3-Container and Dynamic Memory

1. Range-Based For Loops

range-based for loop是一个语法糖，用于遍历支持迭代器遍历的序列，它的语法如下：

```
for (<type> <variable> : <sequence>)
    <body>
```

比如，

```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int item : vec) {
    cout << item << endl;
}
```

编译器会将上述代码转换成迭代器遍历：

```
vector<int> vec = {1, 2, 3, 4, 5};
for (auto it = vec.begin(); it != vec.end(); ++it) {
    int item = *it;
    cout << item << endl;
}
```

上面代码中， `item` 的值是序列元素的拷贝，如果想要修改序列中的元素，我们需要使用引用。比如，将 `vec` 中的所有元素修改为 `42` ：

```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int &item : vec) {
    item = 42;
}
```

它等价于：

```cpp
vector<int> vec = { 1, 2, 3, 4, 5 };
for (auto it = vec.begin(); it != vec.end(); ++it) {
    int &item = *it;
    item = 42;
}
```

2. Arrays

🏅 An *array* is simple collection of objects, built into C++ and many other languages. An array has the following properties:

- It has a fixed size, set when the array is created. This size never changes as long as the array is alive.
- An array holds *elements* that are of the same type.
- The elements of an array are stored in a specific order, with the index of the first element being 0.
- The elements are stored contiguously in memory, one after another.
- Accessing any element of an array takes constant time, regardless of whether the element is at the beginning, middle, or end of the array.

🏅 If a program dereferences a pointer that goes past the bounds of the array, the result is undefined behavior.

There are two general strategies for keeping track of where an array ends.

1. Keep track of the length separately from the array. This can be done with either an integer size or constructing a pointer that is just past the end of an array (by just adding the size of the array to a pointer to the array’ s first element).
2. Store a special *sentinel value* at the end of the array, which allows an algorithm to detect that it has reached the end

3. Array-based Containers

🏅 A *container* is an abstract data type whose purpose is to hold objects of some other type.

接下来，让我们以数组为构建块，实现集合这个ADT。

🏅 In designing our ADT, we will use the following process:

1. Determine which operations our ADT will support.
2. Define the ADT’ s public interface, including both function signatures and documentation.
3. Write some code that uses our ADT. This will help us figure out what the right interface should be, and it will serve as a test case. (Ideally, we would also write unit tests for each part of the public interface, following the pattern of test-driven development.)
4. Come up with a data representation. Define member variables and determine the representation invariants.
5. Write definitions for the ADT’ s constructors and member functions, making sure that they adhere to the representation invariants.

3.1 Set Operations

🏅 The following are the operations our set will support:

- Creating an empty set.
- Inserting a value into a set. We will allow an existing item to be inserted – it will just do nothing.
- Removing a value from a set. We will allow a nonexistent item to be removed – it will just do nothing.
- Check if a value is contained in a set.
- Count the number of items in a set.
- Print a character representation of a set to an output stream.

3.2 Public Interfaces

```
class IntSet {
  public:
    // Maximum size of a set.
    static const int MAX_SIZE = 10; // 编译时常量，能确定数组的大小

    // EFFECTS:  Initializes this set to be empty.
    IntSet();

    // REQUIRES: size() < MAX_SIZE
    // MODIFIES: *this
    // EFFECTS:  Adds value to the set, if it isn't already in the set.
    void insert(int value);

    // MODIFIES: *this
    // EFFECTS:  Removes value from the set, if it is in the set.
    void remove(int value);

    // EFFECTS:  Returns whether value is in the set.
    bool contains(int value) const;

    // EFFECTS:  Returns the number of elements.
    int size() const;

    // EFFECTS:  Prints out the set in an arbitrary order.
    void print(std::ostream &os) const;
};
```

3.3 Code Example

```
#include "intset.h"
#include <iostream>
using namespace std;
int main() {
    IntSet set;
    set.insert(7);
    set.insert(32);
    set.insert(32);

    cout << "Size: " << set.size() << endl; // prints 2
    set.print(cout);                        // prints { 7, 32 } in some arbitrary
                                            // order

    set.insert(42);
    set.remove(32);
    set.remove(32);

    cout << "Contains 32? " << set.contains(32) << endl; // prints 0 (false)
    cout << "Contains 42? " << set.contains(42) << endl; // prints 1 (true)
}
```

3.4 Data Representation

在底层，我们使用数组来表示集合，每个集合的大小为 `MAX_SIZE` ，因为是数组，所以我们可以选择一个变量来跟踪是否到达它的边界。

```cpp
class IntSet{
    ...
private:
    int elements[MAX_SIZE];
    int num_elements;
}
```

确定好数据表示后，我们需要弄清楚哪些值对应于有效的集合，我们使用不变式来指定有效值集合。

对于 `num_elements` 来说，它得是非负的，又因为我们只有 `MAX_SIZE` 的空间来存储数据，所以 `num_elements` 不能超过 `MAX_SIZE` 。所以不变式就是 `0 <= num_elements <= MAX_SIZE` 。

