Tarena Technology
达内科技

模板与STL

C/C++教学体系

Tarena Technology 达内科技

“

个人简介

闵卫
minwei@tarena.com.cn

”

Technology
Tarena
达内科技

“

模板

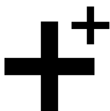
”

Tarena Technology 达内科技

知识导入

全程目标

- 类型参数
- 函数模板与类模板
- 模板特化
- 语法公式
- 编译模型
- 局部特化
- 非类型参数
- 缺省参数
- 模板与继承
- 模板型成员
- 模板递归实例化
- 模板型模板参数
- typename与class
- 容器与迭代器



Tarena Technology 达内科技

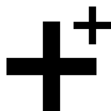
知识讲解

类型参数

- 为同一种算法，定义适用于不同类型的版本

```
int max_int (int a, int b) {
  return a > b ? a : b;
}
string max_string (string a, string b) {
  return a > b ? a : b;
}
```

```
cout << max_int (100, 200) << endl;
cout << max_string ("hello", "world") << endl;
```



知识讲解

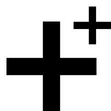
Tarena Technology 达内科技

类型参数

- 借助参数宏摆脱类型限制，同时也丧失了类型安全

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

```
cout << max (100, 200) << endl;
cout << max ("hello", "world") << endl;
```



类型参数

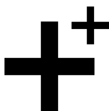
- 让预处理器自动生成针对不同类型的版本

```
#define MAX(T) \
T max_##T (T a, T b) { \
  return a > b ? a : b; \
}
```

```
MAX (int)
MAX (string)
```

```
#define max(T) max_##T
```

```
cout << max(int) (100, 200) << endl;
cout << max(string) ("hello", "world") << endl;
```



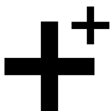
类型参数

- 编写带有参数化类型的通用版本
- 让编译器自动生成针对不同类型的具体版本

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

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

```
cout << max (100, 200) << endl;
cout << max ("hello", "world") << endl;
```



Tarena Technology 达内科技

知识讲解

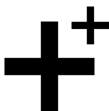
函数模板

- 函数的参数、返回值和局部变量均可使用类型参数

```
template<typename T>
T max (T a, T b) {
  return a > b ? a : b;
}
```

```
cout << max<int> (100, 200) << endl;
cout << max<string> ("hello", "world") << endl;
```

```
cout << max (100, 200) << endl;
cout << max ("hello", "world") << endl;
```

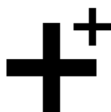


类模板

- 类的成员变量、成员函数、成员类型，甚至基类均可使用类型参数

```
template<typename T>
class Comparator {
public:
  Comparator (T a, T b) : m_a (a), m_b (b) {}
  T max (void) const {
    return m_a > m_b ? m_a : m_b;
  }
private:
  T m_a;
  T m_b;
};
```

```
Comparator<int> ci (100, 200);
cout << ci.max () << endl;
Comparator<string> cs ("hello", "world");
cout << cs.max () << endl;
```



Tarena Technology 达内科技

知识讲解

模板特化

- 对于某些特定类型而言，通用模板可能并不适用

```
template<typename T>
T max (T a, T b) {
  return a > b ? a : b;
}
```

```
char* max (char* a, char* b) {
  return a > b ? a : b; // C风格字符串不能这样比较！
}
```

```
char a[] = "hello",
char b[] = "world";
cout << max<char*> (a, b) << endl;
```

知识讲解

Tarena Technology 达内科技

模板特化

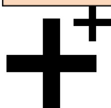
- 可以为通用模板提供一种特殊化定义，作为一般情况之外的特例，为编译器提供一种更为合适的选择

```
template<typename T>
T max (T a, T b) {
  return a > b ?
    a : b;
}
```

```
template<>
char* max (char* a, char* b) {
  return strcmp (a, b) > 0 ?
    a : b;
}
```

```
cout << max<int> (100, 200)
  << endl;
cout << max<string> ("hello",
  "world") << endl;
```

```
char a[] = "hello",
char b[] = "world";
cout << max<char*> (a, b)
  << endl;
```



模板特化

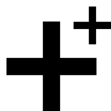
- 可以为完整的类模板提供特化

```
template<typename T>
class Comparator {
public:
  Comparator (T a, T b) :
    m_a (a), m_b (b) {}
  T max (void) const {
    return m_a > m_b ?
      m_a : m_b;
  }
private:
  T m_a;
  T m_b;
};
```

```
template<>
class Comparator<char*> {
public:
  Comparator (char* a, char* b) :
    m_a (a), m_b (b) {}
  char* max (void) const {
    return strcmp (m_a, m_b) > 0 ?
      m_a : m_b;
  }
private:
  char* m_a;
  char* m_b;
};
```

```
Comparator<int> ci (100, 200);
cout << ci.max () << endl;
Comparator<string> cs ("hello",
  "world");
cout << cs.max () << endl;
```

```
char a[] = "hello";
char b[] = "world";
Comparator<char*> cp (a, b);
cout << cp.max () << endl;
```



模板特化

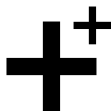
- 也可以只为类模板的部分成员提供特化

```
template<typename T>
class Comparator {
public:
  Comparator (T a, T b) :
    m_a (a), m_b (b) {}
  T max (void) const {
    return m_a > m_b ?
      m_a : m_b;
  }
private:
  T m_a;
  T m_b;
};
```

```
template<>
char* Comparator<char*>::max (
  void) const {
  return strcmp (m_a, m_b) > 0 ?
    m_a : m_b;
}
```

```
Comparator<int> ci (100, 200);
cout << ci.max () << endl;
Comparator<string> cs ("hello",
  "world");
cout << cs.max () << endl;
```

```
char a[] = "hello";
char b[] = "world";
Comparator<char*> cp (a, b);
cout << cp.max () << endl;
```



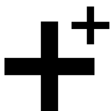
知识讲解

语法公式

- 函数模板
  - 通用版本
    - ✓ 声明同时定义
      template<typename 类型形参1, ...>
      返回类型 函数模板名 (形参表) { ... }
    - ✓ 声明定义分开
      template<typename 类型形参1, ...>
      返回类型 函数模板名 (形参表);

      ---

      template<typename 类型形参1, ...>
      返回类型 函数模板名 (形参表) { ... }

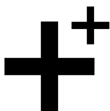


Tarena Technology 达内科技

知识讲解

语法公式

- 函数模板
  - 特化版本
    - ✓ 声明同时定义
      template<>
      返回类型 函数模板名<类型实参1, ...> (形参表) { ... }
    - ✓ 声明定义分开
      template<>
      返回类型 函数模板名<类型实参1, ...> (形参表);
      template<>
      返回类型 函数模板名<类型实参1, ...> (形参表) { ... }



知识讲解

Tarena 达内科技

语法公式

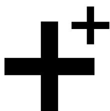
- 类模板
  - 通用版本
    - ✓ 声明同时定义

      template<typename 类型形参1, ...>
      class 类模板名 { ... };
    - ✓ 声明定义分开

      template<typename 类型形参1, ...>
      class 类模板名 { ... };

      ---

      template<typename 类型形参1, ...>
      返回类型 类模板名<类型形参1, ...>::成员函数名 (形参表) { ... }



Tarena Technology 达内科技

知识讲解

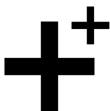
语法公式

- 类模板
  - 特化版本
    - ✓ 声明同时定义

      template<>
      class 类模板名<类型实参1, ...> { ... };
    - ✓ 声明定义分开

      template<>
      class 类模板名<类型实参1, ...> { ... };

      返回类型 类模板名<类型实参1, ...>::成员函数名 (形参表) { ... }

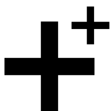


Tarena Technology 达内科技

语法公式

- 类模板
  - 特化版本
    - ✓ 针对成员函数

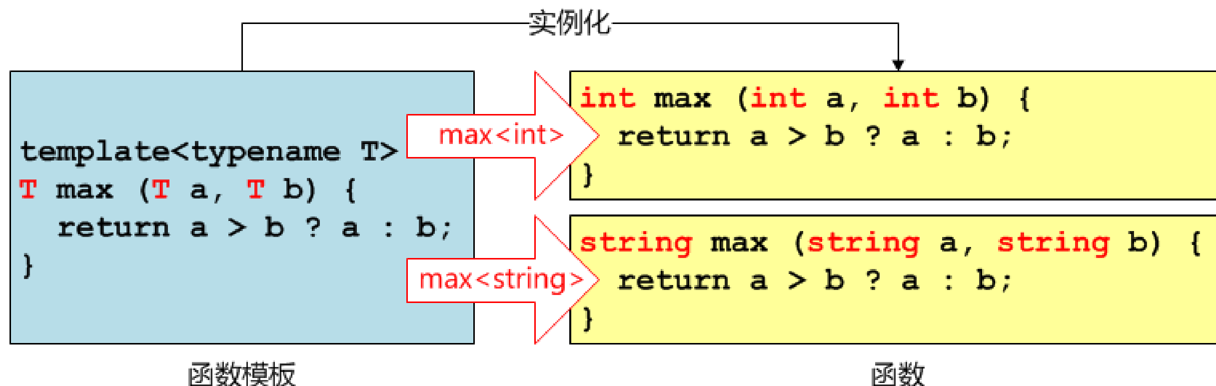
      template<>
      返回类型 类模板名<类型实参1, …>::成员函数名 (形参表) { … }

