

## 第十二章 工具与技术

- ① 命名空间
  - 定义命名空间
  - 使用命名空间
- ② 异常处理
  - 抛出异常
  - 检测异常
  - 捕获异常
  - 使用标准库异常类
- ③ 多重继承与虚继承
  - 多重继承
  - 虚继承
- ④ 12.4 嵌套类
  - 二维数组类
  - 二维数组类
  - 通用计算器
- ⑤ 运行时类型识别

## 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。

# 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

# 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

# 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);
```

# 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

# 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

## 定义命名空间

### 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);
```

# 命名空间

## 定义命名空间

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

# 命名空间

## 定义命名空间

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。

# 命名空间

## 使用命名空间

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
```

# 命名空间

## 使用命名空间

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`

# 命名空间

## 使用命名空间

`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

# 命名空间

## 使用命名空间

`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。

# 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



# 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**

# 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。

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。

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

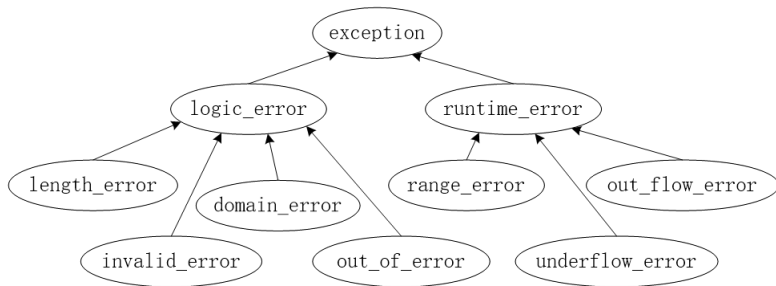
异常被谁捕获？

## 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。

# 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。

# 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 "Oops  
    !"; }  
};
```

## `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

# 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 "Oops  
    !"; }  
};
```

## `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。

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;  
}
```



# 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
```

# 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

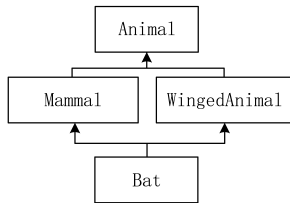
- 多重继承的派生类对象的构造函数只能初始化其**直接基类**成员
- 构造的顺序与**派生列表**中基类出现的先后顺序一致
- 调用基类的构造函数有**隐式**和**显式**两种方式
- 成员析构的顺序与构造的顺序相反

# 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

# 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
```

# 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

# 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

# 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

# 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 [] []  
};
```



# 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]

# 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; }
};
```

# 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
```

## 说明

- 使用嵌套类将 Array1D 的底层实现隐藏

## 使用二维数组

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

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？

# 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

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

之后即可根据运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

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。



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 REGISTRAR 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

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 REGISTRAR 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

# 嵌套类——二维数组类

## 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

# 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**。