员函数：

assign函数：

- str.assign("ABC")——清空字符串，并设置为 "ABC"
- str.assign("ABC",2)——清空字符串，并设置为"AB"，保留两个字符
- str.assign("ABC",1,1)——清空字符串，设置为 "ABC" 中的从 位置1 开始，保留 1个 字符
- str.assign(5, 'A')——清空字符串，然后字符串设置为 5个 'A'

- str.length()——求字符串长度
- str.size()——和 length() 一样
- str.capacity()——获取容量，包含了不用增加内存就能使用的字符数
- str.resize(10)——设置当前 str 的大小为10，若大小与当前串的长度，\0 来填充

• `str.size()`——和 `length()` 一样

- `str.capacity()`——获取容量，包含了不用增加内存就能使用的字符数
- `str.resize(10)`——设置当前 `str` 的大小为10，若大小与当前串的长度，\0 来填充
- `str.resize(10,char c)`——设置当前 `str` 的大小为10，若大小与当前串的长度，字符 `c` 来填充
- `str.reserve(10)`——设置 `str` 的容量 10，不会填充数据
- `str.swap(str1)`——交换 `str1` 和 `str` 的字符串
- `str.push_back('A')`——在 `str` 末尾添加一个字符 'A'，参数必须是字符形式
- `str.append("ABC")`——在 `str` 末尾添加一个字符串 "ABC"，参数必须是字符串形式

insert函数方法:

- `str.insert(2,3,'A')`——在 `str` 下标为2的位置添加 3个 字符'A'
- `str.insert(2,"ABC")`——在 `str` 下标为2的位置添加 字符串 "ABC"
- `str.insert(2,"ABC",1)`——在 `str` 下标为2的位置添加 字符串 "ABC" 中 1个 字符
- `str.insert(2,"ABC",1,1)`——在 `str` 下标为2的位置添加 字符串 "ABC" 中从位置 1 开始的 1 个字符

注: 上个函数参数中加粗的 1，可以是 `string::npos`，这时候最大值，从 位置1 开始后面的全部字符

- `str.insert(iterator pos, size_type count, CharT ch)`——在 `str` 中，迭代器指向的 `pos`位置 插入 `count`个 字符 `ch`

`s4.insert(++str1.begin(),2,'a');` 结果: `s4: ABCD -> AaaBCD`

- `str.insert(iterator pos, InputIt first, InputIt last)`——在 `str` 中，`pos`位置 插入 `str1` 的 开始位置 到 结束为止

`s4.insert(s4.begin(),str1.begin(),str1.end());` 结果: `s4: ABCD str1: abc -> abcABCD`

- `str.erase(2)`——删除 下标2 的位置开始，之后的全删除
- `str.erase(2,1)`——删除 下标2 的位置开始，之后的 1个 删除
- `str.clear()`——删除 `str` 所有
- `str.replace(2,4,"abcd")`——从 下标2 的位置，替换 4个字节，为"abcd"
- `str.empty()`——判空

反转相关:

(位于头文件<algorithm>)

- `reverse(str.begin(),str.end())`——`str`的开始 到 结束字符反转

`reverse(s4.begin,s4.end);` 结果: `s4: ABCD -> DCBA`

查找相关:

查找成功返回位置，查找失败，返回-1

find函数: 从头查找

- `str.find('A')`——查找 'A'
- `str.find("ABC")`——查找 "ABC"

`int n=s4.find("ABC"); s4: ABCD -> n = 0`

- `str.find('B',1)`——从 位置1 处，查找'B'
- `str.find("ABC",1,2)`——从 位置1 处，开始查找 'ABC' 的前 2个 字符

rfind函数: 从尾部查找

- `str.rfind('A')`——查找 'A'
- `str.rfind("ABC")`——查找 "ABC"

`int n=s4.rfind("ABC"); s4: AAAABCD -> n = 3`

- `str.rfind('B',1)`——从 位置1 处，向前查找'B'
- `str.rfind("ABC",1,2)`——从 位置1 处，开始向前查找 'ABC' 的前 2个 字符

- `str.rfind('B',1)`——从 位置1 处, 向前查找'B'
- `str.rfind("ABC",1,2)`——从 位置1 处, 开始向前查找 'ABC' 的前 2个 字符

find_first_of()函数:

查找是否包含有子串中任何一个字符

- `str.find_first_of("abBc")`——查找 "abBc" 和str 相等的任何字符, "abBc" 中有就返回位置
- `str.find_first_of("abBc",1)`——查找 "abBc" 和str 相等的任何字符, 从 位置1 处, 开始查找"abBc" 中的字符, "abBc" 中有的就返回位置
- `str.find_first_of("abBc",1,2)`——查找 "abBc" 和str 相等的任何字符, 从 位置1 处, 开始查找"abBc" 的前 2个 字符, "abBc" 中有的就返回位置

find_last_of()函数:

`find_first_not_of()` 末尾查找, 从末尾处开始, 向前查找是否包含有子串中任何一个字符

- `str.find_last_of("abBc")`——查找 "abBc" 和str 相等的任何字符, 向前查找, "abBc" 中有的返回位置
- `str.find_last_of("abBc",1)`——查找 "abBc" 和str 相等的任何字符, 从 位置1 处, 开始向前查找"abBc" 中的字符, "abBc" 中有的就返回位置
- `str.find_last_of("abBc",10,2)`——查找 "abBc" 和str 相等的任何字符, 从 位置10 处, 开始向前查找"abBc" 的前 2个 字符, "abBc" 中有的就返回位置

拷贝相关的:

- `str1=str.substr(2)`——提取子串, 提取出 str 的下标为2 到末尾, 给 str1
- `str1=str.substr(2,3)`——提取子串, 提取出 str 的下标为2 开始, 提取三个字节, 给 str1
- `const char* s1=str.data()`——将string类转为字符串数组, 返回给s1

`char* s=new char[10]`

- `str.copy(s,count,pos)`——将 str 里的 pos 位置开始, 拷贝 count个 字符,存到 s 里

比较相关的函数: (改部分已经在VS2013中验证, 错误已更正, 如还有, 请指出)

compare函数: (str原串) 与 (str新串) ASCII值相等返回0; (str原串) 小于 (str新串) 返回-1; (str原串) 大于 (str新串) 返回1。

示例对象: `string str("abcd")`

- `str.compare("abcd")`——返回0。
- `str.compare("abce")`——返回-1。
- `str.compare("abcc")`——返回1。
- `str.compare(0,2,str,2,2)`——用str的 下标0 开始的 2个字符 和 str的 下标2 开始的 2个 字符比较——就是用 "ab" 和 "cd" " 比较, 结果返回-1。
- `str.compare(1,2," bcx" ,2)`——用str的 下标1 开始的 2个字符 和 "bcx" 的前 2个 字符比较——就是用 "bc" 和 "bc" " 比较, 返回0。

5.2 vector类

2020年7月1日 16:00

在c++中，vector是一个十分有用的容器，下面对这个容器做一下总结。

1 基本操作

- (1) 头文件#include<vector>.
- (2) 创建vector对象，vector<int> vec;
- (3) 尾部插入数字：vec.push_back(a);
- (4) 使用下标访问元素，cout<<vec[0]<<endl;记住下标是从0开始的。
- (5) 使用迭代器访问元素.

```
vector<int>::iterator it;
for(it=vec.begin();it!=vec.end();it++)
    cout<<*it<<endl;
```

- (6) 插入元素：vec.insert(vec.begin()+i, a);在第i+1个元素前面插入a;
- (7) 删除元素：vec.erase(vec.begin()+2);删除第3个元素
vec.erase(vec.begin()+i, vec.begin()+j);删除区间[i, j-1];区间从0开始
- (8) 向量大小:vec.size();
- (9) 清空:vec.clear();

2

vector的元素不仅仅可以使int, double, string, 还可以是结构体，但是要注意：结构体要定义为全局的，否则会出错。

3 算法

- (1) 使用reverse将元素翻转：需要头文件#include<algorithm>

```
reverse(vec.begin(), vec.end()); //将元素翻转(在vector中，如果一个函数中需要两个迭代器，一般后一个都不包含.)
```

- (2) 使用sort排序：需要头文件#include<algorithm>，

```
sort(vec.begin(), vec.end()); //默认是按升序排列，即从小到大).
```

可以通过重写排序比较函数按照降序比较，如下：

定义排序比较函数：

```
bool Comp(const int &a, const int &b)
{
    return a>b;
}
```

调用时:sort(vec.begin(), vec.end(), Comp)，这样就降序排序。

来自 <<https://www.cnblogs.com/wang7/archive/2012/04/27/2474138.html>>

六、备考总结

2020年8月28日 23:31

- 判断是不是虚函数：
该函数是否与基类的虚函数有相同的 ①函数名称 ②参数个数及 ③对应参数类型 ④返回值 ⑤const
若是虚函数，则发生动态绑定的时候若基类引用或指针引用或者指向派生类，调用的是派生类的相应函数，否则只调用基类的
- 引用、常量、成员对象 必须使用构造函数初始化列表
- return的时候先调用拷贝构造函数，再调用析构函数
- 异常处理的时候，try块中的throw也是一样的。如果有异常类，则需要调用异常类的拷贝构造函数和析构函数，最后还要调用try块中其他对象的析构函数。Summary:一旦有异常抛出，try块中下面的语句就不再执行，并要析构已经存在的对象（包括异常类的对象和其他对象）
- 有new出现的地方，退出的时候一定要加上delete。一般在一段语句的末尾都要加上delete
- 引用被创建的时候就要初始化，并且之后不会被修改他被谁引用了，但是可以通过他改变原始的值 [引用赋值](#)
- 创建了对象，函数返回或者出其函数作用域的时候，就要调用类的析构函数。 但是如果是用指针创建了，比如A *p=new B[2],则delete的时候是删除指针A的，所以会调用A的析构函数——不过这样的做法会不安全
- 静态成员对象和静态成员函数 [static](#)
- 派生类中的析构函数必须写成virtual，也不能不写析构函数