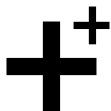


Tarena Technology 达内科技

编译模型

- 模板实例化
  - 从函数模板生成函数的过程，称为函数模板的实例化
  - 当一个函数模板被特定的具体类型实例化时，函数模板中的每个类型形参，都会被具体的类型实参所取代

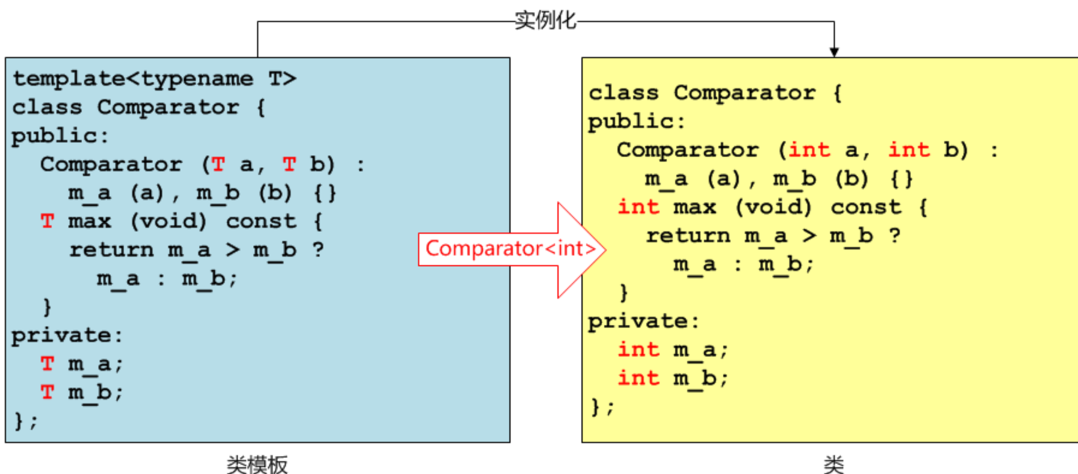


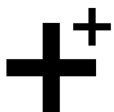


Tarena Technology 达内科技

编译模型

- 模板实例化
  - 从类模板生成类的过程，称为类模板的实例化
  - 当一个类模板被特定的具体类型实例化时，类模板中的每个类型形参，都会被具体的类型实参所取代



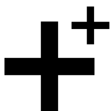


Tarena Technology 达内科技

知识讲解

编译模型

- 后期编译
  - 模板定义只是一种规范描述，而非真正的类型定义
  - 当编译器看到模板定义时，仅做一般性的语法检查，同时保存一份模板的内部表示，并不生成指令代码
  - 只有在编译器看到模板被实例化为具体函数或类时，才真正用模板的内部表示结合类型实参，生成代码
  - 每个C++语言的源文件是单独编译的，因此编译器仅在编译包含模板定义的源文件时保存其内部表示
  - 如果模板的实例化与它的定义不在同一个编译单元中，模板将失去被编译的机会，进而导致链接错误

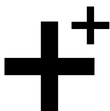


Tarena Technology 达内科技

知识讲解

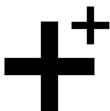
编译模型

- 包含模型
  - 必须保证编译器在编译任何有关模板实例的代码时，一定要能够看到相应模板的完整定义
  - 将模板的声明和定义放在一个头文件中，并在所有使用该模板实例的源文件中包含这个头文件
  - 出于某种原因，可能不得不把模板的定义部分放在独立于其声明头文件的源文件中，为此可以在该头文件中的声明部分之后，包含定义此模板的源文件
  - 包含模型的缺陷
    - ✓ 函数模板或类模板成员函数的定义可能很大
    - ✓ 暴露了用户不希望或不应该了解的实现细节
    - ✓ 被多个源文件包含会增加不必要的编译时间



编译模型

- 分离模型
  - 将模板的声明放在头文件中，而将模板的定义放在源文件中，且该头文件不包含该源文件
  - 使用模板实例的源文件仅包含该模板的头文件
  - 通过对模板的定义使用export关键字，即指明其为导出，意在向编译器表明此处的模板定义可能会在其它源文件中产生实例
  - 编译器对导出模板的内部表示会有更加持久的记忆，即便模板的实例化与它的定义不在同一个编译单元中，也能被正确地编译，不会导致链接错误
  - 分离模型需要更加复杂的编译处理过程，并不是所有的C++编译器都能支持

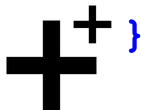


Tarena Technology 达内科技

知识讲解

编译模型

```
// utility.cpp
#include "utility.h"
export template<typename T>
T max (T a, T b) {
    return a > b ? a : b;
}

// comparator.cpp
export template<typename T> class Comparator;
#include "comparator.h"
template<typename T>
T Comparator<T>::max (void) const {
    return m_a > m_b ? m_a : m_b;
}
```



局部特化

- 针对部分类型参数取特定类型的特化
  - 编译器优先选择特化程度最高的版本

```
template<typename T1, typename T2>
class Dual { ... };
```

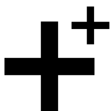
```
Dual<double, double> dual;
```

```
template<typename T1>
class Dual<T1, int> { ... };
```

```
Dual<double, int> dual;
```

```
template<>
class Dual<int, int> { ... };
```

```
Dual<int, int> dual;
```



Tarena Technology 达内科技

知识讲解

局部特化

- 针对类型参数之间某种特殊关系的特化
  - 编译器优先选择匹配程度最高的版本

```
template<typename T1, typename T2, typename T3>
class Trio { ... };
```

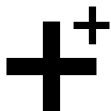
```
Trio<char, short, long> trio;
```

```
template<typename T1, typename T2>
class Trio<T1, T2, T2> { ... };
```

```
Trio<char, short, short> trio;
```

```
template<typename T1>
class Trio<T1, T1*, T1*> { ... };
```

```
Trio<char, char*, char*> trio;
```



局部特化

- 针对类型参数某些特殊属性的特化
  - 编译器优先选择针对指针和数组的特化

```
template<typename T>
class Feeb { ... };
```

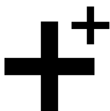
```
Feeb<char> feeb;
```

```
template<typename T>
class Feeb<T*> { ... };
```

```
Feeb<char*> feeb;
```

```
template<typename T>
class Feeb<T[]> { ... };
```

```
Feeb<char[]> feeb;
```



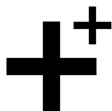
Tarena Technology 达内科技

知识讲解

非类型参数

- 除类型参数外，模板也可以接受非类型参数
- 传递给模板的非类型实参只能是常量、常量表达式或者具有常属性的变量，但不能被volatile修饰

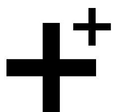
```
template<typename T, size_t S>
class Array {
...
private:
    T m_array[S];
};

Array<int, 256> array;
```



缺省参数

- 模板的参数，无论是类型参数还是非类型参数，都可以带有缺省值
- 如果某一个模板参数带有缺省值，那么该参数后面的所有参数必须都带有缺省值

```
template<typename T1, typename T2 = int,
    size_t S1 = 128, size_t S2 = 256>
class Buffer {
private:
    T1 m_buf1[S1];
    T2 m_buf2[S2]:
};
Buffer<string, double> buffer;
```



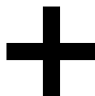
模板与继承

- 从模板继承

```
template<typename V>
class Value {
public:
  Value (const V& val) : m_val (val) {}
  V m_val;
};
```

```
template<typename K, typename V>
class Pair : public Value<V> {
public:
  Pair (const K& key, const V& val) :
    m_key (key), Value<V> (val) {}
  K m_key;
};
```

```
Pair<string, double> pair ("PAI", 3.14);
cout << pair.m_key << '=' << pair.m_val
  << endl;
```



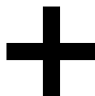
模板与继承

- 向模板派生

```
template<typename K, typename V,
  template<typename T> class Base>
class Value : public Base<K> {
public:
  Value (const K& key, const V& val) :
    Base<K> (key), m_val (val) {}
  V m_val;
};
```

```
template<typename K>
class Pair {
public:
  Pair (const K& key) : m_key (key) {}
  K m_key;
};
```

```
Value<string, double, Pair> pair ("PAI", 3.14);
cout << pair.m_key << '=' << pair.m_val
  << endl;
```



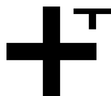
模板型成员

- 模板型成员变量

```
template<typename V>
class Value {
public:
  Value (const V& val) : m_val (val) {}
  V m_val;
};
```

```
template<typename K, typename V>
class Pair {
public:
  Pair (const K& key, const V& val) :
    m_key (key), m_val (val) {}
  K m_key;
  Value<V> m_val;
};
```

```
Pair<string, double> pair ("PAI", 3.14);
cout << pair.m_key << '=' << pair.m_val
  << endl;
```



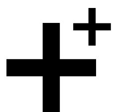
模板型成员

知识讲解

- 模板型成员函数

```
template<typename K>
class Pair {
public:
  Pair (const K& key) : m_key (key) {}
  template<typename V>
  void print (const V& val) {
    cout << m_key << '=' << val << endl;
  }
  K m_key;
};
```

```
Pair<string> pair ("PAI");
pair.print (3.14);
```



