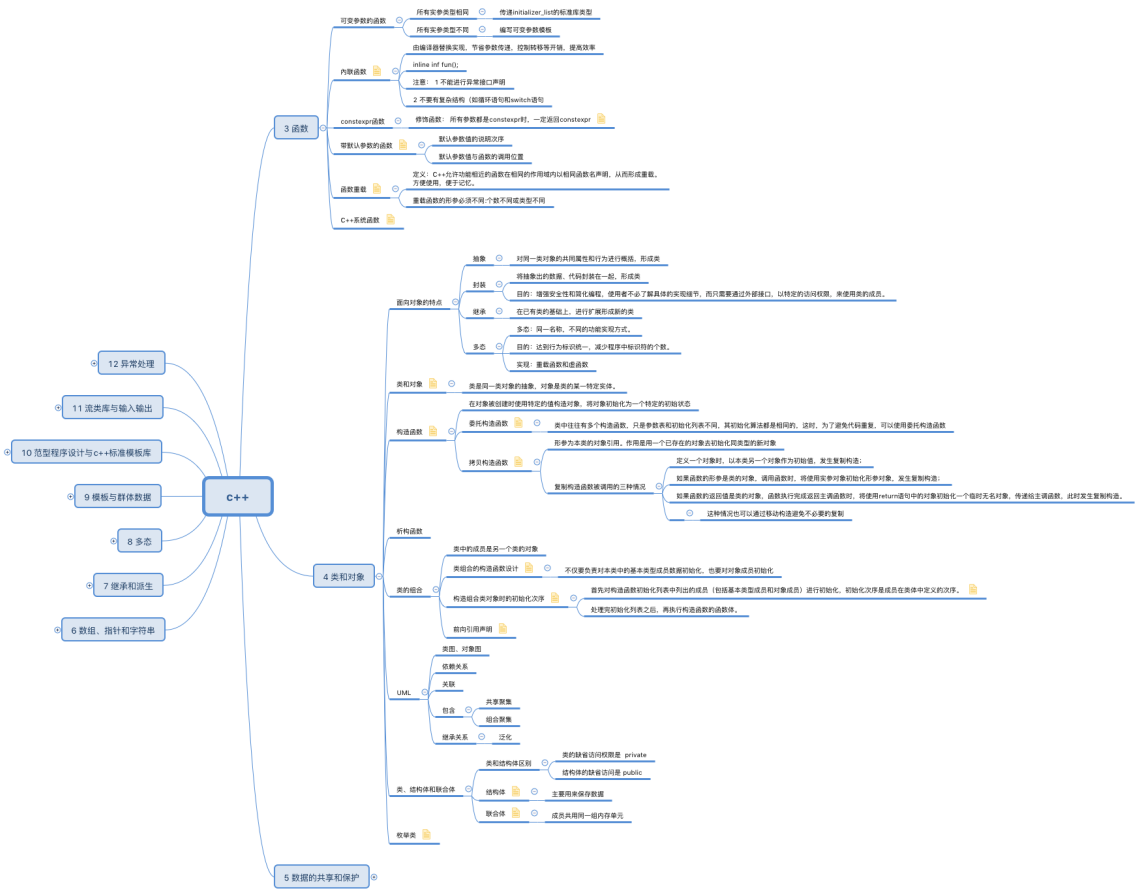
c++



1. 3 函数

1.1. 可变参数的函数

1.1.1. 所有实参类型相同

1.1.1.1. 传递initializer_list的标准库类型

1.1.2. 所有实参类型不同

1.1.2.1. 编写可变参数模板

1.2. 内联函数

声明时使用关键字 inline。

编译时在调用处用函数体进行替换，节省了参数传递、控制转移等开销。

注意：

内联函数体内不能有循环语句和switch语句；

内联函数的定义必须出现在内联函数第一次被调用之前；

对内联函数不能进行异常接口声明。

1.2.1. 由编译器替换实现，节省参数传递，控制转移等开销，提高效率

1.2.2. inline inf fun();

1.2.3. 注意： 1 不能进行异常接口声明

1.2.4. 2 不要有复杂结构（如循环语句和switch语句

1.3. constexpr函数

1.3.1. 修饰函数： 所有参数都是constexpr时，一定返回constexpr

```
constexpr int get_size() {return 20;}

constexpr int foo = get_size();
```

1.4. 带默认参数的函数

默认参数值的说明次序

有默认参数的形参必须列在形参列表的最右，即默认参数值的右面不能有无默认值的参数；

调用时实参与形参的结合次序是从左向右。

例：

int add(int x, int y = 5, int z = 6);//正确

```
int add(int x = 1, int y = 5, int z);//错误
int add(int x = 1, int y, int z = 6);//错误
```

默认参数值与函数的调用位置

如果一个函数有原型声明，且原型声明在定义之前，则默认参数值应在函数原型声明中给出；如果只有函数的定义，或函数定义在前，则默认参数值可以函数定义中给出。

例：

```
int add(int x = 5,int y = 6);
//原型声明在前
int main() {
  add();
}
int add(int x,int y) {
//此处不能再指定默认值
  return x + y;
}
int add(int x = 5,int y = 6) {
//只有定义，没有原型声明
  return  x + y;
}
int main() {
  add();
}
```

1.4.1. 默认参数值的说明次序

1.4.2. 默认参数值与函数的调用位置

1.5. 函数重载

不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆。这样不好：

1.5.1. 定义：C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载。

方便使用，便于记忆。

1.5.2. 重载函数的形参必须不同:个数不同或类型不同

1.6. C++系统函数

C++的系统库中提供了几百个函数可供程序员使用，例如：

求平方根函数（sprt）

求绝对值函数（abs）

使用系统函数时要包含相应的头文件，例如：

cmath

2. 4 类和对象

2.1. 面向对象的特点

2.1.1. 抽象

2.1.1.1. 对同一类对象的共同属性和行为进行概括，形成类

2.1.2. 封装

2.1.2.1. 将抽象出的数据、代码封装在一起，形成类

2.1.2.2. 目的：增强安全性和简化编程，使用者不必了解具体的实现细节，而只需要通过外部接口，以特定的访问权限，来使用类的成员。

2.1.3. 继承

2.1.3.1. 在已有类的基础上，进行扩展形成新的类

2.1.4. 多态

2.1.4.1. 多态：同一名称，不同的功能实现方式。

2.1.4.2. 目的：达到行为标识统一，减少程序中标识符的个数。

2.1.4.3. 实现：重载函数和虚函数

2.2. 类和对象

```
class 类名称
{
  public:
            公有成员（外部接口）
  private:
             私有成员
  protected:
             保护型成员
}
```

类成员的访问控制

公有类型成员

在关键字public后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数。

私有类型成员

在关键字private后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。

如果紧跟在类名称的后面声明私有成员，则关键字private可以省略。

保护类型成员

与private类似，其差别表现在继承与派生时对派生类的影响不同

类的成员函数

在类中说明函数原型；

可以在类外给出函数体实现，并在函数名前使用类名加以限定；

也可以直接在类中给出函数体，形成内联

在类中声明内联成员函数的方式：

将函数体放在类的声明中。

使用inline关键字。

2.2.1. 类是同一类对象的抽象，对象是类的某一特定实体。

2.3. 构造函数

构造函数的形式

函数名与类名相同；

不能定义返回值类型，也不能有return语句；

可以有形式参数，也可以没有形式参数；

可以是内联函数；

可以重载；

可以带默认参数值。

默认构造函数

调用时可以不需要实参的构造函数

参数表为空的构造函数

全部参数都有默认值的构造函数

默认构造函数

调用时可以不需要实参的构造函数

参数表为空的构造函数

全部参数都有默认值的构造函数

下面两个都是默认构造函数，如在类中同时出现，将产生编译错误：

```
Clock();
Clock(int newH=0,int newM=0,int newS=0);
```

隐含生成的构造函数

如果程序中未定义构造函数，编译器将在需要时自动生成一个默认构造函数

参数列表为空，不为数据成员设置初始值；

如果类内定义了成员的初始值，则使用内类定义的初始值；

如果没有定义类内的初始值，则以默认方式初始化；

基本类型的数据默认初始化的值是不确定的。

“=default”

