第十二章 工具与技术

目录

前言

学习目标

1. 理解并掌握命名空间的使用方法；
2. 掌握异常处理的使用方法；
3. 理解和使用多重继承以及虚继承；
4. 了解嵌套类和运行时类型识别的使用方法；
5. 学会使用标准库中一些特殊工具，包括 tuple 类型、bitset 类型以及对日期和时间的处理。

12.1 命名空间

将大量由于共同开发等原因使用的全局名字引入到同一作用域时，不可避免会产生命名冲突，例如：

定义相同名称的不同函数

```
//Foo.h
int doSomething(int x, int y) {
    return x * y;
}
//Goo.h
int doSomething(int x, int y) {
    return x + y;
}
//main.cpp
#include "Foo.h"
#include "Goo.h"
int main() {
    int x = doSomething(2, 1); //错误: doSomething已
      经被定义
}
```

说明

- 用户先在不同头文件里定义了同名的 doSomething 函数
- main.cpp 文件中的代码在引用了均包含 doSomething 函数的不同头文件后，试图调用 doSomething，引发编译错误

12.1 命名空间

定义命名空间

命名空间

命名空间可以将全局作用域内有效的类名、函数名或对象名组织到一个名字下面。即将全局作用域分割为子作用域，每个子域称为一个命名空间。

命名空间示例

```
//Foo.h
namespace Foo {
int doSomething(int x, int y);
class X{ /*...*/ };
}
```

说明

- 定义一个命名空间以关键字 namespace 开始，后面跟命名空间的名字
- 主体由一对花括号括起来的声明和定义组成
- 左侧代码定义了一个名为 Foo 的命名空间，该命名空间包括两个成员：一个函数和一个类

12.1 命名空间

定义命名空间

一个命名空间是可以不连续的。通常情况下，我们将命名空间成员声明放到头文件中，实现放在源文件，从而达到接口和实现分离的目的。例如：

命名空间示例

```
//Foo.h
namespace Foo {
int doSomething(int x, int y);
class X{ /*...*/ };
}

//Foo.cpp
namespace Foo {
X g_x;
int doSomething(int x, int y) { /*...*/ }
}
```

说明

- 已经在头文件 Foo.h 中定义了命名空间 Foo
- Foo.cpp 源文件中，打开命名空间 Foo，同时为它新增成员 g_x 以及函数 doSomething 的实现

~~外部访问命名空间~~

```
int x = Foo::doSomething(2, 1);
```

12.1 命名空间

定义命名空间

在命名空间外部访问它内部的成员时，必须要明确指出成员所属的命名空间：

外部访问命名空间

```
int x = Foo::doSomething(2, 1);
```

说明

- 调用命名空间 Foo 中的成员 doSomething

12.1 命名空间

定义命名空间

在命名空间外部访问它内部的成员时，必须要明确指出成员所属的命名空间：

外部访问命名空间

```
int x = Foo::doSomething(2, 1);
```

说明

- 调用命名空间 Foo 中的成员 doSomething

提示

一个命名空间可以定义在全局作用域内，也可以定义在其它命名空间内，但不能定义在函数或类的内部。

命名空间的嵌套

在一个命名空间内部定义另外一个命名空间

命名空间的嵌套示例

```
namespace Wang {
    namespace Goo {
        int doSomething(int x, int y);
    }
    namespace Boo {
        class Y {/*...*/};
    }
}
```

说明

- 上面的代码在命名空间 Wang 的内部分别定义了另外两个命名空间：Goo 和 Boo
- 要访问内层命名空间的名字，必须要使用嵌套的命名空间名字

嵌套命名空间的访问

```
int x = Wang::Goo::doSomething(2, 1);
```

C++11 标准新增**内联命名空间（inline namespace）**，用来指示命名空间中的名称可以在外层命名空间中直接使用。当一个程序的新版本发布时，我们使用内联命名空间，例如：

内联命名空间示例

```
namespace FirstVersion {
    void fun(int);
}
inline namespace SecondVersion {
    void fun(int);
    void fun(double);
}

FirstVersion::fun(1); //调用早期版本fun函数
fun(1); //调用当前版本fun函数
fun(1.0); //调用当前版本中新增的fun 函数
```

说明

- 关键字 inline 必须出现在一个命名空间第一次定义的地方
- 命名空间 SecondVersion 是内联的，因此它的成员可以在外层作用域（全局作用域）直接访问
- 如果要访问早期版本（非内联）的成员，则必须要指明所属的版本名字

