第5章 函 数

本章将探讨函数重载、函数调用解析、默认/可选实参、临时变量、引用参数、返回值以及 inline 函数。

C++中的函数与其他编程语言中的函数和例程类似。但是，C++函数还支持在某些语言中没有的许多特性，因此这里有必要探讨它们。

5.1 函数重载

正如 1.5 节中指出的，函数的签名由名称和参数表组成。C++中，函数的返回类型不属于函数签名。

我们已经看到，C++中允许重载函数的名称。前面说过，如果在给定的作用域范围内一个函数的名称具有多种含义，那么它就是被重载的。如果两个或者多个函数在某个给定的作用域内的名称相同但它们的签名不同，就称此函数名称被重载了。1.5 节中还说过，如果位于同一作用域中的两个函数具有相同的签名而返回类型不同，则会导致错误。

函数调用解析

如果在某个作用域范围内调用了某个重载函数，则 C++编译器会根据实参来决定应该调用函数的哪一个版本。为此，必须使实参的个数和类型与重载函数的签名完全匹配。为了找出对应的签名，*Type* 和 *Type&*参数必须匹配。

以下是编译器确定应该调用哪一个重载函数的步骤。

1. 如果存在一个完全匹配的函数，则调用此函数。
2. 否则，通过标准的类型提升转换(参见 19.6 节)来进行匹配。
3. 否则，通过转换构造函数或者转换运算符(参见 2.12 节)来进行匹配。
4. 否则，判断是否可以通过省略号(…)进行匹配(参见 5.11 节)。
5. 否则，编译器报错。

示例 5.1 中展示的类有 6 个成员函数，每一个都有不同的签名。需记住的是，每一个成员函数都具有一个额外的隐式参数：this。在参数表之后的关键字 const，能够保护主对象(指向 this 的对象)不随函数的行为而改变，因此也是函数签名的一部分。

示例 5.1 src/functions/function-call.cpp

```
[ . . . . ]

class SignatureDemo {
public:
    SignatureDemo(int val) : m_Val(val) {}
    void demo(int n)
        {cout << ++m_Val << "\tdemo(int)" << endl;}
```

```
    void demo(int n) const                                          1
        {cout << m_Val << "\tdemo(int) const" << endl;}
/*  void demo(const int& n)
      {cout << ++m_Val << "\tdemo(int&)" << endl;}  */              2
    void demo(short s)
        {cout << ++m_Val << "\tdemo(short)" << endl;}
    void demo(float f)
        {cout << ++m_Val << "\tdemo(float)" << endl;}
    void demo(float f) const
        {cout << m_Val << "\tdemo(float) const" << endl;}
    void demo(double d)
        {cout << ++m_Val << "\tdemo(double)" << endl;}
private:
    int m_Val;
};
```

1 对 const 重载。

2 与前一个函数冲突。

示例 5.2 中包含的客户代码测试了 SignatureDemo 中的那些重载函数。

示例 5.2 src/functions/function-call.cpp

```
[ . . . . ]

int main() {
    SignatureDemo sd(5);
    const SignatureDemo csd(17);
    sd.demo(2);
    csd.demo(2);                                                    1
    int i = 3;
    sd.demo(i);
    short s = 5;
    sd.demo(s);
    csd.demo(s);                                                    2
    sd.demo(2.3);                                                   3
    float f(4.5);
    sd.demo(f);
    csd.demo(f);
    //csd.demo(4.5);
    return 0;
}
```

1 调用 const 版本。

2 不能调用非 const short 版本，所以需要进行 int 类型提升，以调用 const int 版本。

3 这里为 double，不是 float。

其运行结果应与下面的类似。

```
6       demo(int)
17      demo(int) const
7       demo(int)
8       demo(short)
17      demo(int) const
```

```
9       demo(double)
10      demo(float)
17      demo(float) const
```

5.1.1 练习：重载函数

1. 体验示例 5.1。首先将第三个成员函数的注释符号删除并编译。
2. 将 main() 结尾处前面的那一行的注释符号删除：

   ```
   //  csd.demo(4.5);
   ```

   会发生什么？对出现的错误消息给出解释。
3. 当链编这个应用时，注意编译器对于未使用的参数给出的警告。将编译器提示的函数首部中未使用的参数名称全部删除(不要删除参数类型)，然后重新链编，观察编译器的输出。如果想告知编译器不是在故意使用某些参数，则这就是一种好的做法。15.2 节中可以看到能够更方便地利用这种技术。
4. 添加其他的函数调用以及其他版本的 demo() 函数，并解释结果。

5.2 可选实参

默认(可选)实参

函数参数可以有默认值，此时它们就是可选的。可选实参的默认值可以是常量表达式或者不涉及局部变量的表达式。

拥有默认实参的参数必须是参数表中最靠右(最后面)的参数。具有默认值的最靠右实参在函数调用时可以忽略，而此时这些参数就会使用默认值进行初始化。

从函数的角度来看，如果调用时缺少一个实参，那么此实参必须对应于参数表中的最后一个参数。如果调用时缺少两个实参，则它们必须正好是参数表中的最后两个参数(以此类推)。

因为可选实参指示符应用在函数的接口上，所以它处于函数声明中；如果函数声明存放在一个单独的头文件中，则可选实参指示符不会出现在函数定义中。带有可选实参的函数可以有多种调用方式。如果所有的函数实参都是可选的，则不用任何实参就可以调用这个函数。声明有 n 个可选实参的函数，可以看成是声明了 $n+1$ 个函数的简化版本，其中每一个函数都对应此函数的一种调用方式。