对于 `elements` ，第一个不变式就是数组的前 `num_elements` 项是集合中的元素；又因为集合禁止重复元素，所以另一个不变式是数组的前 `num_elements` 项中没有重复元素。

综上，

```cpp
class IntSet{
    ...
private:
    int elements[MAX_SIZE];
    int num_elements;
    // INVARIANTS:
    // 0 <= num_elements && num_elements <= MAX_SIZE
    // the first num_elements items in elements are the items in the set
    // the first num_elements items in elements contain no duplicates
}
```

3.5 Implementation

为了维护不变式，我们需要将 `num_elements` 初始化为0。

```cpp
IntSet::IntSet() : num_elements(0) {}
```

接着实现 `contains` ，不变式告诉我们前 `num_elements` 个元素是集合中的元素，所以我们只需要变量前 `num_elements` 个元素。

```cpp
bool IntSet::contains(int value) const {
    for (int i = 0; i < num_elements; ++i) {
        if (elements[i] == value) {
            return true;
        }
    }
    return false;
}
```

对于 `insert` ，我们要先断言REQUIRES，然后插入元素的时候也要维护不变式——不能有重复的元素，如果有，不做任何操作。

```cpp
void IntSet::insert(int value) {
    assert(size() < MAX_SIZE);
    if (!contains(value)) {
        elements[num_elements++] = value;
    }
}
```

要 `remove` 一个元素，我们首先需要检查该值是否在集合中。如果在集合中，我们需要知道它在数组中的位置，这就需要通过遍历数组来找到它。为了避免在 `contains` 和 `remove` 中重复这个算法，我们定义一个私有的辅助函数来完成这个任务，并在这两个地方调用它。

```cpp
int IntSet::indexOf(int value) const {
    for (int i = 0; i < num_elements; ++i) {
        if (elements[i] == value) {
            return i;
        }
    }
    return -1;
}

bool IntSet::contains(int value) const {
    return indexOf(value) != -1;
}

void IntSet::remove(int value) {
    int index = indexOf(value);
    if (index != -1) {
        elements[index] = elements[--num_elements];
    }
}
```

剩下的两个接口如下。

```cpp
int IntSet::size() const {
    return num_elements;
}

void IntSet::print(std::ostream &os) const {
    os << "{ ";
    for (int i = 0; i < num_elements; ++i) {
        os << elements[i] << " ";
    }
    os << "}";
}
```

4. Container ADTs and Polymorphism

4.1 Operator Overloading

🏅 C++ follows the philosophy that user-defined types should have the same access to language facilities as built-in types. Since operators can be used with built-in atomic types, C++ allows operators to be applied to class types through *operator overloading*.

Most operators in C++ can be overloaded. An operator overload requires at least one of the operands to be of class type – the behavior of operators on atomic types cannot be changed. For most operators, the overloadding function can either be a top-level function or a member of the type of the left-most operand, if it is of class type.

比如，对于前面的 `IntSet`，我们可以重载 `+` 来实现并集操作：

```cpp
IntSet IntSet::operator+(const IntSet &rhs) const {
    IntSet result = *this;
    for (int i = 0; i < rhs.num_elements; ++i) {
        if (!contains(rhs.elements[i])) {
            result.insert(rhs.elements[i]);
        }
    }
    return result;
}
```

🏅 When the compiler encounters an operator where at least one operand is of class type, it looks for a function whose name is `operator` followed by the symbol for the actual operator. For example, if `+` is applied to two `IntSet` s, the compiler looks for a function with name `operator+` that can be applied to two `IntSet` objects.

🏅 Though most operators can be overloaded either as top-level or member functions, there are some cases where we must use a top-level function:

- The first operand is of atomic type. Atomic types are not classes, so they do not have member functions.
- The first operand is of class type, but we do not have access to the class definition, so we cannot define a new member function.

比如，对于前面的 `IntSet` ：

```cpp
std::ostream &operator<<(std::ostream &os, const IntSet &set) {
    set.print(os);
    return os;
}
```

对于 `os` 对象，我们不能直接访问到它的定义，所以只能将重载函数写成 `top-level` 。

🏅 We saw previously that inserting to a stream evaluates back to the stream object. To support this properly, our overload returns the given stream object. It must return the object by reference:

- Streams cannot be copied, so the code would not compile if it returned a stream by value.
- Even if streams could be copied, we want to return the original stream object itself, not a copy.
- Even if a copy would work, we would end up with object slicing, since `os` actually will refer to an object of a class that derives from `ostream` .

The parameters are in the same order as the operands, from left to right. The function need only call the `print()` member function on the `IntSet` and then return the given `ostream` object.

