directory

review

- 友元
- 静态成员与常量成员
  - 初始化方法和初始化依赖
- 对象的构造与析构时机
  - 常量/静态对象的构造/析构顺序
  - 参数对象的构造/析构顺序
- 对象的new和delete

常量引用

回顾：引用

- 具名变量的别名：`类型名 & 引用名 变量名`
  - 例：`int v0; int& v1 = v0;` v1是变量v0的引用，它们在内存中是**同一单元**的**两个不同名字**
- 引用必须在**定义时进行初始化**，且不能修改引用指向
- 函数参数可以是引用类型，表示**函数的形式参数与实际参数是同一个变量**，**改变形参将改变实参。**如调用以下函数将交换实参的值：`void swap(int& a, int& b) {int tmp = b; b = a; a = tmp; }`
- 函数返回值可以是引用类型，但不得是函数的临时变量

回顾：常量成员和常量对象

- 使用**const修饰**的数据成员，称为类的**常量数据成员**，在对象的整个生命周期里**不可更改**，且**只能在构造函数的初始化列表或就地初始化**
- **成员函数**用const修饰，该成员函数的实现语句**不能修改**类的数据成员，即不能改变对象状态（内容）
- 若对象被定义为常量，则**只能调用以const修饰的成员函数**

```
class Student {
  const int ID; //常量数据成员
public:
  Student(int id) : ID(id) {}  //通过初始化列表设置
  int getID() const { return ID; } //常量成员函数
};
```

参数中的常量和常量引用

- **最小特权原则**：给函数足够的权限去完成相应的任务，但不要给予他多余的权限。
    - 例如函数`void add(int& a, int& b)`，如果将参数类型定义为int&，则给予该函数在函数体内修改a和b的值的权限
- 如果我们不想给予函数修改权限，则可以在参数中使用**常量/常量引用**
    - `void add(const int& a, const int& b)`, 此时函数中**仅能读取**a和b的值，无法对a, b进行任何修改操作。

```
int a=10;              //非常量左值（有确定存储地址，也有变量名）
const int a1=10;       //常量左值（有确定存储地址，也有变量名）
const int a2=20;       //常量左值（有确定存储地址，也有变量名）

//非常量左值引用
int &b1=a;            //正确，a是一个非常量左值，可以被非常量左值引用绑定
int &b2=a1;           //错误,a1是一个常量左值，不可以被非常量左值引用绑定
int &b3=10;           //错误，10是一个非常量右值，不可以被非常量左值引用绑定
int &b4=a1+a2;        //错误，（a1+a2）是一个常量右值，不可以被非常量左值引用绑定

//常量左值引用
const int &c1=a;      //正确，a是一个非常量左值，可以被非常量右值引用绑定
const int &c2=a1;     //正确,a1是一个常量左值，可以被非常量右值引用绑定
const int &c3=a+a1;   //正确，（a+a1）是一个非常量右值，可以被常量右值引用绑定
const int &c4=a1+a2;  //正确，（a1+a2）是一个常量右值，可以被非常量右值引用绑定
```

可以归纳为：**非常量左值引用只能绑定到非常量左值上；常量左值引用可以绑定到非常量左值、常量左值、非常量右值、常量右值等所有的值类型。**

拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它的参数是语言规定的，是**同类对象的常量引用**
- 拷贝构造函数示例：

```
class Person {
    int id;
    ...
public:
    Person(const Person& src) { id = src.id;  ... }
    ...
};
```

- 作用：用参数对象的内容初始化当前对象
- 拷贝构造函数被调用的三种常见情况：
    - 1、用一个类对象定义另一个新的类对象
        - `Test a; Test b(a); Test c = a;`
    - 2、函数调用时以类的对象为**形参** `Func(Test a)`
    - 3、函数返回**类对象** `Test Func(void)`
    - 编译器会自动调用"拷贝构造函数"，在已有对象基础上**生成新对象。**
- 类的新对象被定义后，会调用构造函数或拷贝构造函数。如果调用拷贝构造函数且当前没有给类显式定义拷贝构造函数，编译器将自动合成"**隐式定义的拷贝构造函数**"，其功能是**调用所有数据成员的拷贝构造函数或拷贝赋值运算符。**