如果程序中未定义构造函数，默认情况下编译器就不再隐含生成默认构造函数。如果此时依然希望编译器隐含生成默认构造函数，可以使用“=default”。

例如

```
class Clock {
public:
    Clock() =default; //指示编译器提供默认构造函数
    Clock(int newH, int newM, int newS); //构造函数
private:
    int hour, minute, second;
};
```

2.3.1. 在对象被创建时使用特定的值构造对象，将对象初始化为一个特定的初始状态

2.3.2. 委托构造函数

Clock类的两个构造函数：

Clock(int newH, int newM, int newS) : hour(newH),minute(newM), second(newS) { //构造函数

}

Clock::Clock(): hour(0),minute(0),second(0) { }//默认构造函数

委托构造函数

委托构造函数使用类的其他构造函数执行初始化过程

例如：

Clock(int newH, int newM, int newS): hour(newH),minute(newM), second(newS){

}

Clock(): Clock(0, 0, 0) { }

2.3.2.1. 类中往往有多个构造函数，只是参数表和初始化列表不同，其初始化算法都是相同的，这时，为了避免代码重复，可以使用委托构造函数

2.3.3. 拷贝构造函数

class 类名 {

public :

类名（形参）；//构造函数

类名（const 类名 &对象名）；//复制构造函数

// ...

}；

类名::类（ const 类名 &对象名）//复制构造函数的实现

{ 函数体 }

2.3.3.1. 形参为本类的对象引用。作用是用一个已存在的对象去初始化同类型的新对象

2.3.3.2. 复制构造函数被调用的三种情况

2.3.3.2.1. 定义一个对象时，以本类另一个对象作为初始值，发生复制构造；

2.3.3.2.2. 如果函数的形参是类的对象，调用函数时，将使用实参对象初始化形参对象，发生复制构造；

2.3.3.2.3. 如果函数的返回值是类的对象，函数执行完成返回主调函数时，将使用return语句中的对象初始化一个临时无名对象，传递给主调函数，此时发生复制构造。

2.3.3.2.4.

2.3.3.2.4.1. 这种情况也可以通过移动构造避免不必要的复制

2.4. 析构函数

2.5. 类的组合

2.5.1. 类中的成员是另一个类的对象

2.5.2. 类组合的构造函数设计

```
类名::类名(对象成员所需的形参，本类成员形参)
    :对象1(参数)，对象2(参数)，.....
{
//函数体其他语句
}
```

2.5.2.1. 不仅要负责对本类中的基本类型成员数据初始化，也要对对象成员初始化

2.5.3. 构造组合类对象时的初始化次序

```
//4_4.cpp
#include <iostream>
#include <cmath>
using namespace std;
class Point {  //Point类定义
public:
    Point(int xx = 0, int yy = 0) {
        x = xx;
        y = yy;
    }
    Point(Point &p);
    int getX() { return x; }
    int getY() { return y; }
private:
    int x, y;
};

Point::Point(Point &p) { //复制构造函数的实现
    x = p.x;
    y = p.y;
    cout << "Calling the copy constructor of Point" << endl;
}

//类的组合
class Line {   //Line类的定义
public:  //外部接口
    Line(Point xp1, Point xp2);
    Line(Line &l);
    double getLen() { return len; }
private: //私有数据成员
    Point p1, p2;  //Point类的对象p1,p2
```

```
    double len;
};

//组合类的构造函数
Line::Line(Point xp1, Point xp2) : p1(xp1), p2(xp2) {
    cout << "Calling constructor of Line" << endl;
    double x = static_cast<double>(p1.getX() - p2.getX());
    double y = static_cast<double>(p1.getY() - p2.getY());
    len = sqrt(x * x + y * y);
}
Line::Line (Line &l): p1(l.p1), p2(l.p2) {//组合类的复制构造函数
    cout << "Calling the copy constructor of Line" << endl;
    len = l.len;
}

//主函数
int main() {
    Point myp1(1, 1), myp2(4, 5); //建立Point类的对象
    Line line(myp1, myp2);  //建立Line类的对象
    Line line2(line); //利用复制构造函数建立一个新对象
    cout << "The length of the line is: ";
    cout << line.getLen() << endl;
    cout << "The length of the line2 is: ";
    cout << line2.getLen() << endl;
    return 0;
}
```

结果；

1 Line - 》构造函数

Line - 》形参

形参 myp2(4, 5) - 》point构造函数

形参 myp1(1, 1) - 》point构造函数

Line - 》构造函数的初始化列表

p1(xp1) - 》point构造函数

p2(xp2) - 》point构造函数

2 Line::Line (Line &l): -》拷贝构造函数

p1(xp1) -》point构造函数

p2(xp2) -》point构造函数

2.5.3.1. 首先对构造函数初始化列表中列出的成员（包括基本类型成员和对象成员）进行初始化，初始化次序是成员在类体中定义的次序。

成员对象构造函数调用顺序：按对象成员的声明顺序，先声明者先构造。

初始化列表中未出现的成员对象，调用用默认构造函数（即无形参的）初始化

2.5.3.2. 处理完初始化列表之后，再执行构造函数的函数体。

2.5.4. 前向引用声明

在提供一个完整的类声明之前，不能声明该类的对象，也不能在内联成员函数中使用该类的对象。

当使用前向引用声明时，只能使用被声明的符号，而不能涉及类的任何细节。

2.6. UML

2.6.1. 类图、对象图

2.6.2. 依赖关系

2.6.3. 关联

2.6.4. 包含

2.6.4.1. 共享聚集

2.6.4.2. 组合聚集

2.6.5. 继承关系

2.6.5.1. 泛化

2.7. 类、结构体和联合体

2.7.1. 类和结构体区别

2.7.1.1. 类的缺省访问权限是 private

2.7.1.2. 结构体的缺省访问是 public

2.7.2. 结构体

```
struct 结构体名称 {
    公有成员
protected:
   保护型成员
private:
    私有成员
};
```

2.7.2.1. 主要用来保存数据

2.7.3. 联合体

```
union 联合体名称 {
   公有成员
protected:
   保护型成员
private:
   私有成员
};
```

2.7.3.1. 成员共用同一组内存单元

2.8. 枚举类

```
enum class 枚举类型名: 底层类型 {枚举值列表};
```

底层类型默认为int

例：

```
enum class Type { General, Light, Medium, Heavy};
enum class Type: char { General, Light, Medium, Heavy};
enum class Category { General=1, Pistol, MachineGun, Cannon};
```

3. 5 数据的共享和保护

3.1. 标识符的作用域和可见性

```
int i; //全局变量，文件作用域
int main() {
   i = 5; //为全局变量i赋值
   {
      int i; //局部变量，局部作用域
```

```
    i = 7;
    cout << "i = " << i << endl;//输出7
  }
  cout << “i = ” << i << endl;//输出5
  return 0;
}
```

3.1.1. 函数原型作用域

3.1.1.1. 函数原型中的参数，其作用域始于"("，结束于")"

3.1.2. 局部作用域(块作用域)

```
void fun(int a) {
  int b = a;
  cin >> b;
  if (b > 0) {
    int c;

    ......
  }
}
```

