C++ 11

C++ 11

- C++ 类
- C++ 指针
- C++ 左值、右值、引用
- C++ 参数传递
- C++ 返回值传递
- std::swap std::move
- C++ 移动构造函数，移动赋值操作符函数
- C++ 模版
- 矩阵例子

C++ 类

- 概念
  - 类 class
  - 成员 member
    - 成员函数 member function (也叫 方法 method)
  - 实例 instance
    - 对象 object
  - 访问权限修饰符
    - public
    - private
  - 构造函数 constructor

```
private:
  int storedValue;
```

C++ 类

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      : storedValue{ initialValue } { }
    int read( ) const
      { return storedValue; }
```

- 概念
  - 默认参数
  - 初始化列表 initialization list
    - 对于 const 数据成员是必须的
    - 对于没有默认构造函数的对象成员是必须的
    - 对于没有默认构造函数的父类是必须的
  - explicit 构造函数
    - 禁止隐式类型转换 implicit type-conversion

```
IntCell obj;      // obj是IntCell型对象
obj = 37;         // 不应编译：类型不匹配
```

- 使用{}代替()是C++11的引入的
  - 令所有初始化列表都统一使用{}

C++ 类

- 常成员函数 const member function
- 访问函数 accessor
  - isEmpty
  - getName
- 修改函数 mutator
  - makeEmpty
  - setName

```
int read( ) const
  { return storedValue; }

void write( int x )
  { storedValue = x; }
```

C++类

- 接口与实现分离
  - 接口放在.h文件中
    - 为了避免接口被编译器多次读取
    - #ifndef NAME
      #define NAME
      …
      #endif
  - 实现文件放在.cpp文件中
    - 特征signature必须完全匹配
      - 类中定义的名字需要使用全名
      - 常成员函数的const
      - :: 作用域解释符 scope resolution operator

```
int IntCell::read( ) const
{
    return storedValue;
}
```

```
1   #ifndef IntCell_H
2   #define IntCell_H
3
4   /**
5    * 一个模拟整数单元的类.
6    */
7   class IntCell
8   {
9     public:
10      explicit IntCell( int initialValue = 0 );
11      int read( ) const;
12      void write( int x );
13
14    private:
15      int storedValue;
16  };
17
18  #endif
```

C++类

- 对象声明　声明都是合法的：

```
IntCell obj1;         // 零参数构造函数
IntCell obj2( 12 );   // 单参数构造函数
```

而另一方面，下列声明都是不正确的：

```
IntCell obj3 = 37;    // 构造函数是explicit的
IntCell obj4( );      // 函数声明
```

  - obj4 的混乱现象是引入{}的原因之一

```
IntCell obj1;         // 零参数构造函数，同前
IntCell obj2{ 12 };   // 单参数构造函数，同前
IntCell obj4{ };      // 零参数构造函数
```

C++类

- vector (作为C++的一级公民，享受下面的特殊初始化语法)

```
vector<int> daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
vector<int> daysInMonth { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

不过，歧义也就伴随着语法出现了，从下面的声明看到

```
vector<int> daysInMonth { 12 };
```

C++11 给了初始化表列优先权

```
vector<int> daysInMonth( 12 );    // 必须使用 () 来调用填写大小的构造函数
```

C++类

- 范围for语句 (range for)
- 保留字 auto
  - 编译器自动推断适合的类型

```
int sum = 0;
for( int x : squares )
    sum += x;
```

```
int sum = 0;
for( auto x : squares )
    sum += x;
```

C++ 指针

- 动态内存分配

```
1   int main( )
2   {
3       IntCell *m;
4
5       m = new IntCell{ 0 };
6       m->write( 5 );
7       cout << "Cell contents: " << m->read( ) << endl;
8
9       delete m;
10      return 0;
11  }
```

```
m = new IntCell( );     // OK
m = new IntCell{ };     // C++11
m = new IntCell;        // 本书首选
```

- C++对于动态分配的内存不进行自动垃圾回收
  - 内存泄漏 (memory leak) 在C++程序中普遍存在
  - 尽量不使用 new，而使用自动变量 automatic variable

C++ 左值、右值、引用

- 左值 lhs
  - 非临时变量
- 右值 rhs
  - 临时变量
    - 存储表达式或函数调用的结果的临时变量
    - 字面值 literal
- 例子中的左值
  - arr, str, arr[x], y, z, ptr, *ptr, (*ptr)[0]
  - x 是一个不能被改变的左值
- 例子中的右值
  - 2, "foo”, x+y, str.substr(0,1)
- 也存在函数和运算符结果是右值的情况？
  - *ptr, ptr[x] *不是一般的运算符； ptr[x]返回的是引用(对引用的操作并不会改变引用)

```
vector<string> arr( 3 );
const int x = 2;
int y;
  ...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

C++ 左值、右值、引用

- 左值引用 left value reference

