# CS100 Recitation 12

GKxx

## Contents

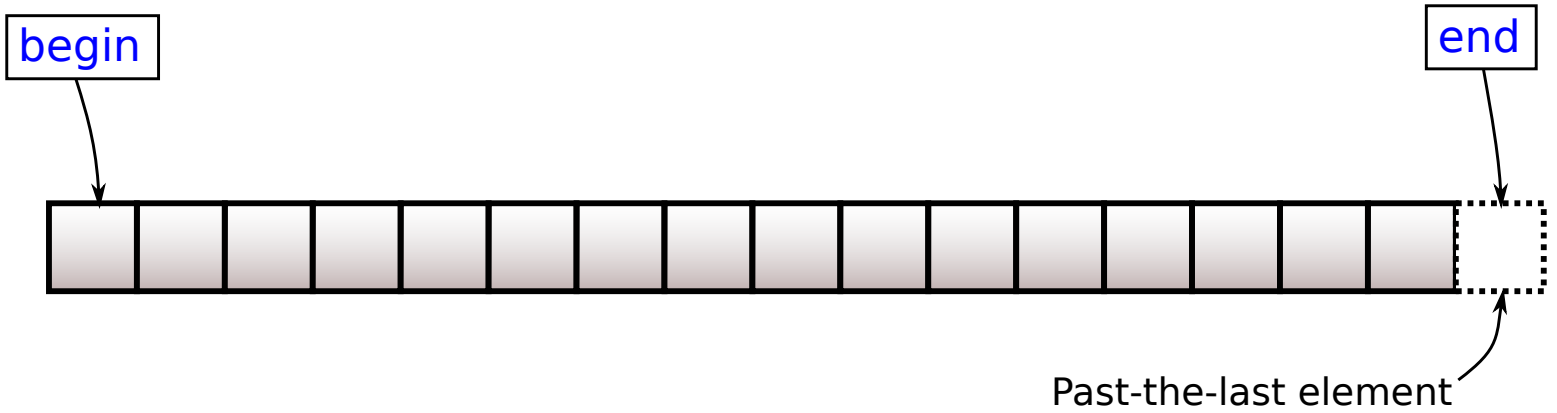
- STL 迭代器和算法
- 继承、多态

# STL 迭代器和算法

# 迭代器

一种用来访问容器中的元素的广义“指针”。

- **每个容器**（以及 `std::string`）都有迭代器，其类型为 `Container::iterator`
- `c.begin()` 返回“首迭代器”：指向开头
- `c.end()` 返回“尾后迭代器”：指向**尾后** (off-the-end) 位置
- 通常用一对迭代器表示一个**左闭右开区间**： `[begin, end)`



# 迭代器

并非所有容器都像 `std::vector` 或数组这样连续存储、支持用下标访问。

- 链表，二叉树，哈希表……

“广义指针”：扩展了指针的“访问元素”和“遍历序列”的功能，在不同的数据结构上 `++iter` 会采用不同的方式“前进到下一位置”。

## 迭代器的基本操作

所有迭代器类型都支持以下操作：

- `*it` ：返回 `it` 所指向的元素的**引用**
- `it->mem` ：等价于 `(*it).mem`
- `++it`, `it++` ：将 `it` 前进一步。
- `it1 == it2`, `it1 != it2` ：检查 `it1` 和 `it2` 是否指向相同的位置。

# 迭代器型别 (Iterator categories)

一个容器的迭代器型别归根到底取决于这个容器的性质。

- **ForwardIterator** 前向迭代器：支持基本操作。
- **BidirectionalIterator** 双向迭代器：在前向迭代器的基础上还支持 `--it` 和 `it--` 。
- **RandomAccessIterator** 随机访问迭代器：在双向迭代器的基础上还支持算术运算和大小比较（类似于指针）
  - `it + n` , `n + it` , `it - n` , `it += n` , `it -= n` , `it[n]` , `it1 - it2` , `<` , `<=` , `>` , `>=`
  - `std::string::iterator` 和 `std::vector<T>::iterator` 都是随机访问迭代器