3.1.2.1. 函数的形参、在块中声明的标识符；

3.1.2.2. 其作用域自声明处起，限于块中。

3.1.3. 类作用域

3.1.4. 文件作用域

3.1.5. 命名空间作用域

3.2. 对象的生存期

3.2.1. 静态生存期

生存期与程序的运行期相同。

在文件作用域中声明的对象具有这种生存期。

在函数内部声明静态生存期对象，要冠以关键字static

3.2.2. 动态生存期

3.2.2.1. 块作用域中声明的，没有用static修饰的对象

3.3. 类的静态数据成员

3.3.1. 为该类的所有对象共享，具有静态生存期

3.3.2. 必须在类外定义和初始化，用：：之名所属类

```
int Point::count = 0;//静态数据成员定义和初始化，使用类名限定
```

3.4. 类的静态成员函数

3.4.1. 类外代码可以使用类名和作用域操作符来调用静态成员函数

3.4.2. 静态成员函数主要用于处理该类的静态数据成员，可以直接调用静态成员函数。

3.4.3. 如果访问非静态成员，要通过对象来访问。

3.5. 类的友元

3.5.1. 会破坏数据封装和数据隐藏

3.5.2. 可以使用友元函数和友元类

3.5.2.1. 友元函数

```
#include <iostream>
#include <cmath>
class Point {  //Point类声明
public:  //外部接口
    Point(int x=0, int y=0) : x(x), y(y) { }
    int getX() { return x; }
    int getY() { return y; }
    friend float dist(Point &a, Point &b);
private: //私有数据成员
    int x, y;
};

float dist( Point& a, Point& b) {
  double x = a.x - b.x;
  double y = a.y - b.y;
  return static_cast<float>(sqrt(x * x + y * y));
}
int main() {
  Point p1(1, 1), p2(4, 5);
  cout <<"The distance is: ";
  cout << dist(p1, p2) << endl;
  return 0;
}
```

3.5.2.1.1. 类声明中由关键字friend修饰说明的非成员函数，

3.5.2.1.2. 在它的函数体中能够通过对象名访问 private 和 protected成员

3.5.2.1.3. 访问对象中的成员必须通过对象名

3.5.2.2. 友元类

```
class A {
  friend class B;
public:
  void display() {
    cout << x << endl;
  }
private:
  int x;
};

class B {
public:
  void set(int i);
  void display();
private:
  A a;
};




void B::set(int i) {
   a.x=i;
}
void B::display() {
   a.display();
};
```

3.5.2.2.1. 若一个类为另一个类的友元，则此类的所有成员都能访问对方类的私有成员

3.5.2.2.2. 声明语法：将友元类名在另一个类中使用friend修饰说明

3.5.2.2.3. 类的友元关系是单向的

3.6. 数据的共享和保护

3.6.1. 对于既需要共享、又需要防止改变的数据应该声明为常类型（用const进行修饰）。

3.6.2. 常类型

3.6.2.1. 常对象：必须进行初始化,不能被更新。

```
class A
{
  public:
    A(int i,int j) {x=i; y=j;}
                    ...
  private:
    int x,y;
};
A const a(3,4); //a是常对象，不能被更新
```

3.6.2.1.1. const 类名 对象名

3.6.2.2. 常成员

常成员函数

使用const关键字说明的函数。

常成员函数不更新对象的数据成员。

常成员函数说明格式：

类型说明符 函数名（参数表）const;

这里，const是函数类型的一个组成部分，因此在实现部分也要带const关键字。

const关键字可以被用于参与对重载函数的区分

通过常对象只能调用它的常成员函数。

常数据成员

```
#include<iostream>
using namespace std;
class R {
public:
  R(int r1, int r2) : r1(r1), r2(r2) { }
  void print();
  void print() const;
private:
  int r1, r2;
};

void R::print() {
  cout << r1 << ":" << r2 << endl;
}
void R::print() const {
  cout << r1 << ";" << r2 << endl;
}
int main() {
  R a(5,4);
  a.print(); //调用void print()
  const R b(20,52);
  b.print(); //调用void print() const
    return 0;
}
```

3.6.2.2.1. 用const进行修饰的类成员：常数据成员和常函数成员

3.6.2.2.2. 类型说明符 函数名（参数表）const

3.6.2.3. 常引用：被引用的对象不能被更新。

3.6.2.3.1. const 类型说明符 &引用名

3.6.2.4. 常数组：数组元素不能被更新。

3.6.2.4.1. 类型说明符 const 数组名[大小]...

3.6.2.5. 常指针：指向常量的指针。

3.6.3. 对于不改变对象状态的成员函数应该声明为常函数。

3.7. 多文件结构和预编译命令

3.7.1. 外部变量

3.7.2. 外部函数

3.7.3. 标准C++库

输入/输出类

容器类与抽象数据类型

存储管理类

算法

错误处理

运行环境支持

3.7.4. 编译预处理

3.7.4.1. #include 包含指令

将一个源文件嵌入到当前源文件中该点处。

#include<文件名>

按标准方式搜索，文件位于C++系统目录的include子目录下

#include"文件名"

首先在当前目录中搜索，若没有，再按标准方式搜索。

3.7.4.2. #define 宏定义指令

3.7.4.3. #undef

3.7.5. 条件编译指令——#if #elif #else #endif

#ifdef 标识符

程序段1

#else

程序段2

#endif

4. 6 数组、指针和字符串

4.1. 数组

4.1.1. 对象数组

4.1.1.1. 初始化时，每个元素都会调用构造函数

4.1.1.2. 基于范围的for循环

```
int main()
{
int array[3] = {1,2,3};
int *p;
for(p = array; p < array + sizeof(array) / sizeof(int); ++p)
{
```

```
    *p += 2;
    std::cout << *p << std::endl;
}

return 0;
}

int main()
{
int array[3] = {1,2,3};
for(int & e : array)
{
    e += 2;
    std::cout<<e<<std::endl;
}

return 0;
}
```

4.1.1.2.1. C++11

4.2. 指针

4.2.1. 常量指针

不能通过指向常量的指针改变所指对象的值，但指针本身可以改变，可以指向另外的对象

```
int a;
    const int *p1 = &a;//p1是指向常量的指针
    int b;
    p1 = &b;   //正确，p1本身的值可以改变
    *p1 = 1;    //编译时出错，不能通过p1改变所指的对象
```

4.2.1.1. 指向常量的指针

4.2.2. 指针常量

4.2.2.1. 指针类型的常量

若声明指针常量，则指针本身的值不能被改变。

```
int a;
int * const p2 = &a;
p2 = &b;   //错误，p2是指针常量，值不能改变
```

4.2.3. 数组指针

4.2.4. 指针数组

4.2.4.1. 指针类型的数组

4.2.5. 函数指针

定义

存储类型 数据类型 (*函数指针名)();

含义

函数指针指向的是程序代码存储区

4.2.5.1. 指向函数的指针

4.2.5.1.1. 函数回调

4.2.6. 指针函数

4.2.6.1. 指针类型的函数

4.2.7. 对象指针

类名 *对象指针名；

例:

```
Point a(5,10);
Piont *ptr;
ptr=&a;
```

通过指针访问对象成员

对象指针名->成员名

例：ptr->getx() 相当于 (*ptr).getx();

4.2.7.1. this指针

4.2.7.1.1. 指向当前对象自己

4.2.8. this指针

4.2.8.1. 指向当前对象自己

4.2.8.2. 隐含于类的每一个非静态成

4.2.8.3. 指出成员函数所操作的对象

当通过一个对象调用成员函数时，系统先将该对象的地址赋给this指针，然后调用成员函数，成员函数对对象的数据成员进行操作时，就隐含使用了this指针。

例如：Point类的getX函数中的语句：

return x;

相当于：

return this->x;

4.2.9. 智能指针

4.2.9.1. unique_ptr

：不允许多个指针共享资源，可以用标准库中的move函数转移指针

4.2.9.2. shared_ptr ：多个指针共享资源

4.2.9.3. weak_ptr ：可复制shared_ptr，但其构造或者释放对资源不产生影响

4.3. 字符串

4.3.1. 字符串常量

4.3.2. string类

string类常用的构造函数

string(); //默认构造函数，建立一个长度为0的串

例：

string s1;

string(const char *s); //用指针s所指向的字符串常量初始化string对象

例：

string s2 = “abc”;

string(const string& rhs); //复制构造函数

例：

string s3 = s2;

4.3.2.1. 对字符数组操作的封装

4.3.3. 输入整行字符串

4.3.3.1. getline

getline可以输入整行字符串（要包string头文件），例如：

getline(cin, s2);

输入字符串时，可以使用其它分隔符作为字符串结束的标志（例如逗号、分号），将分隔符作为getlin

e的第3个参数即可，例如：

getline(cin, s2, ',');

