C++程序

编程题收集

知识点收集

C++面向对象知识

基类 & 派生类

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为基类，新建的类称为派生类。

继承代表了 is a 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

#include <iostream>

using namespace std;

// 基类

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 派生类

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// 输出对象的面积

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。

子主题 1

访问 p ublic protected private

同一个类 yes yes yes

派生类 yes yes no

外部的类 yes no no

继承类型

公有继承（public）：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问，但是可以通过调用基类的公有和保护成员来访问。

保护继承（protected）： 当一个类派生自保护基类时，基类的公有和保护成员将成为派生类的保护成员。

私有继承（private）：当一个类派生自私有基类时，基类的公有和保护成员将成为派生类的私有成员。

多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性。

class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…

{

<派生类类体>

};

// 派生类

class Rectangle: public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

C++ 重载运算符和重载函数

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

C++ 中的函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

C++ 中的运算符重载

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的

成员函数

Box operator+(const Box&);

#include <iostream>

using namespace std;

class Box{

public:

double getVolume(void)

return length \* breadth \* height

void setLength( double len )

length = len;

void setBreadth( double bre

breadth = bre;

void setHeight( double hei ) {

height = hei;

// 重载 + 运算符，用于把两个 Box 对象相加

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

// 程序的主函数

int main( )

{

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

Box Box3; // 声明 Box3，类型为 Box

double volume = 0.0; // 把体积存储在该变量中

// Box1 详述

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// Box2 详述

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// Box1 的体积

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// Box2 的体积

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// 把两个对象相加，得到 Box3

Box3 = Box1 + Box2;

// Box3 的体积

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，

Box operator+(const Box&, const Box&);

C++ 多态

多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。

C++多态性主要是通过虚函数实现的，虚函数允许子类重写override(注意和overload的区别，overload是重载，是允许同名函数的表现，这些函数参数列表/类型不同）。

Overload(重载)：在C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，但参数或返回值不同（包括类型、顺序不同），即函数重载。

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

verride(覆盖)：是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有virtual 关键字。

Overwrite(重写)：隐藏，是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape{

public:

Rectangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};class Triangle: public Shape{

public:

Triangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// 程序的主函数

int main( )

{

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// 存储矩形的地址

shape = &rec;

// 调用矩形的求面积函数 area

shape->area();

// 存储三角形的地址

shape = &tri;

// 调用三角形的求面积函数 area

shape->area();

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Parent class area

Parent class area

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

virtual int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

}

导致错误输出的原因是，调用函数 area() 被编译器设置为基类中的版本，这就是所谓的静态多态，或静态链接 - 函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为 area() 函数在程序编译期间就已经设置好了。

修改后，当编译和执行前面的实例代码时，它会产生以下结果：

Rectangle class area

Triangle class area

# 虚函数

虚函数 是在基类中使用关键字 virtual 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。

纯虚函数

您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

= 0 告诉编译器，函数没有主体，上面的虚函数是纯虚函数。

C++ 数据抽象

数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。

数据抽象有两个重要的优势

类的内部受到保护，不会因无意的用户级错误导致对象状态受损。

类实现可能随着时间的推移而发生变化，以便应对不断变化的需求，或者应对那些要求不改变用户级代码的错误报告。

数据抽象的实例

class Adder{

public:

// 构造函数

Adder(int i = 0)

{

total = i;

}

// 对外的接口

void addNum(int number)

{

total += number;

}

// 对外的接口

int getTotal()

{

return total;

};

private:

// 对外隐藏的数据

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total 60

上面的类把数字相加，并返回总和。公有成员 addNum 和 getTotal 是对外的接口，用户需要知道它们以便使用类。私有成员 total 是用户不需要了解的，但又是类能正常工作所必需的。

C++ 数据封装

C++ 程序都有以下两个基本要素：

程序语句（代码）：这是程序中执行动作的部分，它们被称为函数。

程序数据：数据是程序的信息，会受到程序函数的影响。

数据封装是一种把数据和操作数据的函数捆绑在一起的机制

数据抽象是一种仅向用户暴露接口而把具体的实现细节隐藏起来的机制。

数据封装的实例

class Adder{

public:

// 构造函数

Adder(int i = 0)

{

total = i;

}

// 对外的接口

void addNum(int number)

{

total += number;

}

// 对外的接口

int getTotal()

{

return total;

};

private:

// 对外隐藏的数据

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total 60

上面的类把数字相加，并返回总和。公有成员 addNum 和 getTotal 是对外的接口，用户需要知道它们以便使用类。私有成员 total 是对外隐藏的，用户不需要了解它，但它又是类能正常工作所必需的。

C++ 接口（抽象类）

接口描述了类的行为和功能，而不需要完成类的特定实现。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类

class Box

{

public:

// 纯虚函数

virtual double getVolume() = 0;

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。如果试图实例化一个抽象类的对象，会导致编译错误。

如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数，这也意味着 C++ 支持使用 ABC 声明接口。如果没有在派生类中重载纯虚函数，就尝试实例化该类的对象，会导致编译错误。

// 基类

class Shape

{

public:

// 提供接口框架的纯虚函数

virtual int getArea() = 0;

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

/ 派生类

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public:

int getArea()

{

return (width \* height)/2;

}

};

int main(void)

{

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// 输出对象的面积

cout << "Total Rectangle area: " << Rect.getArea() << endl;

Tri.setWidth(5);

Tri.setHeight(7);

// 输出对象的面积

cout << "Total Triangle area: " << Tri.getArea() << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total Rectangle area: 35

Total Triangle area: 17

从上面的实例中，我们可以看到一个抽象类是如何定义一个接口 getArea()，两个派生类是如何通过不同的计算面积的算法来实现这个相同的函数

C++各个容器比较（vector，deque，list，set，map，queue，stack）

Queue是先进先出的集合而Stack是后进先出的集合。

stack

stack 模板类的定义在<stack>头文件中。

stack 模板类需要两个模板参数，一个是元素类型，一个容器类型，但只有元素类型是必要

的，在不指定容器类型时，默认的容器类型为deque。

定义stack 对象的示例代码如下：

stack<int> s1;

stack<string> s2;

stack 的基本操作有：

入栈，如例：s.push(x);

出栈，如例：s.pop();注意，出栈操作只是删除栈顶元素，并不返回该元素。

访问栈顶，如例：s.top()

判断栈空，如例：s.empty()，当栈空时，返回true。

访问栈中的元素个数，如例：s.size()

queue

queue 模板类的定义在<queue>头文件中。

与stack 模板类很相似，queue 模板类也需要两个模板参数，一个是元素类型，一个容器类

型，元素类型是必要的，容器类型是可选的，默认为deque 类型。

定义queue 对象的示例代码如下：

queue<int> q1;

queue<double> q2;

queue 的基本操作有：

入队，如例：q.push(x); 将x 接到队列的末端。

出队，如例：q.pop(); 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问队首元素，如例：q.front()，即最早被压入队列的元素。

访问队尾元素，如例：q.back()，即最后被压入队列的元素。

判断队列空，如例：q.empty()，当队列空时，返回true。

访问队列中的元素个数，如例：q.size()

#include <cstdlib>

#include <iostream>

#include <queue>

using namespace std;

int main()

{

int e,n,m;

queue<int> q1;

for(int i=0;i<10;i++)

q1.push(i);

if(!q1.empty())

cout<<"dui lie bu kong\n";

n=q1.size();

cout<<n<<endl;

m=q1.back();

cout<<m<<endl;

for(int j=0;j<n;j++)

{

e=q1.front();

cout<<e<<" ";

q1.pop();

}

cout<<endl;

if(q1.empty())

cout<<"dui lie bu kong\n";

system("PAUSE");

return 0;