🏅 In other cases, we need to define an operator as a member function:

- If the overload needs access to private members, a member function would have access because it is part of the class. (In the future, we will see that we can use a friend declaration to give an outside class or function access to private members. Friend declarations are sometimes used with operator overloads.)
- Some operators can only be overloaded as member functions: the assignment operator (`=`), the function-call operator (`()`), the subscript operator (`[]`), and the arrow operator (`->`).

4.2 Parametric Polymorphism

我们的集合只支持 `int` 类型，现在想让它支持 `char` 类型，一种方法就是复制一份写好的代码，然后将类型修改为 `char` ，如果想支持 `double` 类型，我们可以故技重施。但这种解决方案并不好，因为它会导致几乎相同的代码的重复。除了数据类型不一样，复制出来的几份代码本质上是重复的，这不由地让我们想到抽象，我们可以针对数据类型进行抽象，在设计和实现时不指定具体的类型，而是采用一个像变量一样可以取值为任意类型的类型变量——C++为我们提供了 `template` 。

🏅 A template is a model for producing code. We write a generic version, parameterized by one or more *template parameters*. When we use the code with a particular type argument, the compiler will *instantiate* a specific version of the code by substituting arguments for the parameters and compiling the resulting code. We specify a template and its parameters by placing a template header before the entity that we are defining.

```
template <typename/class T>
class UnsortedSet {
    ...
};
```

🏅 The template header can go on the same line or the previous one: whitespace generally does not matter in C++. Since the entity that follows the template header is a class, it is a *class template*. The header begins with the `template` keyword, followed by a parameter list surrounded by angle brackets. Within the parameter list, we introduce a template parameter by specifying the kind of entity the parameter can represent. The `template` or `class` keyword indicates that the parameter is a type parameter. The parameter name then follows. Here, we have chosen the name `T` , since we don’ t have further information about the type. (`Value_type` or `Element_type` are other common names to use with a container.)

一个模板可以有多个参数，并且参数可以是具体的类型，比如标准库中的 `std::array` 是这样声明的：

```cpp
template <typename T, std::size_t N>
class array;
```

🏅 The scope of a template parameter is the entire entity that follows; if it is a class, then the scope is the entire class definition.

🏅 The syntax for using a class template is to follow the name of the template with an argument list enclosed by angle brackets.

比如，

```cpp
array<int, 10> arr10;
```

4.2.1 Function Template

🏅 We can also define a function as a template, resulting in a *function template*.

和定义类模板一样，将模板头放在函数之前，就定义了一个函数模板。比如，

```cpp
template <typename T>
T max(const T &value1, const T &value2) {
    return value2 > value1 ? value2 : value1;
}
```

我们可以在函数定义中的任何位置使用模板形参。函数模板的使用方法和类模板一样——在模板名后面跟上由尖括号括起来的模板实参，但与类模板不同的是，编译器在大多数情况下都能从函数调用的参数中推导出模板实参。

4.2.2 Compeling Template

🏅 We saw previously that the C++ compiler only needs access to declarations when compiling code that uses a class or a function defined in some other source file. However, this is not the case for class and function templates. The compiler must actually instantiate the definitions for each set of template arguments, so it must have access to the full definition of a template.

To provide the compiler with access to the full definition of a template wherever it is used, we must arrange for the header file to contain the full definition. We can just define the template directly in the header file itself; it is still good practice to separate the declarations from the definitions for the benefit of anyone using our code.

```cpp
// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);
...
    template <typename T>
    T max(const T &value1, const T &value2) {
    return value2 > value1 ? value2 : value1;
}
```

🏅 A better organization is to separate the definitions into their own file; common convention is to use a suffix like `.tpp` for this file. We can then use a `#include` directive to pull the code into the header file.

```cpp
// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);

#include "max.tpp"
```

```cpp
// max.tpp

template <typename T>
T max(const T &value1, const T &value2) {
  return value2 > value1 ? value2 : value1;
}
```

🏅 Code that uses the `max` module would then just `#include "max.hpp"`, which would transitively include `max.tpp`.

4.2.3 Include Guards

在复杂的程序中，常常会出现一个文件不小心多次 `#include` 同一个头文件的情况，这可能引发编译器错误，提示某个函数或类被定义了多次。例如，模块A依赖于模块B，因此包含了B的头文件。然后模块C同时依赖于A和B，所以它包含了这两个模块的头文件，这导致模块C多次包含B的头文件，编译器会提示错误。

🏅 To avoid problems with a header being included more than once, headers generally have *include guards* that arrange for the compiler to ignore the code if the header is included a second time. The following is an example of include guards.

```
#ifndef MAX_HPP
#define MAX_HPP

// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);

#include "max.tpp"

#endif /* MAX_HPP */
```

🏅 The `#ifndef` checks whether the *macro* `MAX_HPP` is defined. If not, the compiler processes the code between the `#ifndef` and the `#endif`. The `#define` introduces a definition for `MAX_HPP`. The next time the header is included, `MAX_HPP` is defined, so the `#ifndef` becomes false, and the compiler ignores the code between the `#ifndef` and `#endif`.