4.4. 动态内存分配

new 类型名T（初始化参数列表）

功能：在程序执行期间，申请用于存放T类型对象的内存空间，并依初值列表赋以初值。

结果值：成功：T类型的指针，指向新分配的内存；失败：抛出异常。

4.4.1. 动态申请内存操作符 new

4.4.2. 释放内存操作符delete

delete 指针p

功能：释放指针p所指向的内存。p必须是new操作的返回值。

4.4.3. 分配和释放动态数组

分配：new 类型名T [数组长度]

数组长度可以是任何表达式，在运行时计算

释放：delete[] 数组名p

释放指针p所指向的数组。

4.4.3.1. new 类型名T [数组长度]

4.4.3.2. delete[] 数组名p

4.4.4. 动态创建多维数组

```
#include <iostream>
using namespace std;
int main() {
  int (*cp)[9][8] = new int[7][9][8];
  for (int i = 0; i < 7; i++)
     for (int j = 0; j < 9; j++)
        for (int k = 0; k < 8; k++)
           *(*(*(cp + i) + j) + k) = ( i * 100 + j * 10 + k);
  for (int i = 0; i < 7; i++) {
      for (int j = 0; j < 9; j++) {
         for (int k = 0; k < 8; k++)
          cout << cp[i][j][k] << "  ";
         cout << endl;
      }
      cout << endl;
    }
    delete[] cp;
    return 0;
}
```

4.4.4.1. new 类型名T[第1维长度][第2维长度]

4.4.5. 动态数组类

4.4.6. vector对象

vector对象的定义

vector<元素类型> 数组对象名(数组长度);

例：

vector<int> arr(5)建立大小为5的int数组

vector对象的使用

对数组元素的引用

与普通数组具有相同形式：

vector对象名 [下标表达式]

vector数组对象名不表示数组首地址

获得数组长度

用size函数

数组对象名.size()

4.4.6.1. 基于范围的for - - 使用vector

```
#include <vector>
#include <iostream>
int main()
{
std::vector<int> v = {1,2,3};
for(auto i = v.begin(); i != v.end(); ++i)
    std::cout << *i << std::endl;

for(auto e : v)
std::cout << e << std::endl;
```

}

auto：自动变化类型

4.5. 对象复制与移动

4.5.1. 浅层复制

4.5.1.1. 实现对象间数据元素的一一对应复制

4.5.2. 深层复制

4.5.2.1. 当被复制的对象数据成员是指针类型时，

不是复制该指针成员本身，而是将指针所指对象进行复制

4.5.3. 移动构造

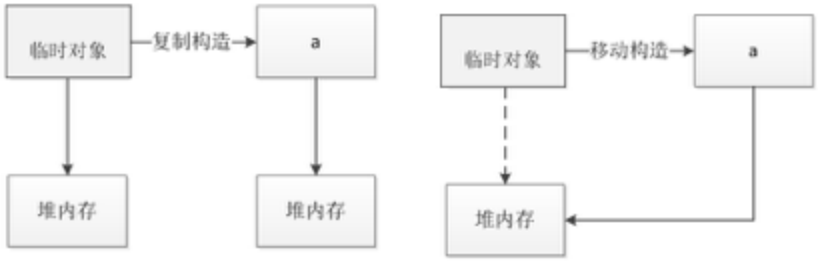
什么时候该触发移动构造？

有可被利用的临时对象

移动构造函数:

class_name (class_name &&)

4.5.3.1.



4.5.3.2. 函数返回含有指针成员的对象

```
#include<iostream>
using namespace std;
class IntNum {
public:
    IntNum(int x = 0) : xptr(new int(x)){ //构造函数
        cout << "Calling constructor..." << endl;
    }
    IntNum(const IntNum & n) : xptr(new int(*n.xptr)){//复制构造函数
        cout << "Calling copy constructor..." << endl;
    }
    IntNum(IntNum && n): xptr( n.xptr){   //移动构造函数
        n.xptr = nullptr;
        cout << "Calling move constructor..." << endl;
    }
    ~IntNum(){ //析构函数
        delete xptr;
        cout << "Destructing..." << endl;
    }
private:
    int *xptr;
};



//返回值为IntNum类对象
IntNum getNum() {
IntNum a;
return a;
}
```

```
int main() {
cout << getNum().getInt() << endl; return 0;
}
```

运行结果：

```
Calling constructor...
Calling move constructor...
Destructing...
0
Destructing...
```

5. 7 继承和派生

5.1. 继承

单继承时派生类的定义

语法

```
class 派生类名：继承方式 基类名
{
   成员声明；
}
```

例

```
class Derived: public Base
{
public:
   Derived ();
   ~Derived ();
};
```

多继承时派生类的定义

语法

```
class 派生类名：继承方式1 基类名1，继承方式2 基类名2，...
{
   成员声明；
```

}

注意：每一个“继承方式”，只用于限制对紧随其后之基类的继承。

例

```
class Derived: public Base1, private Base2
{
public:
   Derived ();
   ~Derived ();
};
```

5.1.1. 目的：实现设计与代码的重用

5.1.2. 保持已有类的特性而构造新类的过程

5.1.3. 继承方式

5.1.3.1. 公有继承

继承的访问控制

1 基类的public和protected成员：访问属性在派生类中保持不变；

2 基类的private成员：不可直接访问。

访问权限

1

派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；

2 通过派生类的对象：只能访问public成员。

5.1.3.2. 私有继承

继承的访问控制

1 基类的public和protected成员：都以private身份出现在派生类中；

2 基类的private成员：不可直接访问。

访问权限

1

派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；

2 通过派生类的对象：不能直接访问从基类继承的任何成员。

5.1.3.3. 保护继承

继承的访问控制

1 基类的public和protected成员：都以protected身份出现在派生类中；

2 基类的private成员：不可直接访问。

访问权限

1

派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；

2 通过派生类的对象：不能直接访问从基类继承的任何成员

5.2. 派生

5.2.1. 目的：当新的问题出现，原有程序无法解决

（或不能完全解决）时，需要对原有程序进行改造

5.2.2. 在已有类的基础上新增自己的特性而产生新类的过程

5.2.3. 构成

5.2.3.1. 吸收基类成员

5.2.3.2. 改造基类成员

5.2.3.3. 添加新的成员

5.2.4. 派生类构造函数

若不继承基类的构造函数

1派生类新增成员：派生类定义构造函数初始化；

2继承来的成员：自动调用基类构造函数进行初始化；

3派生类的构造函数需要给基类的构造函数传递参数。

5.2.4.1. 定义

5.2.4.1.1. 单继承

```
派生类名::派生类名(基类所需的形参，本类成员所需的形参):基类名(参数表), 本类成员初始化列表
{
   //其他初始化；
}；
```

5.2.4.1.2. 多继承

```
派生类名::派生类名(参数表) :
基类名1(基类1初始化参数表),
基类名2(基类2初始化参数表),
...
基类名n(基类n初始化参数表),
本类成员初始化列表
```

```
{
    //其他初始化；
};
```

5.2.4.1.3. 多继承 + 对象成员

```
派生类名::派生类名(形参表):
基类名1(参数), 基类名2(参数), ..., 基类名n(参数),
本类成员（含对象成员）初始化列表
{
    //其他初始化
};
```