示例 5.3 中，Date 类的构造函数有三个参数，每一个都是可选的，其默认值都为 0。

示例 5.3 src/functions/date.h

```
[ . . . . ]
class Date {
public:
    Date(int d = 0, int m = 0, int y = 0);
    void display(bool eoln = true) const;
private:
    int m_Day, m_Month, m_Year;
};
[ . . . . ]
```

示例 5.4 中的构造函数定义看似普通，它们无须再指定可选实参。如果函数调用时任何实参的实际值都为 0，则它会被从当前日期中获取的一个有意义的值替换。

示例 5.4 src/functions/date.cpp

```
#include <QDate>
#include "date.h"
#include <iostream>

Date::Date(int d , int m , int y )
: m_Day(d), m_Month(m), m_Year(y) {

    static QDate currentDate = QDate::currentDate();          1

    if (m_Day == 0) m_Day = currentDate.day();
    if (m_Month == 0) m_Month = currentDate.month();
    if (m_Year == 0) m_Year = currentDate.year();
}


void Date::display(bool eoln) const {
    using namespace std;
    cout << m_Year << "/" << m_Month << '/' << m_Day;
    if (eoln)
        cout << endl;
}
```

1 此处使用 Qt 的 QDate 类的目的是取得当前日期。

示例 5.5 src/functions/date-test.cpp

```
#include "date.h"
#include <iostream>

int main() {
    using namespace std;
    Date d1;
    Date d2(15);
    Date d3(23, 8);
    Date d4(19, 11, 2003);

    d1.display(false);
    cout << '\t';
    d2.display();
    d3.display(false);
    cout << '\t';
    d4.display();
    return 0;
}
```

示例 5.5 表明，实际上是通过定义默认值重载了函数。函数的不同版本执行相同的代码，只不过是靠后的参数接收了不同的值。

如果在 2011 年 5 月 14 日运行这个程序，则其运行结果如下。

```
src/functions> qmake
src/functions> make
[ compiler linker messages ]
src/functions> ./functions
2011/5/14       2011/5/15
2011/8/23       2003/11/19
src/functions>
```

5.3 运算符重载

C++使用关键字 operator 为运算符赋予新的含义，比如运算符+, -, =, *和&。为运算符增加新的含义，是重载的一种特殊形式。运算符重载提供了一种更紧致的函数调用语法，可以在很大程度上提高代码的可读性(假设运算符都按照常规理解的意义来进行操作)。

可以重载 C++中几乎所有已存在的运算符。例如，假设希望定义一个名称为 Complex 的类来表示复数[①]。

为了指定如何用这些对象进行基本的算术运算，可以重载四则算术运算符。当然，也可以重载插入运算符<<，以提高输出语句的可读性。

示例 5.6 中展示了一个包含成员运算符和非成员运算符的类的定义。

示例 5.6 src/complex/complex.h

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

class Complex {
    // binary nonmember friend function declarations
    friend ostream& operator<<(ostream& out, const Complex& c);
    friend Complex operator-(const Complex& c1, const Complex & c2);
    friend Complex operator*(const Complex& c1, const Complex & c2);
    friend Complex operator/(const Complex& c1, const Complex & c2);

    public:
    Complex(double re = 0.0, double im = 0.0);                    1

    // binary member function operators
    Complex& operator+= (const Complex& c);
    Complex& operator-= (const Complex& c);


    Complex operator+(const Complex & c2);                        2

private:
    double m_Re, m_Im;
};
```

1 默认构造函数和转换构造函数。

2 这里应该像其他的非可变运算符一样，是一个非成员友元函数。

示例 5.6 中声明的运算符都是二元的(接收两个操作数)。对成员函数而言，却只有一个形式参数，因为第一个(左边的)操作数都是隐式的：*this。示例 5.7 中给出了这些成员运算符的定义。

---

① 复数的形式为：$a + bi$，其中 a 和 b 为实数，i 表示-1 的平方根。由于复数中存在 $b = 0$ 的情况，所以实数是复数的一个子集。

复数最初是为了描述类似下面这样的方程的解而引入的：

$x^2 - 6x + 25 = 0$

利用二次式，可以很容易地知道这个方程的根为 $3 + 4i$ 和 $3 - 4i$。

示例 5.7 src/complex/complex.cpp

```
[ . . . . ]

Complex& Complex::operator+=(const Complex& c) {
    m_Re += c.m_Re;
    m_Im += c.m_Im;
    return *this;
}

Complex Complex::operator+(const Complex& c2) {
    return Complex(m_Re + c2.m_Re, m_Im + c2.m_Im);
}

Complex& Complex::operator-=(const Complex& c) {
    m_Re -= c.m_Re;
    m_Im -= c.m_Im;
    return *this;
}
```

示例 5.8 中给出了非成员友元函数的定义，定义它们时就像常规的全局函数那样。

示例 5.8 src/complex/complex.cpp

```
[ . . . . ]


ostream& operator<<(ostream& out, const Complex& c) {
    out << '(' << c.m_Re << ',' << c.m_Im << ')' ;
    return out;
}