定义在全局作用域中的名字也是定义在**全局命名空间**中的。我们可以直接使用作用域操作符访问全局命名空间的成员：

访问全局命名空间成员

```
::member_name
```

说明

- 全局命名空间是隐式声明的，每个文件将全局作用域内定义的名字添加到全局命名空间中。

为了简化如下的繁琐的命名空间成员的访问方式，我们可以使用 **using 声明（using declaration）**或 **using 指示（using directive）**

繁琐的命名空间成员访问形式

```
Wang::Goo::doSomething
```

一条 **using 声明**语句用来引用命名空间中的一个成员。例如，我们使用 using 声明引入标准库命名空间 std 中成员 cout：

使用 using 声明

```
using std::cout; //此声明告诉编译器后续cout属于命名空
  间std
cout << "Hello world"; //cout等价于std::cout
```

说明

- using 声明引入的名字的作用域从声明的地方开始，直到 using 声明所在的作用域结束处为止
- 左侧的 using 声明表明在其作用域范围内，所有的 cout 都是指 std::cout

using 声明一次只能引入命名空间的一个成员，如果要引入一个命名空间内所有的成员，我们可以使用 **using 指示**:

使用 using 指示

```
using namespace std;
```

说明

- using 指示意味着 std 中所有成员在此处都可见

注意

- 指示可以出现在全局作用域、局部作用域和命名空间作用域中，但是不能出现在类的作用域中
- using 指示的作用域也是从声明的地方开始，直到 using 语句所在的作用域结束处为止

using 声明一次只能引入命名空间的一个成员，如果要引入一个命名空间内所有的成员，我们可以使用 **using 指示**:

使用 using 指示

```
using namespace std;
```

说明

- using 指示意味着 std 中所有成员在此处都可见

提示

虽然 using 声明只能引入命名空间中的一个成员，但与 using 指示相比，它不易引起命名冲突，是一种更安全的方式。

12.2 异常处理

我们很难保证一个大型程序在运行期间不会出现错误，如果出现了错误，程序很可能无法正确运行，甚至会崩溃。**异常处理（exception handling）**允许我们将异常检测和解决的过程分离开来，程序中某一个模块出现了异常不会导致整个程序无法正确运行。C++ 语言提供了异常内部处理机制，该处理机制涉及到三个关键字：

- try：检测可能产生异常的语句块
- catch：捕获异常
- throw：抛出异常

12.2 异常处理

抛出异常

当程序在运行期间出现异常时，我们可以通过 throw 来抛出一个异常。例如，以下函数返回 a 除以 b 的结果，如果出现除数为 0 的情况：

抛出一个异常

```
double divide(int a, int b){
    if (b == 0)
        throw "Error, division by zero!";
    return a / b;
}
```

抛出各种类型的异常

```
throw -1; // 抛出一个整型数
throw x; // x为double类型对象
throw MyException("Fatal Error"); //MyException为一个
  类类型
```

说明

- throw 可以抛出任何类型对象
- 通常情况下，抛出的异常为错误的编号、错误描述或用户自定义的异常类对象。
- 当执行 throw 语句时，其后面的语句不会被执行。程序的控制权将转移到与之匹配的 catch 模块

12.2 异常处理——检测异常

C++ 语言通过**关键字** try 来检测可能发生异常的代码。通常情况下，我们将可能发生异常的代码放到 try 语句块中，该语句块中的任何异常都可以被检测到。

检测异常

```
try{
    divide(a, b); //函数调用语句
}
```

说明

- 一旦在 try 语句块内部有异常抛出时，系统检查与该 try 块关联的 catch 子句，并寻找与异常相匹配的 catch 子句。

12.2 异常处理——捕获异常

最终，我们通过 **catch 子句**捕获异常，并处理它：

检测异常

```
catch (const char *str) {//捕获一个C风格字符串常量对象
    //任何能够被char *接受的异常都将被捕获
    cerr << "捕获异常" << str << endl; //cerr为标准错
      误ostream对象
}
```

说明

- cerr 为标准错误 ostream 对象用于输出程序错误信息。
- catch 语句中的异常声明类似于只包含一个形参的函数形参列表。
- 异常声明包括类型和名字，其中类型决定了该 catch 子句能够捕获的异常的类型
- 能够捕获的错误类型可以为左值引用，但不能为右值引用。

12.2 异常处理——捕获异常

一个包含 try、catch 和 throw 的异常处理案例如下:

异常处理案例

```
int a = 1, b = 0;
try {
    int c = divide(a, b);
}
catch (const string &str) {
    cerr << str << endl;
}
catch (const char *str) {
    cerr << str << endl;
}
```

说明

异常被抛出后:

1. try 后面的 catch 尝试匹配，匹配则处理
2. 否则沿着调用链向外层逐层检查
3. 无法匹配，则调用 terminate 终止程序

问题

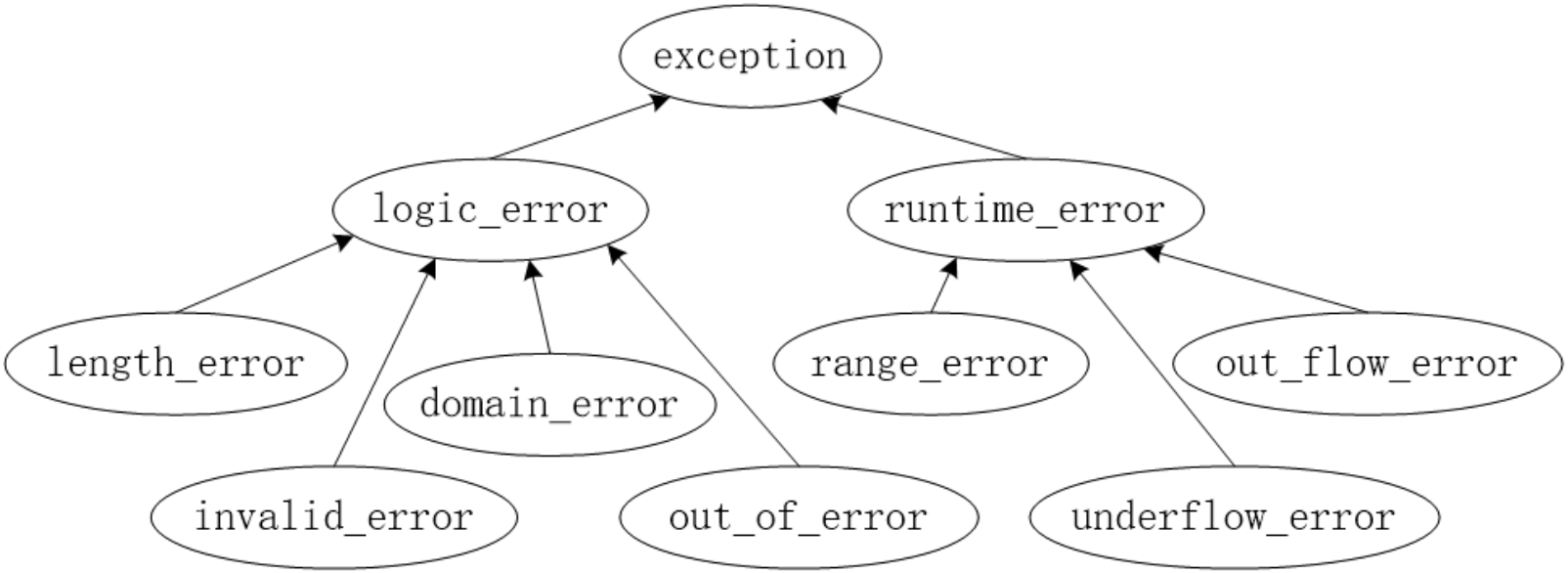
异常被谁捕获?

注意

通常情况下，异常的类型和 catch 声明的类型要求严格匹配，但不包括以下情况：1）非常量到常量的转换；2）派生类向基类的转换；3）数组或函数被转换成指向数组元素或函数的指针。

12.2 异常处理——标准库异常类

C++ 标准库提供了**标准异常类（standard exception）**，使用时需要包含头文件 exception。其继承关系如图：



基类 exception

基类 exception 只定义了默认的构造函数、复制构造函数、赋值运算符、虚析构函数和一个名为 **what** 的虚成员。

what 函数返回一个 const char*，指向一个以 null 结尾的字符数组，用于提示异常类型。我们可以继承 exception 类并重载 what：

自定义版本的 what 成员

```
struct MyException :public exception {
    const char* what() const noexcept { return "Ooops
      !"; }
};
```

noexcept

C++11 标准引入的新关键字，用来指明某个函数不会抛出异常

说明

noexcept 应置于：

- 形参列表后面
- 如修饰成员函数，在 const 限定符之后，final、override 或纯虚函数 =0 之前
- 函数的声明和定义处必须都要出现

12.2 异常处理——标准库异常类

what 函数返回一个 const char*，指向一个以 null 结尾的字符数组，用于提示异常类型。我们可以继承 exception 类并重载 what：

自定义版本的 what 成员

```
struct MyException :public exception {
    const char* what() const noexcept { return "Ooops
      !"; }
};
```

noexcept

C++11 标准引入的新关键字，用来指明某个函数不会抛出异常

提示

noexcept 说明可以优化代码的执行效率。

12.2 异常处理——标准库异常类

下面的代码将抛出一个 MyException 异常对象，该对象可以被异常声明为基类 exception类型的 catch 子句捕获：

使用 MyException 对象

```
try {
    throw MyException();
}
catch (exception &ex) {
    cerr << ex.what() << endl;
    }
```

