四、高级主题

C++ 和标准库的一些附加特性，这些特性在特定情况下很有用。这些特性分成两类：一类对于求解大规模问题很有用；另一类适用于特殊问题而非通用问题。针对特殊问题的特性既有属于 C++的（19章），也有属于标准库的（17章）。

17章介绍四个具有特殊用处的标准库工具：bitset 类和三个新标准库工具（tuple、正则表达式和随机数），以及附加的 IO 库功能（格式控制、未格式化 IO 和随机访问）。

18章介绍异常处理、命名空间和多重继承，这些特性在设计大型程序时很有用。

即使编写一个简单的程序，也能从异常处理机制收益。第5章介绍了异常处理的基本知识。但是，对于大型程序设计，运行时错误处理显得更为重要和难于管理。18章额外介绍一些有用的异常处理工具，详细讨论异常是如何处理的，并展示如何定义和使用自己的异常类，指定一个特定函数不会抛出异常。

大型程序使用多个提供商的代码。为了避免名字冲突，可以在一个 namespace 中定义名字。使用一个来自标准库的名字，实际上都是在使用名为 std 的命名空间中的名字。18章介绍如何定义自己的命名空间。

18章最后介绍多重继承，对复杂的继承层次很有用。

19章介绍几种用于特定类别问题的特殊工具和技术，包括如何重定义内存分配机制；C++对运行时类型识别（run-time type identification，RTTI）的支持——允许我们在运行时才确定一个车表达式的实际类型；以及如何定义和使用指向类成员的指针。类成员指针不同于普通数据或函数指针。普通指针仅根据对象或函数的类型而变化，而类成员指针还必须反应成员所属的类。还介绍三种附加的聚合类型：联合（unions）、嵌套类（nested classes）和局部类（local classes）。最后介绍一组本质上不可移植的语言特性：volatile 修饰符（qualifier）、位域（bit-fields）以及链接指令（linkage directives）。

17. 标准库特殊设施

17.1 tuple 类型

tuple 是类似 pair（11.2.3节）的模板。

当希望将一些数据组合成单一对象，但又不想麻烦地定义一个新数据结构来表示这些数据时，tuple 是非常有用的。

> 可以将 tuple 看做一个快速而随意的数据结构。

17.1.1 定义和初始化tuple

访问 tuple 的成员

关系和相等运算符

17.1.2 使用tuple返回多个值

返回 tuple 的函数

使用函数返回 tuple

17.2 bitset 类型

4.8节介绍了将整数运算对象当作二进制位集合处理的一些内置运算符。标准库还定义了 bitset 类，使得位运算符的使用更为容易，并且能把处理超过最长整型类型大小的位集合。

17.2.1 定义和初始化bitset

用 unsigned 值初始化 bitset

从一个 string 初始化 bitset

17.2.2 bitset操作

提取 bitset 的值

bitste 的 IO 运算符

使用 bitset

17.3 正则表达式

正则表达式（regular expression）是一种描述字符序列的方法，是一种极其强大的计算工具。重点介绍如何使用 C++ 正则表达式（RE库），它是标准库的一部。RE库定义在头文件 regex 中，它包含多个组件。

表17.4：正则表达式库组件

regex	表示有一个正则表达式的类
regex_match	将字符序列与一个正则表达式匹配

regex_search	寻找第一个与正则表达式匹配的子序列
regex_replace	使用给定格式替换一个正则表达式
sregex_iterator	迭代器适配器，调用 regex_search 来遍历一个 string 中所有匹配的子串
smatch	容器类，保存在 string 中搜索的结果
ssub_match	string 中匹配的子表达式的结果

