

## directory

- 3.1 函数重载与缺省值
- 3.2 基础知识
- 3.3 类与对象
- 3.4 成员变量与成员函数
- 3.5 private和public
- 3.6 this指针
- 3.7 内联函数

## 3.1 函数重载与缺省值

### 函数重载

- why: 名一样而义不同
  - 同一任务，但输入**信息的类型**不同
    - `sum(int a, int b);`
    - `sum(double a, double b);`
  - 同一任务，函数**输入信息的存储形式**不同
    - `sort(const char* s);`
    - `sort(string str);`
  - 相似任务，在**抽象概念**层面一致
- def: 同一名称的函数，有两个以上不同的函数实现.
- 编译器将根据函数调用语句的实际参数来决定哪一个函数被调用
- 注意：多个同名的函数实现之间，必须保证至少有一个**函数参数的类型**有区别。
  - **返回值、参数名称等不能**作为区分标识。
  - 函数**参数的个数、类型**等至少有一个不一样

### 内置类型转换

- 如果函数调用语句的**实参**与函数定义中的**形参**数据类型不同，且两种数据类型在C++中可以进行自动类型转换（如int和float），则实参会被**转换为形参的类型**
- 如果形参类型为int，实参类型为float，则**向零取整**
- 自动类型转换也可以通过定义的类型转换运算符来完成

### 函数参数的缺省值

- 函数参数可以在定义时设置默认值（缺省值）
- 在调用该函数时，若不提供相应的实参，则编译自动将相应形参设置成缺省值
- 有缺省值的函数参数，必须是最后一个参数
- 如果有多个带缺省值的函数参数，则这些函数参数都只能在没有缺省值的参数后面出现
- 缺省值冲突问题

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

int fun(int a=1) { return a+1; }
```

```
float fun(float a) { return a; }
int fun(int a, int b) { return a+b; }

int main() {
    float a = 1.5;
    int b = 2;
    cout << fun(a, b) << endl; // 3
    cout << fun(fun(a, b)) << endl; // 4
    cout << fun(fun(a, b)) + fun(fun(a), b) << endl; // 7
    return 0;
}
```

## auto 关键字

- 由编译器根据上下文自动确定变量的类型

```
auto i = 3;      //i是int型变量
auto f = 4.0f;   //f是float型变量
auto a('c');     //a是char型变量
auto b = a;       //b是char型变量
auto *x = new auto(3); //x是int*
```

```
#include <iostream>
#include<typeinfo>
#include<cxxabi.h>
using namespace std;
void show(char* a, int b = 1) { cout << a << " " << b << endl;}

template<typename T>
char* get_type(const T& instance){
    return abi::__cxa_demangle(typeid((instance)).name(), nullptr, nullptr,
    nullptr);
}

int main() {
    auto *a = new auto (1);
    auto b = "123";
    auto c = new int [10]; // int *
    //auto *d = 1; 无法编译通过
    //自动类型推导(auto)需从初始化表达式中推导出变量的类型
    //但是这里的初始化表达式1是一个类型为int的整型字面量，而指针类型的语法要求显式地指定
    一个类型，所以编译器无法推导出指针的类型。
    //如果想声明一个指向整型字面量1的指针，你可以这样写：
    //auto *d = new int(1);
    //将创建一个int类型的动态分配对象，然后将指针d指向这个对象。
    //在不需要时删除它，以避免内存泄漏。
    cout << get_type(a) << endl; // int*
    cout << get_type(b) << endl; // char const*
```

```
    cout << get_type(c) << endl; //int*
}
```

- 追踪返回类型的函数
  - 可以将函数返回类型的声明信息放到函数参数列表的后面进行声明

```
auto func(char* ptr, int val) -> int;
```

- 注意:
  - auto变量必须在编译器确定类型
  - auto变量必须在定义时初始化
    - auto a; //错误
    - auto b4 = 10, b5 = 20.0, b6 = 'a'; //错误,没有推导为同一类型
  - 参数不能被声明为auto
    - void func(auto a) {...} //错误
  - auto不是一个真正的类型, 不能使用一些以类型为操作数的操作符, 如sizeof或typeid
    - cout << sizeof(auto) << endl; //错误

## decltype

- 可以对变量或表达式结果的类型进行推导
- 重用匿名类型
- auto + decltype 自动追踪返回类型