5.2.4.2. 执行顺序

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

class Base1 {//基类Base1，构造函数有参数
public:
    Base1(int i)
  { cout << "Constructing Base1 " << i << endl; }
};

class Base2 {//基类Base2，构造函数有参数
public:
    Base2(int j)
  { cout << "Constructing Base2 " << j << endl; }
};

class Base3 {//基类Base3，构造函数无参数
```

```
public:
    Base3()
  { cout << "Constructing Base3 *" << endl; }
};

class Derived: public Base2, public Base1, public Base3 {
public:
    Derived(int a, int b, int c, int d): Base1(a), member2(d), member1(c), Base2(b)
  //此处的次序与构造函数的执行次序无关
    { }
private:
    Base1 member1;
    Base2 member2;
    Base3 member3;
};

int main() {
    Derived obj(1, 2, 3, 4);
    return 0;
}
```

5.2.4.2.1. 1 调用基类构造函数。

顺序按照它们被继承时声明的顺序（从左向右）。

5.2.4.2.2. 2 对初始化列表中的成员进行初始化。

2.1 顺序按照它们在类中定义的顺序。

2.2

对象成员初始化时自动调用其所属类的构造函数。由初始化列表提供参数

。

5.2.4.2.3. 3 执行派生类的构造函数体中的内容。

5.2.5. 派生类复制构造函数

派生类定义了复制构造函数的情况

1 一般都要为基类的复制构造函数传递参数。

2

复制构造函数只能接受一个参数，既用来初始化派生类定义的成员，也将被传递给基类的复制构造函数

。

3 基类的复制构造函数形参类型是基类对象的引用，实参可以是派生类对象的引用

例如:

C::C(const C &c1): B(c1) {…}

5.2.6. 派生类的析构函数

5.2.6.1. 析构函数不被继承，派生类如果需要，要自行声明析构函数。

5.2.6.2. 声明方法与无继承关系时类的析构函数相同。

5.2.6.3. 不需要显式地调用基类的析构函数，系统会自动隐式调用。

5.2.6.4. 先执行派生类析构函数的函数体，再调用基类的析构函数

5.2.7. 访问基类成员

5.2.7.1. 当派生类与基类中有相同成员时

若未特别限定，则通过派生类对象使用的是派生类中的同名成员。

如要通过派生类对象访问基类中被隐藏的同名成员，应使用基类名和作用域操作符（::）来限定。

```
#include <iostream>
using namespace std;
class Base1 {
public:
```

```
    int var;
    void fun() { cout << "Member of Base1" << endl; }
};
class Base2 {
public:
    int var;
    void fun() { cout << "Member of Base2" << endl; }
};
class Derived: public Base1, public Base2 {
public:
    int var;
    void fun() { cout << "Member of Derived" << endl; }
};

int main() {
    Derived d;
    Derived *p = &d;

  //访问Derived类成员
    d.var = 1;
    d.fun();

    //访问Base1基类成员
    d.Base1::var = 2;
    d.Base1::fun();

    //访问Base2基类成员
    p->Base2::var = 3;
    p->Base2::fun();

    return 0;
}
```

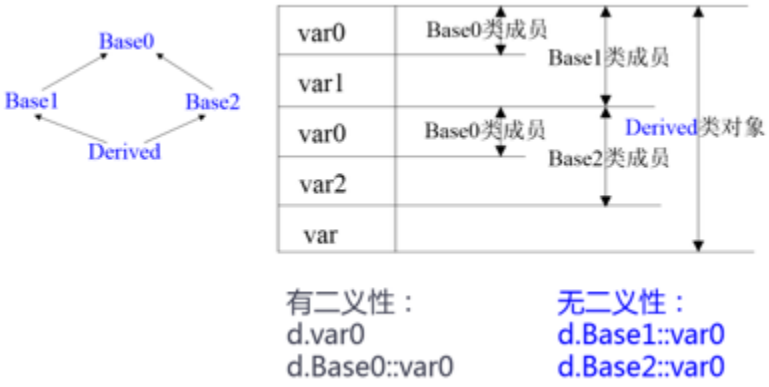
5.2.7.2. 二义性问题

“派生类对象名或引用名.成员名”、“派生类指针->成员名”访问成员存在二义性问题

解决方式：用类名限定

5.2.7.2.1. 如果从不同基类继承了同名成员，但是在派生类中没有定义同名成员

5.2.7.2.2.



5.2.8. 虚基类

5.2.8.1. 问题

当派生类从多个基类派生，而这些基类又共同基类，则在访问此共同基类中的成员时，将产生冗余，并有可能因冗余带来不一致性

5.2.8.2. 定义

以virtual说明基类继承方式

例：

class B1:virtual public B

主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题

为最远的派生类提供唯一的基类成员，而不重复产生多次复制

5.2.8.2.1. 在第一级继承时就要将共同基类设计为虚基类

5.2.8.3. 虚基类的构造函数

1 建立对象时所指定的类称为最远派生类。

2 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。

3
在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中为虚基类的构造函数列出参数。如果未列出，则表示调用该虚基类的默认构造函数。

4
在建立对象时，只有最远派生类的构造函数调用虚基类的构造函数，其他类对虚基类构造函数的调用被忽略。

5.2.8.4. 析构函数为虚函数，派生类析构后，会主动调用基类的析构函数

5.3. 类型转换

5.3.1. 公有派生类对象可以被当作基类的对象使用

派生类的对象可以隐含转换为基类对象；

派生类的对象可以初始化基类的引用；

派生类的指针可以隐含转换为基类的指针。

5.3.2. 通过基类对象名、指针只能使用从基类继承的成员

5.3.3. 不要重新定义，继承而来的非虚函数

6. 8 多态

6.1. 定义

能根据操作环境的不同采用不同的处理方式.

一组具有相同基本语义的方法能在同一接口下为不同的对象服务。

6.2. 实现

6.2.1. 静态绑定

6.2.2. 动态绑定

6.3. 运算符重载

静态编译决定

重载之后运算符的优先级和结合性都不会改变

6.3.1. 重载为类的非静态成员函数

6.3.1.1. 定义

```
函数类型 operator 运算符（形参）
{
    ......
}
```

参数个数=原操作数个数-1 （后置++、--除外）

如果要重载 B 为类成员函数，使之能够实现表达式 oprd1 B oprd2，其中 oprd1 为A 类对象，则 B 应被重载为 A 类的成员函数，形参类型应该是 oprd2 所属的类型。

经重载后，表达式 oprd1 B oprd2 相当于 oprd1.operator B(oprd2)

6.3.1.2. 单目运算符重载

```
#include <iostream>
using namespace std;
class Clock    {//时钟类定义
public:
    Clock(int hour = 0, int minute = 0, int second = 0);
    void showTime() const;
  //前置单目运算符重载
    Clock& operator ++ ();
  //后置单目运算符重载
    Clock operator ++ (int);
private:
    int hour, minute, second;
};

Clock::Clock(int hour, int minute, int second) {
    if (0 <= hour && hour < 24 && 0 <= minute && minute < 60
        && 0 <= second && second < 60) {
        this->hour = hour;
        this->minute = minute;
        this->second = second;
    } else
        cout << "Time error!" << endl;
}
void Clock::showTime() const {  //显示时间
    cout << hour << ":" << minute << ":" << second << endl;
}
```

例8-2重载前置++和后置++为时钟类成员函数

```
Clock & Clock::operator ++ () {
   second++;
   if (second >= 60) {
      second -= 60;  minute++;
      if (minute >= 60) {
        minute -= 60; hour = (hour + 1) % 24;
      }
   }
   return *this;
}

Clock Clock::operator ++ (int) {
   //注意形参表中的整型参数
   Clock old = *this;
   ++(*this);  //调用前置“++”运算符
   return old;
}
```

例8-2重载前置++和后置++为时钟类成员函数