数组 `T[N]` 当然可以被视为一种容器，其迭代器是 `T*` ，也是随机访问迭代器。

实际上还有比 ForwardIterator 更低级的 InputIterator ，以及特殊的 OutputIterator 。

# 从迭代器范围初始化

`std::string` 以及大多数容器都支持从一个迭代器范围初始化：

```
std::vector<char> v = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'};
std::vector v2(v.begin() + 2, v.end() - 3);  // {'c', 'd', 'e', 'f'}
std::string s(v.begin(), v.end());           // "abcdefghi"
```

- CTAD 同样能发挥作用： `v2` 的类型是 `std::vector<char>` ，这里的元素类型 `char` 是根据我们提供的迭代器推断出来的。

# 基于范围的 `for` 语句

基于范围的 `for` 语句本质上是用**迭代器**遍历：

```
std::vector<int> v = something();
for (auto x : v) {
  do_something_with(x);
}
```

```
std::vector<int> v = something();
{
  auto begin = v.begin();
  auto end = v.end();
  for (; begin != end; ++begin) {
    do_something_with(*begin);
  }
}
```

* 如何让 `Dynarray` 也能用基于范围的 `for` 语句遍历？

# 标准库算法

通常接受一对迭代器表示范围 `[begin, end)`

```
std::sort(a, a + N);                  // a 可能是一个数组
std::copy(v.begin(), v.end(), a); // v 可能是一个 std::vector<...>
```

带 `_n` 后缀的函数使用 `[begin, begin + n)`

```
class Dynarray {
public:
  Dynarray(const Dynarray &other)
      : m_length{other.size()}, m_storage{new int[other.size()]} {
    std::copy_n(other.m_storage, other.size(), m_storage);
  }
  Dynarra(std::size_t n, int x) : m_length{n}, m_storage{new int[n]} {
    std::fill_n(m_storage, m_length, x);
  }
};
```

# 标准库算法

通常用迭代器表示位置

```
auto pos = std::find(v.begin(), v.end(), val); // pos 指向 val 第一次出现的位置
auto maxPos = std::max_element(v.begin(), v.end()); // maxPos 指向最大值所在的位置
```

部分算法对迭代器型别有要求，比如

- `std::sort` 接受 RandomAccessIterator ，比较严格。
- `std::copy` 接受任何 InputIterator 作为前两个参数，要求第三个参数是 OutputIterator 。

部分算法对元素类型有要求，比如

- `std::sort` 要求元素类型具有 `<` 运算符。
- `std::equal` 要求元素类型具有 `==` 运算符。

# 算法不修改容器

标准库算法绝不会改变容器的大小（除非传给它的迭代器是某些特殊的迭代器适配器）

例如，`std::copy(begin, end, dest)` 只是**复制**元素，但并不向容器**插入**元素！

```
std::vector<int> a = someValues();
std::vector<int> b(a.size());
std::vector<int> c{};
std::copy(a.begin(), a.end(), b.begin()); // 正确
std::copy(a.begin(), a.end(), c.begin()); // 未定义行为!
```

某种神秘的迭代器适配器：它的 `++` 和赋值操作十分不同寻常，会将 `*iter++ = x` 变为 `c.push_back(x)`

```
std::copy(a.begin(), a.end(), std::back_inserter(c)); // 这是可以的
```

# 一些常见算法

- 不修改序列：`count`，`find`，`find_end`，`find_first_of`，`search` 等
- 修改序列：`copy`，`fill`，`reverse`，`unique` 等
- 划分、排序、归并：`partition`，`sort`，`nth_element`，`merge` 等
- 二分查找：`lower_bound`，`upper_bound`，`binary_search`，`equal_range`
- 堆相关：`make_heap`，`push_heap`，`pop_heap`，`sort_heap` 等
- 大小比较：`min` / `max`，`min_element` / `max_element`，`equal`，`lexicographical_compare` 等
- 数值运算（`<numeric>`）：`accumulate`，`inner_product` 等