regex 类表示一个正则表达式。除了初始化和赋值外，regex 还支持其他一些操作。表17.6，regex支持的操作。

函数 regex_match 和 regex_search 确定一个给定字符序列与一个给定 regex 是否匹配。

表17.5：regex_search 和 regex_match 的参数	

17.3.1 使用正则表达式库

指定 regex 对象的选项

指定或使用正则表达式时的错误

正则表达式类和输入序列类型

17.3.2 匹配与Regex迭代器类型

使用 sregex_iterator

使用匹配数据

17.3.3 使用子表达式

子表达式用于数据验证

使用子匹配操作

17.3.4 使用regex_replace

只替换输入序列的一部分

用来控制匹配和格式的标志

使用格式标志

17.4 随机数

17.4.1 随机数引擎和分布

分布类型和引擎

比较随机数引擎和 rand 函数

引擎生成一个数值序列

设置随机数发生器种子

17.4.2 其他随机数分布

生成随机实数

使用分布的默认结果类型

生成非均匀分布的随机数

bernoulli_distribution 类

17.5 IO库再探

17.5.1 格式化输入与输出

很多操纵符改变格式状态

控制布尔值的格式

指定整型值的进制

在输出中指出进制

控制浮点数格式

指定打印精度

指定浮点数记数法

打印小数点

输出补白

控制输入格式

17.5.2 未格式化的输入/输出操作

单字节操作

将字符放回输入流

从输入操作返回的 int 值

多字节操作

确定读取了多少个字符

17.5.3 流随机访问

seek 和 tell 函数

只有一个标记

重定位标记

访问标记

读写同一个文件

小结

术语

18. 用于大型程序的工具

大规模编程对程序设计语言的要求更高。大规模应用程序的特殊要求包括：

- 在独立开发的子系统之间协同处理错误的能力
- 使用各种库（可能包含独立开发的库）进行协同开发的能力
- 对比较复杂的应用概念建模的能力

本章介绍的三种 C++ 语言特性正好能满足上述要求，它们是：异常处理、命名空间和多重继承。

18.1 异常处理

异常处理（exception handling）机制允许程序中独立开发的部分能够在运行时就出现的问题进行通行并做出相应处理。异常使得我们能够将问题的检测和解决过程分离开来。程序的一部分负责检测问题的出现，然后解决该问题的任务传递给程序的另一部分。检测环节无须知道问题处理模块的所有细节。

想要有效地使用异常处理，必须先了解抛出异常时发生了什么，捕获异常时发生了什么，以及用来传递错误的对象的意义。

18.1.1 抛出异常

C++中，通过抛出（throwing）这条表达式来引发（raised）一个异常。被抛出的表达式的类型以及当前的调用链共同决定了哪段处理代码（handler）将被用来处理该异常。被选中的处理代码是在调用链中与抛出对象类型匹配的最近的处理代码。其中，根据抛出对象的类型和内容，程序的异常抛出部分将会告知异常处理部分到底发生了什么。

当执行一个 throw 时，跟在 throw 后面的语句不再执行。相反，程序的控制权从 throw 转移到与之匹配的 catch 模块。该 catch 可能是同一个函数中的局部 catch，也可能位于直接或间接调用了发生异常的函数的另一个函数中。控制权从一处转移到另一处，这有两个重要含义：

- 沿着调用链的函数可能提早退出
- 一旦程序开始执行异常处理代码，则沿着调momo用链创建的对象将被销毁

throw 后面的语句将不再被执行，所以 throw 语句用法有点类似于 return 语句：它通常作为条件语句的一部分或者作为某个函数的最后（或唯一）一条语句。

栈展开

当抛出一个异常后，程序暂停当前函数的执行过程并立即开始寻找与异常匹配的 catch 子句。当 throw 出现在一个 try 语句块时，检查与该块关联的 catch 子句。如果找到了匹配的 catch，则使用该 catch 处理异常。如果没找到匹配的 catch 且该 try 语句嵌套在其他 try 块中，则继续检查与外层 try 匹配的 catch 子句。如果还是找不到匹配的 catch，则退出当前函数，在调用当前函数的外层函数中继续寻找，以此类推。

上述过程被称为栈展开（stack unwinding）过程。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的 catch 子句为止；或者也可能一直没找到匹配的 catch，则退出主函数后查找过程终止。

如果找到了一个匹配的 catch 子句，则程序进入该子句并执行其中代码。执行完该 catch 子句后，找到与 try 块关联的最后一个 catch 子句之后的点，并从这里继续执行。