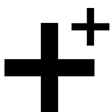
Tarena Technology 达内科技

知识讲解

模板型成员

- 模板型成员函数

```
template<typename K>
class Pair {
public:
   Pair (const K& key) : m_key (key) {}
   template<typename V>
   void print (const V& val);
   K m_key;
};
template<typename K>
   template<typename V>
void Pair<K>::print (const V& val) {
   cout << m_key << '=' << val << endl;
}
```



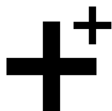
知识讲解

模板型成员

- 模板型成员类型

```
template<typename K>
class Pair {
public:
  Pair (const K& key) : m_key (key) {}
  template<typename V>
  class Value {
  public:
    Value (const V& val) : m_val (val) {}
    V m_val;
  };
  K m_key;
};
```

```
Pair<string> pair ("PAI");
Pair<string>::Value<double> value (3.14);
cout << pair.m_key << '=' << value.m_val
  << endl;
```

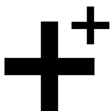


知识讲解

模板型成员

- 模板型成员类型

```
template<typename K>
class Pair {
public:
   Pair (const K& key) : m_key (key) {}
   template<typename V>
   class Value;
   K m_key;
};
template<typename K>
   template<typename V>
class Pair<K>::Value {
public:
   Value (const V& val) : m_val (val) {}
   V m_val;
};
```



知识讲解

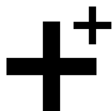
Tarena Technology 达内科技

模板递归实例化

- 模板的类型参数可以是模板的实例

```
template<typename T = int, size_t S = 3>
class Array {
public:
  T& operator[] (size_t i) {
    return m_array[i];
  }
  const T& operator[] (size_t i) const {
    return const_cast<Array&> (*this)[i];
  }
  size_t size (void) const {
    return S;
  }
private:
  T m_array[S];
};
```

```
Array<Array<string, 5>, 4> mat45;
Array<Array<> > mat33;
```



Tarena Technology 达内科技

知识讲解

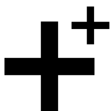
模板型模板参数

- 模板的类型参数可以又是一个模板

```
template<typename T>
class Element {
public:
  Element (const T& data) : m_data (data) {}
  T m_data;
};
```

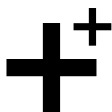
```
template<typename K, typename V,
  template<typename T> class E>
class Pair {
public:
  Pair (const K& key, const V& val) :
    m_key (key), m_val (val) {}
  E<K> m_key;
  E<V> m_val;
};
```

```
Pair<string, double, Element> pair (
  "PAI", 3.14);
cout << pair.m_key.m_data << '='
  << pair.m_val.m_data << endl;
```



typename与class

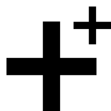
- 在C++引入模板早期，类型参数使用class关键字
  - template<class T> T max (T a, T b) { ... }
  - template<class T> Comparator { ... };
  - 出于兼容性的考虑，截至目前，这样写依然合法
- 在C++中class关键字还可以用来定义类
  - class Student { ... };
- 为了避免混淆，较晚近的C++语言标准，针对模板中的类型参数，又专门引入typename关键字
  - template<typename T> T max (T a, T b) { ... }
  - template<typename T> Comparator { ... };



typename与class

- typename关键字还可用于解决嵌套依赖类型问题
  - 以下代码无法通过编译：

```
class A {
public:
    class B {}; // 嵌套依赖类型
};
template <typename T> void foo (T a) {
    T::B b;     // 此行报错！
}
int main (void) {
    A a;
    foo (a);
}
```



typename与class

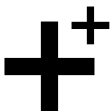
- typename关键字还可用于解决嵌套依赖类型问题
  - 当编译器看到代码行：

    ```
    T::B b;
    ```

    时并不知道T的具体类型，因此它会将B解释为一个非类型标识符，而非类型标识符是不能定义变量的
  - 为了解决这个问题，可以在嵌套依赖类型名B前面加上typename关键字：

    ```
    template <typename T> void foo (T a) {
        typename T::B b;
    }
    ```

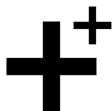
    以显式地告诉编译器B是一个类型名，可以定义变量，具体什么类型，等到模板被实例化了再说



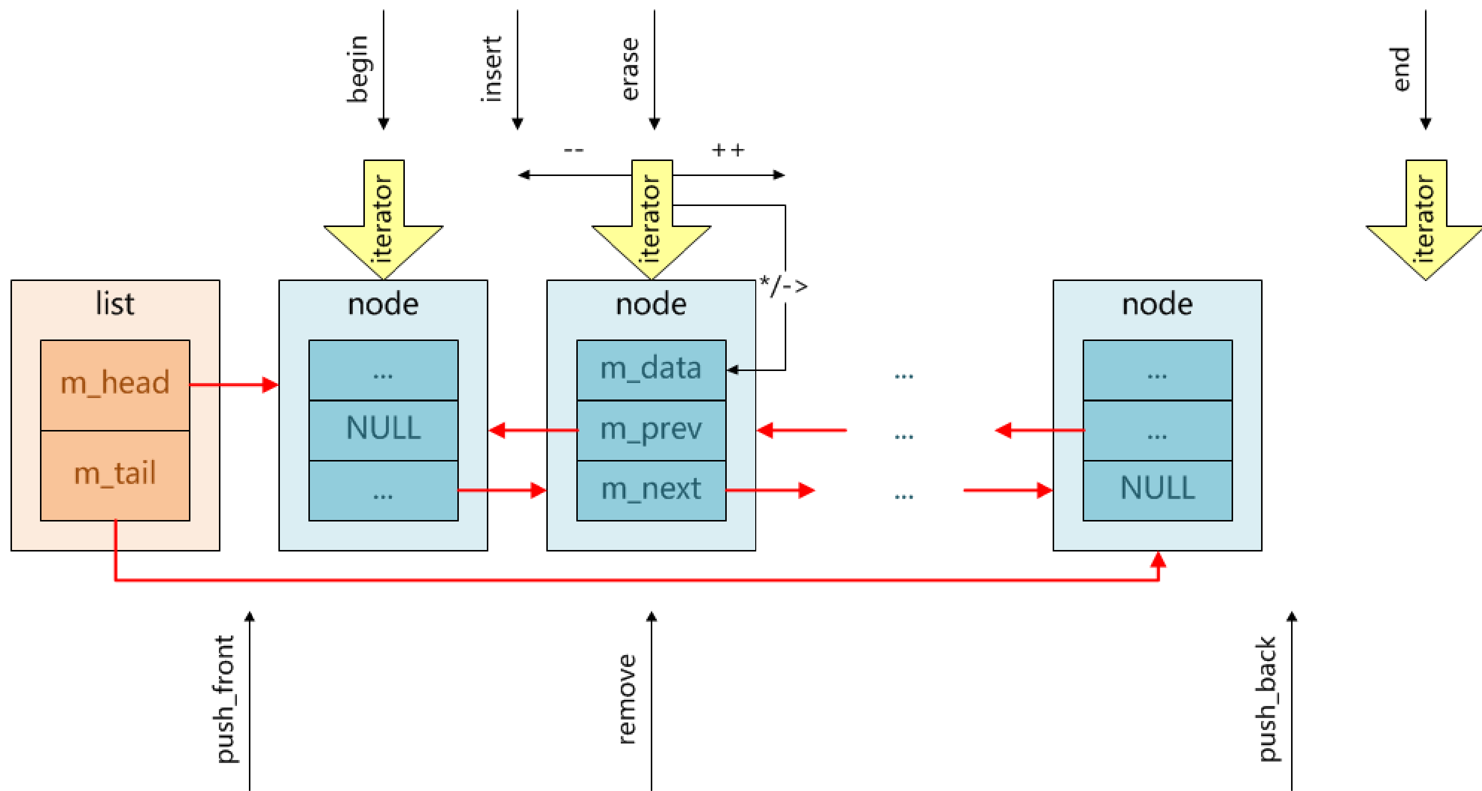
Technology Tarena 达内科技

容器与迭代器

- 支持迭代器的链表模板容器(list)
  - 直接构造为空链表
  - 支持拷贝构造和拷贝赋值，可获得完整的容器副本
  - front()/push_front()/pop_front()等接口函数
  - 四种内置迭代器类型，及基于迭代器的插入/删除函数insert()/erase()
    - ✓ 正向迭代器(list::iterator)
    - ✓ 常正向迭代器(list::const_iterator)
    - ✓ 反向迭代器(list::reverse_iterator)
    - ✓ 常反向迭代器(list::const_reverse_iterator)



容器与迭代器



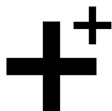
Tarena Technology 达内科技

代码实践

课堂练习

练习时间

编写一个通用型的查找函数find()，可以在任意类型的数组或list容器中，查找特定的元素



Tarena Technology
达内科技

“

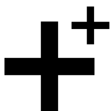
STL

”

Tarena Technology 达内科技

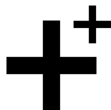
全程目标

- STL概述
- 向量
- 双端队列
- 列表
- 堆栈
- 队列
- 优先队列
- 映射
- 多重映射
- 集合
- 多重集合
- 字符串
- 内存分配器
- 全局迭代器
- 泛型算法