A widely supported but nonstandard alternative is `#pragma once`, which need only be placed at the top of file.

4.2.4 Member-Function Templates

🏅 Defining a member function within the definition for a class template is no different than a member function within a class. Defining a member function outside the definition of a class template differs from that of a regular class; we must inform the compiler that the function is a member of a class template.

```cpp
template <typename T>
bool UnsortedSet<T>::contains(const T &value) const {
  return indexOf(value) != -1;
}

template <typename T>
void UnsortedSet<T>::insert(const T &value) {
  assert(size() < MAX_SIZE);
  if (!contains(value)) {
    elements[num_elements] = value;
    ++num_elements;
  }
}
```

🏅 Here, we must tell the compiler that `contains()` is a member of `UnsortedSet<T>`. However, before we can use the name `T`, we need to introduce it with a template header. The scope of a template header is the entity that immediately follows; thus, each member-function definition needs its own template header.

4.2.5 Static vs. Dynamic Polymorphism

🏅 The template mechanism gives us parametric polymorphism; a template type parameter can take on the form of any type. The compiler instantiates the code at compile time, so we also refer to this as *static polymorphism*. Compare this with subtype polymorphism, where the program resolves virtual function calls at runtime; we therefore use *dynamic polymorphism* to also refer to it.

In designing an ADT, we have the choice of static polymorphism, dynamic polymorphism, or both. However, dynamic polymorphism comes with a runtime performance cost; a virtual function call generally requires two extra pointer dereferences compared to a non-virtual call. For a container, a program may make many calls to insert items, remove items, check whether an item is in the container, and so on. Thus, the general convention in C++ is to avoid dynamic polymorphism with containers.

Another reason why dynamic polymorphism is not a good fit for a container is that we usually make the decision of which container to use when we write our code, not when we run it. However, it is still good practice to write our code in such a way that we can easily change the container type we are using.

4.2.6 Type Aliases

🏅 Ideally, we would have a single location in our code that needs to change if we decide to use something else. We' ve seen this pattern already when it comes to a value that is used in many places: define a constant or variable, and use the associated name rather than the value directly. Thus, introducing a new name for an entity is a powerful technique, allowing us to avoid hardcoding the entity everywhere in our code. C++ gives us this capability for types with type aliases.

A *type alias* is a new name for an existing type. We can introduce a type alias with a *using declaration.*

```
using new_name = existing type;
```

🏅 A using declaration (but not a typedef) can also define an *alias template*, which introduces a new name that is a template for a set of types.

比如，

```
template <typename T>
using Set = UnsortedSet<T>;

template <typename T>void fill_from_array(Set<T> &set, const T *arr, int n);

// 使用别名
int main () {
    Set<int> set1;
    int arr1[4] = { 1, 2, 3, 2 };
    fill_from_array(set1, arr1, 4);
    Set<char> set2;
    char arr2[3] = { 'a', 'b', 'a' };
    fill_from_array(set2, arr2, 3);
}

```

如果我们想要使用 `SortedSet<T>`，只需要修改一个地方：

```
template <typename T>
using Set = SortedSet<T>;
```

4.3 Memory Models and Dynamic Memory

🏅 An object is a region of data storage in memory, located at some address in memory. An object also has a *storage duration* that determines its lifetime. We consider three storage durations:

- *static*: the lifetime is essentially the whole program
- *automatic* (also called *local*): the lifetime is tied to a particular scope, such as a block of code
- *dynamic*: the object is explicitly created and destroyed by the programmer

The first two durations are controlled by the compiler, and an object with static or automatic scope is associated with a variable. Objects with dynamic storage are managed by the programmer, and they are not directly associated with variables.

The following variables refer to objects with static storage duration:

- global variables
- static member variables
- static local variables

The lifetime of these objects is essentially the whole program, so they can be used at any point in time in the program.

4.3.1 Static Local Variables

🏅 A *static local variable* is a local variable declared with the `static` keyword. Rather than living in the activation record for a function call, it is located in the same region of memory as other variables with static storage duration, and there is one copy of each static local variable in the entire program.