12.3 多重继承与虚继承——多重继承

多重继承

为一个派生类指定多个基类的继承结构称为多重继承

多重继承

```
class Mammal {
public:
    virtual void feedMilk() {} //母乳喂养
};
class WingedAnimal {
public:
    virtual void flap() {} //振翅飞翔
};
class Bat: public Mammal, public WingedAnimal { };
```

说明

- Bat 类的派生列表中有两个以逗号分隔的基类
- Bat 类对象将具有 Mammal 和 WingedAnimal 两种动物的行为

~~多重继承~~

```
~~Bat b;~~
~~b.feedMilk(); //母乳喂养~~
~~b.flap(); //振翅飞翔~~
```

12.3 多重继承与虚继承——多重继承

多重继承

```
class Bat: public Mammal, public WingedAnimal { };
```

调用基类构造函数

```
Bat::Bat() {} //隐式调用Mammal和WingedAnimal的默认构造函数
Bat::Bat() :Mammal(), WingedAnimal() {} //显式调用基类的默认构造函数
```

说明

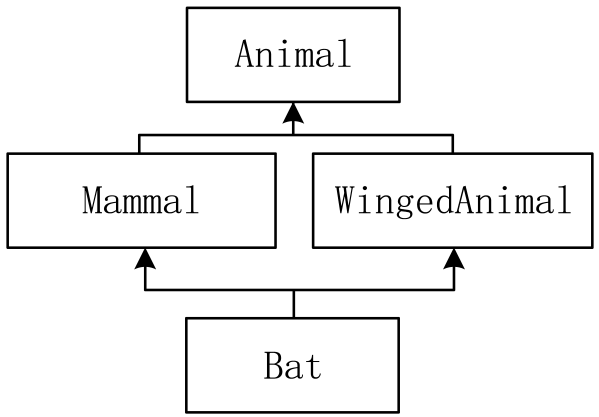
- 多重继承的派生类对象的构造函数只能初始化其直接基类成员
- 构造的顺序与派生列表中基类出现的先后顺序一致
- 调用基类的构造函数有隐式和显式两种方式
- 成员析构的顺序与构造的顺序相反

12.3 多重继承与虚继承——虚继承

我们将 Mammal 和 WingedAnimal 进一步抽象，设计一个公共基类 Animal：

加入公共基类 Animal

```
class Animal {
protected:
    int m_age;
public:
    Animal(int n = 0) :m_age(n) {}
    virtual void eat() {}
};
class WingedAnimal: public Animal{
public:
    virtual void feedMilk() {}
};
class Mammal: public Animal{
public:
    virtual void flap() {}
};
class Bat: public Mammal, public WingedAnimal { };
```



死亡钻石

菱形继承关系造成的二义性问题

12.3 多重继承与虚继承——虚继承

当存在如下调用的时候，会产生二义性问题：

加入公共基类 Animal

```
Bat b;
b.eat(); //错误：二义性访问
Animal a = b; //错误：类型无法转换
```

12.3 多重继承与虚继承——虚继承

C++ 通过**虚继承（virtual inheritance）**的机制来解决上述问题：

加入公共基类 Animal

```
class WingedAnimal: virtual public Animal {/*...*/};
class Mammal : virtual public Animal {/*...*/};
```

说明

- 通过在派生列表中添加关键字virtual来指定虚基类
- 不论该虚基类在继承体系中出现多少次，在派生类中只包含唯一一份共享的虚基类成员

12.3 多重继承与虚继承——虚继承

虚继承的基类由最底层的派生类进行初始化：

虚继承对象的构造

```
class Bat : public Mammal, public WingedAnimal {
public:
    Bat() :Animal(1), Mammal(), WingedAnimal(){}
};
```

说明

此处初始化顺序如下：

- Bat 类的构造函数提供的初始化列表初始化 Animal 成员
- 构造 Mammal 成员
- 构造 WingedAnimal 成员

12.4 嵌套类

嵌套类

在一个类的内部定义的类

说明

- 嵌套类是独立的类，与外层类在语法上没有关联
- 嵌套类和外层类的访问控制遵循普通类之间的访问控制原则

12.4 嵌套类——二维数组类

任务

实现一个二维数组类，该类可以像普通二维数组那样支持两个下标操作：

```
int arr[2][2];
arr[0][0] = 1;
```

存在的难点

C++ 仅支持一维下标操作符的重载：

```
class Array2D{
    /*...*/
    int operator [][](/*...*/); //错误：C++没有运算符[][]
};
```

