20220421-C++

1.过程描述

-基础知识篇

1.Const

1.1 含义

1.2 作用

1.3 const对象默认为文件局部变量

1.4 指针与const

1）指向常量的指针

2）常指针

3）指向常量的常指针

1.5 函数中使用const

1）const修饰函数的返回值

2）const修饰函数参数

1.6 类中使用const

2. Static

2.1 静态变量

1）函数中的静态变量

2）类中的静态变量

2.2 静态对象

2.3 类中的静态函数

2.4 限定访问范围

3.this指针

3.1 作用

3.2 使用

3.3 示例

-容器篇

1.Vector

2.队列

3.Stack

4.集合

5.Map

6.List

2.结果输出

1.过程描述

–基础知识篇

1.Const

1.1 含义

常类型是值使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的

1.2 作用

- 定义常量

```C++
const int a=10;
```

- 类型检查
  - const常量与#define的区别在于：const常量具有类型，编译器可以进行安全检查，而#define宏定义没有数据类型，只是简单的字符串替换，不能进行安全检查
  - const定义的变量只有类型为整形或枚举、且以常量表达式初始化时才能作为常量表达式。其它情况下它只是一个const限定的变量，不要将其与常量混淆
- 防止修改，起保护作用，增加程序健壮性

C++ 复制代码

```cpp
void f(const int i)
{
    i++;//error
}
```

- 可以节省空间，避免不必要的内存分配
  - const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是像#define一样给出的是立即数
  - const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝

1.3 const对象默认为文件局部变量

注意：非const变量默认为extern，要使const变量能够在其它文件中访问，必须在文件中显式地指定为extern

C++ 复制代码

```cpp
未被const修饰：
// file1.cpp
int ext;
// file2.cpp
#include<iostream>

extern int ext;
int main(){
    std::cout<<(ext+10)<<std::endl;
}

被const修饰
//extern_file1.cpp
extern const int ext=12;//重点在这里的extern
//extern_file2.cpp
#include<iostream>
extern const int ext;
int main(){
    std::cout<<ext<<std::endl;
}
```

可以发现，未被const修饰的变量不需要extern显式声明，而const常量需要显式声明extern，并且需要做初始化（针对file1而言）。因为常量在定义后就不能被修改，所以定义时必须初始化。

1.4 指针与const

与指针相关的const有四种：

C++ 复制代码

```
const char* a;//指向const对象的指针或者说指向常量的指针
char const* a;//同上
char* const a;//指向类型对象的const指针
const char* const a;//指向const对象的const指针
```

如果const位于*的左侧，说明const是用来修饰指针指向的变量，即指针指向为常量；
如果const位于*的右侧，说明const就是修饰指针本身，即指针本身是常量

1）指向常量的指针

ptr是一个指向int类型const对象的指针，const定义的是int类型，也就是ptr所指向的对象类型，而不是ptr本身。所以ptr可以不用赋初始值，但是不能通过ptr去修改所指对象的值

C++ 复制代码

```
const int *ptr;
*ptr=10;//error
```

不能使用void*指针保存const对象的地址，必须使用const void*类型的指针

C++ 复制代码

```
const int p=10;
const void * vp=&p;
void *vp=&p;//error
```

有一个重点：允许把非const对象的地址赋给指向const对象的指针
不能通过ptr指针来修改b的值，但可以用其它方式来修改b的值

C++ 复制代码

```
const int *ptr;
int b=10;
ptr=&b;
*ptr=50;//error
int c = 20;
b = c;
```

2）常指针

const指针必须进行初始化，且const指针的值不能修改

C++ 复制代码

```
int main()
{
    int num = 0;
    int* const ptr = &num;
    int num2 = 10;
    ptr = &num2;//error
}
```

此外，当把一个const常量的地址赋值给ptr时，由于ptr指向的是一个变量，而不是const常量，所以会报错

C++ 复制代码

```
int main(){
    const int num=0;
    int * const ptr=&num; //error! const int* -> int*
    cout<<*ptr<<endl;
}
```

3）指向常量的常指针

ptr是一个const指针，然后指向了一个int类型的const对象

```cpp
const int p = 3;
const int * const ptr = &p;
```

1.5 函数中使用const

1）const修饰函数的返回值

```cpp
const int func1();//这个本身无意义，因为函数返回本身就是赋值给其它的变量；

const int* func2();//指针指向的内容不变

int* const func2();//指针本身不可变
```

2）const修饰函数参数

- 传递过来的参数及指针本身在函数内不可变，无意义

```cpp
void func(const int var);//传递过来的参数不可变
void func(int *const var);//指针本身不可变
```

表明参数在函数体内不能被修改，但此处没有任何意义，var本身就是形参，在函数内不会改变。传入的形参是指针也是一样。
输出参数采用值传递，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加const修饰