Complex operator-(const Complex& c1, const Complex& c2) {
    return Complex(c1.m_Re - c2.m_Re, c1.m_Im - c2.m_Im);
}
```

前面已经讲解了定义 C++代码中四则代数运算的数学规则，它们的细节被封装并隐藏了，这样客户代码就不必处理它们。示例 5.9 中给出的客户代码演示并测试了这个 Complex 类。

示例 5.9 src/complex/complex-test.cpp

```
#include "complex.h"
#include <iostream>

int main() {
    using namespace std;
    Complex c1(3.4, 5.6);
    Complex c2(7.8, 1.2);

    cout << c1 << " + " << c2 << " = " << c1 + c2 << endl;
    cout << c1 << " - " << c2 << " = " << c1 - c2 << endl;
    Complex c3 = c1 * c2;
    cout << c1 << " * " << c2 << " = " << c3 << endl;
    cout << c3 << " / " << c2 << " = " << c3 / c2 << endl;
    cout << c3 << " / " << c1 << " = " << c3 / c1 << endl;

    return 0;
}
```

以下是示例 5.9 的输出。

```
(3.4,5.6) + (7.8,1.2) = (11.2,6.8)
(3.4,5.6) - (7.8,1.2) = (-4.4,4.4)
(3.4,5.6) * (7.8,1.2) = (19.8,47.76)
(19.8,47.76) / (7.8,1.2) = (3.4,5.6)
(19.8,47.76) / (3.4,5.6) = (7.8,1.2)
```

成员运算符与全局运算符的比较

前面已经看到，可以将运算符重载为成员函数或者全局函数。首先应注意，它们的主要差异是调用它们的方式。特别地，成员函数运算符要求有一个用作左操作数的对象，而全局函数允许对任何一个操作数进行某种类型转换。

示例 5.10 中给出了为什么 `Complex::operator+()` 更适合作为非成员函数的理由。

示例 5.10 src/complex/complex-conversions.cpp

```
#include "complex.h"

int main() {
    Complex c1 (4.5, 1.2);
    Complex c2 (3.6, 1.5);

    Complex c3 = c1 + c2;
    Complex c4 = c3 + 1.4;                    1
    Complex c5 = 8.0 - c4;                    2
    Complex c6 = 1.2 + c4;                    3
}
```

1 提升右操作数。

2 提升左操作数。

3 错误：左操作数没有针对成员运算符而提升。

运算符重载存在一些限制。只有内置的运算符才能被重载，因此无法对类似于“$”、“"”、“'”等未拥有运算符定义的符号重新进行定义。此外，尽管可以给内置运算符赋予新的定义，但是它们的结合性和优先级无法改变。

可以重载除下面这些运算符以外的全部一元或者二元内置运算符：

- 三元条件运算符 `testExpr ? valueIfTrue : valueIfFalse`
- 作用域解析运算符 `::`
- 成员选择运算符 `.` 和 `.*`

提示

有一种方法可以帮助记住哪一个运算符可以被重载。如果符号中的某个位置有一个点(.)，则它可能就不能被重载。

注意

重载逗号运算符是允许的，但是建议不要这样做，除非你是 C++行家。

19.1 节中给出了运算符以及它们的特性的一个完整列表。

注意

可以为内置运算符赋予一种新的含义，这样它就可以使用具有不同类型的操作数。但是，无法改变内置运算符的结合性和优先级。

5.3.1 练习：运算符重载

1. 继续开发 Fraction 类，添加重载的加、减、乘、除以及各种比较运算符。在每一种情况下，参数都应声明为 const Fraction&。编写客户代码，测试这些新的运算符。
2. 如果要使其真正有用，则 Fraction 对象应该能够与其他类型的数交互。扩展 Fraction 类的定义，使得上一题中的运算符也能够用于 int 类型和 double 类型。应能够清楚地对表达式进行求值，例如 frac + num。当 frac 为分数而 num 是 int 类型时，应该如何处理表达式 num + frac? 编写客户代码，测试这些新的函数。
3. 为 Complex 类添加算术运算符和比较运算符。编写客户代码，测试这个扩展的类。

5.4 按值传递参数

默认情况下，C++参数是按值传递的。当调用函数时，会生成每一个实参对象的一个临时(局部)副本并放入程序栈中。只有这些临时副本在函数内部进行操作，而调用块中的实参对象不会受这些操作的影响。当函数返回时，这些临时的栈变量就会被销毁。可以通过一种有效的方法来理解值参数：值参数仅仅是用函数调用时指定的对应实参对象进行初始化的局部变量。如示例 5.11 所演示的那样。

示例 5.11 src/functions/summit.cpp

```
#include <iostream>

int sumit(int num) {
    int sum = 0;
    for (; num ; --num)                    1
        sum += num;
    return sum;
}

int main() {
    using namespace std;
    int n = 10;
    cout << n  << endl;
    cout << sumit(n) << endl;
    cout << n << endl;                     2
    return 0;
}
```

1 参数值减少到 0。

2 看一看 sumit() 对 n 做了什么?

输出如下所示。

```
10
55
10
```

如果将指针传递给函数，那么这个指针的一个临时副本会被放入栈中。对这个指针的任何改变都不会对调用块中的指针产生影响。例如，临时指针可能会被赋予一个不同的值(参见示例 5.12)。

示例 5.12 src/functions/pointerparam.cpp

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

void messAround(int* ptr) {
    *ptr = 34;                                          1
    ptr = 0;                                            2
}


int main() {
    int n(12);                                                  3
    int* pn(&n);                                                4
    cout << "n = " << n << "\tpn = " << pn << endl;
    messAround(pn);                                             5
    cout << "n = " << n << "\tpn = " << pn << endl;
    return 0;
}
```

1 改变所指向的值。

2 改变由 ptr 保存的地址。更好的做法是不解引用它。

3 初始化一个 int 值。

4 初始化指向 n 的指针。

5 看一看 messAround()改变了什么?

