

8 函数探幽

内容提要

- C++内联函数
- 引用变量
- 默认参数
- 函数重载
- 函数模板

1 C++内联函数

➤ 函数调用

- 存储该指令的内存地址，并将函数参数复制到堆栈
- 跳到标记函数起点的内存单元，执行函数代码（可能还需将返回值放入到寄存器中）
- 然后跳回到地址被保存的指令处

内联函数，C++为提高程序运行速度所做改进

➤ 在函数声明/定义前加上 `inline` ([P8.1 inline.cpp](#))

- 内联。编译器将使用相应的函数代码替换函数调用
- 内联代码无需跳到另一个位置处执行代码再跳回来。运行速度比常规函数稍快。但需要占用更多内存
 - 减少函数调用的开销
 - 保持函数的形式
- 不能递归
- 推荐选项，编译器不一定满足要求

```
1. #include <iostream>
2. // an inline function definition
3. inline double square(double x) { return x * x; }
4. int main()
5. {
6.     using namespace std;
7.     double a, b;
8.     double c = 13.0;
9.
10.    a = square(5.0);
11.    b = square(4.5 + 7.5); // can pass expressions
12.    cout << "a = " << a << ", b = " << b << "\n";
13.    cout << "c = " << c;
14.    cout << ", c squared = " << square(c++) << "\n";
15.    cout << "Now c = " << c << "\n";
16.    return 0;
17. }
```

2 引用变量

- 引用是已定义的变量的**别名**！
- 主要用途
 - 用作函数的形参（改变形参的值，参数值回传）
 - 通过将引用变量用作参数，函数将使用原始数据，而不是其副本
 - 除指针之外，引用也为函数处理大型结构提供了一种非常方便的途径

2.1 创建引用变量

[P8.2 firstref.cpp](#)

- rats和rodents完全一样
 - 别名，指向相同的内存
- 引用和取地址符
 - 函数参数中
 - 只有引用，没有取地址。
 - 和“=”关联
 - 引用一定在“=”左边
 - 取地址一般在“=”右边
- 某些特殊场合
 - 比较复杂的函数返回值（举比较复杂的例子）

```
1. #include <iostream>
2. int main(){
3.     using namespace std;
4.     int rats = 101;
5.     int & rodents = rats;    // rodents is a reference
6.     cout << "rats = " << rats;
7.     cout << ", rodents = " << rodents << endl;
8.     rodents++;
9.     cout << "rats = " << rats;
10.    cout << ", rodents = " << rodents << endl;
11.    // some implementations require type casting the
        following addresses to type unsigned
12.    cout << "rats address = " << &rats;
13.    cout << ", rodents address = " << &rodents << endl;
14.    // cin.get();
15.    return 0;
16. }
```