```
string str = "hell";
string & rstr = str;                    // rstr 是 str 的另一个名字
rstr += 'o';                            // 把 str 改成"hello"
bool cond = (&str == &rstr);            // true; str 和 rstr 是同一对象
string & bad1 = "hello";                // 非法: "hello" 不是可修改的左值
string & bad2 = str + "";               // 非法: str+"" 不是左值
string & sub = str.substr( 0, 4 ); // 非法: str.substr( 0, 4 ) 不是左值
```

- 右值引用 right value reference

```
string str = "hell";
string && bad1 = "hello";               // 合法
string && bad2 = str + "";              // 合法
string && sub = str.substr( 0, 4 ); // 合法
```

C++ 左值、右值、引用

- auto

```
for( auto x : arr )   // 行不通
    ++x;

for( auto & x : arr ) // 行得通
     ++x;
```

C++ 参数传递

- 传值调用 call by value
- 传引用调用 call by reference

```
void swap( double & a, double & b );      // 交换 a 和 b; 参数类型正确
```

- 传常量引用调用 call by constant reference

```
string randomItem( const vector<string> & arr ); // 返回 arr 中的一个随机项
```

- 传右值引用调用 call by right value reference

```
string randomItem( const vector<string> & arr );  // 返回左值 arr 中的随机项
string randomItem( vector<string> && arr );       // 返回右值 arr 中的随机项

vector<string> v { "hello", "world" };
cout << randomItem( v ) << endl;                          // 调用左值的方法
cout << randomItem( { "hello", "world" } ) << endl; // 调用右值的方法
```

C++ 返回值传递

- 传值返回 return by value
- 传常量引用返回 return by constant reference
- 在C++11中，返回对象可以移动到调用函数中
  - result被移动到sum
  - 不可移动的情况

```
LargeType randomItem1( const vector<LargeType> & arr )
{
    return arr[ randomInt( 0, arr.size( ) - 1 ) ];
}
```

- 传引用返回
  - T & operator [] (int)

```
vector<int> partialSum( const vector<int> & arr )
{
    vector<int> result( arr.size( ) );

    result[ 0 ] = arr[ 0 ];
    for( int i = 1; i < arr.size( ); ++i )
        result[ i ] = result[ i - 1 ] + arr[ i ];

    return result;
}

    vector<int> vec;
      ...
    vector<int> sums = partialSum( vec ); // 在老的C++ 中是复制；在 C++11 中是移动
```

std::swap std::move

- swap 函数的高效实现

```
void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = std::move( x );
    x = std::move( y );
    y = std::move( tmp );
}
```

```
13 int main() {
14     vector<int> v = {1,2,3};
15     auto x = move(v);
16     cout << v.size(); // 0
17 }
```

C++ 移动构造函数，移动赋值操作符函数

- 五大函数
  - destructor
    - 运行超出自动变量的范围scope，动态内存变量被delete
  - copy constructor
    - 拷贝左值：初始化，传值调用，传值返回
  - move constructor
    - 拷贝右值
    - 从另个反面理解右值：可以被取代的 (move的实际意义就是取代)

```
IntCell B = C;   // 若C是左值则调用拷贝构造函数；若C是右值则调用移动构造函数
IntCell B { C }; // 若C是左值则调用拷贝构造函数；若C是右值则调用移动构造函数
```

  - copy assignment operator
  - move assignment operator

```
~IntCell( );                                    // 析构函数
IntCell( const IntCell & rhs );                 // 拷贝构造函数
IntCell( IntCell && rhs );                      // 移动构造函数
IntCell & operator= ( const IntCell & rhs );    // 拷贝赋值
IntCell & operator= ( IntCell && rhs );         // 移动赋值
```

C++ 移动构造函数，移动赋值操作符函数

- 指明使用默认的实现(C++自动合成的实现)

```
~IntCell( ) { cout << "Invoking destructor" << endl; }     // 析构函数
IntCell( const IntCell & rhs ) = default;                  // 拷贝构造函数
IntCell( IntCell && rhs ) = default;                       // 移动构造函数
IntCell & operator= ( const IntCell & rhs ) = default;     // 拷贝赋值
IntCell & operator= ( IntCell && rhs ) = default;          // 移动赋值
```

- 禁止对象复制和赋值