输出如下所示。

```
n = 12  pn = 0xbffff524
n = 34  pn = 0xbffff524
```

输出结果中显示了指针 pn 的十六进制值以及 n 的十进制值，这样对函数动作产生的改变就不存在任何疑问了。

注意

总结如下:

- 当将对象按值传递给函数时，会产生该对象的一个副本。
- 这个副本被函数当成局部变量。
- 函数返回时，会销毁这个副本。

5.5 按引用传递参数

不应该将大型对象或者具有大量复制构造函数的对象进行按值传递，因为副本的创建会消耗大量不必要的机器周期和时间。在 C 语言中，可以通过指针来传递对象，以避免这类对象的复制。但是,使用指针的语法与使用对象的语法有所不同。此外，指针即使是偶尔的误用都可能引起数据的崩溃，从而导致难以发现和修复的运行时错误。在 C++(以及 C99)中，可以按引用传递参数，这种机制提供了与用指针传递参数相同的性能。对于对象而言，可以使用点运算符(.)来间接地访问成员。

引用参数也是参数，只不过它是其他对象的一个别名。为了将一个参数声明为引用参数，只须在类型名称与参数名称之间添加一个和符号(&)。

函数的引用参数使用函数调用时传递的实际实参进行初始化。正如其他所有的引用那样，这个实参必须是一个非 const 的左值。在函数中对非 const 引用参数进行的改变，将导致用来初始化此参数的实参对象的相应变化。这一特性可用于定义函数，通常情况下函数最多返回一个值，而在多个实参对象中的变化能够有效地使函数返回多个值。示例 5.13 中展示了如何针对整型变量使用引用参数。

示例 5.13 src/reference/swap.cpp

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

void swap(int& a, int& b) {
    int temp = a;
    cout << "Inside the swap() function:\n"
         << "address of a: " << &a
         << "\taddress of b: " << &b
         << "\naddress of temp: " << &temp << endl;
    a = b;
    b = temp;
}


int main() {
    int n1 = 25;
    int n2 = 38;
    int n3 = 71;
    int n4 = 82;
    cout << "Initial values:\n"
         << "address of n1: " << &n1
         << "\taddress of n2: " << &n2
         << "\nvalue of n1: " << n1
         << "\t\t\tvalue of n2: " << n2
         << "\naddress of n3: " << &n3
         << "\taddress of n4: " << &n4
         << "\nvalue of n3: " << n3
         << "\t\t\tvalue of n4: " << n4
         << "\nMaking the first call to swap()" << endl;
```

```
    swap(n1,n2);
    cout << "After the first call to swap():\n"
         << "address of n1: " << &n1
         << "\taddress of n2: " << &n2
         << "\nvalue of n1: " << n1
         << "\t\t\tvalue of n2: " << n2
         << "\nMaking the second call to swap()" << endl;
    swap(n3,n4);
    cout << "After the second call to swap():\n"
         << "address of n3: " << &n3
         << "\taddress of n4: " << &n4
         << "\nvalue of n3: " << n3
         << "\tvalue of n4: " << n4 << endl;
    return 0;
}
```

这个程序中包含有许多额外的输出语句，以帮助跟踪重要变量的地址。

```
Initial values:
address of n1: 0xbffff3b4        address of n2: 0xbffff3b0
value of n1: 25                  value of n2: 38
address of n3: 0xbffff3ac        address of n4: 0xbffff3a8
value of n3: 71                  value of n4: 82
```

开始时，程序栈可能与图 5.1 相似。随着程序的执行，输出可能是这样的：

```
Making the first call to swap()
Inside the swap() function:
address of a: 0xbffff3b4         address of b: 0xbffff3b0
address of temp: 0xbffff394
```

当将引用传递给函数时，压入栈的值是地址而不是右值。本质上，按引用传递非常类似于按指针传递。现在，程序栈可能与图 5.2 相似。

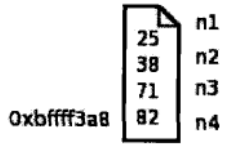

图 5.1 执行第一个 swap()之前的程序栈

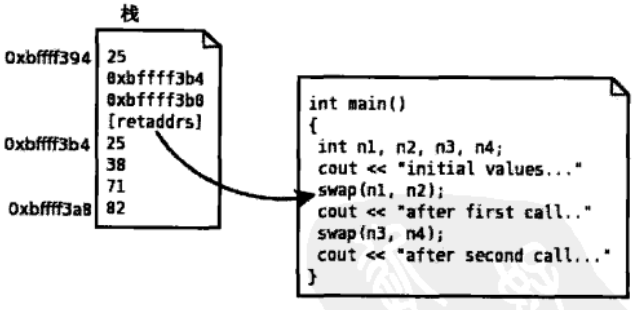

图 5.2 执行第一个 swap()时的程序栈