- 参数指针所指内容为常量不可变

```cpp
void stringcopy(char* dst,const char* src);
```

当函数体内的语句试图改动src指向的内容，编译器将指出错误

- 参数为引用，为了增加效率同时防止修改

```cpp
void func(const A &a)
```

对于非内部数据类型的参数而言，像void func(int a)这样声明的函数注定效率比较低，因为函数体内将产生A类型的临时对象用于复制复制参数a，而临时对象的构造、复制、析构过程都将小号时间。
而按引用传递则不会产生临时对象。但如果不加const，参数a的值有可能改变。以此类推，是否必须将void func(int x)改写为void func(const int & x)以提高效率。完全没必要，因为像int这样的内部数据类型的参数不存在构造、析构的过程，复制也非常快，按值传递和引用传递的效率几乎相当

1.6 类中使用const

在一个类中，任何不会修改数据成员的函数都应该声明为const类型。如果在编写const成员函数时，不慎修改数据成员t或调用了其它非const成员函数，编译器将指出错误，这能提高程序的健壮性
使用const关键字进行说明的成员函数，称为常成员函数，只有常成员函数才有资格操作常量或常对象，没有const关键字进行说明的成员函数不能用来操作常对象。（ex：void take(int num) const）
对于类中的const成员变量必须通过初始化列表进行初始化：

```cpp
class Apple{
private:
    int people[100];
public:
    Apple(int i);
    const int apple_number;
};

Apple::Apple(int i):apple_number(i)
{

}
```

const对象只能访问const成员函数，而非const对象可以访问任意的成员函数，包括const成员函数：

C++ 复制代码

```cpp
//apple.cpp
class Apple
{
private:
    int people[100];
public:
    Apple(int i);
    const int apple_number;
    void take(int num) const;
    int add();
    int add(int num) const;
    int getCount() const;

};
//main.cpp
#include<iostream>
#include"apple.cpp"
using namespace std;

Apple::Apple(int i):apple_number(i)
{

}
int Apple::add(){
    take(1);
    return 0;
}
int Apple::add(int num) const{
    take(num);
    return num;
}
void Apple::take(int num) const
{
    cout<<"take func "<<num<<endl;
}
int Apple::getCount() const
{
    take(1);
    add();  // error
    return apple_number;
}
int main(){
    Apple a(2);
    cout<<a.getCount()<<endl;
    a.add(10);
```

```
46        return 0;
47    }
```

这里由于add并非const修饰，所以运行报错

C++ 复制代码

```
1    const Apple b(3);
2    b.add();//error
```

const对象只能访问const成员函数
上面实现的类除了利用参数列表进行常量的初始化外，也可以直接使用以下方法：
第一：将常量定义与static结合

C++ 复制代码

```
1    static const int apple_number
```

第二：在外面初始化（只有静态成员才允许在类以外定义）

C++ 复制代码

```
1    const int Apple::apple_number=10;
```

C++11允许直接在定义初始化，可以写成：

C++ 复制代码

```
1    static const int apple_number=10;
2    // 或者
3    const int apple_number=10;
```

此外，还有一点，static静态成员变量不能在类的内部初始化。在类的内部只是声明，通常在类的实现文件中初始化

```cpp
class Apple
{
private:
    int people[100];
public:
    static int apple_number=10;//error

};
```

```cpp
class Apple
{
private:
    int people[100];
public:
    static int apple_number;

};
int Apple::apple_number = 10;//OK
```

2. Static

2.1 静态变量

1）函数中的静态变量

当变量声明为static时，空间将在程序的生命周期内分配。即时多次调用该函数，静态变量的空间也只分配一次。前一次调用中的变量值通过下一次函数调用传递。主要适用于那些需要存储先前函数状态的场景

```cpp
void demo()
{
    static int count=0;
    count++;
}
```

2）类中的静态变量

由于声明为static的变量只被初始化一次，因为他们在单独的静态存储中分配空间，因此类中的静态变量由所有对象共享。对于不同的对象，不能由相同静态变量的多个副本，也是因为这个原因，静态变量不能使用构造函数初始化。

C++ 复制代码

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

class Apple
{
public:
    static int i;

    Apple()
    {
        // Do nothing
    };
};

int main()
{
Apple obj1;
Apple obj2;
obj1.i =2;
obj2.i = 3;

// prints value of i
cout << obj1.i<<" "<<obj2.i;
}
//会报错
```

C++ 复制代码

```cpp

#include<iostream>
using namespace std;

class Apple
{
public:
    static int i;