```
// c++11
auto func(int x, int y) -> decltype(x + y)
{
    return x + y;
}
// c++14
// 不需要显式指定返回类型
auto func(int x, int y)
{
    return x + y;
}
```

## 内存申请与释放

- 内存的动态申请与释放 `new delete`

```
int * ptr = new int(10); // 单个变量
int * array = new int[10]; // 10元素数组
delete ptr; // 删除指针变量所指单个内存单元
delete[] array; // 删除多个单元组成的内存块
```

- 零指针
  - NULL表示空指针，是一个int型变量
  - nullptr是严格意义上的空指针

## for循环

- 基于范围的for循环 由冒号分为两个部分，第一部分是用于迭代的变量，第二部分表示被迭代的范围

```
#include <iostream>
using namespace std;
int main() {
    int arr[3] = {1, 3, 9};
    for (int e : arr) // auto e : arr 也可以
        cout << e << endl;
    return 0;
}
```

## 3.2 基础知识

- oop的基本特征：封装（数据 + 函数）

## 3.3 类与对象

- class 用户自定义的类型
  - 包含函数与数据的特殊“结构体”，用于扩充C++语言的类型体系
  - 类中包含的函数：“成员函数”
  - 包含的数据：“成员变量”

## 3.4 成员变量与成员函数

- 成员函数必须在类内声明，但定义（实现）可以在类内或者类外。
  - 通常，**类的声明放在头文件中，而类的成员函数实现（定义）则放在实现文件中。**
  - 一般将不同的类分别保存为**不同的**头文件和实现文件。
- 类= “属性/数据” + “服务/函数”
- 为了方便解决依赖关系，复杂的成员函数声明和定义一般是分离的，很少使用类内定义的方式。

## 3.5 类成员的访问权限

- public：可以在类外访问
- private：可以在类外访问
  - class中成员的**缺省属性**为private，即不声明默认为private
- protected

```
// matrix.h
class Matrix {
```

```

public:
    void fill(char dir);
private:
    int data[6][6];
}; // <1>
//或者
class Matrix {
    int data[6][6]; // class中成员的缺省属性为private
public:
    void fill(char dir);
}; // <2>

```

- 不允许在类外(非该类的成员函数)操作访问对象的私有成员和保护成员，只能访问它的公有属性的成员（函数、数据）。

```

// main.cpp
#include "matrix.h" // Matrix类的声明
int main()
{
    Matrix obj; // 定义变量（对象）
    obj.fill('u'); // 访问公有成员
    obj.data[1][1] = 23; // ERROR! 不能在类外访问私有成员和保护成员
    return 0;
}

```

- 可以在类内用“.”操作访问**同一类**下的私有成员

```

// matrix.h
class Matrix {
private:
    int data[6][6];
    void add(Matrix a);
public:
    void fill(char dir);
};

```

```

// matrix.cpp
#include "matrix.h"
void Matrix::add(Matrix a) {
    for(int i=0; i<6; i++) {
        for(int j=0; j<6; j++) {
            data[i][j] += a.data[i][j]; // 可以在类内用“.”操作访问同一类下的私有成员
        }
    }
}

```

```

class P {
private:
    int data = 1;
    void add(P a);
public:
    void add(int i) { data += i; }
};
void P::add(P a) { data += a.data; } // A: 在P类私有函数内访问同类P对象的私有成员,
类内访问
// 虽然私有成员是不可直接访问的, 但是同一个类的成员函数可以访问该类的所有成员, 包括私有成员。
//在这个例子中, add(P a) 是 P 类的一个成员函数, 所以它可以访问 P 类的私有成员 data。

//需要注意的是, add(P a) 是私有成员函数, 只能被该类内的其他成员函数调用, 而不能被外部调用。
//因此, 这个函数只能在类内部被使用, 而不会对类外部的代码造成影响。
class Q {
private:
    void add(P a) { data += a.data; } //B: Q类函数内访问P类对象的私有成员
public:
    int data = 2;
};
int main() {
    P a, b; Q c;
    int d = c.data; // C: Q类对象的公有数据成员
    a.add(b); // D: 在main函数内, P类对象的私有函数
    a.add(d); // E: P类对象的公有函数
    return 0;
}

```

## 3.6 this指针

- 所有成员函数的参数中, 隐含着一个指向当前对象的指针变量, 其名称为this
- 是成员函数和普通函数的重要区别