12.4 嵌套类——二维数组类

任务

实现一个二维数组类，该类可以像普通二维数组那样支持两个下标操作：

```
int arr[2][2];
arr[0][0] = 1;
```

存在的难点

C++ 仅支持一维下标操作符的重载：

```
class Array2D{
    /*...*/
    int operator [][](/*...*/); //错误：C++没有运算符[][]
};
```

分析

arr[0][0] 等价于 (arr[0])[0]

12.4 嵌套类——二维数组类

代码清单 12.1 二维数组类

```
template<typename T>
class Array2D {
private:
    class Array1D {
        ... //下页内容
    };
    size_t m_size; //第一维长度
    Array1D *m_arr; //元素类型为Array1D
    public:
    Array2D(size_t s1, size_t s2) :m_size(s1), m_arr(new Array1D[s1]) {
        for (int i = 0; i<m_size; i++) {
            m_arr[i].m_size = s2;
            m_arr[i].m_arr = new T[s2];
        }
    }
    ~Array2D() { delete[] m_arr; }
    Array1D & operator[](int idx) { return m_arr[idx]; }
    size_t size() { return m_size; }
};
```

12.4 嵌套类——二维数组类

代码清单 12.1 二维数组类

```
...//上页内容
class Array1D {
    friend class Array2D; //声明为友元类
public:
    ~Array1D() { delete[] m_arr; }
    T & operator[](int idx) { return m_arr[idx]; }
private:
    size_t m_size = 0; //第二维长度
    T *m_arr = nullptr;
};
...//上页内容
```

使用二维数组

```
Array2D<int> arr(2, 2);
arr[0][0] = 1;
```

说明

- 使用嵌套类将 Array1D 的底层实现隐藏

12.4 嵌套类——二维数组类

9.3.5 节中我们实现的计算器程序有以下的代码：

计算器

```
/*...*/
unique_ptr<Operator> oo;
if (o == '+')
    oo = make_unique<Plus>();
else if (o == '-')
    oo = make_unique<Minus>();
/*...*/
```

思考

如果我们想要为计算器添加新的运算符，我们要怎么做?

12.4 嵌套类——二维数组类

为了有利程序扩展，我们希望根据运算符的名字来自动创建相应运算符类对象。首先我们需要实现一个**类注册机制**，我们将用如下数据结构来实现：

类注册机制核心数据结构

```
map<char, function<unique_ptr<Operator>()>>
  ms_operator;
```

类注册机制

保存类名（字符串）和类实例获取方法的映射关系，使程序能够根据名称得到类的实例。

说明

- map 对象的关键字为 char 类型，用于根据字符调用对应的 function 对象
- function 对象将返回一个指向 Operator 对象的 unique_ptr，用于生成对应运算符对象

12.4 嵌套类——二维数组类

下面代码将自动注册 Plus 类：

类注册机制核心数据结构

```
ms_operator.emplace('+',[]() { return make_unique<
  Plus>(); });
```

说明

- emplace 函数用于向 map 插入一组 char 到 Operator 对象生成器的映射
- lambda 表达式用于初始化 function 对象

之后即可根据用算符名字自动创建该运算符类对象：

类注册机制核心数据结构

```
ms_operator.emplace(' +' ,[]() { return make_unique<
  Plus>(); });
```

根据名字自动创建运算符

```
unique_ptr<Operator> oo = ms_operator[' +' ]();
```

说明

- 下标运算符调用 function 对象
- function 对象通过 lambda 表达式调用 make_unique
- make_unique 将返回对应下标运算符接收的字符所对应的 Operator 的 unique_ptr

12.4 嵌套类——二维数组类

为了方便用户注册，我们实现一个对象工厂（object factory）：

代码清单 12.2 Operator 类对象工厂 1

```
class Factory{
public:
template<typename T>
struct RegisterClass {
RegisterClass(char opr) {
Factory::ms_operator.emplace(opr,[]{return
  make_unique<T>();});
}//在Factory静态成员ms_operator作用域范围内，可以直接
  访问
};
static unique_ptr<Operator> create(char opr) {
    auto it = ms_operator.find(opr);
    if (it != ms_operator.end())
        return it->second(); //调用与opr相关联的lambda
          表达式
}
private://静态成员
static map<char, function<unique_ptr<Operator>()>>
  ms_operator;
};
```

说明

- 工厂通过嵌套类 RegisterClass 的构造函数实现对其静态成员 ms_operator 的类型注册功能
- it->second() 调用与 opr 相关的 lambda 表达式，返回对应的对象。