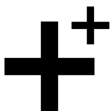
STL概述

- 标准模板库(STL)主要包括三部分
  - 容器
    - ✓ 存储和管理对象的集合
  - 迭代器
    - ✓ 在不暴露容器内部表示的前提下，遍历其中的元素
  - 泛型算法
    - ✓ 借助于迭代器，以泛型的方式处理容器内的元素
- STL的所有组件都是用模板定义的，全面支持泛型
- STL所强调的是让数据的结构和算法独立于其类型
- STL所追求的是在尽量小的框架内实现最大的弹性



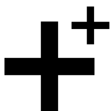
STL概述

- 十大容器
  - 线性容器(3)
    - ✓ 向量(vector)、列表(list)、双端队列(deque)
    - ✓ 元素按先后顺序呈线性排列
    - ✓ 支持某种形式的next操作，从一个元素移至下个元素
  - 适配器容器(3)
    - ✓ 堆栈(stack)、队列(queue)、优先队列(priority_queue)
    - ✓ 以线性容器作为底层容器，对其通用接口加以限制
    - ✓ 只要满足上层容器对接口的要求，也可以用自己定义的容器类型作为适配器容易的底层容器



STL概述

- 十大容器
  - 关联容器(4)
    - ✓ 映射(map)、多重映射(multimap)、集合(set)、多重集合(multiset)
    - ✓ 以key-value对为存储单元，按key的升序排列
    - ✓ 根据key存储或访问与之相关联的value
    - ✓ 通常采用二叉树的逻辑结构

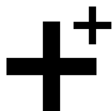


Tarena Technology 达内科技

知识讲解

STL概述

- 容器的共同特征
  - 所有容器都支持完整意义上的拷贝构造和拷贝赋值，可以将容器对象作为一个整体，用于构造或赋值给另一个同类型的容器对象
  - 一个容器等于“==”另一个容器的条件
    - ✓ 容器和容器元素的类型相同
    - ✓ 元素数量相同
    - ✓ 对应位置的元素均满足相等性“==”条件
  - 一个容器小于“<”另一个容器的条件
    - ✓ 容器和容器元素的类型相同
    - ✓ 一个容器的每个元素都小于“<”另一个容器对应位置的元素

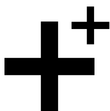


Tarena Technology 达内科技

知识讲解

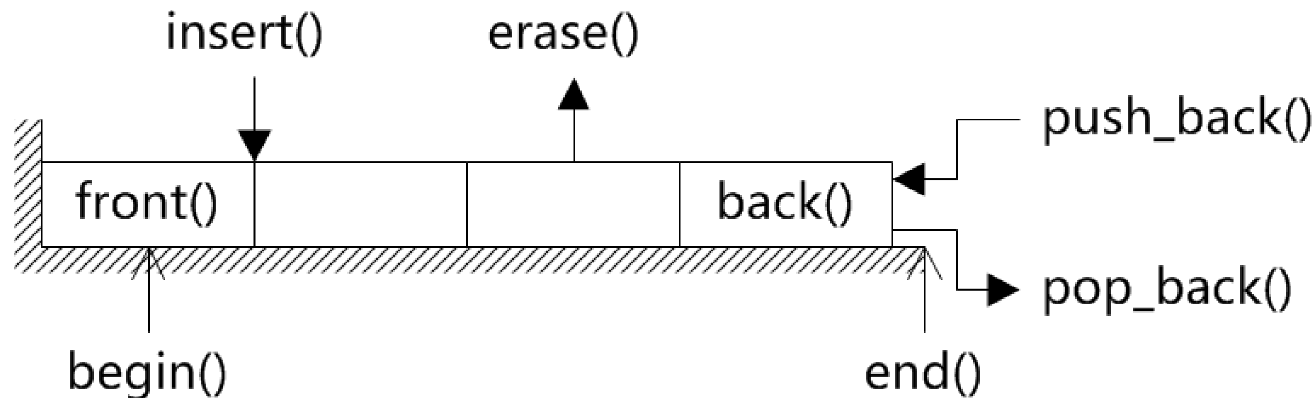
STL概述

- 容器的共同特征
  - 容器的其它关系运算符，如“>”、“>=”、“<=”和“!=”，都有类似于“==”和“<”的重载定义
  - 将对象插入容器，容器实际存储的是该对象的一份拷贝而非其本身
  - 能够被放入容器的对象类型，必须要能够支持完整意义上的拷贝构造和拷贝赋值，以实现对象间的复制
  - 智能指针(auto_ptr)，因其特有的转移语义，不能用作容器的元素

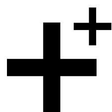


知识讲解

向量



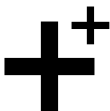
vector



Technology Tarena 达内科技

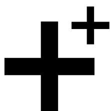
向量

- 基本特性
  - 连续内存与下标访问
    - ✓ 向量中的元素被存储在一段连续的内存空间中
    - ✓ 通过下标随机访问向量元素的效率与数组相当
  - 动态内存分配
    - ✓ 向量的内存空间会随着新元素的加入而自动增长
    - ✓ 内存空间的连续性不会妨碍向量元素的持续增加
    - ✓ 如果当前内存空间无法满足连续存储的需要，向量会自动开辟新的足够的连续内存空间，并在把原空间中的元素复制到新空间中以后，释放原空间



向量

- 基本特性
  - 通过预分配空间减小额外开销
    - ✓ 向量的增长往往伴随着内存的分配与释放、元素的复制与销毁等额外开销
    - ✓ 如果能够在创建向量时合理地为它预分配一些空间，将在很大程度上缓解这些额外开销
  - 随机访问与插入删除
    - ✓ 向量具有数组的随机访问性，在其尾部进行插入或删除，效率极高
    - ✓ 向量具有链表的灵活性，在除尾部以外的其它位置也可以进行插入或删除，但效率不高

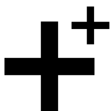


Tarena Technology 达内科技

知识讲解

向量

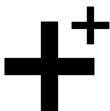
- 实例化
  - vector<元素类型> 向量对象;
    - ✓ 定义空向量
      vector<int> vi;
  - vector<元素类型> 向量对象 (初始大小);
    - ✓ 基本类型元素，用适当类型的零初始化
      vector<int> vi (10);
    - ✓ 类类型的元素，用缺省构造函数初始化
      vector<Student> vs (10);
  - vector<元素类型> 向量对象 (初始大小, 元素初值);
    - ✓ 根据指定的元素初值，初始化向量元素
      vector<double> vd (10, 1.23);



向量

- 实例化
  - vector<元素类型> 向量对象 (起始迭代器, 终止迭代器);
    - ✓ 用另一个容器起始迭代器和终止迭代器之间的元素，初始化向量中的元素

```
int a[5] = {1, 2, 3, 4, 5};
vector<int> vi (&a[0], &a[5]);
```

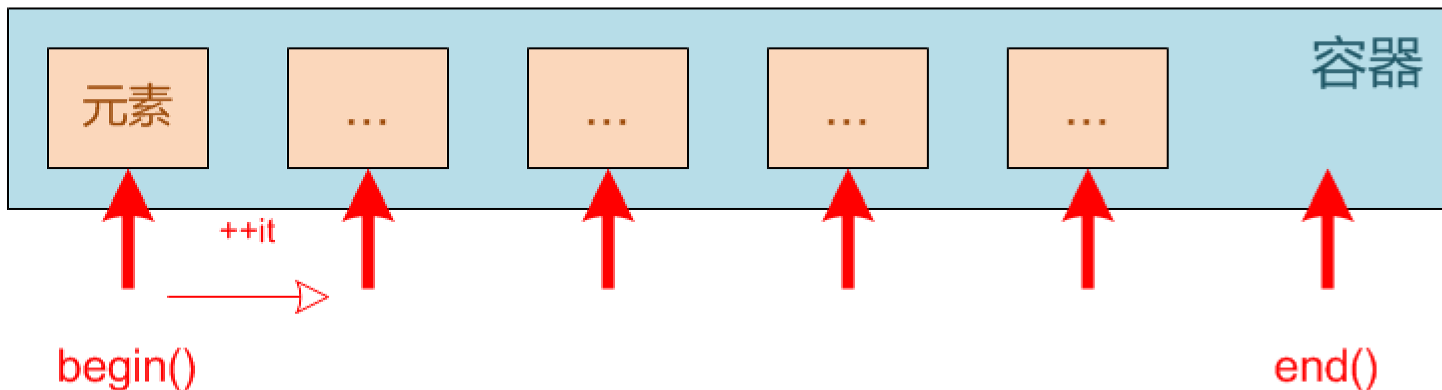


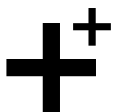
Tarena
Technology
达内科技

知识讲解

向量

- 迭代器(Iterator)



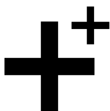


Tarena Technology 达内科技

知识讲解

向量

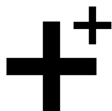
- 迭代器(Iterator)
  – 迭代器和容器的关系，相当于指针和数组的关系
  – 迭代器是一个类类型的对象，但因其重载了若干与指针一致的运算符，如*/->/++/--等，故可将其视为指向容器元素的“指针”
  – 通过迭代器，可以在完全不了解容器内部结构的前提下，以完全一致且透明的方式，访问其中的元素
  – 迭代器和指针之间的一个重要区别就是不存在值为NULL的迭代器，即不能用NULL或0初始化迭代器