}

priority\_queue

在<queue>头文件中，还定义了另一个非常有用的模板类priority\_queue(优先队列）。优先队

列与队列的差别在于优先队列不是按照入队的顺序出队，而是按照队列中元素的优先权顺序

出队（默认为大者优先，也可以通过指定算子来指定自己的优先顺序）。

priority\_queue 模板类有三个模板参数，第一个是元素类型，第二个容器类型，第三个是比

较算子。其中后两个都可以省略，默认容器为vector，默认算子为less，即小的往前排，大

的往后排（出队时序列尾的元素出队）。

定义priority\_queue 对象的示例代码如下：

priority\_queue<int> q1;

priority\_queue< pair<int, int> > q2; // 注意在两个尖括号之间一定要留空格。

priority\_queue<int, vector<int>, greater<int> > q3; // 定义小的先出队

priority\_queue 的基本操作与queue 相同。

初学者在使用priority\_queue 时，最困难的可能就是如何定义比较算子了。

如果是基本数据类型，或已定义了比较运算符的类，可以直接用STL 的less 算子和greater

算子——默认为使用less 算子，即小的往前排，大的先出队。

如果要定义自己的比较算子，方法有多种，这里介绍其中的一种：重载比较运算符。优先队

列试图将两个元素x 和y 代入比较运算符(对less 算子，调用x<y，对greater 算子，调用x>y)，

若结果为真，则x 排在y 前面，y 将先于x 出队，反之，则将y 排在x 前面，x 将先出队。

#include <iostream>

#include <queue>

using namespace std;

class T

{

public:

int x, y, z;

T(int a, int b, int c):x(a), y(b), z(c)

{

}

};

bool operator < (const T &t1, const T &t2)

{

return t1.z < t2.z; // 按照z 的顺序来决定t1 和t2 的顺序

}

main()

{

priority\_queue<T> q;

q.push(T(4,4,3));

q.push(T(2,2,5));

q.push(T(1,5,4));

q.push(T(3,3,6));

while (!q.empty())

{

T t = q.top(); q.pop();

cout << t.x << " " << t.y << " " << t.z << endl;

}

return 1;

}

输出结果为(注意是按照z 的顺序从大到小出队的)：

3 3 6

2 2 5

1 5 4

4 4 3

# List

#include <iostream>

#include <list>

#include <numeric>

#include <algorithm>

/\*

STL中提供的list类，是一个双向循环链表类。从每一个元素（节点），都可以访问前面一个元素和后面一个元素。

在STL中，list提供了一系列的成员函数，实则完成的就是链表中的基本运算，以及其他在数据结构中不作为基本运算，

但在工程中常用到的功能。

\*/

using namespace std;

//创建一个list容器的实例LISTINT

typedef list<int> LISTINT;

//创建一个list容器的实例LISTCHAR

typedef list<char> LISTCHAR;

int main()

{

//--------------------------

//用list容器处理整型数据

//--------------------------

//用LISTINT创建一个名为listOne的list对象

LISTINT listOne;

//声明i为迭代器

LISTINT::iterator i;

//从前面向listOne容器中添加数据

listOne.push\_front (2);

listOne.push\_front (1);

//从后面向listOne容器中添加数据

listOne.push\_back (3);

listOne.push\_back (4);

//从前向后显示listOne中的数据

cout<<"listOne.begin()--- listOne.end():"<<endl;

for (i = listOne.begin(); i != listOne.end(); ++i)

cout << \*i << " ";

cout << endl;

//从后向后显示listOne中的数据

LISTINT::reverse\_iterator ir;

cout<<"listOne.rbegin()---listOne.rend():"<<endl;

for (ir =listOne.rbegin(); ir!=listOne.rend(); ir++)

{

cout << \*ir << " ";

}

cout << endl;

//使用STL的accumulate(累加)算法

int result = accumulate(listOne.begin(), listOne.end(),0);

cout<<"Sum="<<result<<endl;

cout<<"------------------"<<endl;

//--------------------------

//用list容器处理字符型数据

//--------------------------

//用LISTCHAR创建一个名为listOne的list对象

LISTCHAR listTwo;

//声明i为迭代器

LISTCHAR::iterator j;

//从前面向listTwo容器中添加数据

listTwo.push\_front ('A');

listTwo.push\_front ('B');