如果没找到匹配的 catch 子句，程序将退出。因为异常被认为是妨碍程序正常执行的事件，所以一旦引发了某个异常，就不能不理它。找不到匹配的 catch 时，程序将调用标准库函数 teminate，

teminate 负责终止程序的执行过程。

> 一个异常如果没有被捕获，则它将终止当前的程序。

栈展开过程中对象被自动销毁

析构函数与异常

异常对象

> 抛出指针要求在任何对应的处理代码存在的地方，指针所指的对象都必须存在。

18.1.2 捕获异常

> 通常，如果 catch 接受的异常与某个继承体系有关，最好将该 catch 的参数定义成引用类型。

查找匹配的代码

> 如果在多个 catch 语句的类型之间存在着继承关系，则应该把继承链最低端的类（most derived type）放在前面，而将继承链最顶端的类放在后面。

重新抛出

有时一个单的的 catch 语句不能完整处理某个异常。执行了某些校正操作之后，当前的 catch 可能决定由调用链更上一层的函数接着处理异常。一条 catch 语句通过重新抛出（rethrowing）的操作将异常传递给另外 catch 语句。

捕获所有异常的处理代码

为了捕获所有异常，使用省略号作为异常声明，这样的处理代码称为捕获所有异常（catch-all）的处理代码，形如 catch（...）。一条捕获所有异常的语句可以与任意类型的异常匹配。

catch(...) 通常与重新抛出语句一起使用，其中 catch 执行当前局部能完成的工作，随后重新抛出异常：

```
1 void mainp() {
2     try {
3         // 这里的操作将引发并抛出一个异常
4     }
5     catch (...) {
```

```
        // 处理异常的某些特殊操作
        throw;
    }
}
```

> 如果 catch(...) 与其他几个 catch 语句一起出现，则 catch(...) 必须在最后。

18.1.3 函数try语句块与构造函数

> 处理构造函数初始值异常的唯一方法是将构造函数写成函数 try 语句块。

18.1.4 noexcept异常说明

对于用户和编译器来说，预先直到某个函数捕获抛出异常大有用处。首先，直到函数不会抛出异常有助于简化调用该函数的代码；其次如果编译器确认函数不会抛出异常，它就能执行某些特殊的优化操作，而这些优化操作并不适用于可能出错的代码。

可以通过 noexcept 说明指定某个函数不会抛出异常。

```
void recoup(int) noexcept;     //不会抛出异常
void alloc(int);               //可能抛出异常
```