- 对于基础类型来说，默认的拷贝方式为**位拷贝**(Bitwise Copy)，即直接对整块内存进行复制。
- 隐式定义的拷贝构造函数示例
  - 当定义Test类的对象时`(Test a; Test b=a;)`，使用自动合成的隐式定义的拷贝构造函数
  - 编译器使用**位拷贝**初始化b的数据成员`b.data = a.data, b.buffer = a.buffer`

> 位拷贝原本是C中的概念。在C++中，只有基础类型（int, double等）才会进行位拷贝；对于自定义类，编译器会**递归调用所有数据成员的拷贝构造函数或拷贝赋值运算符。**但一些教材中仍然把这种行为称为“位拷贝”，以区别用户自定义的拷贝方法。

```
class Test {
    int data;
    char* buffer;
public:
    Test() { } //默认构造函数
    ~Test() { } //析构函数
};
```

- 注意：**隐式定义拷贝构造函数在遇到指针类型成员时可能会出错**,导致多个指针类型的变量指向同一个地址(析构时可能重复析构)

执行顺序

- 以下述的func函数为例，调用该函数时，函数中各类构造函数和析构函数的执行顺序如下：

```
Myclass func(Myclass c) {
  Myclass tmp;
  return tmp;
}
```

- 1 `Myclass func(Myclass c)`拷贝构造函数(以类的对象为形参)
- 2 `Myclass tmp;`默认构造函数
- 3 `return tmp;`拷贝构造函数(返回类对象)
- 4 tmp的析构函数
- 5 c的析构函数

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

class Test {
public:
    Test() { //构造函数
        cout << "Test()" << endl;
    }
    Test(const Test& src) { //拷贝构造
        cout << "Test(const Test&)" << endl;
    }
    ~Test() { //析构函数
```

```
        cout << "~Test()" << endl;
    }
};
Test copyObj(Test obj) {
    cout << "func()..." << endl;
    return Test();
}

int main() {
    cout << "main()..." << endl;
    Test t;
    t = copyObj(t);
    return 0;
}
/*
输出:
main()...
Test()
Test(const Test&)
func()...
Test()
~Test()
~Test()
~Test()
*/
```

- 隐式定义拷贝会使得对象a, b的指针成员m_arr指向同一个内存地址
- 当类内**含指针类型的成员**时，为避免**指针被重复删除**，不应使用隐式定义的拷贝构造函数

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

class Pointer {
    int *m_arr;
    int m_size;
public:
    Pointer(int i):m_size(i) { //构造
        m_arr = new int[m_size];
        memset(m_arr, 0, m_size*sizeof(int));
    }
    ~Pointer(){delete []m_arr;} //析构
    void set(int index, int value) {
        m_arr[index] = value;
    }
    void print();
};
void Pointer::print()
{
    cout << "m_arr: ";
    for (int i = 0; i < m_size; ++ i)
```

```
    {
        cout << " " << m_arr[i];
    }
    cout << endl;
}

int main() {
  Pointer a(5);
  Pointer b = a; //调用默认的拷贝构造
    a.print();
    b.print();
    b.set(2, 3);
    b.print();
    a.print();
    return 0;
}
```

- 拷贝构造有什么问题?
    - 当对象很大的时候?
    - 当对象含有指针的时候?
- 频繁的拷贝构造会造成程序效率的显著下降
- 正常情况下，应尽可能避免使用拷贝构造函数
    - 解决方法： （1）使用**引用/常量引用传参数或返回对象**； （2）将拷贝构造函数声明为**private**；（3）用**delete**关键字让编译器**不生成拷贝构造函数的隐式定义版本。**
- (1)
    - 引用或常量引用传递参数 `func(MyClass a) --> func(const MyClass& a)`
    - 返回值为引用 `MyClass func(…) --> MyClass& func(…)`
- (2)
    - 拷贝构造函数私有化

```
class MyClass {
    MyClass(const MyClass&){}
    public:
        MyClass() = default;
    ...
}
```

- (3)
    - 拷贝构造函数显式删除

```
class MyClass {
    public:
```

```
        MyClass() = default;
        MyClass(const MyClass&) = delete;
        ...
}
```

右值引用

- 多数情况下，我们更需要对象的“移动”，而非对象的“拷贝”。C++11为此提供了一种新的构造函数，即**移动构造函数。**
- 为理解移动构造函数的工作原理，首先要引入C++11的另一个新特性——右值引用。