4.3.2 Static Member variables

前面在实现 `IntSet` 时， `MAX_SIZE` 被声明为 `static` ，这表示它是一个静态成员变量。它和静态局部变量一样，它的生命周期是整个程序，并且只有一份拷贝。它的出现是为了不破坏封装性。

当静态成员变量被声明为 `const` 时，它就是编译时常量了。

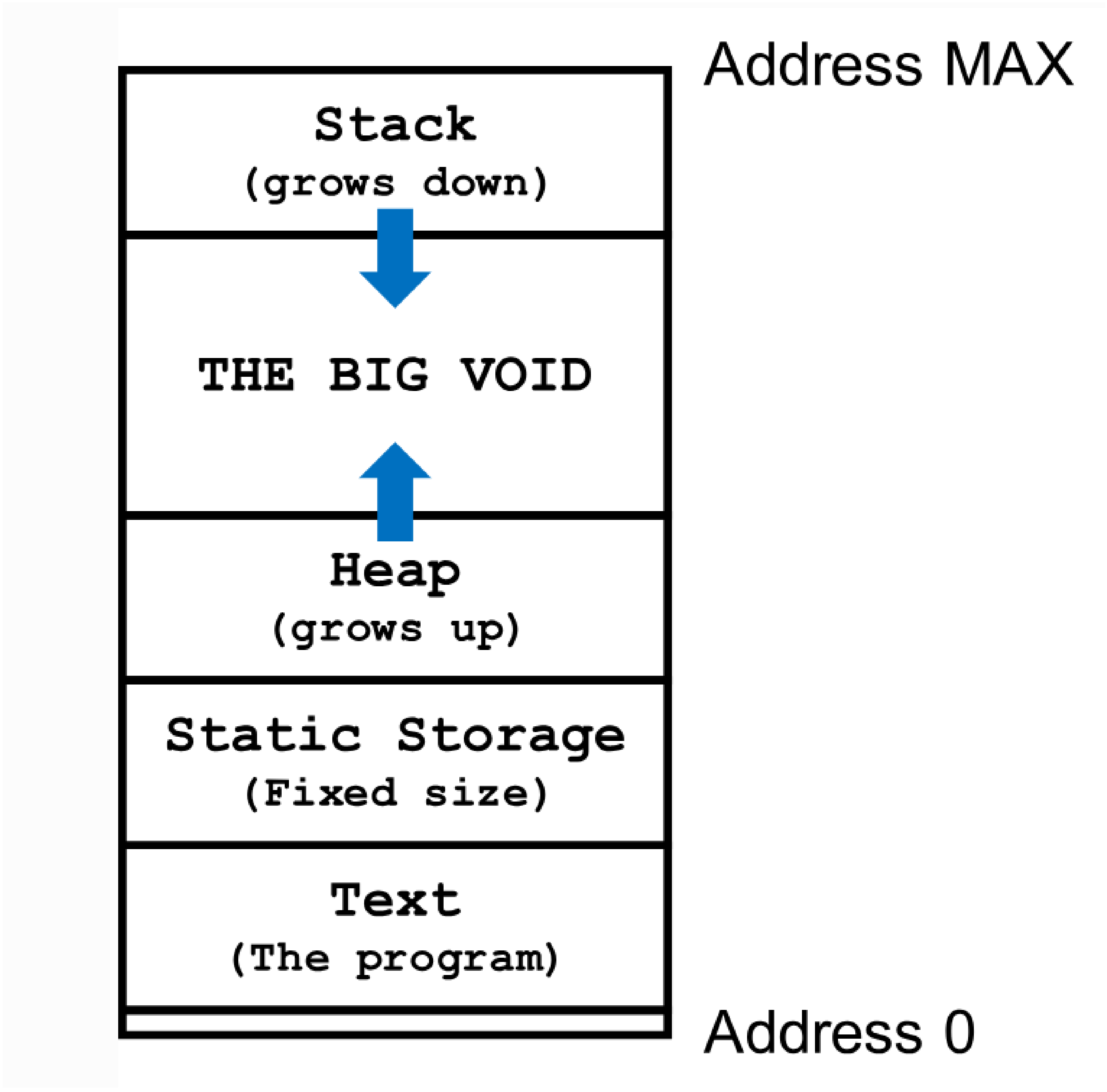
> Member variables that are `static` need not be compile-time constants. However, a member variable that isn't a compile-time constant cannot be initialized at the point of declaration (unless it is declared as `inline` in C++17 and onward). The variable must be defined outside of the class, generally in a source (`.cpp`) file. As with a member function, the scope-resolution operator is used to define a member variable outside the class (e.g. `int IntSet::foo = 3;`).

🏅 A static member can be accessed by name from within the class, the same as a non-static member. It can be accessed from outside the class with the scope-resolution operator.

4.3.3 Automatic Storage Duration

🏅 Function parameters and local variables have automatic storage duration, also called local storage duration. The lifetime of the associated object corresponds to the variable’s *scope*, the region of code where it is valid to refer to the variable. The scope of a variable begins at its point of declaration and ends at the end of the scope region in which it is defined. A scope region can be the body of a function, a whole loop or conditional, the body of a loop or conditional, or just a plain *block* denoted by curly braces.

4.3.4 Memory Models - Address Spaces



4.3.5 The `new` and `delete` Operators

> 🏅 The syntax of a `new` expression consists of the `new` keyword, followed by a type, followed by an optional initialization using parentheses or curly braces.