//从后面向listTwo容器中添加数据

listTwo.push\_back ('x');

listTwo.push\_back ('y');

//从前向后显示listTwo中的数据

cout<<"listTwo.begin()---listTwo.end():"<<endl;

for (j = listTwo.begin(); j != listTwo.end(); ++j)

cout << char(\*j) << " ";

cout << endl;

//使用STL的max\_element算法求listTwo中的最大元素并显示

j=max\_element(listTwo.begin(),listTwo.end());

cout << "The maximum element in listTwo is: "<<char(\*j)<<endl;

while(1);

return 0;

}

Vector

vector基本操作：

　 1.头文件 #include<vector>。 注：一定要加上using namespace std;

　 2.vector对象的创建： vector<int/char/string/基本数据类型/自定义类型/结构体类型/.....>vec ; (以下以int类型为例)

　 3.vector对象尾部插入数据： vec.push\_back(a); 注意： vector对象的下标从0开始！！！！！

　 4.vector对象尾部删除数据： vec.pop\_back(a);

　 5.vector中定位函数： vec.at(i); //相当于vec[i];

　 6.vector中第一个元素的指针： vec.begin();

　 7.vector中最后一个元素+1的指针：vec.end();

　 8.vector中得到第一个元素的值： vec.front();

　 9.vector中得到最后一个元素的值： vec.back();

　 10.判断vector是否为空： vec.empty();

　 11.交换vector两个容器的值： vector<int>a.swap(vec);

　 12.vector对象的访问：vec[0],vec[1],vec[2]..............

迭代器的用法,iterator

1. 迭代器(iterator)是一中检查容器内元素并遍历元素的数据类型。

(1) 每种容器类型都定义了自己的迭代器类型，如vector:

vector<int>::iterator iter;这条语句定义了一个名为iter的变量，它的数据类型是由vector<int>定义的iterator类型。

(2) 使用迭代器读取vector中的每一个元素：

vector<int> ivec(10,1);

for(vector<int>::iterator iter=ivec.begin();iter!=ivec.end();++iter)

{

\*iter=2; //使用 \* 访问迭代器所指向的元素

}

const\_iterator:

只能读取容器中的元素，而不能修改。

for(vector<int>::const\_iterator citer=ivec.begin();citer!=ivec.end();citer++)

{

cout<<\*citer;

//\*citer=3; error

}

vector<int>::const\_iterator 和 const vector<int>::iterator的区别

const vector<int>::iterator newiter=ivec.begin();

\*newiter=11; //可以修改指向容器的元素

//newiter++; //迭代器本身不能被修改

(3) iterator的算术操作：

iterator除了进行++,--操作，可以将iter+n,iter-n赋给一个新的iteraor对象。还可以使用一个iterator减去另外一个iterator.

const vector<int>::iterator newiter=ivec.begin();

vector<int>::iterator newiter2=ivec.end();

cout<<"\n"<<newiter2-newiter;

一個很典型使用vector的STL程式:

1 #include <vector>

2 #include <iostream>

3

4 using namespace std;

5

6 int main() {

7 vector<int> ivec;

8 ivec.push\_back(1);

9 ivec.push\_back(2);

10 ivec.push\_back(3);

11 ivec.push\_back(4);

12

13 for(vector<int>::iterator iter = ivec.begin();1. iter != ivec.end(); ++iter)

14 cout << \*iter << endl;

15 }

# C++中容器Vector的用法

https://www.cnblogs.com/LGJC1314/p/6680054.html

vector基本操作：

　 1.头文件 #include<vector>。 注：一定要加上using namespace std;

　 2.vector对象的创建： vector<int/char/string/基本数据类型/自定义类型/结构体类型/.....>vec ; (以下以int类型为例)

　 3.vector对象尾部插入数据： vec.push\_back(a); 注意： vector对象的下标从0开始！！！！！

　 4.vector对象尾部删除数据： vec.pop\_back(a);

　 5.vector中定位函数： vec.at(i); //相当于vec[i];

　 6.vector中第一个元素的指针： vec.begin();

　 7.vector中最后一个元素+1的指针：vec.end();

　 8.vector中得到第一个元素的值： vec.front();

　 9.vector中得到最后一个元素的值： vec.back();

　 10.判断vector是否为空： vec.empty();

　 11.交换vector两个容器的值： vector<int>a.swap(vec);

　 12.vector对象的访问：vec[0],vec[1],vec[2]..............

vector(向量): C++中的一种数据结构,确切的说是一个类.它相当于一个动态的数组,当程序员无法知道自己需要的数组的规模多大时,用其来解决问题可以达到最大节约空间的目的.