12.4 嵌套类——二维数组类

下面是用于注册的宏和静态成员的初始化：

代码清单 12.2 Operator 类对象工厂 2

```
#define REGISTRAR(T, Key) Factory::RegisterClass<T>
  reg_##T(Key);
map<char, function<unique_ptr<Operator>()>> Factory
  ::ms_operator;
```

示例：注册 Plus 类

```
REGISTRAR(Plus, '+');
```

等价于：

```
Factory::RegisterClass<Plus> reg_Plus('+');
```

说明

- 通过宏 REGISTRAER 创建嵌套类 RegisterClass 的全局对象同时完成注册
- ## 用来连接两个语言符号，产生一个对象名
- lambda 表达式用于初始化 function 对象

12.4 嵌套类——二维数组类

其他运算符类注册方式类似：

代码清单 12.2 Operator 类对象工厂 2

```
REGISTRAR(Minus, '-');
REGISTRAR(Multiply, '*');
REGISTRAR(Divide, '/');
REGISTRAR(Equal, '=');
```

说明

- 通过宏 REGISTRAER 创建嵌套类 RegisterClass 的全局对象同时完成注册
- ## 用来连接两个语言符号，产生一个对象名
- lambda 表达式用于初始化 function 对象

修改后 Calculator 类的 doIt 函数

```
double Calculator::doIt(const string & exp) {
    for (auto it = exp.begin(); it != exp.end();) {
        if (isNum(it))
            m_num.push(readNum(it));
        else {
            auto oo=Factory::create(*it++); //产生一个
              当前运算符类对象
            while (oo->precedence() <= m_opr.top()->
              precedence()) {
                if (m_opr.top()->symbol() == '#')
                    break;
                calculate();
            }
            if (oo->symbol() != '=')
                m_opr.push(std::move(oo));
        }
    }
    double result = m_num.top();
    m_num.pop();
    return result;
}
```

说明

第 6 行代码调用 Factory::create 函数，用来构造相应类型的对象

问题

如果形参未注册，调用 create 将会返回什么?

思考

体会修改后代码和原始代码的区别以及改进之处

12.5 运行时类型识别

运行时类型识别

运行时类型识别（run-time type identification, RTTI）指的是通过基类的指针或引用来检查其指向的派生类型。

RTTI 提供如下两个运算符：

- `typeid`
- `dynamic_cast`

这两个运算符作用于基类的指针或引用，如果该类型：

- 含有虚函数，则返回基类指针或引用的动态类型
- 不含有虚函数，则返回该类型的静态类型

12.5 运行时类型识别——dynamic_cast 运算符

dynamic_cast 运算符用于基类和派生类之间的安全转换，其使用方式有：

dynamic_cast 用法

```
dynamic_cast<type*>(expr)
dynamic_cast<type&>(expr)
dynamic_cast<type&&>(expr)
```

说明

三种使用方式中，expr 依次必须为：

- 有效的指针
- 左值
- 右值

注意

成功转换的前提：

expr 的类型必须是 type 的基类、派生类或 type 本身

否则：

- 指针类型返回空指针
- 引用类型抛出 std::bad_cast 异常

12.5 运行时类型识别——dynamic_cast 运算符

dynamic_cast 运算符用于基类和派生类之间的安全转换，其使用方式有：

定义基类及其派生类

```
struct Base{
    virtual ~Base() {}
};
struct Derived:Base{
    void name() {}
};
```

说明

下方代码：

- 第一个转换中 b1 与基类对象绑定，转换失败
- 第二个转换中 b2 与派生类对象绑定，转换成功，执行 name 调用

指针类型的 dynamic_cast

```
Base *b1 = new Base, *b2 = new Derived;
if (Derived *d = dynamic_cast<Derived*>(b1))
    d->name(); //转换失败，d为nullptr，不会执行此调用
if (Derived *d = dynamic_cast<Derived*>(b2))
    d->name(); //转换成功，执行此调用
```

建议

把 dynamic_cast 操作放到条件定义里，避免出现使用未绑定指针的不安全操作

12.5 运行时类型识别——dynamic_cast 运算符

dynamic_cast 运算符用于基类和派生类之间的安全转换，其使用方式有：

引用类型的 dynamic_cast

```
try { //转换失败，抛出std::bad_cast异常
    Derived &d = dynamic_cast<Derived&>(*b1);
}catch(std::bad_cast){
    cout << "downcast failed" << endl;
}
```