```
int main() {
   Clock myClock(23, 59, 59);
   cout << "First time output: ";
   myClock.showTime();
   cout << "Show myClock++:    ";
   (myClock++).showTime();
   cout << "Show ++myClock:    ";
   (++myClock).showTime();
   return 0;
}
```

6.3.1.2.1. 前置 ++p

如果要重载 U 为类成员函数，使之能够实现表达式 U oprd，其中 oprd 为A类对象，则 U 应被重载为 A 类的成员函数，无形参。

经重载后，表达式 U oprd 相当于 oprd.operator U()

6.3.1.2.2. 前置单目运算符，重载函数没有形参

6.3.1.2.3. 后置 p++

如果要重载 ++或--为类成员函数，使之能够实现表达式 oprd++ 或 oprd-- ，其中 oprd 为A类对象，则 ++或-- 应被重载为 A 类的成员函数，且具有一个 int 类型形参。

经重载后，表达式 oprd++ 相当于 oprd.operator ++(0)

6.3.1.2.4. 后置++运算符，重载函数需要有一个int形参

6.3.2. 重载为非成员函数

6.3.2.1. 定义

参数个数=原操作数个数（后置++、--除外）

至少应该有一个自定义类型的参数。

后置单目运算符 ++和--的重载函数，形参列表中要增加一个int，但不必写形参名。

如果在运算符的重载函数中需要操作某类对象的私有成员，可以将此函数声明为该类的友元。

双目运算符 B重载后，表达式oprd1 B oprd2 等同于operator B(oprd1,oprd2)

前置单目运算符 B重载后，表达式 B oprd 等同于operator B(oprd)

后置单目运算符 ++和--重载后，表达式 oprd B 等同于operator B(oprd,0)

cout重载为非成员函数

6.4. 虚函数

动态编译

6.4.1. 定义

虚函数：

虚函数是实现运行时多态性基础

C++中的虚函数是动态绑定的函数

虚函数必须是非静态的成员函数，虚函数经过派生之后，就可以实现运行过程中的多态。

一般成员函数可以是虚函数

构造函数***不能***是虚函数

析构函数***可以***是虚函数

虚函数成员：

虚函数的声明

virtual 函数类型 函数名（形参表）;

虚函数声明只能出现在类定义中的函数原型声明中，而不能在成员函数实现的时候。

在派生类中可以对基类中的成员函数进行覆盖。

虚函数一般不声明为内联函数，因为对虚函数的调用需要动态绑定，而对内联函数的处理是静态的。

作用：

通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数

6.4.1.1. 作用

6.4.1.1.1. 通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数

6.4.2. 属于对象，而不是属于类

6.4.3. 派生类可以不显式地用virtual声明虚函数

6.4.4. 虚析构函数

6.4.4.1. 通过基类的指针来销毁对象，需要正确识别对象类型

6.4.5. 构造函数不能是虚函数

6.4.5.1. 从存储空间角度

6.4.5.1.1. vtable未初始化，vbtl在构造函数调用后才建

6.4.6. 虚表与动态绑定

6.4.6.1. 虚表

6.4.6.1.1. 每个多态类有一个虚表（virtual table）

6.4.6.1.2. 虚表中有当前类的各个虚函数的入口地址

6.4.6.1.3. 每个对象有一个指向当前类的虚表的指针（虚指针vptr）

6.4.6.2. 动态绑定实现

构造函数中为对象的虚指针赋值

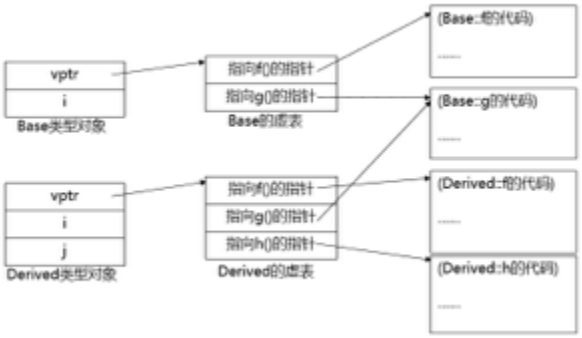
通过多态类型的指针或引用调用成员函数时，通过虚指针找到虚表，进而找到所调用的虚函数的入口地址

通过该入口地址调用虚函数

6.4.6.2.1.

```
class Base {
public:
   virtual void f();
   virtual void g();
private:
   int i;
};

class Derived: public Base {
public:
   virtual void f(); //覆盖Base::f
   virtual void h(); //新增的虚函数
private:
   int j;
};
```



6.4.7. 抽象类

6.4.7.1. 定义

纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，

纯虚函数的声明格式为：

virtual 函数类型 函数名(参数表) = 0;

带有纯虚函数的类称为抽象类

6.4.7.1.1. 纯虚函数是一个在基类中声明的虚函数

6.4.7.1.2. 带有纯虚函数的类称为抽象类

6.4.7.2. 不能定义抽象类的对象

6.4.8. 显示函数覆盖

6.4.8.1. override

6.4.8.1.1. 必须在基类中找到相应的重载函数

6.4.8.2. final

6.4.8.2.1. 用来避免类被继承

6.4.8.2.2. 或是基类的函数被改写

7. 9 模板与群体数据

7.1. 模板

7.1.1. 定义

语法形式：

template <模板参数表>

函数定义

模板参数表的内容：

类型参数：class（或typename） 标识符（T或者Type）

常量参数：类型说明符 标识符

模板参数：template <参数表> class 标识符

7.1.2. 类模板

类模板：

template <模板参数表>

class 类名

{类成员声明};

如果需要在类模板以外定义其成员函数，则要采用以下的形式：

template <模板参数表>

类型名 类名<模板参数标识符列表>::函数名（参数表）

```
template <class T>
class Store {//类模板：实现对任意类型数据进行存取
private:
    T item;  // item用于存放任意类型的数据
    bool haveValue;  // haveValue标记item是否已被存入内容
public:
    Store();
    T &getElem();   //提取数据函数
    void putElem(const T &x);  //存入数据函数
};