```
IntCell( const IntCell & rhs ) = delete;                   // 无拷贝构造函数
IntCell( IntCell && rhs ) = delete;                        // 无移动构造函数
IntCell & operator= ( const IntCell & rhs ) = delete;      // 无拷贝赋值
IntCell & operator= ( IntCell && rhs ) = delete;           // 无移动赋值
```

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      { storedValue = new int{ initialValue }; }

    ~IntCell( )                                                    // 析构函数
      { delete storedValue; }

    IntCell( const IntCell & rhs )                                 // 拷贝构造函数
      { storedValue = new int{ *rhs.storedValue }; }

    IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue }     // 移动构造函数
      { rhs.storedValue = nullptr; }

    IntCell & operator= ( const IntCell & rhs )                    // 拷贝赋值
    {
        if( this != &rhs )
            *storedValue = *rhs.storedValue;
        return *this;
    }

    IntCell & operator= ( IntCell && rhs )                         // 移动赋值
    {
        std::swap( storedValue, rhs.storedValue );
        return *this;
    }
```

C++ 移动构造函数，移动赋值操作符函数

- C++中常使用拷贝课交换来实现拷贝赋值

```
IntCell & operator= ( const IntCell & rhs )          // 拷贝赋值
{
    IntCell copy = rhs;
    std::swap( *this, copy );
    return *this;
}
```

  - 注意：如果swap中又调用了上面的拷贝赋值，会出现无限递归
    - C++中预期：swap通过3次move实现

C++ 移动构造函数，移动赋值操作符函数

- 成员中包含非基本数据类型时，移动构造和赋值的实现
  - Items是非基本数据类型

```
IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue },        // 移动构造函数
                            items{ std::move( rhs.items ) }
  { rhs.storedValue = nullptr; }
```

- 移动赋值操作符可通过逐个成员交换实现

```
IntCell & operator= ( IntCell && rhs )
{
    std::swap( storedValue, rhs.storedValue );
    return *this;
}
```

C++ 移动构造函数，移动赋值操作符函数

```
5  class A
6  {
7  public:
8      A() {}
9      A(A && x) { cout << "move constructor" << endl; }
10     A & operator =(A && x) { cout << "move assignment" << endl;
11                              return *this; }
12 };
13
14 int main() {
15     A a, b;
16     b = move(a);
17     A c(move(b));
18     swap(a,b);
19 }
```

```
move assignment
move constructor
move constructor
move assignment
move assignment
```

C++ 移动构造函数，移动赋值操作符函数

- move 做的事情就是类型转换，使得编译器被骗然后调用move constructor或move assignment

```
1   void swap( vector<string> & x, vector<string> & y )
2   {
3       vector<string> tmp = static_cast<vector<string> &&>( x );
4       x = static_cast<vector<string> &&>( y );
5       y = static_cast<vector<string> &&>( tmp );
6   }
7
8   void swap( vector<string> & x, vector<string> & y )
9   {
10      vector<string> tmp = std::move( x );
11      x = std::move( y );
12      y = std::move( tmp );
13  }
```

C++ 移动构造函数，移动赋值操作符函数

- 无限递归
  - swap 调用
    - move constructor
    - move assignment

```
5  class A
6  {
7  public:
8      A() {}
9      A(A && x) { swap(*this,x); }
10     A & operator =(A && x) { swap(*this,x);
11                              return *this; }
12 };
13
14 int main() {
15     A a, b;
16     b = move(a);
17 }
```

C++ 模版

- 实现类型无关操作
  - 如顺序查找
- 当算法和数据机构与元素类型无关，使用模版实现一次代码编写
- 泛型generics: 能泛化到不同元素类型的(容器)类型
  - 在C++中泛型通过模版类实现
  - 如vector

C++ 模版

- 除了模版类，C++还支持模版函数

```
1  /**
2   * 返回数组a中的最大项.
3   * 假设a.size( ) > 0.
4   * 可比较的对象必须提供operator<和operator=
5   */
6  template <typename Comparable>
7  const Comparable & findMax( const vector<Comparable> & a )
8  {
9      int maxIndex = 0;
10
11     for( int i = 1; i < a.size( ); ++i )
12         if( a[ maxIndex ] < a[ i ] )
13             maxIndex = i;
14     return a[ maxIndex ];
15 }
```

C++ 模版

- 模版的自动展开
  - 展开后静态检查
    - 类型缺少成员则报错
  - 代码膨胀 code bloat
- 匹配
  - 非模版会被优先

```
1   int main( )
2   {
3       vector<int>     v1( 37 );
4       vector<double>  v2( 40 );
5       vector<string>  v3( 80 );
6       vector<IntCell> v4( 75 );
7
8       // 填入到未显示的那些 vector 中的附加代码
9
10      cout << findMax( v1 ) << endl;  // OK: Comparable = int
11      cout << findMax( v2 ) << endl;  // OK: Comparable = double
12      cout << findMax( v3 ) << endl;  // OK: Comparable = string
13      cout << findMax( v4 ) << endl;  // 非法; operator< 未定义
14
15      return 0;
16  }
```