说明

引用类型的 dynamic_cast 转换失败时将抛出 std::bad_cast 异常

12.5 运行时类型识别——typeid 运算符

关键字 `typeid` 用来查询一个类型的信息

typeid 使用格式

```
typeid(type)
typeid(expr)
```

说明

- 如果表达式类型是类类型且包含虚成员函数，那么需要在运行时计算并返回表达式的动态类型；
- 否则，typeid 运算将返回表达式的静态类型，在编译时获得。

12.5 运行时类型识别——typeid 运算符

typeid 操作符的返回结果是名为 type_info 的标准库类型对象的引用。其支持的操作如下：

typeid 使用格式

```
t1 == t2 \\如果两个type_info对象t1和t2类型相同，则返回真；否则返回假
t1 != t2 \\如果两个type_info对象t1和t2类型不同，则返回真；否则返回假
t.name() \\返回类型的C风格字符串，类型名字用系统相关的方法产生
t1.before(t2) \\返回一个bool值，表示t1类型是否出现在t2类型之前
```

注意

name 成员返回的类型名与程序中使用的类型名并不一定一致，具体由编译器的实现决定

12.5 运行时类型识别——typeid 运算符

typeid 常用于比较两个表达式的类型是否相同，或一个表达式的类型是否与指定类型一致：

typeid 比较

```
Derived *d = new Derived;
Base *b = d;
if (typeid(*d) == typeid(*b)) { /*检查d和p是否指向同
  一类型对象*/}
if (typeid(*b) == typeid(Derived)) {/*检查b是否指向
  Derived类型对象*/}
```

注意

此处 typeid 作用于 *b 而非 b。

12.6 union 类型

union 类型

一种特殊的复合数据类型，可以实现一个对象用作多种数据类型的作用

说明

- 可以包含多个共享同一段内存空间的数据成员
- 占用的空间大小取决于内存占用最大的数据成员类型
- 使用时只有一个数据成员处于激活状态

12.6 union 类型——定义 union 类型

union 类型定义方法

union 类型定义

```
union ID {
    char char_type;
    int int_type;
    long long llong_type;
};
```

说明

- 定义了名为 ID 的 union 类型
- ID 类对象可用于存储 char、int、long 类型数据
- ID 类型对象占用内存长度和 long 类型长度相同

12.6 union 类型——使用 union 类型

union 类型定义方法：

union 类型定义

```
ID wang = {' a' }; // 构造了一个ID类对象，为其第一个成
  员赋值
cout << wang.char_type << endl; //输出a
wang.int_type = 1001; // 激活使用第二个成员
cout << wang.int_type << endl; //输出1001
wang.llong_type = 20171001001; // 激活使用第三个成员
cout << wang.llong_type << endl; //输出20171001001
```

说明

- 使用花括号可对其第一个数据成员赋值
- 赋值后其他成员将处于未定义状态
- ID 类型对象占用内存长度和 long 类型长度相同

注意

class 的大多特征适用于 union：

- 其数据成员可用 private、public 和 protected 修饰，默认为 public
- 可以定义成员函数

但另一些特征 union 不具备：

- 不能包含引用类型的数据成员
- union 类不具备继承特性

12.6 union 类型——使用 union 类型

union 类型定义方法：

union 类型定义

```
ID wang = {' a' }; // 构造了一个ID类对象，为其第一个成
  员赋值
cout << wang.char_type << endl; //输出a
wang.int_type = 1001; // 激活使用第二个成员
cout << wang.int_type << endl; //输出1001
wang.llong_type = 20171001001; // 激活使用第三个成员
cout << wang.llong_type << endl; //输出20171001001
```

说明

- 使用花括号可对其第一个数据成员赋值
- 赋值后其他成员将处于未定义状态
- ID 类型对象占用内存长度和 long 类型长度相同

注意

另外：

- 如成员为类类型，状态变化时将调用其构造/析构函数
- 使用时必须清楚使用的是哪一个成员，错误使用会导致崩溃

12.7 标准库特殊工具——tuple 类型

tuple

tuple（元组）是固定大小的异类值集合，它是泛化的 pair

定义 tuple 对象

```
tuple<string, double, int,list<string>> book("title1
  ",58.99,2017,{"Mandy","Lisha","Rosieta"});
```

说明

tuple 对象必须直接初始化

make_tuple 函数

```
auto book2 = make_tuple(string("title2"), 68.99,
  2017, list<string>{"Mandy","Lisha","Rosieta"});
```

说明

make_tuple 使我们可以利用auto 推导类型而不必显式指定

12.7 标准库特殊工具——tuple 类型

访问 tuple 成员

```
auto item1= get<0>(book); //访问第一个元素
get<1>(book) = 48.99; //对第二个元素赋值
for (auto &i : get<3>(book)) //使用范围for访问第四个
  成员的所有元素
    cout << i << "␣";
```