template <class T>
T &Store<T>::getElem() { }
```

7.1.3. 数组类模板

7.1.3.1. 动态数组 + 构造函数 来实现

7.1.3.1.1. 优点： 更通用，数组大小可变

7.1.3.2. 模板参数实现 C++11

<<C++ Primer Plus>> p578

7.1.3.2.1. 优点：自动变量维护的栈，执行速度快

7.1.3.2.2. 缺点： 每种数组大小都生成自己的模板

7.2. 群体

7.2.1. 线性

7.2.1.1. 直接访问

7.2.1.1.1. 数组

7.2.1.2. 顺序访问

7.2.1.2.1. 链表

7.2.1.2.2. 队列

7.2.1.2.3. 栈

7.2.1.3. 索引访问

7.2.2. 非线性

7.2.2.1. 不用位置顺序来标识元素

7.3. 排序

8. 10 范型程序设计与c++标准模板库

8.1. 范型程序设计

泛型程序设计的基本概念：

编写不依赖于具体数据类型的程序

将算法从特定的数据结构中抽象出来，成为通用的

C++的模板为泛型程序设计奠定了关键的基础

8.2. STL

8.2.1. 定义

标准模板库（Standard Template Library，简称STL

8.2.2. 基本组件

8.2.2.1. 容器（container）

容器类是容纳、包含一组元素或元素集合的对象。

基于容器中元素的组织方式：顺序容器、关联容器

按照与容器所关联的迭代器类型划分：可逆容器□随机访问容器

8.2.2.1.1. 容器的功能

容器的通用功能

用默认构造函数构造空容器

支持关系运算符：==、!=、<、<=、>、>=

begin()、end()：获得容器首、尾迭代器

clear()：将容器清空

empty()：判断容器是否为空

size()：得到容器元素个数

s1.swap(s2)：将s1和s2两容器内容交换

相关数据类型（S表示容器类型）

S::iterator：指向容器元素的迭代器类型

S::const_iterator：常迭代器类型

对可逆容器的访问

STL为每个可逆容器都提供了逆向迭代器，逆向迭代器可以通过下面的成员函数得到：

rbegin() ：指向容器尾的逆向迭代器

rend()：指向容器首的逆向迭代器

逆向迭代器的类型名的表示方式如下：

S::reverse_iterator：逆向迭代器类型

S::const_reverse_iterator：逆向常迭代器类型

随机访问容器

随机访问容器支持对容器的元素进行随机访问

s[n]：获得容器s的第n个元素

8.2.2.1.2. 容器的分类

8.2.2.1.2.1. 顺序容器

8.2.2.1.2.1.1. array（数组）、vector（向量）、deque（双端队列）、forward_list（单链表）、list（列表）

8.2.2.1.2.1.2. 功能

在逻辑上可看作是一个长度可扩展的数组

顺序容器：

向量（vector）

双端队列（deque）

列表（list）

单向链表（forward_list）

数组（array）

元素线性排列，可以随时在指定位置插入元素和删除元素。

必须符合Assignable这一概念（即具有公有的拷贝构造函数并可以用“=”赋值）。

array对象的大小固定，forward_list有特殊的添加和删除操作。

顺序容器的接口（不包含单向链表（forward_list）和数组（array））：

构造函数

赋值函数

assign

插入函数

insert， push_front（只对list和deque）， push_back，emplace，emplace_front

删除函数

erase，clear，pop_front（只对list和deque），pop_back，emplace_back

首尾元素的直接访问

front，back

改变大小

resize

向量（Vector）

特点

一个可以扩展的动态数组

随机访问、在尾部插入或删除元素快

在中间或头部插入或删除元素慢

向量的容量

容量(capacity)：实际分配空间的大小

s.capacity() ：返回当前容量

s.reserve(n)：若容量小于n，则对s进行扩展，使其容量至少为n

双端队列（deque）

特点

在两端插入或删除元素快

在中间插入或删除元素慢

随机访问较快，但比向量容器慢

列表(list)

特点

在任意位置插入和删除元素都很快

不支持随机访问

接合(splice)操作

s1.splice(p, s2, q1, q2)：将s2中[q1, q2)移动到s1中p所指向元素之前

单向链表（forward_list）

单向链表每个结点只有指向下个结点的指针，没有简单的方法来获取一个结点的前驱；

未定义insert、emplace和erase操作，而定义了insert_after、emplace_after和erase_after操作，其参数与list的insert、emplace和erase相同，但并不是插入或删除迭代器p1所指的元素，而是对p1所指元素之后的结点进行操作；

不支持size操作。

数组（array）

array是对内置数组的封装，提供了更安全，更方便的使用数组的方式

array的对象的大小是固定的，定义时除了需要指定元素类型，还需要指定容器大小。

不能动态地改变容器大小

顺序容器的比较

STL所提供的顺序容器各有所长也各有所短，我们在编写程序时应当根据我们对容器所需要执行的操作来决定选择哪一种容器。

如果需要执行大量的随机访问操作，而且当扩展容器时只需要向容器尾部加入新的元素，就应当选择向量容器vector；

如果需要少量的随机访问操作，需要在容器两端插入或删除元素，则应当选择双端队列容器deque；

如果不需要对容器进行随机访问，但是需要在中间位置插入或者删除元素，就应当选择列表容器list或forward_list；

如果需要数组，array相对于内置数组类型而言，是一种更安全、更容易使用的数组类型。

8.2.2.1.2.2. (有序)关联容器

8.2.2.1.2.2.1. set（集合）、multiset（多重集合）、map（映射）、multi map（多重映射）

8.2.2.1.2.3. 无序关联容器

8.2.2.1.2.3.1. unordered_set
（无序集合）、unordered_multiset（无序多重集合）

8.2.2.1.2.3.2. unordered_map（无序映射）、unorder_multimap（无序多重映射）

8.2.2.1.2.4. 容器适配器

8.2.2.1.2.4.1. stack（栈）、queue（队列）、priority_queue（优先队列）

栈(stack)：最先压入的元素最后被弹出

队列(queue)：最先压入的元素最先被弹出

优先级队列(priority_queue)：最“大”的元素最先被弹出

栈和队列模板

栈模板

template <class T, class Sequence = deque<T> > class stack;

队列模板

template <class T, class FrontInsertionSequence = deque<T> > class queue;

栈可以用任何一种顺序容器作为基础容器，而队列只允许用前插顺序容器（双端队列或列表）

栈和队列共同支持的操作

s1 op s2

op可以是==、!=、<、<=、>、>=之一，它会对两个容器适配器之间的元素按字典序进行比较

s.size() 返回s的元素个数

s.empty() 返回s是否为空

s.push(t) 将元素t压入到s中

s.pop()

将一个元素从s中弹出，对于栈来说，每次弹出的是最后被压入的元素，而对于队列，每次被弹出的是最先被压入的元素

不支持迭代器，因为它们不允许对任意元素进行访问

栈和队列不同的操作

栈的操作

s.top() 返回栈顶元素的引用

队列操作

s.front()获得队头元素的引用

s.back()获得队尾元素的引用

优先级队列

优先级队列也像栈和队列一样支持元素的压入和弹出，但元素弹出的顺序与元素的大小有关，每次弹出的总是容器中最“大”的一个元素。

template <class T, class Sequence = vector<T> > class priority_queue;

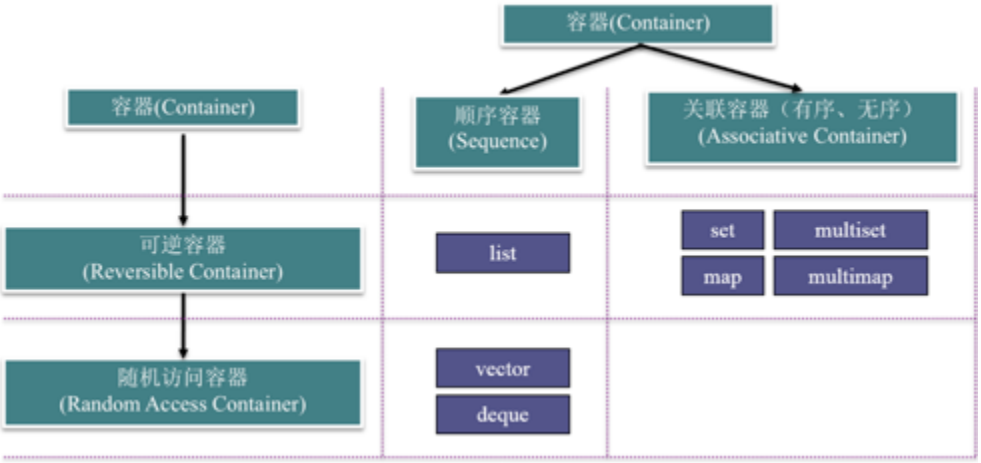
优先级队列的基础容器必须是支持随机访问的顺序容器。

支持栈和队列的size、empty、push、pop几个成员函数，用法与栈和队列相同。

优先级队列并不支持比较操作。

与栈类似，优先级队列提供一个top函数，可以获得下一个即将被弹出元素（即最“大”的元素）的引用。

8.2.2.1.3.



8.2.2.2. 迭代器（iterator）

提供了顺序访问容器中每个元素的方法；

可以使用“++”运算符来获得指向下一个元素的迭代器；

可以使用“*”运算符访问一个迭代器所指向的元素，如果元素类型是类或结构体，还可以使用“->”运算符直接访问该元素的一个成员；

有些迭代器还支持通过“--”运算符获得指向上一个元素的迭代器；

迭代器是泛化的指针：指针也具有同样的特性，因此指针本身就是一种迭代器；

使用独立于STL容器的迭代器，需要包含头文件<iterator>。

8.2.2.2.1. 泛化的指针，提供了顺序访问容器中每个元素的方法

8.2.2.2.1.1. 解除引用读取、写入、++i、i++、--i 、i-- 、i[n]、 i+n 、i=n 、i+=n、 i-=n

8.2.2.2.2. 迭代器的分类

8.2.2.2.2.1. 输入流迭代器

8.2.2.2.2.1.1. istream_iterator<T>

以输入流（如cin）为参数构造

可用*(p++)获得下一个输入的元素

8.2.2.2.2.1.2. 可以用来从序列中读取数据，如输入流迭代器

8.2.2.2.2.2. 输出流迭代器

8.2.2.2.2.2.1. ostream_iterator<T>

构造时需要提供输出流（如cout）

可用(*p++) = x将x输出到输出流

8.2.2.2.2.2.2. 允许向序列中写入数据，如输出流迭代器

8.2.2.2.2.3. 正向迭代器

8.2.2.2.2.3.1. 既是输入迭代器又是输出迭代器，并且可以对序列进行单向的遍历

8.2.2.2.2.4. 双向迭代器

8.2.2.2.2.4.1. 在两个方向上都可以对数据遍历

8.2.2.2.2.5. 随机访问迭代器

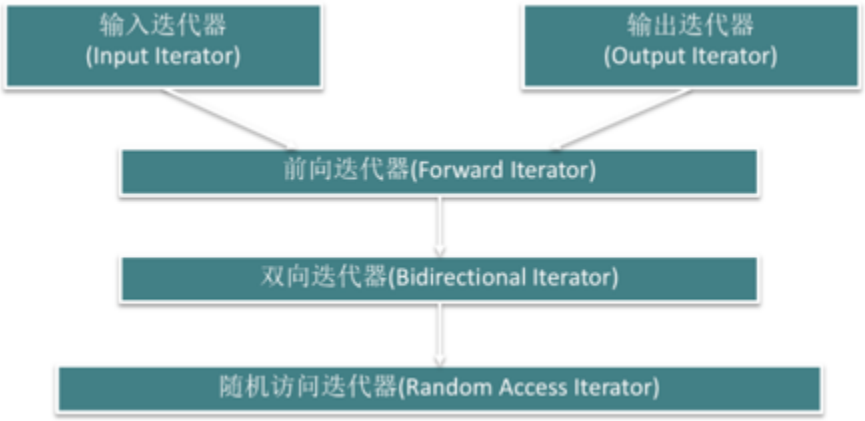
8.2.2.2.2.5.1. 也是双向迭代器，但能够在序列中的任意两个位置之间进行跳转

8.2.2.2.2.5.2. 如指针、使用vector的begin()、end()函数得到的迭代器

8.2.2.2.3.

8.2.2.2.3.1.

8.2.2.2.3.1.1.



8.2.2.2.3.2. 辅助函数

8.2.2.2.3.2.1. advance(p, n)

8.2.2.2.3.2.1.1. 对p执行n次自增操作

8.2.2.2.3.2.2. distance(first, last)

8.2.2.2.3.2.2.1. 计算两个迭代器first和last的距离，即对first执行多少次“++”操作后能够使得first == last

8.2.2.3. 函数对象（function object）

一个行为类似函数的对象，对它可以像调用函数一样调用。

函数对象是泛化的函数：任何普通的函数和任何重载了“()”

运算符的类的对象都可以作为函数对象使用

使用STL的函数对象，需要包含头文件<functional>

8.2.2.4. 算法（algorithms）

STL包括70多个算法

例如：排序算法，消除算法，计数算法，比较算法，变换算法，置换算法和容器管理等

可以广泛用于不同的对象和内置的数据类型。

使用STL的算法，需要包含头文件<algorithm>。

8.2.2.4.1. 模板使算法独立于存储的数据类型

8.2.2.4.2. 迭代器使算法独立于使用的容器类型

9. 11 流类库与输入输出

9.1. 输出流

9.1.1. 定义

9.1.1.1. 重要的三个输出流

9.1.1.1.1. ostream

9.1.1.1.2. ofstream

9.1.1.1.3. ostringstream

9.1.1.2. 预先定义的输出流对象

9.1.1.2.1. cout 标准输出

9.1.1.2.2. cerr 标准错误输出，没有缓冲，发送给它的内容立即被输出。

9.1.1.2.3. clog 类似于cerr，但是有缓冲，缓冲区满时被输出。

9.1.1.3. 标准输出换向