# 谓词 (Predicate)

许多算法接受一个谓词 (predicate)，即一个返回 `bool` 的**可调用对象**，来定制操作。

- 需要比较元素的算法通常默认采用 `<` 进行比较，但也提供接受一个二元谓词 `cmp` 的版本，用 `cmp(a, b)` 代替 `a < b` 。
  - `std::sort(begin, end, cmp)`
  - `std::max_element(begin, end, cmp)`
- 带有 `_if` 后缀的函数接受一个一元谓词 `cond` ，只关心那些 `cond(element)` 为真的元素 `element` 。
  - `std::find_if(b, e, [](int x) { return x % 2 == 0; })` 找第一个偶数
  - `std::copy_if(b, e, d, [](int x) { return x % 2 == 0; })` 只拷贝偶数

# 可调用对象

C++ 中的可调用对象有函数、函数指针、类型为重载了调用运算符 `operator()` 的类类型的对象。

Lambda 本质上是让编译器帮你合成一个重载了 `operator()` 的类型并创建一个该类型的对象。

```
auto cmp =
    [](const std::string &a,
       const std::string &b) {
         return a.size() < b.size();
       };
```

```
struct LambdaType_cmp {
  bool operator()(const std::string &a,
                  const std::string &b) {
    return a.size() < b.size();
  }
};
LambdaType_cmp cmp;
```

# 继承 (Inheritance)、多态 (Polymorphism)

## 继承

- 一个子类对象里一定有一个完整的父类对象*
- 禁止“坑爹”：继承不能破坏父类的封装性

# 继承

- 一个子类对象里一定有一个完整的父类对象*
  - 父类的**所有成员**（除了构造函数和析构函数）都被继承下来，无论能否访问
  - 子类的构造函数必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员
  - 子类的析构函数在析构了自己的成员之后，必然调用父类的析构函数
- 禁止“坑爹”：继承不能破坏父类的封装性
  - 子类不可能改变继承自父类成员的访问限制级别
  - 子类不能随意初始化父类成员，必须经过父类的构造函数
    - 先默认初始化后赋值是可以的

* 除非父类是空的，这时编译器会做“空基类优化 (Empty Base Optimization, EBO)”。

# 子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员

```
class DiscountedItem : public Item {
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double discount)
      : Item(name, price), m_minQuantity(minQ), m_discount(discount) {}
};
```

可以在初始值列表里调用父类的构造函数

- 如果没有调用？

# 子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员

```
class DiscountedItem : public Item {
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double discount)
      : Item(name, price), m_minQuantity(minQ), m_discount(discount) {}
};
```

可以在初始值列表里调用父类的构造函数

- 如果没有调用，则自动调用父类的默认构造函数
    - 如果父类不存在默认构造函数，则报错。

合成的默认构造函数的行为？

# 子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员

```
class DiscountedItem : public Item {
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double discount)
      : Item(name, price), m_minQuantity(minQ), m_discount(discount) {}
};
```

可以在初始值列表里调用父类的构造函数

- 如果没有调用，则自动调用父类的默认构造函数
    - 如果父类不存在默认构造函数，则报错。

合成的默认构造函数：先调用父类的默认构造函数，再 ……

## 子类的析构函数

析构函数在执行完函数体之后：

- 先按成员的声明顺序倒序销毁所有成员。对于含有 non-trivial destructor 的成员，调用其 destructor。
- 然后调用父类的析构函数销毁父类的部分。

# 子类的拷贝控制

自定义：不要忘记拷贝/移动父类的部分

```
class Derived : public Base {
public:
  Derived(const Derived &other) : Base(other), /* ... */ { /* ... */ }
  Derived &operator=(const Derived &other) {
    Base::operator=(other);
    // ...
    return *this;
  }
};
```

合成的拷贝控制成员（不算析构）的行为？

## 子类的拷贝控制

合成的拷贝控制成员（不算析构）：先父类，后子类自己的成员。

- 如果这个过程中调用了任何不存在/不可访问的函数，则合成为 implicitly deleted

# 动态绑定

向上转型：

- 一个 `Base *` 可以指向一个 `Derived` 类型的对象
  - `Derived *` 可以向 `Base *` 类型转换
- 一个 `Base &` 可以绑定到一个 `Derived` 类型的对象
- 一个 `std::shared/unique_ptr<Base>` 可以指向一个 `Derived` 类型的对象
  - `std::shared/unique_ptr<Derived>` 可以向 `std::shared/unique_ptr<Base>` 类型转换