说明

- 使用标准库函数模板 get 来获取 tuple 的成员
- 使用 get 时需提供一个整型常量表达式实参，如 get<0> 表示第一个成员

12.7 标准库特殊工具——tuple 类型

比较 tuple 对象

```
if(book < book2) {/*...*/}
if(book == book2) {/*...*/}
```

说明

两个 tuple 可比较的条件：

- 成员数量相等
- 对应成员可以比较

比较的规则为按字典顺序比较 tuple 中的值

12.7 标准库特殊工具——tuple 类型

使用 tie 函数

```
string title;
double price;
int year;
list<string> author;
auto book3 = tie(title, price, year, author); //创建
  一个tuple对象
```

说明

标准库 tie 函数可以将多个对象的左值引用组合为一个 tuple 对象

tuple 对象的解包操作

```
tie(title, price, year, author) = book;
cout << title << "␣" << price << "␣"<< year;
```

输出:

```
title1 48.99 2017
```

说明

- 第一条语句将 book 中的成员依次赋值给 tie 函数创建的 tuple 对象
- 由于 tie 函数创建的 tuple 包含 4 个对象的引用，所以有最终的输出结果

12.7 标准库特殊工具——bitset 类型

bitset

固定长度的二进制序列，可以方便地进行逻辑运算，及与字符串和整数相互转换。

定义 bitset

```
bitset<12> b(1002); //b的二进制序列[0011 1110 1010]
bitset<12> b2("110010"); //二进制序列为[0000 0011
  0010]
```

说明

- 模板参数为表示位长度的整型表达式
- bitset 可以由整型值初始化，整形将转换为 unsigned long long 再转化为位模式存储
- bieset 也可以由 0 和 1 组成的字符串常量初始化

高位舍弃

```
bitset<8> b1(1002); //舍弃高位，b1 的二进制序列[1110
  1010]
```

说明

- unsigned long long 位数如大于 bitset 长度，则高位会被舍弃
- 否则多余高位设为 0

12.7 标准库特殊工具——bitset 类型

bitset 的一些成员函数如下所示：

bitset 成员函数——统计

```
bitset<8> b3("01010101");
b3.any(); //是否至少有一位为1
b3.none(); //是否所有位全为0
b3.all(); //是否所有位全为1
b3.count(); //值为1的位的数量
b3.size(); //位集合的长度
```

bitset 成员函数——位操作

```
b3.flip(); //所有位取反（0变1, 1变0）
b3.reset(); //所有位设为0
b3.set(); //所有位设为1
b3.set(i); //第i位设为1
b3.reset(i); //第i位设为0
b3.flip(i); //第i位取反
b3.test(i); //第i 位是否为1
b3[i] = 0; //第i位设为0
```

12.7 标准库特殊工具——bitset 类型

bitset 转化为整型值

```
cout << b.to_ullong() << endl; //输出1002
```

说明

bitset 的成员函数 to_ulong 和 to_ullong 可将位集合的二进制序列转化成整型值

注意

待转换的位集合的长度不能大于 long 或 long long 的位数

12.7 标准库特殊工具——日期和时间

C++ 提供 chrono 库对时间和日期进行操作，其包含三种时钟类：

- `system_clock`
- `steady_clock`
- `high_resolution_clock`

其中 system_clock 和 high_resolution_clock 是**实时时钟（real time clock）**，会随如夏令时的时间调整而改变，而 steady_clock 是**单调时钟（monotonic clock）**，不随外界时间调整而改变。

12.7 标准库特殊工具——日期和时间

输出时间点

```
using namespace chrono;
time_t tt = system_clock::to_time_t(system_clock::
  now());
cout << put_time(gmtime(&tt), "%F␣%T" ) << endl;
```

在我们测试中，上述代码输出为：
2017-12-09 20:15:08

说明 1

- 成员函数 now 可获得数据成员的时间点 time_point
- high_resolution_clock 相比 system_clock 精度要高

说明 2

获取标准日历形式输出：

- 通过 to_time_t 将 time_point 转换为 time_t 类型，然后通过 gmtime 转换为日历时间
- 再借助 iomanip 库中的 put_time 函数将日历时间转换成各种时间书写形式

12.7 标准库特殊工具——日期和时间

输出时间间隔

```
auto start = steady_clock::now();
doSomething(); // 执行某种算法
auto end = steady_clock::now();
auto interval = duration_cast<milliseconds>(end -
  start);
cout << interval.count() << endl;
```

说明

- 两个 time_point 类型相减得到 duration 类型用于表示时间段
- 转换函数 duration_cast 将时间段转换成特定的时间单位，如毫秒等
- 最后使用 duration 的成员函数 count 获取最终结果

注意

考虑到测量时间间隔需要稳定不变的时钟，代码中使用的时钟是 steady_clock。

本章结束