2.2 将引用用作函数参数

P8.4 swaps.cpp

➤ 参数传递方法

➤ 按值传递：函数使用调用实参的拷贝

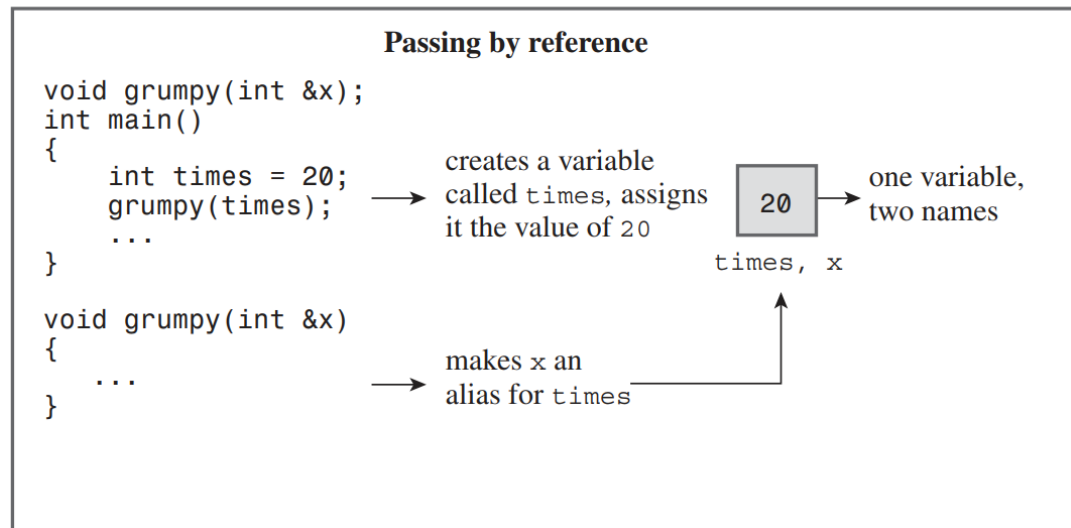
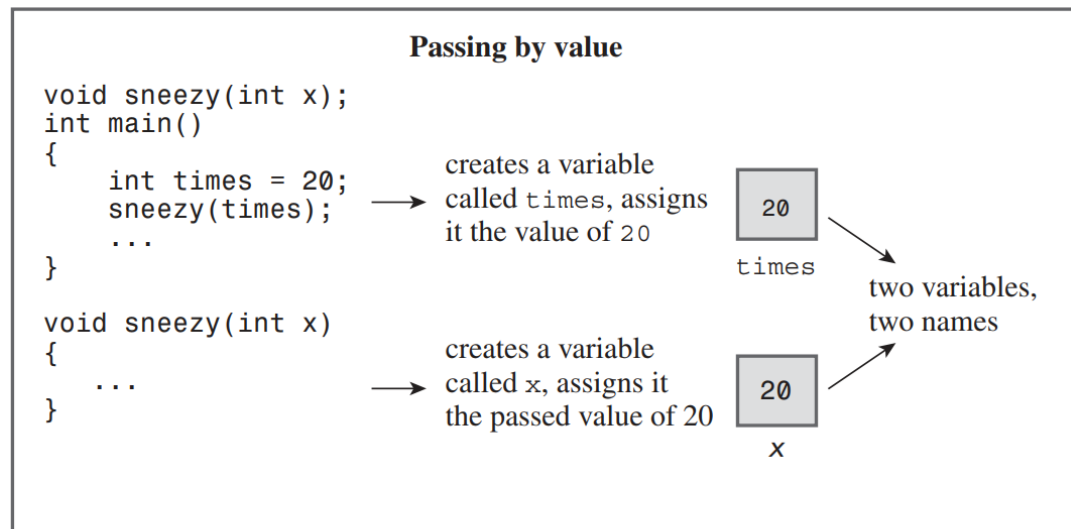
➤ 指针：间接传递

➤ 按引用传递：别名，就是调用实参本身

➤ 函数调用修改实参

➤ 指针方式

➤ 引用



2.3 引用的属性和特别之处

[P8.5 cubes.cpp](#)

- 引用有可能修改参数的值，防止修改参数
 - `const &`，尽可能使用
- 实参为常数/表达式（左值），形参必须用 `const &`
 - 否则出现编译错误
 - 原因：不用 `const` 意味着实参可被改变，而左值不可以被改变
- 因此
 - 尽量实参和形参类型一致！
 - 根据是否修改实参，决定是否用 `const`

2.4 将引用用于结构

[P8.6 strtref.cpp](#), 对比C的用法

- 目的：减少传递开销
- 返回引用（尽量不用）
 - 返回引用的函数实际上是被引用的变量的别名
 - 返回函数可以左值！
 - 返回引用时最重要的一点是，应避免返回函数终止时不再存在的内存单元引用

```
const free_throws &clone2(free_throws &ft)
{
    free_throws newguy; // first step to big error
    newguy = ft;         // copy info
    return newguy;       // return reference to copy
}
```

```
1. struct sysop{
2.     char name[26];
3.     char quote[64];
4.     int used;
5. };

6. const sysop & use(sysop & sysopref){
7.     sysopref.used++;
8.     return sysopref;
9. }

10. int main(){
11.     sysop looper = {"Rick", "I'm", 0};
12.     use(looper); // looper is type sysop
13.     sysop copycat;
14.     copycat = use(looper);
15.     cout << "use: " << use(looper).used << "\n";
16.     return 0;
17. }
```

2.5 将引用用于类对象

- `string`, `ostream`, `istream`, `ofstream`, `ifstream`
- 8.7, `version3`, 返回局部变量引用

2.6 对象，继承和引用

- 基类引用指向具体派生类，用于面向对象设计

2.7 何时使用引用参数

➤ 根本原因

- 函数调用时，参数可以回传
- 提高参数传递的效率

何时使用引用参数

➤ 参数值需要被修改

- 内置类型，指针
- 数组，指针或引用
- 结构，指针或引用
- 类对象，引用（也可以指针）

➤ 参数值不做修改

- 数据对象很小，按值
- 数组，使用指针
- 较大的数据结构，`const &`
- 类对象，使用`const &`

3 默认参数

[P8.9 left.cpp](#)

- 当函数调用中省略了实参时自动使用的一个值
- 从右往左添加默认值（可以多个默认值）

```
int harpo(int n, int m = 4, int j = 5);      // VALID
int chico(int n, int m = 6, int j);         // INVALID
int groucho(int k = 1, int m = 2, int n = 3); // VALID

beeps = harpo(2);      // same as harpo(2,4,5)
beeps = harpo(1, 8);   // same as harpo(1,8,5)
beeps = harpo(8, 7, 6); // no default arguments used
```