1.文件包含:

首先在程序开头处加上#include<vector>以包含所需要的类文件vector

还有一定要加上using namespace std;

变量声明:

2.1 例:声明一个int向量以替代一维的数组:vector <int> a;(等于声明了一个int数组a[],大小没有指定,可以动态的向里面添加删除)。

2.2 例:用vector代替二维数组.其实只要声明一个一维数组向量即可,而一个数组的名字其实代表的是它的首地址,所以只要声明一个地址的向量即可,即:vector <int \*> a.同理想用向量代替三维数组也是一样,vector <int\*\*>a;再往上面依此类推.

动态内存分配New, Malloc()

栈：在函数内部声明的所有变量都将占用栈内存。

堆：这是程序中未使用的内存，在程序运行时可用于动态分配内存。

New 与Malloc的不同

new 与 malloc() 函数相比，其主要的优点是，new 不只是分配了内存，它还创建了对象。

New() Delete()例程

#include <iostream>

using namespace std;

int main ()

{

double\* pvalue = NULL; // 初始化为 null 的指针

pvalue = new double; // 为变量请求内存

\*pvalue = 29494.99; // 在分配的地址存储值

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // 释放内存

return 0;

}

#include <iostream>

using namespace std;

class Box

{

public:

Box() {

cout << "调用构造函数！" <<endl;

}

~Box() {

cout << "调用析构函数！" <<endl;

}

};

int main( )

{

Box\* myBoxArray = new Box[4];

delete [] myBoxArray; // Delete array

return 0;

}

Malloc() free 例程

class Obj

{

public:

Obj( )

{ cout << "Initialization" << endl; }

~ Obj( )

{ cout << "Destroy" << endl; }

void Initialize( )

{ cout << "Initialization" << endl; }

void Destroy( )

{ cout << "Destroy" << endl; }

}obj;

void UseMallocFree( )

{

Obj \* a = (Obj \*) malloc( sizeof ( obj ) ); // allocate memory

a -> Initialize(); // initialization

// …

a -> Destroy(); // deconstruction

free(a); // release memory

}

C++ 异常处理

C++ 异常处理涉及到三个关键字：try、catch、throw。

throw: 当问题出现时，程序会抛出一个异常。这是通过使用 throw 关键字来完成的。

catch: 在您想要处理问题的地方，通过异常处理程序捕获异常。catch 关键字用于捕获异常。

try: try 块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个 catch 块。

try 和 catch 关键字：try 块中放置可能抛出异常的代码，try 块中的代码被称为保护代码

try

{

// 保护代码

}catch( ExceptionName e1 )

{

// catch 块

}catch( ExceptionName e2 )

{

// catch 块

}catch( ExceptionName eN )

{

// catch 块

}

抛出异常

您可以使用 throw 语句在代码块中的任何地方抛出异常。

double division(int a, int b)

{

if( b == 0 )

{

throw "Division by zero condition!";

}

return (a/b);

}

#include <iostream>

using namespace std;

double division(int a, int b){

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

int main (){

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

}catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

}

由于我们抛出了一个类型为 const char\* 的异常，因此，当捕获该异常时，我们必须在 catch 块中使用 const char\*。当上面的代码被编译和执行时，它会产生下列结果：

Division by zero condition!