比如，

```cpp
// default initialization
new int;
// value initialization
new int();
new int{};
// explicit initialization
new int(3);
new int{3};
```

🏅 If no initialization is provided, the newly created object is default initialized. For an atomic type, nothing is done, so the object’ s value is whatever is already there in memory. For a class type, the default constructor is called.

Empty parentheses or curly braces result in *value initialization*. For an atomic type, the object is initialized to a zero value. For a class type, the default constructor is called.

Explicit initialization can be done by supplying values in parentheses or curly braces. For atomic types, the value is used to initialize the object. For C-style ADTs, curly braces must be used, and the values within are used to initialize the member variables. For C++ ADTs, both parentheses and curly braces invoke a constructor with the values within as arguments.

A `new` expression does the following:

- Allocates space for an object of the given type on the heap.
- Initializes the object according to the given initialization expression.
- Evaluates to the address of the newly created object.

The address is how we keep track of the new object; it is not associated with a variable name, so we have to store the address somewhere to be able to use the object.

```cpp
int main() {
  int *ptr = new int(3);
  cout << *ptr << endl;    // prints 3
  ...
}
```

🏅 The newly created object’ s lifetime is not tied to a particular scope. Rather, it begins when the `new` expression is evaluated and ends when the `delete` operator is applied to the object’ s address.

```cpp
delete ptr;
```

🏅 The expression `delete ptr;` does **not** kill the `ptr` object – it is a local variable, and its lifetime ends when it goes out of scope. Rather, `delete` follows the pointer to the object it is pointing to and kills the latter. We can continue to use the pointer object itself.

4.3.6 Dynamic Arrays

🏅 We saw previously that local and member-variable arrays must have a size that is known at compile time, so the compiler can reason about the sizes of activation records and class-type objects. This restriction does not apply to dynamic arrays – since they are located on the heap rather than in an activation record or directly within a class-type object, the compiler does not need to know their size.

The syntax for creating a dynamic array is the `new` keyword, an element type, square brackets containing an integer expression, and an optional initializer list. The expression within the square brackets need not be a compile-time constant.

```
int main() {
  cout << "How many elements? ";
  int count;
  cin >> count;
  int *arr = new int[count];
  for (int i = 0; i < count; ++i) {
    arr[i] = i;
  }
  ...
}
```

🏅 A `new` array expression does the following:

- Allocates space for an array of the given number of elements on the heap.
- Initializes the array elements according to the given initialization expression. If no initialization is provided, the elements are default initialized.
- Evaluates to the address of the first element of the newly created array.

🏅 The lifetime of a dynamic array begins when it is created by a `new` array expression. It ends when the array-deletion operator `delete[]` is applied to the address of the array’ s first element.

```cpp
delete[] arr;
```

4.3.7 Life-time of Class Objects

🏅 When a class-type object is created, a constructor is run to initialize it. When a class-type object dies, its destructor is run to clean up its data. For a local object, this is when the associated variable goes out of scope. For a dynamic object, it is when `delete` is applied to its address.

If a class-type object is an element of an array, it dies when the array dies, so its destructor runs when the array is dying.

The lifetime of member variables of a class-type object is tied to the lifetime of the object as a whole. When the object dies, its members die as well; if they are of class-type themselves, their destructors run. Specifically, the members of a class-type object are automatically destroyed after the destructor of the object is run, in reverse order of their declarations.

The order in which the bodies of the destructors are run is the reverse of the constructors – we get the same “socks-and-shoes” ordering that we saw with base-class and derived-class constructors and destructors.

5. Manage Dynamic Memory-RAII

动态资源是由程序员管理的，容易出问题，比如内存泄露。而对于 `automatic duration` 的对象，它的生命周期是由编译器来管理的，只要编译器设计好了，是不会忘记释放它的。所以我们可以考虑将动态资源与自动内存管理结合起来。

🏅 We can leverage automatic memory management and destructors to wrap up the management of a dynamic resource in a class. In particular, we will arrange for the constructor of a class-type object to allocate a resource and for the destructor to release that resource. Doing so ties the management of the resource to the lifetime of the class-type object. This strategy is called *resource acquisition is initialization (RAII)*, and it is also known as *scope-based resource management*.

6. The Big Three

当用已经存在的对象去初始化另一个对象，会发生值的拷贝，称为拷贝初始化(initailization as copy)。比如，我们有一个 `Person` 类，它包括 `name` ， `age` 和 `is_ninja` 三个成员变量。

```cpp
Person elise = {"Elise", 22, true};
Person tail = elise;
```

当用 `elise` 对象初始化 `tail` 对象时， `elise` 成员的值会分别拷贝给 `tail` 的成员。C++提供了几种拷贝初始化的形式：

```cpp
Person tali = elise;
Person tali(elise);
Person tali{elise};
Person tali = {elise};
```

当给函数以 `pass by value` 的方式传参时，也会发生拷贝初始化。

```cpp
void func(Person p);

int main() {
    Person elise = { "Elise", 22, true };
    func(elise);
}
```

当 `func` 被调用时，形参 `p` 会被初始化为 `elise` 的值。同样，一个函数 `return by value` 时，也可能发生拷贝初始化。

```cpp
Person func2() {
     Person elise = { "Elise", 22, true };
     return elise;
}

int main() {
    Person tali = func2();
}
```

`tail` 会发生拷贝初始化。

6.1.1 Copy Constructors

🏅 We also previously discussed that a constructor is always called when a class-type object is created (except for C-style ADTs when the members are initialized directly, like `elise` above). Copy initialization of a class-type object also invokes a constructor, specifically the *copy constructor* for the class.