向量

- 迭代器(Iterator)
  - 迭代器是容器模板的公有内置类型
    - ✓ vector<int>::iterator it;
  - 起始迭代器指向容器的第一个元素
    - ✓ vector<int>::iterator it = vi.begin ();
  - 终止迭代器指向容器最后一个元素的下一个位置
    - ✓ vector<int>::iterator it = vi.end ();
  - 通过迭代器遍历容器中的元素
    - ✓

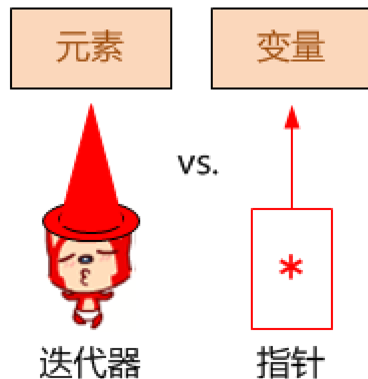
```
for (it = vn.begin (); it != vn.end (); it++)
    cout << *it << endl;
```

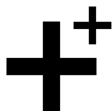


知识讲解

向量

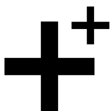
- 迭代器(Iterator)
  - 迭代器的指针操作
    - ✓ */->：访问目标元素本身或其成员
    - ✓ --/++：向前或向后移动一个位置
    - ✓ -/-=/+/+=：向前或向后移动若干位置(随机迭代器)
    - ✓ ==/!=：是否指向相同或不同的目标元素
    - ✓ </>：是否更靠前或更靠后(随机迭代器)
    - ✓ 拷贝构造/拷贝赋值：只复制迭代器本身，不复制目标元素





向量

- 迭代器(Iterator)
  - 双向迭代器与随机迭代器
    - ✓ 双向迭代器：双向行进，递增前进，递减后退
      list/map/multimap/set/multiset
    - ✓ 随机迭代器：双向行进+随机访问
      vector/deque/string
  - 正向迭代器与反向迭代器
    - ✓ 正向迭代器：递增向尾，递减向头
      iterator/const_iterator
    - ✓ 反向迭代器：递增向头，递减向尾
      reverse_iterator/const_reverse_iterator

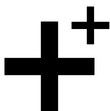


Tarena Technology
达内科技

知识讲解

向量

- 迭代器(Iterator)
  - 左值迭代器与右值迭代器
    - ✓ 左值迭代器：目标元素可修改
      iterator/reverse_iterator
    - ✓ 右值迭代器：目标元素不可改
      const_iterator/const_reverse_iterator



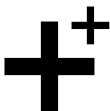
向量

- 迭代器(Iterator)
  - 迭代器的使用
    - ✓ 获取首/尾正向迭代器：begin()/end()
    - ✓ 获取首/尾反向迭代器：rbegin()/rend()
    - ✓ 从非常容器获取左值迭代器，可转换为右值迭代器

```
vector<int> vi;
vector<int>::iterator it = vi.begin ();
vector<int>::const_iterator cit = vi.begin ();
```

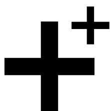
    - ✓ 从常容器获取右值迭代器，不可转换为左值迭代器

```
const vector<int>& cv = vi;
vector<int>::const_iterator cit = cv.begin ();
```



向量

- 迭代器(Iterator)
  - 迭代器的使用
    - ✓ 用两个迭代器表示容器的整体或特定区域：
下限迭代器指向容器或区域的第一个元素
上限迭代器指向容器或区域最后一个元素的下一个位置
    - ✓ 设计泛型算法时尽量避免使用针对随机迭代器的操作
    - ✓ 从某种意义上说，指针也可以被看做是一种迭代器，就象迭代器可被视作指针一样
    - ✓ 迭代器允许程序员在完全不知道容器内部结构细节的前提下，近乎透明地访问其中的数据元素，是构成STL泛型算法的基础



Tarena Technology 达内科技

知识讲解

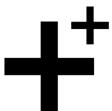
向量

- 随机访问
  - 基于迭代器的随机访问

```
vector<int> vi (5, 100);
vector<int>::iterator it = vi.begin ();
*(it + 1) = 10;
*(it + 2) += 20;
++*(it + 3);
cout << *(it + 4) << endl;
```

  - 基于下标的随机访问

```
vi[1] = 10;
vi[2] = 20;
++vi[3];
cout << vi[4] << endl;
```



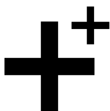
向量

- 常用成员函数
  - 获取首/尾元素

    value_type& front (void);
    const value_type& front (void) const;
    value_type& back (void);
    const value_type& back (void) const;
  - 压入/弹出元素

    void push_back (const value_type& val);
    void pop_back (void);
  - 插入/删除元素

    iterator insert (iterator loc, const value_type& val);
    iterator erase (iterator loc);

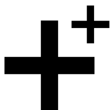


向量

- 常用成员函数
  - 获得正/反向起/止迭代器

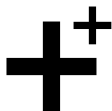
    ```
    iterator begin (void);
    const_iterator begin(void) const;
    iterator end (void);
    const_iterator end (void) const;
    reverse_iterator rbegin (void);
    const_reverse_iterator rbegin (void) const;
    reverse_iterator rend (void);
    const_reverse_iterator rend (void) const;
    ```

  - 任何可能导致容器结构发生变化的函数被调用以后，先前获得的迭代器都可能因此失效，重新初始化以后再使用才是安全的



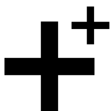
向量

- 大小和容量
  - 获取大小，即元素个数
    size_type size (void) const;
  - 改变大小，可增可减，增构造，减析构
    void resize (size_type num,
        const value_type& val = value_type ());
  - 清空向量，相当于resize (0)
    void clear (void);
  - 是否为空，空返回true，否则返回false
    bool empty (void) const;
  - 获取容量，即最多能容纳多少个元素
    size_type capacity (void) const;
  - 改变容量，只增不减，新增部分不做初始化
    void reserve (size_type size);



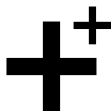
向量

- 大小和容量
  - 向量大小可增可减，改变向量大小的成员函数除resize()外，还有push_back()、pop_back()、insert()和erase()等
  - 向量容量只增不减，改变向量容量的成员函数只有reserve()
  - 向量大小的增加可能会导致其容量的增加，但向量容量的变化不会影响其大小
  - 通过resize()成员函数增加向量的大小，新增部分被初始化
  - 通过reserve()成员函数增加向量的容量，新增部分不初始化
  - 位于向量的容量范围内但不在其大小范围内的元素，也可以通过下标或迭代器访问，但其值未定义
  - 由resize()和reserve()成员函数所引起的，向量大小和容量的变化，都发生在向量的尾部



向量

- 查找和排序
  - 在特定区域内查找
    iterator find (iterator begin, iterator end,
        const value_type& val);
    iterator find (iterator begin, iterator end,
        const value_type& val, equal cmp);
    成功返回匹配元素的迭代器，失败返回上限迭代器
  - 在特定区域内排序
    void sort (iterator begin, iterator end);
    void sort (iterator begin, iterator end, less cmp);



向量

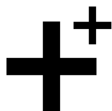
- 查找和排序
  - 等于比较器(equal)，形如：

```
bool cmp (const int& a, const int& b) {
    return a == b;
}
```

的函数，或形如：

```
class Cmp {
    bool operator () (const int& a, const int& b) {
        return a == b;
    }
};
Cmp cmp;
```

的函数对象，定义容器元素的等于规则，以供查找之需



知识讲解

Tarena Technology 达内科技

向量

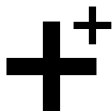
- 查找和排序
  - 小于比较器(less)，形如：

```
bool cmp (const int& a, const int& b) {
    return a < b;
}
```

的函数，或形如：

```
class Cmp {
    bool operator () (const int& a, const int& b) {
        return a < b;
    }
};
Cmp cmp;
```

的函数对象，定义容器元素的小于规则，以供排序之需

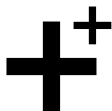


Tarena Technology 达内科技

知识讲解

向量

- 类对象的向量
  - 如果一个类类型对象需要被存储在向量中，那么该类至少应支持缺省构造，以确保向量内存的初始化
  - 该类还需要支持完整意义上的拷贝构造和拷贝赋值
  - 该类可能还需要支持“==”和“<”两个关系运算符，用于元素间的比较，其它关系运算可据此推断
  - 向量中保存的永远只是对象的副本，而非对象本身
  - 改变大小将导致对象被构造，改变容量只分配内存
  - 删除向量中的一个元素，相应类型的析构函数将被调用，同时，其后所有元素依次向前拷贝赋值一次

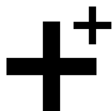


Technology
Tarena
达内科技

知识讲解

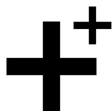
向量

- 类对象的向量
  - 两参数版本的sort()函数，通过元素类型的“<”运算符对向量进行排序
  - 三参数版本的sort()函数，通过第三个参数接受一个小于比较器，以确定元素间的比较方法，实现排序