违反异常说明

> 通常编译器不能也不必再编译时验证异常说明。

异常说明的实参

noexcept 运算符

异常说明与指针、虚函数和拷贝控制

18.1.5 异常类层次

标准库异常类（5.6.3节）构成了图18.1所示的继承体系

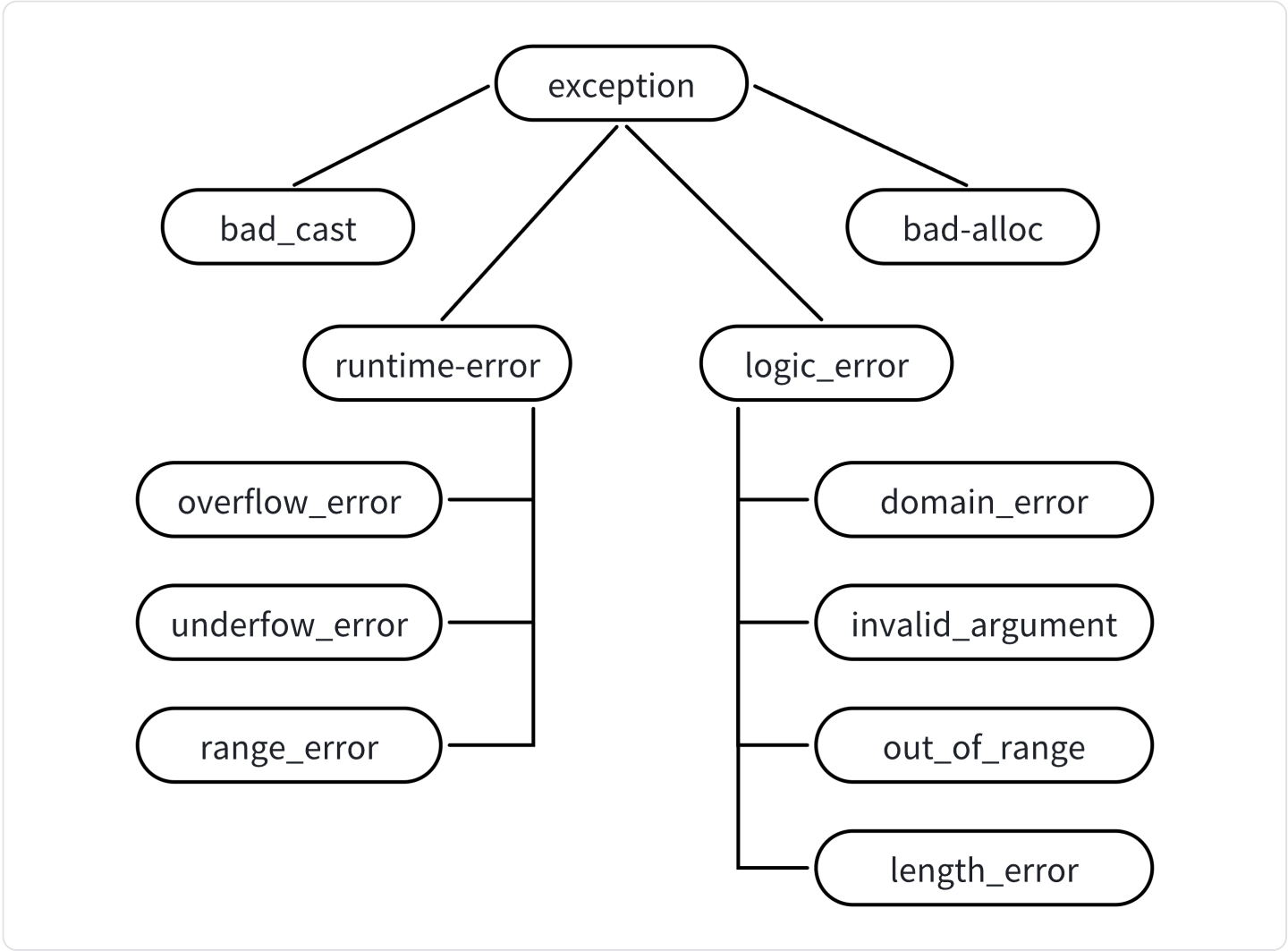

图18.1：标准 exception 类层次

类型 exception 仅仅定义了拷贝构造函数、拷贝赋值运算符、一个虚析构函数和一个名为 what 的虚成员。其中 what 函数返回一个 const char*，该指针指向一个以 unll 结尾的字符数组，并且确保不会抛出任何异常。

类 exception、bad_cast 和 bad_alloc 定义了默认构造函数。类 runtime_error 和 logic_error 没有默认构造函数，但有一个可以接受 C 风格字符串或标准库 string 类型实参的构造函数，这些实参负责提供关于错误的更多信息。这些类中， what 负责返回用于初始化异常对象的信息。因为 what 是虚函数，所以当捕获基类的引用时，对 what 函数的调用将执行与异常对象动态类型对应的版本。

书店应用程序的异常类

实际应用程序通常会自定义 exception （或者 exception 的标准库派生类）的派生类以扩展其继承体系。这些面向应用的异常类表示了与应用相关的异常条件。

构建一个书店应用程序，其中的类将复杂得多。复杂的一个方面就是如何处理异常。实际上，很可能需要建立一个自己的异常类体系，用它来表示与应用相关的各种问题。我们设计的异常类可能如下所示：

```
1  // 为某个书店应用程序设定的异常类
```

```
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
                            std::logic_error(s) { }
    isbn_mismatch(const std::string &s,
        const std::string &lhs, const std::string &rhs) :
        cosnt std::logic_error(s), left(lhs), right(rhs) { }
    const std::string left, right;
};
```

面向应用的异常类继承自标准异常类。和其他继承体系一样，异常类也可以看作按照层次关系组织的。层次越低，表示的异常情况越特殊。例如，异常类继承体系中最顶层的是 exception，它表示某处出错了，错误细节未做描述。

继承体系第二层将 exception 分为两个类别：运行时错误（只有在程序运行时才能检测到的错误）和逻辑错误（我们可以在程序代码中发现的错误）。

书店程序进一步细分上述异常类别。out_of_stock 的类表示在运行时可能发生的错误，比如某些顺序无法满足；isbn_mismatch 类表示 logic_error 的一个特例，程序可以通过比较对象的 isbn() 结果来阻止或处理这一错误。