```
After the first call to swap():
address of n1: 0xbffff3b4        address of n2: 0xbffff3b0
value of n1: 38                  value of n2: 25
Making the second call to swap()
Inside the swap() function:
```

图 5.3 给出的是上面的行输出之后的栈状态。

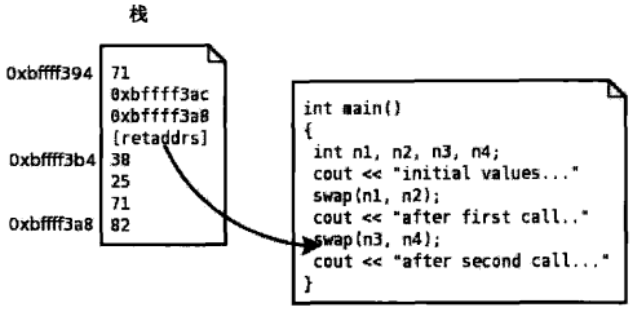

图 5.3 执行第二个 swap() 时的程序栈

```
address of a: 0xbffff3ac        address of b: 0xbffff3a8
address of temp: 0xbffff394
After the second call to swap():
address of n3: 0xbffff3ac       address of n4: 0xbffff3a8
value of n3: 82 value of n4: 71
```

第一次调用时，swap() 函数实际上只作用于 n1 和 n2，第二次调用时，作用的是 n3 和 n4。

按引用传递的语法提供了按指针传递的一种替代方式。本质上，按引用传递就是用指针实现的，不会复制值。二者的主要区别是：对于指针，必须解引用它；对于引用，可以用访问被引用实体同样的方式访问按引用传递的对象。

按指针传递还是按引用传递

如果可以选择，则通常更倾向于使用引用而不是指针，因为这样可以降低程序偶然发生内存崩溃的概率。只有在管理那些需要对指针进行操作的对象时(创建、销毁或者添加到一个托管容器中)，才会选择使用指针，并且，通常可以将这些例程封装为成员函数。

5.6 const 引用

将一个引用参数声明为 const，就是通知编译器，要确保函数不会试图改变这个对象。对大于指针的对象，const 引用要比值参数高效，因为不需要进行数据复制。示例 5.14 中包含了三个函数，每一个函数都以不同的方式接收参数。

示例 5.14 src/const/reference/constref.cpp

```
#include <QString>

class Person {

public:
    void setNameV( QString newName) {
        newName += " Smith";                    1
        m_Name = newName;
    }

    void setNameCR( const QString& newName) {
//      newName += " Python";                   2
```

```
        m_Name = newName;
    }
    void setNameR( QString& newName) {
        newName += " Dobbs";                                    3
        m_Name = newName;
    }
private:
    QString m_Name;
};

#include <QDebug>

int main() {
    Person p;
    QString name("Bob");
    p.setNameCR(name);                                          4
//  p.setNameR("Monty");                                        5
    p.setNameCR("Monty");                                       6
    p.setNameV("Connie");                                       7
    p.setNameR(name);                                           8
    qDebug() << name;
}
```

1 改变一个将被销毁的临时变量。

2 错误：无法转换为 const&。

3 改变原始的 QString。

4 没有创建临时变量。

5 错误：不能转换为 QString&。

6 将一个 char*变量转换为临时变量，并获得由 const 引用传递的值。

7 创建第一个临时 QString，将 char*转换成 QString。当按值传递时，创建第二个临时变量。

8 没有创建临时变量。

5.7 函数返回值

当执行完给它设计的任务时，有些函数会返回一个值。为返回对象临时准备的场所通常为一个寄存器(如果可以容纳的话)，但是有时也可以是栈上分配的一个对象。当执行 return 语句时，会初始化临时的返回对象，且其存在时间只为满足包含该函数调用的任何表达式的使用。对函数而言，这个对象副本通常是局部的，或者是在 return 语句的表达式中构造的一个对象。

5.8 从函数返回引用

有时候将函数设计成返回引用是非常有用的。例如，可以将多个操作“链”起来：

```
cout << thing1 << thing2 << thing3 ... ;
```

返回引用(尤其是*this)通常用来给成员函数提供左值行为。

采用引用参数，就可以通过将对象的别名指定成 const 来保护返回的引用。

示例 5.15 中展示了返回引用的本质。

示例 5.15 src/reference/maxi.cpp

```
#include <iostream>
using namespace std;
int& maxi(int& x, int& y) {

    return (x > y) ? x : y;
}

int main() {
    int a = 10, b = 20;
    maxi(a,b) = 5;                                    1
    maxi(a,b) += 6;                                   2
    ++maxi(a, b) ;                                    3
    cout << a << '\t' << b << endl;
    return 0;
}
```

1 给 b 赋值 5。

2 给 a 增加 6，a 现在的值为 16。

3 给 a 增加 1。

输出如下所示。

```
17    5
```

正如在 main() 函数中看到的那样，函数 maxi() 的引用返回值使表达式 maxi(a,b) 成为了一个可修改的左值。

警告

要小心，不要让函数返回一个指向临时(局部)对象的引用。只需稍加思考即可知晓这种限制的缘由：当函数返回时，所有的局部变量都销毁了。