下面是一个显式定义拷贝构造函数的例子：

```cpp
class Example {
  public:
    Example(int x_in, double y_in)
        : x(x_in), y(y_in) {}

    Example(const Example &other)
        : x(other.x), y(other.y) {
        cout << "Example copy ctor: " << x << ", " << y << endl;
    }

    int get_x() const {
        return x;
    }

    double get_y() const {
        return y;
    }

  private:
    int x;
    double y;
};
```

🏅 The second constructor above is the copy constructor, and it takes a reference to an existing object as the parameter. The parameter must be passed by reference – otherwise, a copy will be done to initialize the parameter, which itself will invoke the copy constructor, which will call the copy constructor to initialize its parameter, and so on.

The C++ compiler provides an implicit copy constructor if a user-defined one is not present. The implicit copy constructor just does a member-by-member copy, so in the case of `Example`, it acts like the following.

```cpp
Example(const Example &other)
    : x(other.x), y(other.y) {}
```

6.1.2 Assignment Operator

当用已经存在的对象对另一个已经存在的对象赋值时，也会发生拷贝。比如，

```cpp
int main() {
    Example e1(2, -1.3);
    Example e2(3, 4.1);
    e2 = e1;
}
```

🏅 An assignment expression consists of a left-hand-side object, the `=` operator, and a right-hand-side object or value. The expression evaluates the right-hand side, copies it into the left-hand-side object, and then evaluates back to the left-hand-side object. We can then use the result in a larger expression.

```cpp
cout << (e2 = e1).get_x() << endl;  // prints 2
```

🏅 Like most operators, the assignment operator can be overloaded; the overload must be a member function of the type of the left-hand side.

```cpp
class Example {
  public:
    Example &operator=(const Example &rhs);
    ...
};
```

🏅 The function takes in an `Example` by reference to const, corresponding to the right-hand operand of the assignment (In C++, a reference to const can bind to a temporary, allowing the right-hand operand of the assignment to be a value rather than an object). The return value will be the left-hand-side object itself, and it needs to be returned by reference rather than by value (the latter would make a copy rather than returning the object itself). Thus, the return type is `Example &`.

The following is a definition of the overloaded assignment operator that just does a member-by-member copy.

```cpp
Example &Example::operator=(const Example &rhs) {
    x = rhs.x;
    y = rhs.y;
    return *this;
}
```

🏅 Like the copy constructor, the compiler provides an implicit definition of the assignment operator if a user-defined one is not present. Like the implicitly defined copy constructor, the implicit assignment operator performs a member-by-member copy.

6.1.3 Shallow and Deep Copies

🏅 For most class types, a member-by-member copy is sufficient, and there is no need to write a custom copy constructor or assignment operator. However, for a type that manages a dynamic resource, a member-by-member copy generally results in incorrect sharing of a resource.