4 函数重载

- 函数多态（函数重载）能够使用多个同名的函数
 - 术语“多态”指的是有多种形式，因此函数多态允许函数可以有多种形式。
 - 术语“函数重载”指的是可以有多个同名的函数，因此对名称进行了重载。
 - 这两个术语指的是同一回事，但通常使用函数重载。

([P8.10 leftover.cpp](#))

- 通过函数重载来设计一系列函数：它们完成相同的工作，但使用不同的参数列表函数名相同
 - 参数不同(不是返回值)
 - 类型
 - 数目
 - 目的：类似的功能，类似的接口/调用方式。
- 参数列表-函数特征标

函数重载

- 函数重载的关键是函数的参数列表，也称为函数特征标(function signature)
 - 如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，变量名无关（编译器的角度）。

```
void print(const char *str, int width); // #1
void print(double d, int width);       // #2
void print(long l, int width);         // #3
void print(int i, int width);          // #4
void print(const char *str);           // #5

print("Pancakes", 15);                 // use #1
print("Syrup");                        // use #5
print(1999.0, 10);                     // use #2
print(1999, 12);                       // use #4
print(1999L, 15);                      // use #3
```

➤ 一些看起来彼此不同的特征标是不能共存的

- 编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标

```
double cube(double x);
```

```
double cube(double & x);
```

- 匹配函数时，区分const 和非const 变量

```
1. void dribble(char * bits);           // overloaded
2. void dribble (const char *cbits);    // overloaded
3. void dabble(char * bits);            // not overloaded
4. void driv1(const char * bits);       // not overloaded

5. const char p1[20] = "How's the weather?";
6. char p2[20] = "How's business?";
7. dribble(p1); // dribble(const char *);
8. dribble(p2); // dribble(char *);
9. dabble(p1);  // no match
10. dabble(p2); // dabble(char *);
11. driv1(p1);  // driv1(const char *);
12. driv1(p2);  // driv1(const char *);
```

互斥和重载引用参数

- 是特征标，而不是函数类型使得可以对函数进行重载

```
long gronk(int n, float m);    // same signatures,  
double gronk(int n, float m); // hence not allowed  
long gronk(int n, float m);    // different signatures,  
double gronk(float n, float m); // hence allowed
```

- 重载引用参数

```
void staff(double &rs);        // matches modifiable lvalue  
void staff(const double &rCs); // matches rvalue, const lvalue  
void stove(double &r1);        // matches modifiable lvalue  
void stove(const double &r2);  // matches const lvalue  
void stove(double &&r3);       // matches rvalue  
double x = 55.5;  
const double y = 32.0;  
stove(x);    // calls stove(double &)  
stove(y);    // calls stove(const double &)  
stove(x + y); // calls stove(double &&)
```

5 函数模板

[P8.11 funtemp.cpp](#)

- 数模板是通用的函数描述，即，使用泛型来定义函数
 - 其中的泛型可用具体的类型替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。
 - 由于模板允许以泛型（而不是具体类型）的方式编写程序，因此有时也被称为通用编程
 - 由于类型是用参数表示的，因此模板特性有时也被称为参数化类型(parameterized types)
- 通过泛型来定义函数
- 通过将类型作为参数传递给模板
- 例子：交换两个数的值
 - int
 - double
 - 更复杂的，如某个结构体/类

函数模板

[P8.11 funtemp.cpp](#)

```
template <class T>
void swap(T &a, T &b);
```

- typename和class等价
- 可以有具体参数
- 编译器会生成函数的具体版本（实例化）

```
1. / function template definition
2. template <typename T> // or class T
3. void Swap(T &a, T &b)
4. {
5.     T temp; // temp a variable of type T
6.     temp = a;
7.     a = b;
8.     b = temp;
9. }
```

5.1 重载的模板

[P8.12 twotemps.cpp](#)

- 可以像重载常规函数定义那样重载模板定义

5.2 模板的局限性

- 可能无法处理某些类型

- 如数组类型，赋值=

- `a = b;`

5.3 显示具体化

[P8.13 twoswap.cpp](#)

➤ 高级性能，不做要求

5.4 实例化和具体化

➤ Pass

示例

```
1. #include <iostream>
2. // an inline function definition
3. inline double square(double x)
4. {
5.     return x * x;
6. }
7.
8. int main()
9. {
10.     using namespace std;
11.     double a, b;
12.     double c = 13.0;
13.     a = square(5.0);
14.     b = square(4.5 + 7.5); // can pass expressions
15.     return 0;
16. }
```