左值和右值

- 左值：**可以取地址、有名字的值。**
- 右值：不能取地址、没有名字的值;常见于**常值、函数返回值、表达式**

```
int a = 1;
int b = func();
int c = a + b;
// 其中a、b、c为左值，1、func函数返回值、a+b的结果为右值。
```

- 左值**可以取地址**，并且**可以被&引用**(左值引用)

```
int *d = &a; // 正确
int &d = a; // 正确
int *e = &(a + b); // 错误
int &e = a + b; // 错误
```

- 右值引用
    - 虽然右值无法取地址，但可以被&&引用(右值引用)：`int &&e = a + b;// 正确`
    - 右值引用无法绑定左值：`int &&e = a; // 错误`
- 总结
    - 左值引用能绑定左值，右值引用能绑定右值
    - 例外：常量左值引用能也绑定右值（为什么这么设计？）

```
const int &e = 3;  // 正确
const int &e = a*b;  // 正确
```

引用的绑定

- 非常量左值引用：非常量左值

- 常量左值引用：非常量左值、常量左值、右值
- 右值引用：右值
- 注意：**所有的引用（包括右值引用）本身都是左值**，结合该规则和上表便可判断各种构造函数、赋值运算符中传递参数和取返回值的引用绑定情况
- 右值引用示例：

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

void ref(int &x) {
  cout << "left " << x << endl;
}

void ref(int &&x) {
  cout << "right " << x << endl;
}

int main() {
  int a = 1;
  ref(a);
  ref(2); //2是一个常量
  return 0;
}
/*
Output:
left 1
right 2

int &x代表左值引用参数;
int &&x代表右值引用参数,
对2的引用是右值引用。

如果没有定义 ref(int &&x) 函数会发生什么?
编译错误:
[Error] invalid initialization of non-const reference of type 'int&' from an
rvalue of type 'int'
*/
```

```
#include <iostream>

using namespace std;

void ref(int &x) {
  cout << "left " << x << endl;
}

void ref(int &&x) {
  cout << "right " << x << endl;
  ref(x); //调用哪一个函数?
}
```

```
int main() {
  ref(1); //1是一个常量
  return 0;
}
/*
right 1
left 1
Output:
right 1
left 1

ref(1)首先调用ref(int &&x)函数,
此时右值引用x为左值,
因此ref(x)调用ref(int &x)函数。
*/
```

移动构造函数

- 右值引用可以延续即将销毁变量的生命周期，用于构造函数可以提升处理效率，在此过程中尽可能少地进行拷贝。 使用右值引用作为参数的构造函数叫做移动构造函数。
- 拷贝构造函数：`ClassName(const ClassName& VariableName);`
- 移动构造函数：`ClassName(ClassName&& VariableName);`
- 移动构造函数与拷贝构造函数最主要的差别就是类中**堆内存**是重新开辟并拷贝，还是**直接将指针指向那块地址**。
- 对于一些即将析构的临时类，移动构造函数**直接利用了原来临时对象中的堆内存**，新的对象**无需开辟内存**，**临时对象无需释放内存**，从而大大提高计算效率。

```
class Test {
public:
  int * buf; //// only for demo.
  Test() {
    buf = new int[10]; //申请一块内存
    cout << "Test(): this->buf @ " << hex << buf << endl;
  }
  ~Test() {
    cout << "~Test(): this->buf @ " << hex << buf << endl;
    if (buf) delete[] buf;
  }
  Test(const Test& t) : buf(new int[10]) {
    for(int i=0; i<10; i++) buf[i] = t.buf[i]; //拷贝数据
    cout << "Test(const Test&) called. this->buf @ " << hex << buf << endl;
  }
  Test(Test&& t) : buf(t.buf) { //直接复制地址，避免拷贝
    cout << "Test(Test&&) called. this->buf @ "<< hex << buf << endl;
    t.buf = nullptr; //将t.buf改为nullptr，使其不再指向原来内存区域
  }
```