```
int& max(int i,int j) {
    int retval = i > j ? i : j;

    return retval;
}
```

如果足够幸运，上述代码会导致一个编译器警告，但还不会使编译器认为是一个错误。以下是稍新的 C++版本给出的警告信息：

```
badmax.cpp:4: warning: reference to local variable 'retval' returned
```

示例 5.16 是返回引用的好处的一个实际例子，其中定义了针对矢量的一些常见运算符。

5.9 对 const 重载

const 改变了成员函数的签名，这意味着这种函数可以对 const 进行重载。示例 5.16 是自定义矢量类的一个例子，其成员函数就进行了 const 重载。

示例 5.16 src/const/overload/constoverload.h

```
#ifndef CONSTOVERLOAD_H
#define CONSTOVERLOAD_H

#include <iostream>

class Point3 {                                                    1
 public:
    friend std::ostream& operator<<(std::ostream& out, const Point3& v);
    Point3(double x = 0, double y = 0, double z = 0);
    double& operator[](int index);
    const double& operator[](int index) const;                    2
    Point3 operator+(const Point3& v) const;
    Point3 operator-(const Point3& v) const;
    Point3 operator*(double s) const;                             3
 private:
    static const int cm_Dim = 3;
    double m_Coord[cm_Dim];
};

#endif
```

1 一个(double 类型的)三维点。

2 对 const 重载。

3 标量乘法。

示例 5.17 中给出了这些运算符函数的定义。

示例 5.17 src/const/overload/constoverload.cpp

```
[ . . . . ]
const double& Point3::operator[](int index) const {
    if ((index >= 0) && (index < cm_Dim))
        return m_Coord[index];
    else
        return zero(index);
}

double& Point3::operator[](int index) {
    if ((index >= 0) && (index < cm_Dim))

        return m_Coord[index];
    else
       return zero(index);
 }

[ . . . . ]
```

两个函数体相同这一事实值得考虑。如果 index 位于范围之内，则每一个函数都返回 m_Coord[index]。二者有什么不同呢？需重点理解的是，这个运算符的非 const 版本的行为与示例 5.15 中的 maxi()函数非常类似。

5.9.1 练习：对 const 重载

1. 示例 5.18 中，编译器能够根据对象的 const 属性区别对 operator[]的 const 和非 const 版本的调用。

示例 5.18 src/const/overload/constoverload-client.cpp

```
#include "constoverload.h"
#include <iostream>

int main( ) {
    using namespace std;
    Point3 pt1(1.2, 3.4, 5.6);
    const Point3 pt2(7.8, 9.1, 6.4);
    double d ;
    d = pt2[2];                                          1
    cout << d << endl;
    d = pt1[0];                                          2
    cout << d << endl;
    d = pt1[3];                                          3
    cout << d << endl;
    pt1[2] = 8.7;                                        4
    cout << pt1 << endl;
    //  pt2[2] = 'd';
    cout << pt2 << endl;
    return 0;
}
```

1 _____

2 _____

3 _____

4 _____

每一个注释处分别调用的是哪一个运算符？

2. 为什么最后一个赋值被注释掉了？

5.10 inline 函数

为了避免函数调用带来的开销(例如，创建包含实参副本、引用参数地址以及返回地址的栈帧)，C++允许将函数声明为 inline(内联)的。这样的声明会要求编译器将所有对该函数的调用都以函数完全展开之后的代码来替换。例如：

```
inline int max(int a, int b){
  return a > b ? a : b ;
}

int main(){
   int temp = max(3,5);
    etc....
}
```

编译器会将 max 的代码展开成如下的样子：

```
int main() {
  int temp;
  {
    int a = 3;
    int b = 5;
    temp = a > b ? a : b;
  }
  etc.......
}
```

如果要重复地调用(例如，在一个大型循环中)，则 inline 函数可极大地提高性能。inline 函数的缺点是会使编译代码变得更大，在运行时会占用更多内存。对于需调用许多次的小型函数，将其声明成 inline 对内存的影响是微小的，而潜在的性能提升收益会很大。

inline 函数是否会提升程序的性能，这个问题没有简单的答案。这一方面取决于编译器的优化设置，另一方面也取决于程序的性质。程序是否会给处理器造成很重的负荷？是否会大量使用系统内存？是否花费大量的时间与慢速设备(例如，输入/输出设备)交互？这些问题的答案将决定是否应该采用 inline 函数，在此不再深入地探讨，而是将其作为一个高级课题。可以访问 Marshall Cline 的 FAQ Lite 网站，大体了解此问题的复杂性[①]。

inline 函数与#define 宏类似，但有一个重大差异：对#define 宏的替换过程是由预处理器处理的，预处理器本质上就是一个文本编辑器。对 inline 函数的替换过程是由编译器处理的，它会执行更智能的操作，进行正确的类型检查。下一节中将更详细地探讨这种差异。

inline 函数的一些规则

- inline 函数必须在被调用之前定义(仅仅声明它是不够的)。
- 在一个源代码模块中只能有一次 inline 定义。
- 如果类成员函数的定义出现在类定义之内，则成员函数就是隐含 inline 的。

如果函数太复杂，或者编译器的选项改变了，则编译器可能会忽略 inline 指令。大多数编译器会拒绝包含如下语句的 inline 函数：

- while, for, do...while 语句
- switch 语句
- 超过一定数量的代码行

如果编译器拒绝了 inline 函数，则会将其当成常规函数，并会生成常规的函数调用。

5.10.1 inline 函数与宏扩展的比较

宏扩展是一种通过如下的预处理器指令植入代码的机制：