练习时间

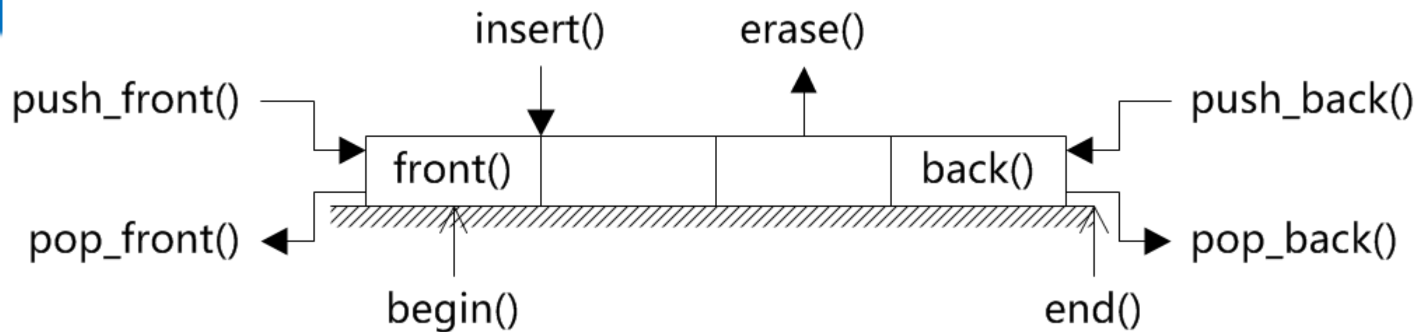
电影票房排行榜



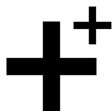
Tarena Technology 达内科技

知识讲解

双端队列

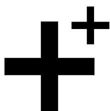


deque



双端队列

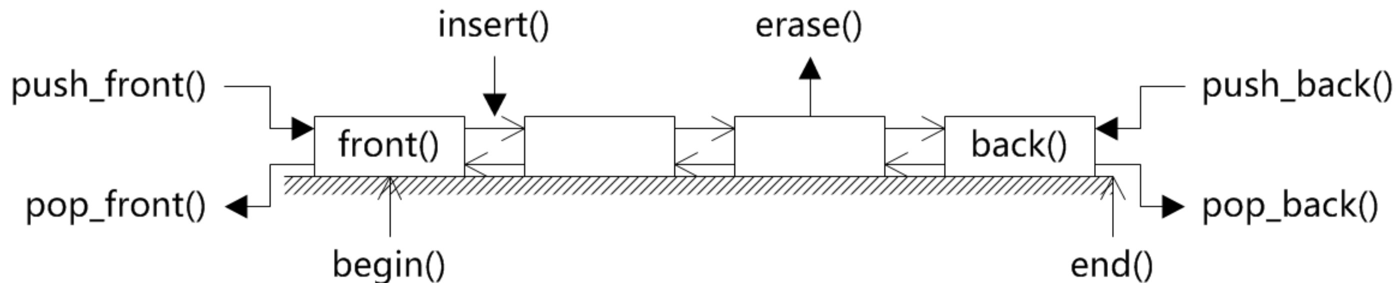
- 双端队列具有向量的所有功能，除了capacity()和reserve()
- 在双端队列的头部与在其尾部进行插入/删除(insert()/erase())的效率一样高
- 和向量相比，双端队列的内存开销要大一些，对元素下标访问的效率也要略低一些
- 双端队列所提供的push_front()/pop_front()成员函数，可以与push_back()/pop_back()成员函数相类似的方式，在容器的头部压入/弹出元素，其效率也是相当的



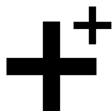
Tarena Technology
达内科技

知识讲解

列表



list

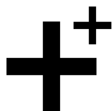


Tarena Technology 达内科技

知识讲解

列表

- 列表是按照链式线性表(链表)的形式进行存储的
- 在列表的任何位置插入/删除元素的效率都很高
- 无法通过下标或迭代器对列表中的元素做随机访问
- 常用成员函数
  - front/push_front/pop_front
  - back/push_back/pop_back
  - insert/erase
  - size/resize/clear/empty
  - begin/end/rbegin/rend

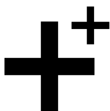


知识讲解

Tarena Technology 达内科技

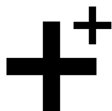
列表

- 常用成员函数
  - 删除所有匹配元素
    void remove (const value_type& val);
  - 连续重复出现的元素只保留一个
    void unique (void);
    利用“==”运算符判断相等性
    void unique (equal cmp);
    利用等于比较器判断相等性



列表

- 常用成员函数
  - 将参数列表的部分或全部元素剪切到调用列表中
    void splice (iterator pos, list& lst);
    将lst列表全部元素剪切到调用列表pos处
    void splice (iterator pos, list& lst, iterator del);
    将lst列表del处元素剪切到调用列表pos处
    void splice (iterator pos, list& lst, iterator begin,
    iterator end);
    将lst列表从begin到end之间的元素剪切到调用列表pos处

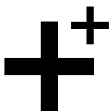


Technology Tarena 达内科技

知识讲解

列表

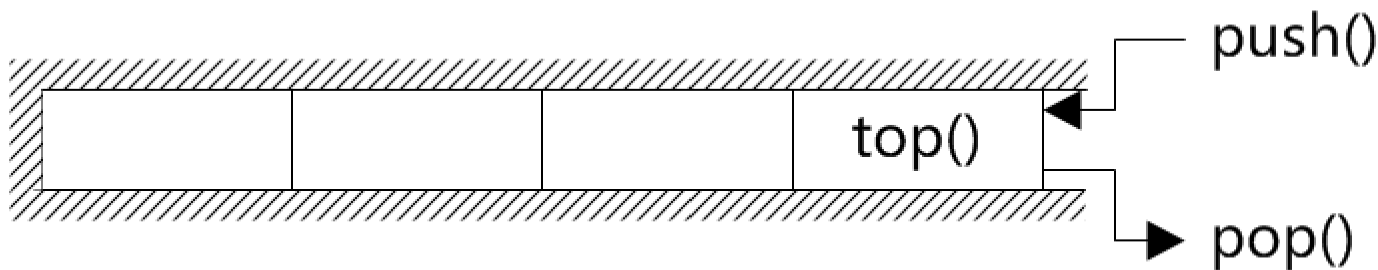
- 常用成员函数
  - 将有序参数列表合并到有序调用列表中，合并后，调用列表依然有序，参数列表为空
    void merge (list& lst);
    利用“<”运算符比大小
    void merge (list& lst, less cmp);
    利用小于比较器比大小
  - 排序
    void sort (void);
    利用“<”运算符比大小
    void sort (less cmp);
    利用小于比较器比大小



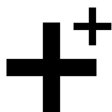
Tarena Technology 达内科技

知识讲解

堆栈

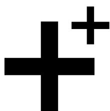


stack



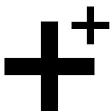
堆栈

- 堆栈作为一种适配器容器，可以用任何支持back/push_back/pop_back/size/empty 等操作的底层容器进行适配
- 除了STL的三种线性容器外，程序员自定义的容器，只要提供了上述接口函数，也可用于适配堆栈
  - value_type& top (void);
    - ✓ value_type& back (void);
  - void push (const value_type& val);
    - ✓ void push_back (const value_type& val);



堆栈

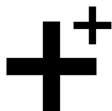
- 除了STL的三种线性容器外，程序员自定义的容器，只要提供了上述接口函数，也可用于适配堆栈
  - void pop (void);
    - ✓ void pop_back (void);
  - size_type size (void) const;
    - ✓ size_type size (void) const;
  - bool empty (void) const;
    - ✓ bool empty (void) const;



知识讲解

堆栈

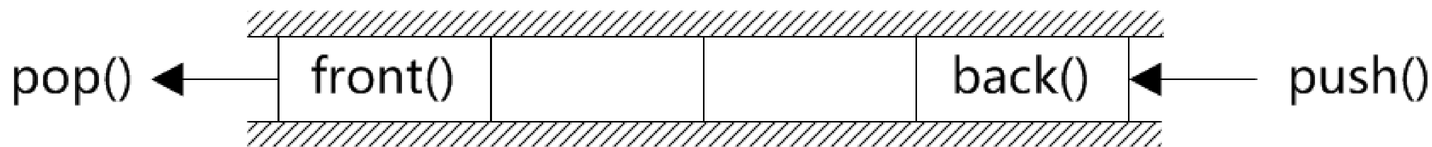
- 定义堆栈容器时可以指定底层容器的类型
  - stack<string, vector<string> > ss;
    注意两个“>”之间的至少要留一个空格，否则编译器会把“>>”理解为右移运算符，导致编译错误
- 如果没有指定底层容器，那么堆栈将使用双端队列作为其底层容器



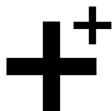
Technology Tarena 达内科技

知识讲解

队列

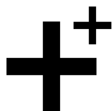


queue



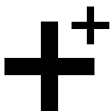
队列

- 队列作为一种适配器容器，可以用任何支持back/front/push_back/pop_front/size/empty等操作的底层容器进行适配
- 除了STL的两种线性容器外，程序员自定义的容器，只要提供了上述接口函数，也可用于适配队列
  - value_type& back (void);
    - ✓ value_type& back (void);
  - const value_type& back (void) const;
    - ✓ const value_type& back (void) const;



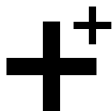
队列

- 除了STL的两种线性容器外，程序员自定义的容器，只要提供了上述接口函数，也可用于适配队列
  - value_type& front (void);
    - ✓ value_type& front (void);
  - const value_type& front (void) const;
    - ✓ const value_type& front (void) const;
  - void push (const value_type& val);
    - ✓ void push_back (const value_type& val);
  - void pop (void);
    - ✓ void pop_front (void);