Static 和 const

# static的作用：

## 1.局部变量：

在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。

1）内存中的位置：静态存储区

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

注：当static用来修饰局部变量的时候，它就改变了局部变量的存储位置（从原来的栈中存放改为静态存储区）及其生命周期（局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问），但未改变其作用域。

## 2.全局变量

在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。

1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

注：static修饰全局变量，并为改变其存储位置及生命周期，而是改变了其作用域，使当前文件外的源文件无法访问该变量，好处如下：（1）不会被其他文件所访问，修改（2）其他文件中可以使用相同名字的变量，不会发生冲突。对全局函数也是有隐藏作用。

对类中的

## 1.成员变量

　　　　用static修饰类的数据成员实际使其成为类的全局变量，会被类的所有对象共享，包括派生类的对象。因此，static成员必须在类外进行初始化(初始化格式： int base::var=10;)，而不能在构造函数内进行初始化，不过也可以用const修饰static数据成员在类内初始化 。

　　　　特点：

不要试图在头文件中定义(初始化)静态数据成员。在大多数的情况下，这样做会引起重复定义这样的错误。即使加上#ifndef #define #endif或者#pragma once也不行。

静态数据成员可以成为成员函数的可选参数，而普通数据成员则不可以。

静态数据成员的类型可以是所属类的类型，而普通数据成员则不可以。普通数据成员的只能声明为 所属类类型的指针或引用。

## 2.成员函数

用static修饰成员函数，使这个类只存在这一份函数，所有对象共享该函数，不含this指针。

静态成员是可以独立访问的，也就是说，无须创建任何对象实例就可以访问。base::func(5,3);当static成员函数在类外定义时不需要加static修饰符。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员。因为静态成员函数不含this指针。

不可以同时用const和static修饰成员函数。

C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this\*。

但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。

static的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态

const的作用：

1.限定变量为不可修改。

2.限定成员函数不可以修改任何数据成员。

3.const与指针：

const char \*p 表示 指向的内容不能改变。

char \* const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

指针和引用的区别

本质区别

指针是一个新的变量，只是这个变量存储的是另一个变量的地址，我们通过访问这个地址来修改变量。

而引用只是一个别名，还是变量本身。对引用进行的任何操作就是对变量本身进行操作，因此以达到修改变量的目的。

int a=1;int \*p=&a;

int a=1;int &b=a;

(2)可以有const指针，但是没有const引用；

(3)指针可以有多级，但是引用只能是一级（int \*\*p；合法 而 int &&a是不合法的）

(4)指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；

(6)"sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；

(5)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。

C/C++程序的内存分区

1）、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其

操作方式类似于数据结构中的栈。

2）、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回

收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3）、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的

全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另

一块区域。 - 程序结束后由系统释放。

4）、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放

5）、程序代码区—存放函数体的二进制代码。

栈区与堆区的区别：

1）堆和栈中的存储内容：栈存局部变量、函数参数等。堆存储使用new、malloc申请的变量等；

2）申请方式：栈内存由系统分配，堆内存由自己申请；

3）申请后系统的响应：栈——只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆——首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表 中删除，并将该结点的空间分配给程序；

4）申请大小的限制：Windows下栈的大小一般是2M，堆的容量较大；

5）申请效率的比较：栈由系统自动分配，速度较快。堆使用new、malloc等分配，较慢；

总结：栈区优势在处理效率，堆区优势在于灵活；

进程跟线程

多线程

多线程是多任务处理的一种特殊形式，多任务处理允许让电脑同时运行两个或两个以上的程序。一般情况下，两种类型的多任务处理：基于进程和基于线程。

基于进程的多任务处理是程序的并发执行。

基于线程的多任务处理是同一程序的片段的并发执行。

多线程程序包含可以同时运行的两个或多个部分。这样的程序中的每个部分称为一个线程，每个线程定义了一个单独的执行路径。

C++ 不包含多线程应用程序的任何内置支持。相反，它完全依赖于操作系统来提供此功能。

Linux 操作系统

#include <pthread.h>

pthread\_create (thread, attr, start\_routine, arg)