```

class Matrix {
public:
    void fill(char dir) { // <1> 在类内定义成员函数
        ...
        this->data[0][0] = 1; //等价于 data[0][0] = 1;
    }
    ...
};
-----
void Matrix::fill(char dir) { // <2> 在类外定义成员函数
    this->data[0][0] = 1; // 等价于 data[0][0] = 1;
    ... ;
}

```

## 3.7 内联函数

- 为什么用内联函数
  - 函数调用要进行一系列准备和后处理工作(压栈、跳转、退栈、返回等)，所以函数调用是一个比较慢的过程。
  - 使用内联函数，编译器自动产生等价的表达式。

```
// <1> 速度慢
cout << max(a, b) << endl;
// <2>
cout << (a > b ? a : b) << endl;
// <3> 与<2>等价
inline int max(int a, int b) {
    return a > b ? a : b;
}
cout << max(a, b) << endl;
```

- 内联函数与宏定义的区别
  - 编译器会将所有宏定义的代码，直接拷贝到被调用的地方。
  - 宏代码容易出错，编译预处理器在拷贝代码时，可能产生意想不到的边界效应。

```
#define MAX(a, b) (a) > (b) ? (a) : (b)

cout << (a) > (b) ? (a) : (b) << end;
```

```
#include <iostream>
using namespace std;
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main() {
    int a = 2, b = 1;
    cout << MAX(a++, b) + 2 << endl; // 5 自增两次，但不适用自增第二次之后的值，但打印
    a可以看到两次自增结果
    cout << a << endl; // 4
    cout << ((a++) > (b) ? (a++) : (b)) + 2 << endl; // a被两次求值 7
    cout << a << endl; // 6
}
```

- 内联函数和宏定义的区别
  - 内联函数可以执行类型检查，进行编译期错误检查。
  - 内联函数可调试，而宏定义的函数不可调试。
    - 在Debug版本，内联函数没有真正内联，而是和一般函数一样，因此在该阶段可以被调试。
    - 在Release版本，内联函数实现了真正的内联，增加执行效率。宏定义的函数无法操作私有数据成员。
  - 宏使用的最常见场景：字符串定义、字符串拼接、标志粘贴(教材第9章p221~222)

- 再次强调
  - 宏定义只是拷贝代码到被调用的地方。
  - 内联函数生成的是，和函数等价的表达式。
- 注意事项：
  - 避免对大段代码使用内联修饰符
    - 相当于把该函数在所有被调用的地方拷贝了一份
    - 所以大段代码内联修饰会增加负担
  - 避免对包含循环或复杂控制结构的函数使用内联
  - 避免将内联函数的声明和定义分开
  - 一般构造函数、析构函数被定义为内联函数
  - 编译器“有权”拒绝不合理的请求，例如编译器认为某个函数不值得内联，就会忽略内联修饰符。
  - 编译器会对一些没有内联修饰符的函数，自行判断可否转化为内联函数，一般会选择短小的函数。

```
//test.h
class Test {
    int* data;
public:
    void setdata(const int* d) {data = d; } //内联函数
    const int* getdata() {return this->data;} //内联函数
    void operation1(int);
    Test(int i){
        if (i>0)
            data = new int[i];
        else
            data = nullptr;
    } //内联函数
    ~Test(){delete[] data;} //内联函数
};
```

## 课后

### 如何在c++中打印变量类型

- `abi::__cxa_demangle` 函数
  - 不是标准库中的函数
  - 是由GNU C++ 提供的扩展
  - 跨平台运行不要使用这个函数，可能不受其他编译器的支持