# 虚函数

继承父类的某个函数 `foo` 时，我们可能希望在子类提供一个新版本（override）。

我们希望在一个 `Base *`, `Base &` 或 `std::shared/unique_ptr<Base>` 上调用 `foo` 时，可以根据**动态类型**来选择正确的版本，而不是根据静态类型调用 `Base::foo`。

在子类里 override 一个虚函数时，函数名、参数列表、 `const` ness 必须和父类的那个函数**完全相同**。

- 返回值类型必须**完全相同**或者随类型 *协变* (covariant)。

加上 `override` 关键字：让编译器帮你检查它是否真的构成了 override

# 虚函数

**除了 override，不要以其它任何方式定义和父类中某个成员同名的成员。**

- 阅读以下章节，你会看到违反这条规则带来的后果。

  《Effective C++》条款 33：避免遮掩继承而来的名称

  《Effective C++》条款 36：绝不重新定义继承而来的 non-`virtual` 函数

  《Effective C++》条款 37：绝不重新定义继承而来的缺省参数值

## 纯虚函数

通过将一个函数声明为 `=0`，它就是一个**纯虚函数** (pure virtual function)。

一个类如果有某个成员函数是纯虚函数，它就是一个**抽象类**。

- 不能定义抽象类的对象，不能调用无定义的纯虚函数*。

* 事实上一个纯虚函数仍然可以拥有一份定义，阅读《Effective C++》条款 34（**必读**）

## 纯虚函数

纯虚函数通常用来定义**接口**：这个函数在所有子类里都应该有一份自己的实现。

如果一个子类继承了某个纯虚函数而没有 override 它，这个成员函数就仍然是纯虚的，这个类仍然是抽象类，无法被实例化。

# 运行时类型识别 (RTTI)

`dynamic_cast` 可以做到"向下转型"：

- 它会在运行时检测这个转型是否能成功
- 如果不能成功，`dynamic_cast<T*>` 返回空指针，`dynamic_cast<T&>` 抛出 `std::bad_cast` 异常。
- **非常非常慢**，你几乎总是应该先考虑用一组虚函数来完成你想要做的事。

`typeid(x)` 可以获取表达式 `x` （忽略顶层 `const` 和引用后）的动态类型信息

- 通常用 `if (typeid(*ptr) == typeid(A))` 来判断这个动态类型是否是 `A`。

https://quick-bench.com/q/E0LS3gJgAHlQK0Em_6XzkRzEjnE

根据经验，如果你需要获取某个对象的动态类型，通常意味着设计上的缺陷，你应当修改设计而不是硬着头皮做 RTTI。

# 设计

`public` 继承建模出“is-a”关系：A discounted item **is an** item.

但有些时候英语上的“is-a”具有欺骗性：

- Birds can fly. A penguin is a bird.
- A square is a rectangle. 但矩形的长宽可以随意更改，而正方形不可以。

阅读《Effective C++》条款 32（**必读**）。

## 继承的访问权限

这个 `public` 是什么意思？

```
class DiscountedItem : public Item {};
```

# 继承的访问权限

```
class DiscountedItem : public Item {};
```

继承的访问权限：**这个继承关系（“父子关系”）是否对外公开**。

如果采用 `private` 继承，则在外人眼里他们不是父子，任何依赖于这一父子关系的行为都将失败（有哪些？）。

# 继承的访问权限

```
class DiscountedItem : public Item {};
```

继承的访问权限：**这个继承关系（“父子关系”）是否对外公开**。

如果采用 `private` 继承，则在外人眼里他们不是父子，任何依赖于这一父子关系的行为都将失败。

- 访问继承而来的成员（本质上也是向上转型）
- 向上转型（包括动态绑定等等）、向下转型

`private` 继承：建模“is-implemented-in-terms-of”。阅读《Effective C++》条款 38、39（选读）。

# 继承的访问权限

```
struct A : B {}; // public inheritance
class C : B {};  // private inheritance
```

`struct` 和 `class` 仅有两个区别：

- 默认的成员访问权限是 `public` / `private` 。
- 默认的继承访问权限是 `public` / `private` 。