    Apple()
    {
        // Do nothing
    };
};

int Apple::i = 1;

int main()
{
    Apple obj;
    // prints value of i
    cout << obj.i;
}
//OK
```

2.2 静态对象

就像变量一样，对象也在声明为static时具有范围，直到程序的声明周期

C++ 复制代码

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

class Apple
{
    int i;
    public:
        Apple()
        {
            i = 0;
            cout << "Inside Constructor\n";
        }
        ~Apple()
        {
            cout << "Inside Destructor\n";
        }
};

int main()
{
    int x = 0;
    if (x==0)
    {
        static Apple obj;
    }
    cout << "End of main\n";
}
//程序输出为:
Inside Constructor
End of main
Inside Destructor
```

2.3 类中的静态函数

与静态变量类似，静态成员函数也不依赖于类的对象，可以用对象和"."来调用静态成员函数，但建议使用类名和范围解析运算符"::"来调用静态成员。

静态成员函数仅允许访问静态成员或其它静态成员函数，它们无法访问类的非静态数据成员或函数

C++ 复制代码

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

class Apple
{
    public:
        // static member function
        static void printMsg()
        {
            cout<<"Welcome to Apple!";
        }
};

// main function
int main()
{
    // invoking a static member function
    Apple::printMsg();
}
```

2.4 限定访问范围

static还具有限定访问范围的作用

C++ 复制代码

```cpp
//source1.cpp
extern void sayhello();
static const char* msg="Hello world!\n";//这里加了static，限定了msg的访问范围
int main()
{
    sayHello();
    return 0;
}

//source2.cpp
extern char* msg;
void sayhello()
{
    printf("%s",msg);
}
//会报错
```

3.this指针

3.1 作用

this在成员函数的开始执行前构造，在成员函数的执行结束后清楚。this指针的作用包括：

- 一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果
- this作用域在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数，对各成员的访问均通过this进行

3.2 使用

this指针的使用：

- 当类的非静态成员函数中返回类对象本身的时候，使用return *this；
- 当参数与成员变量名相同时，如this->n=n；

3.3 示例

C++ 复制代码

```cpp
#include<iostream>
#include<cstring>

using namespace std;
class Person {
public:
    typedef enum {
        BOY = 0,
        GIRL
    }SexType;
    Person(const char* n, int a, SexType s) {
        name = new char[strlen(n) + 1];
        strcpy_s(name,strlen(n)+1, n);
        age = a;
        sex = s;
    }
    int get_age() const {

        return this->age;
        //解析为get_age(const A* const this)
    }
    Person& add_age(int a) {
        age += a;
        return *this;
        //解析为add_age(A* const this,int a)
    }
    ~Person() {
        delete[] name;
    }
private:
    char* name;
    int age;
    SexType sex;
};


int main() {
    Person p("zhangsan", 20, Person::BOY);
    cout << p.get_age() << endl;
    cout << p.add_age(10).get_age() << endl;
    return 0;
}
```

–容器篇

1.Vector

C++ | 复制代码

```cpp
#include <iostream>
#include <vector>
#include <array>
using namespace std;

int main()
{
    vector<int> int_vec(3,100);//创建一个有3个100的vector
    for (int i = 0; i < 5; i++)
    {
        int_vec.push_back(i);//在末尾加入一个元素
    }
    cout <<int_vec.front()<< endl;//返回vector当前第一个元素值
    cout << int_vec[2] << endl;//vector支持随机访问，可以像数组一样用[ ]取值
    cout << int_vec.back() << endl;//返回当前最后一个元素值
    cout << int_vec.size() << endl;//返回当前vector的长度
    int_vec.pop_back();//删除末尾的元素
    cout << int_vec.back() << endl;
    int_vec.clear();
    if (int_vec.empty())
    {
        cout << "int_vec is empty" << endl;
    }
    //insert操作
    int_vec.insert(int_vec.begin(), 0);//这里.begin返回的是一个iterator
    int_vec.insert(int_vec.begin()+1, 1);
    int_vec.insert(int_vec.end(), 2);
    cout << int_vec.back() << endl;

    vector<int> anothervector(2, 400);
    int_vec.insert(int_vec.end(), anothervector.begin(), anothervector.end());//插入另外一个vector
    cout << int_vec.back() << endl;

    /*
    int demo[] = {900,800,700};
    int_vec.insert(int_vec.back(), demo, demo + 3);无法编译
    */

    for (auto it = int_vec.begin(); it != int_vec.end(); it++)
    {
        cout << *it <<",";//iterator本质是指针，通过for循环对vector中的元素进行遍历
    }

```

```
44        //emplace操作
45        vector<int> newVec={100};
46        int i = 0;
47        while (i < 5)
48        {
49            newVec.emplace_back(i);//构造一个对象添加到容器中，省去了拷贝的副本，效率更高
50            i++;
51        }
52        cout << "\n"<<newVec.back() << endl;
53        newVec.emplace(newVec.end(), 800);
54        cout << "\n" << newVec.back() << endl;
55
56        return 0;
57
58    }
```

2.队列

C++ | 复制代码