使用自己的异常类型

使用自定义异常类的方式和使用标准异常类的方式完全一样。程序在某处抛出异常类型的对象，在另外的地方捕获并处理这些出现的问题。举例，可以为 Sales_data 类定义一个复合加法运算符，当检测到参与加法的两个 ISBN 编号不一致时抛出名为 isbn_mismatch 的异常：

```
// 如果参与加法的两个对象并非同一书籍，则抛出一个异常
Sales_data &
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong isbns", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

```

使用复合加法运算符的代码能检测到这一错误，进而输出一条相应的错误信息并继续完成其他任务：

```
// 使用之前设定的书店程序异常类
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) {
    try {
        sum = item1 + item2;
        // 此处使用 sum
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn(" << e.left
             << ") right isbn(" << e.right << ")" << endl;
    }
}
```

18.2 命名空间

大型程序使用多个独立开发的库，这些库会定义大量的全局名字，如类、函数和模板等。当程序用到多个供应商提供的库时，不可避免地会发生某些名字相互冲突的情况。多个库将名字放在全局命名空间中将引发命名空间污染（namespace pollution）。

传统上，程序员通过将其定义的全局实体名字设的很长来避免命名空间污染问题。

命名空间（namespace）为防止名字冲突提供了更加可控的机制。命名空间分割了全局命名空间，其中每个命名空间是一个作用域。通过在某个命名空间中定义库的名字，库的作者（以及用户）可以避免全局名字固有的限制。

18.2.1 命名空间定义

一个命名空间的定义包含两部分：关键字 namespace，和命名空间的名字。命名空间名字后面是{} 括起来的声明和定义。只要能出现在全局作用域中的声明就能置于命名空间内，主要包括：类、变量（及其初始化操作）、函数（及其定义）、模板和其他命名空间：

```
namespace cplusplus_primer {
    class Sales_data { /* ... */ };
    Sales_data operator+(const Sales_data&,
                         const Sales_data&);
    class Query { /* ... */ };
    class Query_base { /* ... */ };
} // 命名空间结束后无须分号，与块类似
```

和其他名字一样，命名空间买的名字也必须在它的作用域内保持唯一。命名空间既可以定义在全局作用域内，也可以定义在其他命名空间中，但是不能定义在函数或类的内部。

每个命名空间都是一个作用域

命名空间可以是不连续的

命名空间的定义可以不连续的特性使得我们可以将几个独立的接口和实现文件组成一个命名空间。此时，命名空间的组织方式类似于管理自定义类及函数的方式：

- 命名空间的一部分成员的作用是定义类，以及声明作为类接口的函数和对象，则这些成员应该置于头文件中，这些头文件被包含在使用了这些成员的文件中
- 命名空间成员的定义部分则置于另外的源文件中

定义本书的命名空间

定义命名空间成员

模板特例化

全局命名空间

全局作用域中定义的名字（即在所有类、函数及命名空间之外定义的名字）也就是定义在全局命名空间中。全局命名空间以隐式的方式声明，并且在所有程序中都存在。全局作用域中定义的名字被隐式添加到全局命名空间中。

作用域运算符同样可以用于全局作用域的成员，因为全局作用域是隐式的，所以它没有名字。

```
1  ::member_name
```

表示全局命名空间的一个成员。

嵌套的命名空间

内联命名空间

和嵌套命名空间不同，内联命名空间（inline namespace）中的名字可以被外层命名空间直接使用。无须在内联命名空间的名字前添加表示该命名空间的前缀，通过外层命名空间的名字就可以直接访问它。

```
inline namespace FifthEd {
    // 该命名空间表示本书第 5 版的代码
}
namespace FifthEd {
    class Query_base { /* ... */ };
    // 其他与 Query 有关的声明
}
```

关键字 inline 必须出现在命名空间第一次定义的地方，后续再打开命名看什么的时候可以写 inline，也可以不写。

当程序的代码在一次发布和另一次发布之间发生了改变时，常常用到内联命名空间。例如，可以把本书当前版本的所有代码都放在一个内联命名空间中，而之前版本的代码都放在一个非内联命名空间中：

```
namespace FourthEd {
    class Item_base { /* ... */ };
    class Query_base { /* ... */ };
    // 本书第4版用到的其他代码
}
```