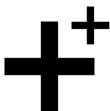
队列

- 除了STL的两种线性容器外，程序员自定义的容器，只要提供了上述接口函数，也可用于适配队列
  – size_type size (void) const;
    ✓ size_type size (void) const;
  – bool empty (void) const;
    ✓ bool empty (void) const;
- 定义队列容器时可以指定底层容器的类型
  – queue<string, list<string> > qs;
    注意两个“>”之间的至少要留一个空格，否则编译器会把“>>”理解为右移运算符，导致编译错误



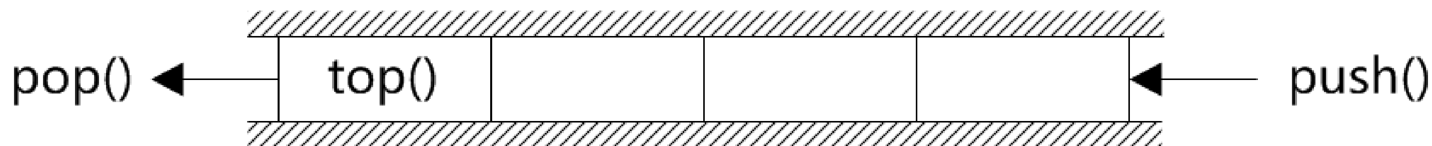
队列

- 如果没有指定底层容器，那么队列将使用双端队列作为其底层容器
- 不能用向量作为队列的底层容器，因其缺少front()和pop_front()接口

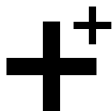


知识讲解

优先队列



priority_queue

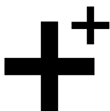


Tarena Technology 达内科技

知识讲解

优先队列

- 每当向优先队列压入一个元素后，队列中具有最高优先级的元素就被自动排列到队列的最前面
- 利用“<”运算符判断优先级，大者优先
  - priority_queue<string> ps;
  - priority_queue<string, vector<string> > ps;
- 利用小于比较器判断优先级，大者优先
  - priority_queue<string, vector<string>, Prioritize> ps;
- 优先队列的缺省底层容器是双端队列



Tarena Technology 达内科技

知识讲解

映射

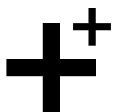
8 | value

4 | value　　12 | value

2 | value　　6 | value　　10 | value　　14 | value

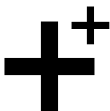
1 | value　　3 | value　　5 | value　　7 | value　　9 | value　　11 | value　　13 | value　　15 | value

map



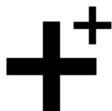
映射

- 映射是一个key-value对的序列，其中每个key都是唯一的
- 映射中所有的key-value对，按key的升序排列，以有序二叉树的形式存储
- 映射可以根据key快速地找到与之对应的value
- 利用key类型的“<”运算符比大小
  - `map<string, int> si;`
- 利用针对key的小于比较器比大小
  - `map<string, int, Less> si;`



映射

- 映射支持以key为索引的下标操作
  - value_type& operator[] (const key_type& key);
    返回与参数key相对应的value，若key不存在，则新建一个key-value对，value按缺省方式初始化
- 映射的基本访问单元为pair，pair只有两个成员变量，first和second，分别表示key和value
  -
```
map<string, int> si;
si.insert (pair<string, int> ("beijing", 1));
si.insert (make_pair ("tianjin", 2));
si["shanghai"] = 3;
map<string, int>::iterator it = si.begin ();
cout << it->first << '-' << it->second << endl;
```



知识讲解

映射

- 常用成员函数
  - 插入元素

    ```
    pair<iterator, bool> insert (
        const pair<key_type, value_type>& pair);
    ```

    插入位置由映射根据key的有序性确定
  - 删除元素

    ```
    void erase (iterator pos);
    ```

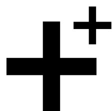
    删除指定位置的元素

    ```
    void erase (iterator begin, iterator end);
    ```

    删除指定范围的元素

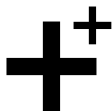
    ```
    size_type erase (const key_type& key);
    ```

    删除指定key的元素



映射

- 常用成员函数
  - 查找匹配元素，失败返回终止迭代器
    iterator find (const key_type& key);
    在映射中查找既不需要“==”运算符，也不需要等于比较器，仅支持针对key的小于比较即可



Tarena Technology 达内科技

知识讲解

多重映射

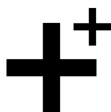
key	value
8	value

key	value	key	value
4	value	12	value

key	value	key	value	key	value	key	value
3	value	5	value	11	value	13	value

key	value	key	value	key	value	key	value	key	value	key	value	key	value	key	value
2	value	4	value	4	value	6	value	10	value	12	value	12	value	14	value

multimap

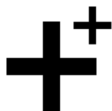


知识讲解

Tarena Technology 达内科技

多重映射

- 允许有重复key的映射
- 不支持以key为索引的下标操作
- 常用成员函数
  - 获取匹配下限，失败返回终止迭代器
    iterator lower_bound (const key_type& key);
    返回值指向第一个key大于或等于参数key的元素
  - 获取匹配上限，失败返回终止迭代器
    iterator upper_bound (const key_type& key);
    返回值指向第一个key大于参数key的元素
  - 获取匹配范围，失败返回终止迭代器对
    pair<iterator, iterator> equal_range (
    const key_type& key);



Tarena Technology 达内科技

知识讲解

集合

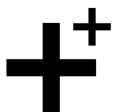
8

4 12

2 6 10 14

1 3 5 7 9 11 13 15

set

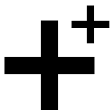


Tarena
Technology
达内科技

知识讲解

集合

- 集合可以看做是没有value的映射，其中每个元素都是唯一的
- 集合中所有的元素，按升序排列，以有序二叉树的形式存储
- 集合可以根据匹配条件快速地找到与之对应的元素
- 利用元素类型的“<”运算符比大小
  - set<string> ss;
- 利用针对元素的小于比较器比大小
  - set<string, Less> ss;

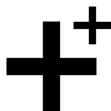


Tarena Technology 达内科技

知识讲解

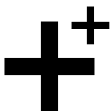
集合

- 常用成员函数
  - 插入元素
    pair<iterator, bool> insert (const key_type& key);
    插入位置由集合根据元素的有序性确定
  - 删除元素
    void erase (iterator pos);
    删除指定位置的元素
    void erase (iterator begin, iterator end);
    删除指定范围的元素
    size_type erase (const key_type& key);
    删除指定元素



集合

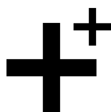
- 常用成员函数
  - 查找匹配元素，失败返回终止迭代器
    iterator find (const key_type& key);
    在集合中查找既不需要“==”运算符，也不需要等于比较器，仅支持针对元素的小于比较即可



知识讲解

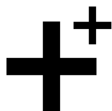
多重集合

multiset



多重集合

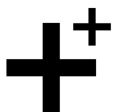
- 允许有重复元素的集合
- 常用成员函数
  - 获取匹配下限，失败返回终止迭代器
    iterator lower_bound (const key_type& key);
    返回值指向第一个大于或等于val的元素
  - 获取匹配上限，失败返回终止迭代器
    iterator upper_bound (const key_type& key);
    返回值指向第一个大于val的元素
  - 获取匹配范围，失败返回终止迭代器对
    pair<iterator, iterator> equal_range (
    const key_type& key);



知识讲解

字符串

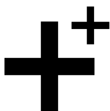
- C++中的string类型是basic_string模板的实例类
  - #include <string>
  - template<typename charT, ...>
    class basic_string { ...};
  - typedef basic_string<char> string;
- 为了支持Unicode等宽字符集，增加了wstring
  - typedef basic_string<wchar_t> wstring;
  - UC下的wchar_t为4字节，支持UCS-4字符集
  - VC下的wchar_t为2字节，支持UCS-2字符集
- 基于char的string，可以很好地支持UTF-8字符集



Technology Tarena 达内科技

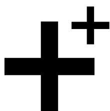
字符串

- 字符串实例化
  - 在栈中隐式实例化
    - ✓ string str;
    - ✓ string str ("");
    - ✓ string str = "";
    - ✓ string str ("Hello, World !");
    - ✓ string str = "Hello, World !";
  - 在栈中显示实例化
    - ✓ string str = string ("");
    - ✓ string str = string ("Hello, World !");



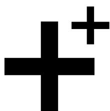
字符串

- 字符串实例化
  - 在堆中显示实例化
    - ✓ string* str = new string;
    - ✓ string* str = new string ();
    - ✓ string* str = new string ("");
    - ✓ string* str = new string ("Hello, World !");
  - 从C字符串到C++字符串
    - ✓ str = psz;
  - 从C++字符串到C字符串
    - ✓ psz = str.c_str ();



字符串

- 字符串运算符
  - 字符串拼接
    - ✓ +
  - 拷贝赋值与复合赋值
    - ✓ =/+=
  - 关系比较
    - ✓ </<=/>/>=/==/!=
  - 下标访问单个字符
    - ✓ []
  - 输入/输出
    - ✓ >>/<< (VC6不支持)

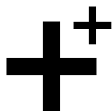


Tarena Technology 达内科技

知识讲解

字符串

- 大小和容量
  - 获取大小，即字符数
    size_type size (void) const;
  - 改变大小，可增可减
    void resize (size_type size, const char& ch = '\0');
  - 清空字符
    void clear (void);
  - 是否为空，空返回true，否则返回false
    bool empty (void) const;
  - 获取容量，即最多能容纳多少个字符
    size_type capacity (void) const;
  - 改变容量，只增不减，新增部分不做初始化
    void reserve (size_type size);