```cpp
#include <iostream>
#include <queue>

using namespace std;
int main()
{
    queue<char> char_que;
    for (int i = 0; i < 5; i++)
    {
        char_que.push(i + 'a');//
    }
    cout << char_que.front() << endl;
    cout << char_que.back() << endl;
    cout << char_que.size() << endl;
    char_que.pop();//从队首删除一个元素
    cout << char_que.front() << endl;
    /*
    queue不支持随机访问，不能像数组一样任意取值；
    queue不可以用clear函数清空，必须一个一个弹出
    queue不支持遍历，无论是数组型遍历还是迭代器型遍历都不支持，所以没有begin(),end()函数
    */

    //双向队列
    deque<char> char_deque;
    for (int i = 0; i < 5; i++)
    {
        char_deque.push_front(i + 'a');
        char_deque.push_back(i + 'A');
    }
    cout << char_deque.front() << endl;
    cout << char_deque.back() << endl;
    cout << char_deque[3] << endl;
    cout << char_deque.size() << endl;
    char_deque.pop_front();
    char_deque.pop_back();
    cout << char_deque.front() << endl;
    cout << char_deque.back() << endl;

    for (auto it = char_deque.begin(); it != char_deque.end(); it++)
    {
        cout << *it << " ";
    }
    char_deque.clear();
    cout << char_deque.size() << endl;
```

```
45
46          /*
47          双向队列支持双向操作，可以在队首或队尾插入或删除元素
48          并且支持随机访问，可以通过deuqe[index]访问某一位置的元素
49          同时也有begin(),end()等函数
50          */
51      }
```

3.Stack

C++ | 复制代码

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

int main()
{
    stack<int> int_stack;
    for (int i = 0; i < 5; i++)
    {
        int_stack.push(i);
    }
    for (int i = 0; i < 5; i++)
    {
        cout << int_stack.size() << endl;
        cout << int_stack.top() << endl;
        int_stack.pop();
    }
}
```

4.集合

C++ 复制代码

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

int main()
{
    set<int> int_set;
    for (int i = 0; i < 5; i++)
    {
        int_set.insert(i);
    }
    for (auto start = int_set.begin(); start != int_set.end(); start++)
    {
        //set.begin()函数和set.end()函数返回set的首尾迭代器
        cout << *start << endl;
    }
    cout << int_set.size() << endl;
    auto it = int_set.find(2);
    cout << *it << endl;
    int_set.erase(2);
    int_set.emplace(1);
    cout << int_set.size() << endl;//元素互异性
    return 0;
    //set容器中的元素是默认排好顺序的（按升序排列），这与数学中集合的无序性不同
}
```

5.Map

C++ | 复制代码

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

int main()
{
    map<int, char>my_map;
    my_map[0] = 'a';
    my_map[1] = 'b';
    my_map.insert(map<int, char>::value_type(2, 'c'));//插入元素
    my_map[3] = 'd';
    auto it = my_map.find(2);
    cout << "key: " << it->first << " value: " << it->second << endl;
    my_map.erase(0);
    my_map.erase(it);
    //my_map.erase('d');失败
    cout << my_map.size() << endl;
    for (auto it = my_map.begin(); it != my_map.end(); it++)
//map.begin()、map.end()：返回map的首、尾迭代器
    {
        cout << "key: " << it->first << " value: " << it->second << endl;
    }
    my_map.clear();
 //map.clear()：删除map所有元素
    if (my_map.empty())
 //map.empty()：返回map是否为空，为空返回1否则0
    {
        cout << "my_map is empty !" << endl;
    }
    return 0;
}
```

6.List

C++ | 复制代码

```cpp
#include <iostream>
#include <list>
using namespace std;
void PrintList(list<int> my_list)
{
    for (auto i = my_list.begin(); i != my_list.end(); i++)
    {
        cout << *i << " ";
    }
}

int main()
{
    list<int> my_list;
    for (int i = 0; i < 5; i++)
    {
        my_list.push_back(i);
        my_list.push_front(i);
    }
    PrintList(my_list);


    cout << "\n";
    my_list.sort();
    PrintList(my_list);


    cout << "\n";
    my_list.insert(my_list.begin(), 8);//不允许在出begin和end以外的其它位置插入
    my_list.insert(my_list.end(), 8);
    PrintList(my_list);


    cout << "\n";
    my_list.remove(2);//删除某个元素
    PrintList(my_list);


    //list为双向链表
}

```

2.结果输出

今天主要看了一点C++那些事的内容，把const static this指针以及STL中的一些常用容器给过了一遍。还是没能开始数据结构和算法的学习，自打十巴掌。