命名空间 cplusplus_primer 将同时使用这两个命名空间。例如，假定每个命名空间都定义在同名的头文件中，则可以把命名空间 cplusplus_primer 定义成如下形式：

```
namespace cplusplus_primer {
#include "FifthEd.h"
#include "FourthEd.h"
}
```

FifthEd 是内联的，所以 cplusplus_primer:: 的代码可以直接获得 FifthEd 的成员。如果想使用早期的代码，则必须像其他嵌套的命名空间一样加上完整的外层命名空间名字，如 cplusplus_primer::FourthEd::Query_base。

未命名的命名空间

未命名的名字空间（unnamed namespace）是指关键字 namespace 后紧跟 {} 括起来的声明语句。未命名的名字空间中定义的变量拥有静态生命周期：它们在第一次使用前创建，并且直到程序结束才销毁。

> 和其他命名空间不同，unnamed namespace 仅在特定的文件内部有效，其作用范围不会横跨多个不同的文件。

18.2.2 使用命名空间成员

namespace_name::member_name 这样使用命名空间的成员非常繁琐。还有其他方法，如命名空间别名和 using 指示等。

命名空间的别名

```
1 namespace cplusplus_primer { ... }
2 namespace primer = cplusplus_primer;
```

> 一个命名空间可以有好几个同义词或别名，所有别名都与命名空间原来的名字等价。

using 声明：扼要概述

using 指示

using 指示与作用域

using 指示示例

头文件与 using 声明或指示

头文件如果在其顶层作用域中含有 using 指示或 using 声明，则会将名字注入到所有包含了该头文件的文件中。通常，头文件应该只负责定义接口部分的名字，而不定义实现部分的名字。因此，头文件最多只能在它的函数或命名空间内使用 using 指示或 using 声明（3.1节）。

> 避免 using 指示
>
> using指示一次性注入某个命名空间的所有名字。如果程序使用了多个不同的库，而这些库中的名字通过 using 指示变得可见，则全局命名空间污染问题将重现。
>
> 而且，当引入库的新版本后，正在工作的程序很可能会编译失败。如果新版本引入了一个与应用程序正在使用的名字冲突的名字，就会出现这个问题。
>
> 另一个风险是由 using 指示引发的二义性错误只有在使用了冲突命名的地方才能被发现。这种延后检测意味着可能在特定库引入了很久之后才爆发冲突。直到程序开始使用该库的新部分后，之前未被检测到的错误才会出现。
>
> 相对使用 using 指示，在程序中对命名空间的每个成员分别使用 using 声明效果更好，这样可以减少注入到命名空间中的名字数量。using 声明引起的二义性问题在声明处就能发现，无需等到使用名字的地方，这对检测并修改错误大有益处。
>
> using 指示也并非一无是处，例如在命名空间本身的实现文件中就可以使用 using 指示。

18.2.3 类、命名空间与作用域

实参相关的查找与类类型形参

查找与 std::move 和 std::forward

友元声明与实参相关的查找

18.2.4 重载与命名空间

与实参相关的查找与重载

重载与 using 声明

重载与 using 指示

跨越多个 using 指示的重载

18.3 多重继承与虚继承

18.3.1 多重继承

多重继承的派生类从每个基类中继承状态

派生类构造函数初始化所有基类

继承的构造函数与多重继承

析构函数与多重继承

多重继承的派生类的拷贝与移动操作

18.3.2 类型转换与多个基类

基于指针类型或引用类型的查找

18.3.3 多重继承下的类作用域

18.3.4 虚继承

另一个 Panda 类

使用虚基类

支持向基类的常规类型转换

虚基类成员的可见性

18.3.5 构造函数与虚继承

虚继承的对象的构造方式

构造函数与析构函数的次序

小结

C++的某些特性特别适合处理超大规模问题，这些特性包括：异常处理、命名空间以及多重继承或虚继承。

异常处理使得我们可以将程序的错误检测部分与错误处理部分分隔开来。当程序抛出一个异常时，当前正在执行的函数暂时中止，开始查找最近邻的与异常匹配的 catch 语句。作为异常处理的一部分，如果查找 catch 语句的过程中退出了某些函数，则函数中定义的局部变量也随之销毁。