参数 描述

thread 一个不透明的、唯一的标识符，用来标识例程返回的新线程。

attr 一个不透明的属性对象，可以被用来设置线程属性。您可以指定线程属性对象，也可以使用默认值 NULL。

start\_routine C++ 例程，一旦线程被创建就会执行。

arg 一个可能传递给 start\_routine 的参数。它必须通过把引用作为指针强制转换为 void 类型进行传递。如果没有传递参数，则使用 NULL。

终止线程

pthread\_exit (status)

#include <iostream>

// 必须的头文件是

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

// 线程的运行函数

void\* say\_hello(void\* args)

{

cout << "Hello w3cschool！" << endl;

}

int main()

{

// 定义线程的 id 变量，多个变量使用数组

pthread\_t tids[NUM\_THREADS];

for(int i = 0; i < NUM\_THREADS; ++i)

{

//参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数

int ret = pthread\_create(&tids[i], NULL, say\_hello, NULL);

if (ret != 0)

{

cout << "pthread\_create error: error\_code=" << ret << endl;

}

}

//等各个线程退出后，进程才结束，否则进程强制结束了，线程可能还没反应过来；

pthread\_exit(NULL);

}

进程

1. 进程的优点

1）健壮性比线程好；

2）由于进程间各自互相独立，所以如果多进程程序设计，当有一个进程挂掉的时候，不会影响其他进程的执行。

2.进程的缺点

1）创建进程比创建线程会消耗更多的系统性能； 2）由于进程创建需要系统分配更多的资源，所以创建会更加复杂； 3）进程间的通信比线程间的通信要复杂； 4）进程间的通信，如消息队列，共享内存，信号量，socket等，通信非常的复杂，而线程共享虚拟内存，可以直接访问全局变量

进程与线程的区别

从形态角度 一个进程可包含一个或多个线程

从调度角度 1）进程是资源分配的基本单位； 2）线程是处理器调度的独立单位。

# Compare用法

方法一：#include <algorithm> sort

sort(begin,end)

int main()

{

int a[20]={2,4,1,23,5,76,0,43,24,65},i;

for(i=0;i<sizeof(a)/sizeof(int);i++)

cout<<a[i]<<" ";

sort(a,a+20);

cout<<endl;

for(i=0;i<sizeof(a)/sizeof(int);i++)

cout<<a[i]<<" ";

cout<<endl;

while(1);

return 0;

}

sort(begin,end,compare)

bool compare011(int a,int b)

{

return a>b; //升序排列，如果改为return a>b，则为降序

}

int main()

{

int a[20]={2,4,1,23,5,76,0,43,24,65},i;

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

sort(a,a+20,compare011);

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

while(1);

return 0;

}

enum Enumcomp{ASC,DESC};

class compare

{

private:

Enumcomp comp;

public:

compare(Enumcomp c):comp(c) {};

bool operator () (int num1,int num2) //()operator overload

{

switch(comp){

case ASC:

return num1<num2;

case DESC:

return num1>num2;

}

}

};

int main()

{

int a[20]={2,4,1,23,5,76,0,43,24,65},i;

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

sort(a,a+20,compare(DESC));

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

while(1);

return 0;

}

qsort():

原型:

\_CRTIMP void \_\_cdecl qsort (void\*, size\_t, size\_t,int (\*)(const void\*, const void\*));

解释: qsort ( 数组名 ，元素个数，元素占用的空间(sizeof)，比较函数)

int compare(const void \*a,const void \*b)

{

return \*(int\*)b-\*(int\*)a;

}

int main()

{

int a[20]={2,4,1,23,5,76,0,43,24,65},i;

for(i=0;i<20;i++)

cout<<a[i]<<endl;

qsort((void \*)a,20,sizeof(int),compare);

for(i=0;i<20;i++)

cout<<a[i]<<endl;

return 0;

}

升序：sort(begin,end,less<data-type>());

降序：sort(begin,end,greater<data-type>()). 库函数Functional

int main()

{

int a[20]={2,4,1,23,5,76,0,43,24,65},i;

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

sort(a,a+20,greater<int>());

for(i=0;i<20;i++)

cout<<a[i]<<" ";

cout<<endl;

while(1);

return 0;