```
ofstream fout("b.out");
streambuf*  pOld  =cout.rdbuf(fout.rdbuf());
//…
cout.rdbuf(pOld);
```

9.1.1.4. 构造输出流对象

ofstream类支持磁盘文件输出

如果在构造函数中指定一个文件名，当构造这个文件时该文件是自动打开的

```
ofstream myFile("filename");
```

可以在调用默认构造函数之后使用open成员函数打开文件

```
ofstream myFile; //声明一个静态文件输出流对象
myFile.open("filename");   //打开文件，使流对象与文件建立联系
```

在构造对象或用open打开文件时可以指定模式

```
ofstream myFile("filename", ios_base::out | ios_base::binary);
```

9.1.1.5. 文件输出流成员函数

open函数

把流与一个特定的磁盘文件关联起来。

需要指定打开模式。

put函数

把一个字符写到输出流中。

write函数

把内存中的一块内容写到一个文件输出流中

seekp和tellp函数

操作文件流的内部指针

close函数

关闭与一个文件输出流关联的磁盘文件

错误处理函数

在写到一个流时进行错误处理

9.1.2. 向文本文件输出

9.1.2.1. 操纵符（manipulator）

iomanip.h

插入运算符与操纵符一起工作

控制输出格式。

很多操纵符都定义在

ios_base类中（如hex()）、<iomanip>头文件（如setprecision()）。

控制输出宽度

在流中放入setw操纵符或调用width成员函数为每个项指定输出宽度。

setw和width仅影响紧随其后的输出项，但其它流格式操纵符保持有效直到发生改变。

dec、oct和hex操纵符设置输入和输出的默认进制。

9.1.2.2. 控制输出精度

9.1.2.2.1. setprecision

9.1.2.2.1.1. 未指定fixed或scientific

9.1.2.2.1.1.1. 有效位数字

9.1.3. 向二进制文件输出

9.1.3.1. ios_base::binary

9.1.4. 向字符串输出

9.1.4.1. 用ostringstream将数值转换为字符串

9.2. 输入流

9.2.1. 定义

9.2.1.1. 重要的输入流类

9.2.1.1.1. istream类最适合用于顺序文本模式输入。cin是其实例。

9.2.1.1.2. ifstream类支持磁盘文件输入。

9.2.1.1.3. istringstream

9.2.1.2. 构造输入流对象

如果在构造函数中指定一个文件名，在构造该对象时该文件便自动打开。

ifstream myFile("filename");

在调用默认构造函数之后使用open函数来打开文件。

ifstream myFile; //建立一个文件流对象

myFile.open("filename"); //打开文件"filename”

打开文件时可以指定模式

ifstream myFile("filename", ios_base::in | ios_base::binary);

9.2.1.3. 输入流相关函数

open 把该流与一个特定磁盘文件相关联。

get 功能与提取运算符（>>）很相像，主要的不同点是get函数在读入数据时包括空白字符。

getline

功能是从输入流中读取多个字符，并且允许指定输入终止字符，读取完成后，从读取的内容中删除终止字符。

read

从一个文件读字节到一个指定的内存区域，由长度参数确定要读的字节数。当遇到文件结束或者在文本模式文件中遇到文件结束标记字符时结束读取。

seekg 用来设置文件输入流中读取数据位置的指针。

tellg 返回当前文件读指针的位置。

close 关闭与一个文件输入流关联的磁盘文件。

9.2.2. 从字符串输入

9.2.2.1. 字符串输入流（ istringstream ）

用于从字符串读取数据

在构造函数中设置要读取的字符串

功能

支持ifstream类的除open、close外的所有操作

典型应用

将字符串转换为数值

9.3. 输入/输出流

9.3.1. 两个重要的输入/输出流

一个iostream对象可以是数据的源或目的。

两个重要的I/O流类都是从iostream派生的，它们是fstream和stringstream。这些类继承了前面描述的istream和ostream类的功能

9.3.2. fstream类

9.3.3. stringstream类

10. 12 异常处理

10.1. 主要思想

抛出异常的程序段：

throw 表达式

捕获并处理异常的程序段：

try

复合语句（保护段）

catch （异常声明）

复合语句（异常处理流程）

catch （异常声明）

复合语句

10.2. 异常接口声明

10.3. 异常处理中的构造与析构

10.3.1. 自动的析构

找到一个匹配的catch异常处理后

1 初始化异常参数。

2 将从对应的try块开始到异常被抛掷处之间构造（且尚未析构）的所有自动对象进行析构。

3 从最后一个catch处理之后开始恢复执行。

```
//12_2.cpp
#include <iostream>
#include <string>
using namespace std;
class MyException {
public:
    MyException(const string &message) : message(message) {}
    ~MyException() {}
    const string &getMessage() const { return message; }
private:
    string message;
};

class Demo {
public:
    Demo() { cout << "Constructor of Demo" << endl; }
    ~Demo() { cout << "Destructor of Demo" << endl; }
};
void func() throw (MyException) {
    Demo d;
    cout << "Throw MyException in func()" << endl;
    throw MyException("exception thrown by func()");
}

int main() {
```

```
    cout << "In main function" << endl;
    try {
        func();
    } catch (MyException& e) {
        cout << "Caught an exception: " << e.getMessage() << endl;
    }
    cout << "Resume the execution of main()" << endl;
    return 0;
}
```

运行结果：

```
In main function
Constructor of Demo
Throw MyException in func()
Destructor of Demo
Caught an exception: exception thrown by func()
Resume the execution of main()
```

10.4. 标准程序库异常处理

10.4.1.

10.4.1.1.