命名空间一种管理大规模复杂应用程序的机制，这些应用可能是由多个独立的供应商分别编写的代码组合而成。一个命名空间是一个作用域，可以在其中定义对象、类型、函数、模版以及其他命名空间。标准库定义在名为 std 的命名空间中。

多重继承非常简单：一个派生类可以从多个直接基类继承而来。在派生类对象中既包含派生类部分，也包含与每个基类对应的基类部分。实际上多重继承的细节非常复杂。特别是对多个基类的继承可能会引入新的名字冲突，并造成来自于基类部分的名字的二义性问题。

如果一个类是从多个基类直接继承而来的，那么有可能这些基类本身又共享了另一个基类。这种情况下，中间类可以选择使用虚继承，从而声明愿意与层次中虚继承同一基类的其他类共享虚基类。用该方法，后代派生类中将只有一个共享虚基类的副本。

术语

19. 特殊工具与技术

19.1 控制内存分配

19.1.1 重载new和delete

operator new 接口和 operator delete 接口

malloc 和 free 函数

19.1.2 定位new表达式

显式的析构函数调用

19.2 运行时类型识别

19.2.1 dynamic_cast运算符

指针类型的 dynamic_cast

引用类型的 dynamic_cast

19.2.2 typeid运算符

使用 typeid 运算符

19.2.3 使用RTTI

类的层次关系

类型敏感的相等运算符

虚 equal 函数

基类 equal 函数

19.2.4 type_info类

19.3 枚举类型

枚举成员

和类一样，枚举也定义新的类型

指定 enum 的大小

枚举类型的前置声明

形参匹配与枚举类型

19.4 类成员指针

19.4.1 数据成员指针

使用数据成员指针

返回数据成员指针的函数

19.4.2 成员函数指针

使用成员函数指针

使用成员指针的类型别名

成员指针函数表

19.4.3 将成员函数用作可调用对象

使用 function 生成一个可调用对象

使用 mem_fn 生成一个可调用对象

使用 bind 生成一个可调用对象

19.5 嵌套类

声明一个嵌套类

在外层类之外定义一个嵌套类

定义嵌套类的成员

嵌套类的静态成员定义

嵌套类作用域中的名字查找

嵌套类和外层类是相互独立的

19.6 union：一种节省空间的类

定义 union

使用 union 类型

匿名 union

含有类类型成员的 union

使用类管理 union 成员

管理判别式并销毁 string

管理需要拷贝控制的联合成员

19.7 局部类

局部类不能使用函数作用域中的变量

常规的访问保护规则对局部类同样适用

局部类中的名字查找

嵌套的局部类

19.8 固有的不可移植的特性

19.8.1 位域

使用位域

19.8.2 volatile 限定符

合成的拷贝对 volatile 对象无效

19.8.3 链接指示：extern "C"

声明一个非 C++ 的函数

链接指示与头文件

指向 extern "C" 函数的指针

链接指示对整个声明都有效

导出 C++ 函数到其他语言

重载函数与链接指示

小结

C++ 为解决某些特殊问题设置了一系列特殊的处理机制。

有的程序需要精确控制内存分配过程，它们可以通过在类的内部或全局作用域中自定义 operator new 和 operator delete 来实现这一目的。

有的程序需要在运行时直接获取对象的动态类型，运行时类型识别（RTTI）为这种程序提供了语言级别的支持。RTTI 只对定义了虚函数的类有效；对没有定义虚函数的类虽然也可以得到其类型信息，但只是静态类型。

定义指向类成员的指针时，在指针类型中包含了该指针所指成员所属类的类型信息。成员指针可以绑定到该类当中任意一个具有指定类型的成员上。当解引用成员指针时，必须提供获取成员所需的对象。

C++定义了另外几种聚集类型：

- 嵌套类，定义在其他类的作用域中，嵌套类通常作为外层类的实现类。
- union，是一种特殊的类，它可以定义几个数据成员但是在任意时刻只有一个成员有值，union 通常嵌套在其他类的内部。
- 局部类，定义在函数的内部，局部类的所有成员都必须定义在类内，局部类不能含有静态数据成员。

C++支持几种固有的不可移植的特性，其中位域和 volatile 使得程序更容易访问硬件；链接指示使得程序更容易访问用其他语言编写的代码。

术语