```
#define MACRO_ID expr
```

这不同于 inline 函数。

宏扩展不对实参进行类型检查。本质上，它是一种编辑操作：在出现 MACRO_ID 的每一个地方都用 expr 替换。在宏中必须注意圆括号的使用，以避免优先级错误。但是，圆括号

① 参见 http://www.parashift.com/c++-faq-lite/inline-functions.html。

无法解决与宏有关的全部问题，见示例 5.19。由宏引起的错误可以导致奇怪的(不明晰的)编译器错误，甚至是更危险的无效结果。示例 5.19 演示了后一种情况。

示例 5.19 src/functions/inlinetst.cpp

```
// Inline functions vs macros

#include <iostream>
#define  BADABS(X)    (((X) < 0)? -(X) : X)
#define  BADSQR(X) (X * X)
#define  BADCUBE(X) (X) * (X) * (X)

using namespace std;

inline double square(double x) {
    return x * x ;
}

inline double cube(double x) {
    return x * x * x;
}

inline int absval(int n) {
    return (n >= 0) ? n : -n;
}

int main() {
    cout << "Comparing inline and #define\n" ;
    double  t = 30.0;
    int i = 8, j = 8, k = 8, n = 8;
    cout << "\nBADSQR(t + 8) = " << BADSQR(t + 8)
            << "\nsquare(t + 8) = " << square(t + 8)
            << "\nBADCUBE(++i) = " << BADCUBE(++i)
            << "\ni = " << i
            << "\ncube(++j) = " << cube(++j)
            << "\nj = " << j
            << "\nBADABS(++k) = " << BADABS(++k)
            << "\nk = " << k
            << "\nabsval(++n) = " << absval(++n)
            << "\nn = " << n << endl;
}
```

以下是输出结果。

```
BADSQR(t + 8) = 278
square(t + 8) = 1444
BADCUBE(++i) = 1100
 i = 11
cube(++j) = 729
j = 9
BADABS(++k) = 10
k = 10
absval(++n) = 9
n = 9
```

BADSQR(t+8)的结果是错误的，因为：

```
    BADSQR(t + 8)
=   (t + 8 * t + 8)          (preprocessor)
=   (30.0 + 8 * 30.0 + 8)    (compiler)
=   (30 + 240 + 8)           (runtime)
=   278
```

不过，更麻烦的是由 BADCUBE 和 BADABS 产生的错误，尽管它们都使用了足够多的圆括号来防止发生在 BADSQR 上的那一种错误。以下是 BADCUBE(++i)的运行情况：

```
    BADCUBE(++i)
=   ((++i) * (++i)) * (++i)    // left associativity
=   ((10) * (10)) * (11)
=   1100
```

一般而言，应该避免使用代码替换宏，大多数严肃的 C++程序员都将其视为危险的。预处理器宏主要用于下面几种情况。

1. 使用#ifndef/#define/#endif 将头文件包裹起来，以避免多次包含某个头文件。
2. 使用#ifdef/#else/#endif 对某些代码部分进行条件编译。
3. __FILE__宏和__LINE__宏用于调试并给出框架信息。

作为一个规则，一般应使用 inline 函数而不是宏来进行代码替换。这个规则的一个例外情况是使用 Qt 宏来向使用某些 Qt 类的程序插入代码。这也就容易理解为什么有些 C++专家怀疑 Qt 中宏的使用。

5.11 带变长实参表的函数

在 C 和 C++中，可以定义其参数表以省略号结尾的函数。省略号使调用者能够指定参数的数量以及类型。这种函数的一个例子来自于<stdio.h>：

```
int printf(char* formatStr, ...)
```

这种灵活机制使得可以进行下面这样的调用：

```
printf("Eschew Obfuscation!\n");
printf("%d days hath %s\n", 30, "September");
```

为了定义使用省略号的函数，需要包含 cstdarg 库，其中的一个宏集合用于访问 std 命名空间中实参表的各项。除省略号之外，参数表中必须至少还有另外一个参数。通常会用一个 va_list 类型的变量 ap(表示“实参指针”)来遍历未命名的实参表。宏

```
va_start(ap, p)
```

中的 p 是参数表中的最后一个命名参数，初始化 ap，使得它指向未命名实参中的第一个。宏

```
va_arg(ap, typename)
```

返回 ap 指向的那个实参，并使用 typename 来确定(用 sizeof)找到下一个实参应前进多少。宏

```
va_end(ap)
```

必须在已经处理完全部的未命名实参后才能调用。它清除未命名实参的栈，并确保在函数终止后程序依然会运行正常。

示例 5.20 中演示了如何使用这些特性。

示例 5.20　src/ellipsis/ellipsis.cpp

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

double mean(int n ...) {                                          1
    va_list ap;                                                   2
    double sum(0);
    int count(n);
    va_start(ap, n);                                              3
    for (int i = 0; i < count; ++i) {
        sum += va_arg(ap, double);
    }
    va_end(ap);                                                   4
    return sum / count;
}

int main() {
    cout << mean(4, 11.3, 22.5, 33.7, 44.9) << endl;
    cout << mean (5, 13.4, 22.5, 123.45, 421.33, 2525.353) << endl;
}
```