```
#include <iostream>
#include<typeinfo>
```



```
#include<cxxabi.h>
using namespace std;

template<typename T>
char* get_type(const T& instance){
    return abi::__cxa_demangle(typeid((instance)).name(), nullptr, nullptr,
    nullptr);
}

int main() {
    auto *x = "abc";
    cout << get_type(x) << endl;
    cout << typeid(x).name() << endl;
}
```

- typeid

## 实现class

- 下面是一个类的声明
  - 请提供成员函数的实现（注意实现和定义分开）
  - 并测试这个类是否正常运行（你还需要测试访问权限）

```
// move.h
class Move {
private:
    double x;
    double y;
    double calc(); // calculate x * y
public:
    Move(double a = 0, double b = 0);
    void display(); // display the result of calc()
    double get_x(); // return x
    double get_y(); // return y
    void reset(double a = 0, double b = 0); // reset x and y to a and b
};
```

```
// move.cpp
#include "move.h"
#include <iostream>

double Move::calc() {
    return x * y;
}

Move::Move(double a, double b) {
    x = a;
    y = b;
}
```

```

void Move::display() {
    std::cout << "Result of calc(): " << calc() << std::endl;
}

double Move::get_x() {
    return x;
}

double Move::get_y() {
    return y;
}

void Move::reset(double a, double b) {
    x = a;
    y = b;
}

```

```

// main.cpp
#include <iostream>
#include "move.h" // 头文件, 包含类的声明
using namespace std;
int main() {
    Move m(2.5, 3.5);

    // 访问私有成员变量 x 和 y 的值
    cout << "x = " << m.get_x() << endl; // 2.5
    cout << "y = " << m.get_y() << endl; // 3.5

    // 显示计算结果
    cout << "calc() = "; // 8.75
    m.display();

    // 重置 x 和 y 的值
    m.reset(1.0, 2.0);

    // 显示计算结果
    std::cout << "calc() = ";
    m.display(); // 2

    return 0;
}

```

## 函数重载, 缺省值

- 函数重载、参数默认值
  - 编写一个接受一个字符串参数, 并打印该字符串的函数。
  - 如果提供了第二个参数 (bool 类型), 且该参数为 true, 则该函数打印 n 次字符串, 其中 n 是该函数被调用的次数

```

#include <iostream>
#include <string>
using namespace std;
void f(const string& str, bool repeat = false) {
    cout << str << endl;
    if (repeat) {
        static int count = 0; // 静态变量, 记录函数被调用的次数
        ++count;
        for (int i = 1; i < count; ++i) {
            cout << str << endl;
        }
    }
}

int main() {
    f("OOP0");
    f("OOP1");
    f("OOP2", true);
    f("OOP3", false);
    f("OOP4", true);
    return 0;
}

```

## 内联函数

- 内联函数 创建两个功能相同的函数f1() f2()。f1() 是内联函数, f2() 是非内联函数。使用 <ctime> 中的标准 C 函数 clock() 标记这两个函数的开始点和结束点, 比较它们哪一个运行的更快, 为了得到有效数字, 也许需要在计时循环中重复调用这两个函数。

注意: 由于编译器可能会忽略inline或自动执行其他优化。该特性和编译器优化指令有关, 如"-O2"。

```

#include <iostream>
#include <ctime>
using namespace std;
int x = 0;
int y = 0;
int z = 0;
inline void f1() {
    for (int i = 0; i < 1000000; i++) {
        x += 1;
    }
}

void f2() {
    for (int i = 0; i < 1000000; i++) {
        y += 1;
    }
}

int main() {

```

```
clock_t start, end;
double cpu_time_used;

// Timing f1
start = clock();
for (int i = 0; i < 1000; i++) {
    f1();
}
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
cout << "Time taken by f1: " << cpu_time_used << " seconds\n"; // Time taken
by f1: 1.72 seconds

// Timing f2
start = clock();
for (int i = 0; i < 1000; i++) {
    f2();
}
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
cout << "Time taken by f2: " << cpu_time_used << " seconds\n"; // Time taken
by f2: 1.744 seconds

// Timing f3
start = clock();
for (int i = 0; i < 1000; i++) {
    for(int i = 0; i < 1000000; i++) {
        z += 1;
    }
}
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
cout << "Time taken by f2: " << cpu_time_used << " seconds\n"; // Time taken
by f2: 1.814 seconds

return 0;
}
```