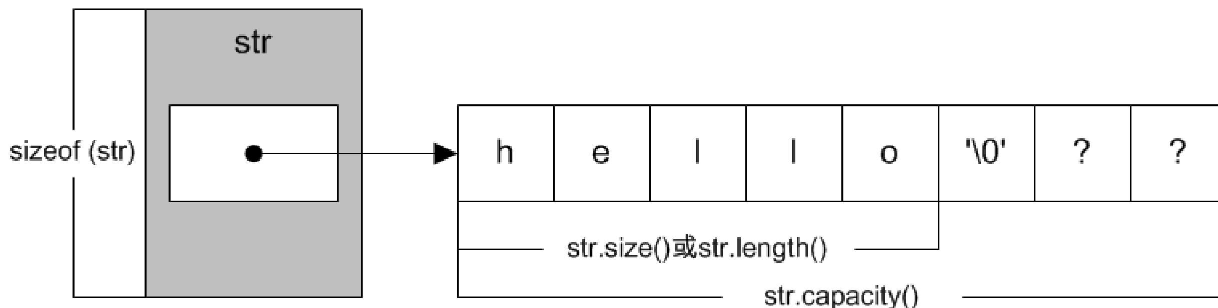


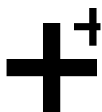
知识讲解

Tarena Technology 达内科技

字符串

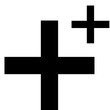
- 大小和容量
  - 获取长度，即字符数，同size()
    size_type length (void) const;
  - 注意字符串的大小和容量，以及与字符串对象大小之间的区别





字符串

- 大小和容量
  - 某些C++实现，string实际上只包含一个char*类型的非静态成员变量，该变量保存一个以'\0'字符结尾的字符数组首地址。因此，一个string对象的大小，实际上就是它唯一一个非静态成员变量的大小，4个字节(32位系统)
  - 另一些C++实现，string还包括一个unsigned int型的非静态成员变量，用于保存字符串的长度。这种情况下，其指针成员变量所指向的字符数组就不再需要'\0'字符结尾了。而string对象的大小，是它两个非静态成员变量大小之和，8个字节(32位系统)



知识讲解

Tarena Technology 达内科技

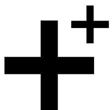
字符串

- 拼接、搜索和子串
  - 字符串拼接

```
string& append (const string& str);
string& append (const string& str, size_type pos,
        size_type len);
string& append (size_type num, char ch);
```

  - 搜索字符，失败返回string::npos

```
size_type find_first_of (char ch, size_type pos = 0);
size_type find_first_of (const string& str,
        size_type pos = 0);
size_type find_first_not_of (char ch, size_type pos = 0);
size_type find_first_not_of (const string& str,
        size_type pos = 0 );
```

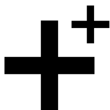


Tarena Technology 达内科技

知识讲解

字符串

- 拼接、搜索和子串
  - 搜索字符，失败返回string::npos
    size_type find_last_of (char ch, size_type pos = npos);
    size_type find_last_of (const string& str,
        size_type pos = npos);
    size_type find_last_not_of (char ch, size_type pos = npos);
    size_type find_last_not_of (const string& str,
        size_type pos = npos);
  - 截取子串
    string substr (size_type pos, size_type len = npos);
    string (const string& str, size_type pos,
        size_type len = npos);

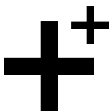


Tarena Technology 达内科技

知识讲解

字符串

- 单字符访问
  - char& at (size_type index);
    const char& at (size_type index) const;
    下标溢出抛out_of_range异常
  - char& operator[] (size_type index);
    不检查下标溢出
- 查找与替换
  - size_type find (const string& str, size_type pos);
    size_type rfind (const string& str, size_type pos);
    失败返回string::npos
  - string& replace (size_type pos, size_type len,
        const string& str);

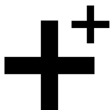


Technology Tarena 达内科技

知识讲解

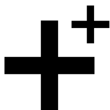
字符串

- 比较与排序
  - int compare (const string& str);
    调用对象</=/>参数对象，返回值</=/>0
  - void qsort (void* base, size_t nmemb, size_t size,
        int (*compar) (const void*, const void*));
- 插入与删除
  - string& insert (size_type pos, const string& str);
  - string& erase (size_type pos = 0,
        size_type len = npos);



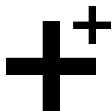
字符串

- 交换与复制
  - void swap (container& from);
  - string& assign (const string& str);
  - string& assign (const string& str, size_type pos, size_type len);
- 其它
  - begin/end/rbegin/rend
  - push_back
  - c_str/data
  - copy



内存分配器

- STL容器通过内存分配器分配/释放内存空间，构造/析构元素对象
- 每个STL容器都有一个缺省内存分配器allocator，一般情况下已足够适用
- 在某些特殊情况下，程序员也可以自己定义内存分配器，完成某些特殊的操作
- 为了让STL容器使用自定义内存分配器，需要在其模板实参中显示指明所用内存分配器的名称
  - vector<int, MyAllocator<int> > vi;
  - map<string, int, less<string>,
MyAllocator<pair<string, int> > > si;

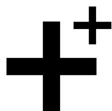


知识讲解

内存分配器

- 自定义内存分配器
  – 模板框架

```
template<typename T>
class MyAllocator {
public:
    // ...
};
```



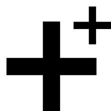
知识讲解

Tarena Technology 达内科技

内存分配器

- 自定义内存分配器
  - 类型定义

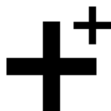
```
typedef size_t      size_type;
typedef ptrdiff_t difference_type;
typedef T*          pointer;
typedef const T*    const_pointer;
typedef T&          reference;
typedef const T&    const_reference;
typedef T           value_type;
```



内存分配器

- 自定义内存分配器
  - 非元素类型内存分配器

```
template<typename U>
class rebind {
public:
    typedef MyAllocator<U> other;
};
```



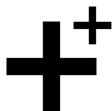
知识讲解

Tarena Technology 达内科技

内存分配器

- 自定义内存分配器
  - 构造/析构函数

```
MyAllocator (void) throw ();
MyAllocator (const MyAllocator& that)
    throw ();
template<typename U>
MyAllocator (const MyAllocator<U>& that)
    throw ();
~MyAllocator (void) throw ();
```



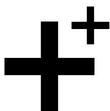
Tarena Technology 达内科技

知识讲解

内存分配器

- 自定义内存分配器
  - 获取元素地址和最大容量

```
pointer address (reference value) const;
const_pointer address (
    const_reference value) const;
size_type max_size (void) const throw ();
```



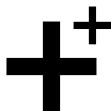
Tarena Technology 达内科技

知识讲解

内存分配器

- 自定义内存分配器
  - 分配/释放内存

```
pointer allocate (size_type num,
    const void* hint = 0);
void deallocate (pointer p,
    size_type num);
```



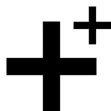
Tarena Technology 达内科技

知识讲解

内存分配器

- 自定义内存分配器
  - 构造/析构元素

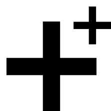
```
void construct (pointer p,
    const_reference value);
void destroy (pointer p);
```



内存分配器

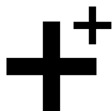
- 自定义内存分配器
  - 相等/不等运算符

```
template<typename T1, typename T2>
bool operator== (
    const MyAllocator<T1>& a,
    const MyAllocator<T2>& b) throw ();
template<typename T1, typename T2>
bool operator!= (
    const MyAllocator<T1>& a,
    const MyAllocator<T2>& b) throw ();
```



全局迭代器

- I/O流迭代器
  - istream_iterator
  - ostream_iterator
- 插入迭代器
  - front_insert_iterator
  - back_insert_iterator
  - insert_iterator

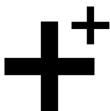


Tarena Technology 达内科技

知识讲解

泛型算法

- STL算法主要完成查找、排序、计数、合并、填充、比较、变换、删除以及划分等任务
- STL算法通常以迭代器作为参数，无需了解所操作容器的内部细节，因此STL算法又被称为泛型算法
- 一个泛型算法能否应用于一种容器，完全取决于该容器是否支持这个算法所需要的迭代器类型
- STL中共有60种算法，包括23种非修改算法，如find()，和37种修改算法，如sort()
- 通常涉及比大小和判断相等的泛型算法，都会同时提供基于运算符的和基于比较器的，两种解决方案

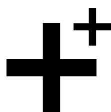


知识讲解

Tarena Technology 达内科技

泛型算法

- 泛型算法举例
  - 复制
    void copy (src_iterator begin, src_iterator end,
            dst_iterator dst);
  - 打印
    void print (iterator begin, iterator end);
  - 遍历
    void for_each (iterator begin, iterator end,
            unary_function fun);
  - 排序
    void sort (iterator begin, iterator end);
    void sort (iterator begin, iterator end, less cmp);



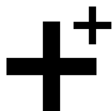
Technology Tarena 达内科技

代码实践

课堂练习

练习时间

验证自定义的泛型排序算法函数sort()是否适用于前面实现的链表容器list，体会迭代器在泛型算法中的作用



Tarena Technology 达内科技

“ 再见 ”