1 第一个参数是实参的数量。
2 依次指向每一个未命名实参。
3 现在，ap 执行第一个未命名实参。
4 返回之前清除栈。

5.12　练习：加密

1. 示例 5.16 中声明了 Point3 类的三个运算符但没有实现它们。为这三个运算符添加实现代码，并在客户代码中添加相应的测试代码。
2. 这个练习中，需复用来自于<cstdlib>(参见附录 B)的 random() 函数。
   这个函数生成 0～RAND_MAX(通常是 2 147 483 647)之间的一个伪随机整数。编写函数
   ```
   int myRand(int min, int max);
   ```
   使其返回 min 到 max - 1 之间的一个伪随机整数。
3. 编写函数
   ```
   QVector<int> randomPerm(int n, unsigned key);
   ```
   它使用 myRand() 函数(用 key 为种子)产生 0,…,n 的一个排列。
4. 加密与隐私正变得越来越重要。加密的一种思路是将一个字符串送入多个转换函数，经过这些转换函数转换的结果是一段可以更加安全地进行传送或保存的密文。此加密字符串的接收者可以使用转换函数的逆向过程来对密文进行逆向操作(即解密)，这样就能够获得原始字符串。加密字符串的发送者必须与接收者共享一些信息(即一个密钥)，以便解密字符串。下面的练习探索了一些简单的转换函数的设计。这些练习利

用了这样一个事实：random()返回的值序列完全由初始值(种子值)决定，因此这个序列是可重复的。

a. 编写函数

```
QString shift(const QString& text, unsigned key) ;
```

其中 shift()函数通过调用 srandom(),使用参数 key 设置随机函数的种子值。对于给定字符串 text 中的每一个字符 ch，加上下一个伪随机整数来获得一个移位的字符。然后，将这个移位的字符放入新字符串中对应的位置。当 text 中的全部字符都处理完之后，shift()返回这个新的字符串。

在将随机整数添加到用于处理字符的代码中时，必须进行“mod n”的加法运算，其中 n 是所使用的底层字符集中的字符数。对于这个练习，可以假定使用的是 ASCII 字符集，它有 128 个字符。

b. 下一个要编写的函数是

```
QString unshift(const QString& cryptext, unsigned key);
```

这个函数逆转上一个练习中描述的过程。

c. 编写代码，测试上面给出的 shift()函数和 unshift()函数。

d. 另外一种加密的方法(可以与上面描述的方法结合使用)是改变给定字符串中字符的顺序。编写函数

```
QString permute(const QString& text, unsigned key);
```

它使用 randomPerm()函数来生成原始字符串 text 的一个转换字符序列。

e. 编写函数

```
QString unpermute(const QString& scrtext, unsigned key);
```

它反转上面描述的 permute()函数的动作。

f. 编写代码，测试 permute()函数和 unpermute()函数。

g. 编写代码，测试对 shift()函数和 permute()函数使用同一个字符串的结果，再测试 unpermute()函数和 unshift()函数。

5. 实现封装前面练习中的函数的 Crypto 类。可以从图 5.4 中使用的 UML 框图开始。m_OpSequence 是一个 QString，它由代表 permute()函数和 shift()函数的字符'p'和's'组成。encrypt()函数将这两个函数用于给定的字符串，使其以在 m_OpSequence 字符串中出现的顺序排列。示例 5.21 中包含了测试这个类的一些代码。

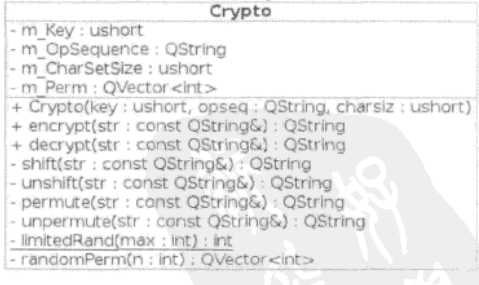

图 5.4 Crypto 的 UML 类框图

示例 5.21 src/functions/crypto-client.cpp

```
#include <QTextStream>
#include "crypto.h"

int main()  {
   QTextStream cout(stdout);
```

```
    QString str1 ("asdfghjkl;QWERTYUIOP{}|123456&*()_+zxcvnm,,, ./?"),
            str2;
    cout << "Original string: " << str1 << endl;
    cout << "length: " << str1.length() << endl;
    QString seqstr("pspsp");
    ushort key(12579);
    Crypto crypt(key, seqstr);
    str2 = crypt.encrypt(str1);
    cout << "Encrypted string: " << str2 << endl;
    cout << "Recovered string: " << crypt.decrypt(str2) << endl;
}
```

关于加密的主题将在 17.1.4 节中再次探讨，这一节中将介绍一个进行加密哈希操作的 Qt 类。

5.13 复习题

1. 函数声明与函数定义有什么不同?
2. 为什么默认实参指示符出现在函数声明中而不在其定义中?
3. 如果位于同一作用域中的两个函数具有相同的签名而返回类型不同，则会导致错误。请给出解释。
4. 如果要对 Fraction 对象重载算术运算符(+, -, *, /)，则成员函数与非成员全局运算符相比哪一个更可取? 给出理由。
5. 对于重载的左修饰运算符(比如-=和+=)，成员函数运算符与(非成员)全局运算符相比，哪一个更可取?
6. 解释按值传递与按引用传递的差异。在哪些情况下应使用其中的一个而不是另一个?
7. 解释预处理器宏与 inline 函数的差异。