}

string 就可以使用反向迭代器

int main()

{

string str("cvicses");

string s(str.rbegin(),str.rend());

cout << s <<endl;

return 0;

}

# C语言优先级

[编辑](https://baike.baidu.com/item/%E8%BF%90%E7%AE%97%E7%AC%A6%E4%BC%98%E5%85%88%E7%BA%A7/javascript:;)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 优先级 | 运算符 | 名称或含义 | 使用形式 | 结合方向 | 说明 |
| 1 | [] | 数组下标 | 数组名[常量表达式] | 左到右 |  |
| () | 圆括号 | （表达式）/函数名(形参表) |  |
| . | 成员选择（对象） | 对象.成员名 |  |
| -> | 成员选择（指针） | 对象指针->成员名 |  |
| 2 | - | 负号运算符 | -常量 | 右到左 | 单目运算符 |
| (类型) | 强制类型转换 | (数据类型)表达式 |  |
| ++ | 自增运算符 | ++变量名/变量名++ | 单目运算符 |
| -- | 自减运算符 | --变量名/变量名-- | 单目运算符 |
| \* | 取值运算符 | \*指针变量 | 单目运算符 |
| & | 取地址运算符 | &变量名 | 单目运算符 |
| ! | 逻辑非运算符 | !表达式 | 单目运算符 |
| ~ | 按位取反运算符 | ~表达式 | 单目运算符 |
| sizeof | 长度运算符 | sizeof(表达式) |  |
| 3 | / | 除 | 表达式/表达式 | 左到右 | 双目运算符 |
| \* | 乘 | 表达式\*表达式 | 双目运算符 |
| % | 余数（取模） | 整型表达式/整型表达式 | 双目运算符 |
| 4 | + | 加 | 表达式+表达式 | 左到右 | 双目运算符 |
| - | 减 | 表达式-表达式 | 双目运算符 |
| 5 | << | 左移 | 变量<<表达式 | 左到右 | 双目运算符 |
| >> | 右移 | 变量>>表达式 | 双目运算符 |
| 6 | > | 大于 | 表达式>表达式 | 左到右 | 双目运算符 |
| >= | 大于等于 | 表达式>=表达式 | 双目运算符 |
| < | 小于 | 表达式<表达式 | 双目运算符 |
| <= | 小于等于 | 表达式<=表达式 | 双目运算符 |
| 7 | == | 等于 | 表达式==表达式 | 左到右 | 双目运算符 |
| != | 不等于 | 表达式!= 表达式 | 双目运算符 |
| 8 | & | 按位与 | 表达式&表达式 | 左到右 | 双目运算符 |
| 9 | ^ | 按位异或 | 表达式^表达式 | 左到右 | 双目运算符 |
| 10 | | | 按位或 | 表达式|表达式 | 左到右 | 双目运算符 |
| 11 | && | 逻辑与 | 表达式&&表达式 | 左到右 | 双目运算符 |
| 12 | || | 逻辑或 | 表达式||表达式 | 左到右 | 双目运算符 |
| 13 | ?: | 条件运算符 | 表达式1? 表达式2: 表达式3 | 右到左 | 三目运算符 |
| 14 | = | 赋值运算符 | 变量=表达式 | 右到左 |  |
| /= | 除后赋值 | 变量/=表达式 |  |
| \*= | 乘后赋值 | 变量\*=表达式 |  |
| %= | 取模后赋值 | 变量%=表达式 |  |
| += | 加后赋值 | 变量+=表达式 |  |
| -= | 减后赋值 | 变量-=表达式 |  |
| <<= | 左移后赋值 | 变量<<=表达式 |  |
| >>= | 右移后赋值 | 变量>>=表达式 |  |
| &= | 按位与后赋值 | 变量&=表达式 |  |
| ^= | 按位异或后赋值 | 变量^=表达式 |  |
| |= | 按位或后赋值 | 变量|=表达式 |  |
| 15 | , | 逗号运算符 | 表达式,表达式,… | 左到右 | 从左向右顺序运算 |