```
};
Test GetTemp() {
  Test tmp;
  cout << "GetTemp(): tmp.buf @ " << hex << tmp.buf << endl;
  return tmp;
}
void fun(Test t) {
  cout << "fun(Test t): t.buf @ " << hex << t.buf << endl;
}
int main() {
  Test a = GetTemp();
  cout << "main() : a.buf @ " << hex << a.buf << endl;
  fun(a);
  return 0;
}
```

右值引用：移动语义

- 如何加快左值初始化的构造速度
  - 移动构造函数加快了右值初始化的构造速度。
  - 如何对左值调用移动构造函数以加快左值初始化的构造速度?
- `std::move`函数
  - 输入：左值（包括变量等，该左值一般不再使用）
  - 返回值：该左值对应的右值

```
Test a;
Test b = std::move(a)
//对于上个实例中定义的Test类，该处调用移动构造函数对b进行初始化
```

- 注意：move函数本身不对对象做任何操作，仅做类型转换，即转换为右值。移动的具体操作在移动构造函数内实现。
- 右值引用结合std::move可以显著提高swap函数的性能。
  - std::move引起移动构造函数或移动赋值运算的调用
  - **避免3次不必要的拷贝操作**

```
template <class T>
swap(T& a, T& b) {
     T tmp(a); //copy a to tmp
     a = b; //copy b to a
     b = tmp; //copy tmp to b
}
```

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

```
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

拷贝/移动构造函数的调用时机

- 判断依据：引用的绑定规则
    - 拷贝构造函数的形参类型为**常量左值引用**，可以绑定常量左值、左值和右值
    - 移动构造函数的形参类型为**右值引用**，可以绑定右值
    - 引用的绑定存在**优先级**，例如常量左值引用和右值引用均能绑定右值，当传入实参类型为右值时优先匹配形参类型为右值引用的函数
- 拷贝构造函数的常见调用时机
    - 用一个类对象/引用/常量引用初始化另一个新的类对象
    - 以类的对象为函数形参，传入实参为类的对象/引用/常量引用
    - 函数返回类对象（类中未显式定义移动构造函数，不进行返回值优化）
- 移动构造函数的常见调用时机
    - 用一个类对象的右值初始化另一个新的类对象（常配合`std::move`函数一起使用）：`Test b = func(a); Test b = std::move(a);` 与`Test b = a;` 不同
    - 以类的对象为函数形参，传入实参为类对象的右值（常配合std::move函数一起使用）：`func(Test());func(std::move(a));`与`func(a)`不同
    - 函数返回类对象（类中显式定义移动构造函数，不进行返回值优化）：`{return Test(); or return tmp;}`均调用移动构造

拷贝赋值运算符

- 已定义的对象之间相互赋值，可通过调用对象的“拷贝赋值运算符函数”来实现的

```
ClassName& operator= (const ClassName& right) {
   if (this != &right) {// 避免自己赋值给自己
   // 将right对象中的内容拷贝到当前对象中...
  }
   return *this;
}
```

- 注意区分下面两种代码：

```
// 赋值
ClassName a;
ClassName b;
a = b;
```

```
// 拷贝构造
ClassName a = b;
```

```
Test& operator= (const Test& right) {
  if (this == &right)  cout << "same obj!\n";
  else {
    for(int i=0; i<10; i++)
      buf[i] = right.buf[i];  //拷贝数据
    cout << "operator=(const Test&) called.\n";
  }
  return *this;
}
```

- 赋值重载函数必须要是类的非静态成员函数(non-static member function)，不能是友元函数。
- 和移动构造函数原理类似

```
Test& operator= (Test&& right) {
  if (this == &right)  cout << "same obj!\n";
  else {
    this->buf = right.buf;  //直接赋值地址
    right.buf = nullptr;
    cout << "operator=(Test&&) called.\n";
  }
  return *this;
}
```

```
swap(Test& a, Test& b) {
     Test tmp(std::move(a)); // 第一行调用移动构造函数
     a = std::move(b);       // std::move的结果为右值引用,
     b = std::move(tmp); // 后两行均调用移动赋值运算
}
```

类型转换

- 当编译器发现表达式和函数调用所需的数据类型和实际类型不同时，便会进行自动类型转换。
- 自动类型转换可通过定义特定的转换运算符和构造函数来完成。
- 除自动类型转换外，在有必要的时候还可以进行强制类型转换。

```
void print(int d) { }

int main()
{
  print(3.5);
```

```
    print('c');
    return 0;
  }
```