C++ 模版

- 例子

```
1  /**
2   * 一个模拟内存单元的类.
3   */
4  template <typename Object>
5  class MemoryCell
6  {
7    public:
8      explicit MemoryCell( const Object & initialValue = Object{ } )
9        : storedValue{ initialValue } { }
10     const Object & read( ) const
11       { return storedValue; }
12     void write( const Object & x )
13       { storedValue = x; }
14   private:
15     Object storedValue;
16 };
```

```
1  int main( )
2  {
3      MemoryCell<int>    m1;
4      MemoryCell<string> m2{ "hello" };
5
6      m1.write( 37 );
7      m2.write( m2.read( ) + "world" );
8      cout << m1.read( ) << endl << m2.read( ) << endl;
9
10     return 0;
11 }
```

C++ 模版

- 很多情况下整个模版类连同它的实现必须放在同一个文件里
  - STL 流行的实现方法遵循这个规则
    - #include <iostream>
  - 由于模版的分离式编译
    - 增加很多额外的语法负担
    - 在很多平台上不能很好工作

C++ 模版

- 常用模版参数
  - Object
    - 零参数构造函数
    - operator =
    - 拷贝构造函数
  - Comparable
    - operator <
    - operator == 不一定用到
  - Comparator
    - 用来给不是Comparable的对象T提供比较函数
    - bool isLessThan(const T &, const T &)

```
1   // 泛型findMax, 带有一个函数对象, Version #1
2   // 前提: a.size( ) > 0
3   template <typename Object, typename Comparator>
4   const Object & findMax( const vector<Object> & arr, Comparator cmp )
5   {
6       int maxIndex = 0;
7
8       for( int i = 1; i < arr.size( ); ++i )
9           if( cmp.isLessThan( arr[ maxIndex ], arr[ i ] ) )
10              maxIndex = i;
11
12      return arr[ maxIndex ];
13  }
14
15  class CaseInsensitiveCompare
16  {
17    public:
18      bool isLessThan( const string & lhs, const string & rhs ) const
19        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
20  };
21
22  int main( )
23  {
24      vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
25      cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
26
27      return 0;
28  }
```

C++ 模版

- 常用模版参数
  - Object
    - 零参数构造函数
    - operator =
    - 拷贝构造函数
  - Comparable
    - operator <
    - operator == 不一定用到
  - Comparator
    - 用来给不是Comparable的对象T提供比较函数
    - ~~bool isLessThan(const T &, const T &)~~
    - bool operator() (const T &, const T &)

```
1   // 泛型 findMax， 用到一个函数对象，C++ 风格
2   // 前提: a.size( ) > 0
3   template <typename Object, typename Comparator>
4   const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
5   {
6       int maxIndex = 0;
7
8       for( int i = 1; i < arr.size( ); ++i )
9           if( isLessThan( arr[ maxIndex ], arr[ i ] ) )
10              maxIndex = i;
11
12      return arr[ maxIndex ];
13  }
14
15  // 泛型findMax， 使用默认的排序
16  #include <functional>
17  template <typename Object>
18  const Object & findMax( const vector<Object> & arr )
19  {
20      return findMax( arr, less<Object>{ } );
21  }
22
23  class CaseInsensitiveCompare
24  {
25    public:
26      bool operator( )( const string & lhs, const string & rhs ) const
27        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
28  };
```

C++ 模版

- 矩阵例子
  - 五大函数均可被自动处理

```
1   #ifndef MATRIX_H
2   #define MATRIX_H
3
4   #include <vector>
5   using namespace std;
6
7   template <typename Object>
8   class matrix
9   {
10    public:
11      matrix( int rows, int cols ) : array( rows )
12      {
13          for( auto & thisRow : array )
14              thisRow.resize( cols );
15      }
16
17      matrix( vector<vector<Object>> v ) : array{ v }
18        { }
19      matrix( vector<vector<Object>> && v ) : array{ std::move( v ) }
20        { }
21
```

C++ 模版

- 矩阵例子

```
22        const vector<Object> & operator[]( int row ) const
23          { return array[ row ]; }
24        vector<Object> & operator[]( int row )
25          { return array[ row ]; }
26
27        int numrows( ) const
28          { return array.size( ); }
29        int numcols( ) const
30          { return numrows( ) ? array[ 0 ].size( ) : 0; }
31      private:
32        vector<vector<Object>> array;
33    };
34    #endif
```