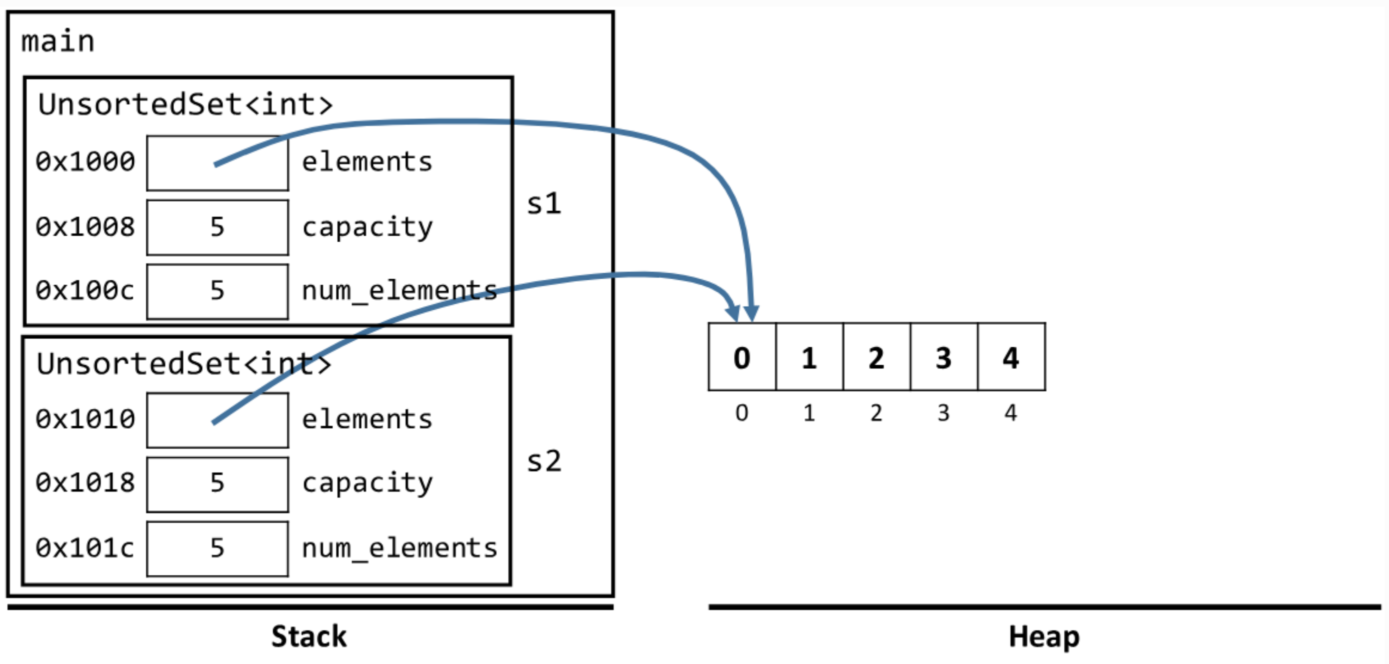


Figure 59 *The implicit copy constructor copies each member one by one, resulting in a shallow copy.*

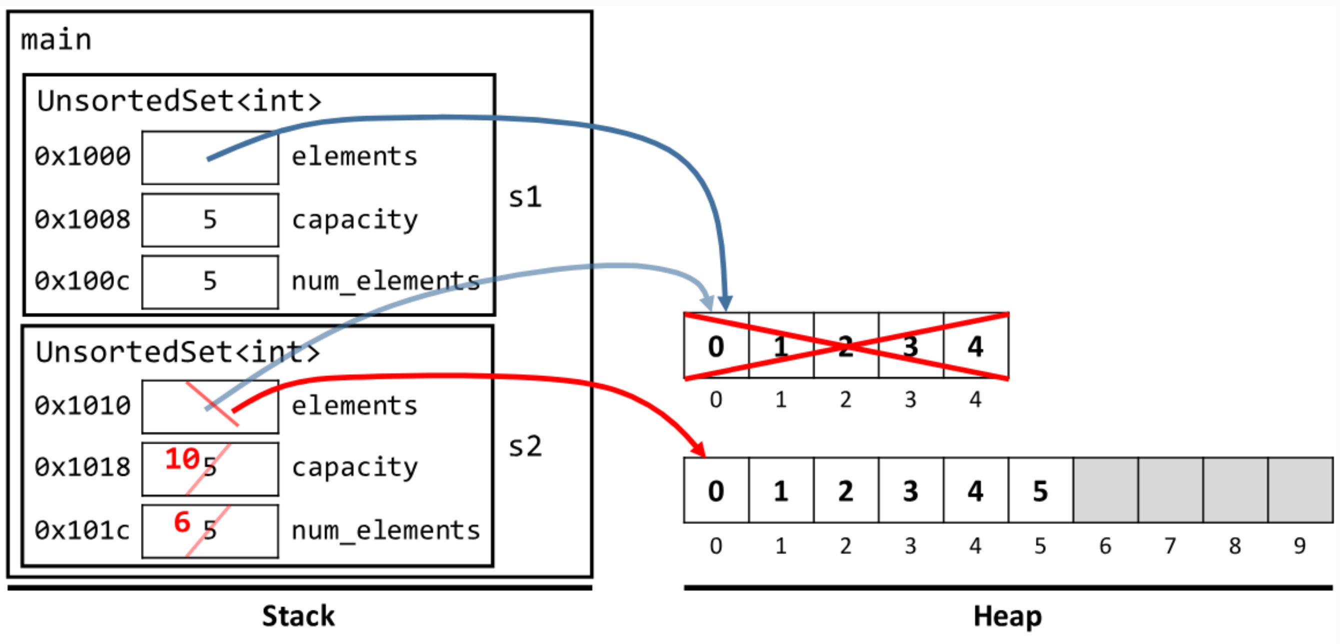


Figure 60 *A subsequent* `grow()` *results in one of the sets using a dead array.*

🏅 The fundamental problem is that the implicitly defined member-by-member copy is a *shallow copy*: it copies the pointer to the array of elements, rather than following it and copying the array as a whole. Instead, we need a *deep copy*, where we make a copy of the underlying resource rather than having the two sets share the same resource.

赋值运算符也是类似的。不过对于赋值运算符我们要注意 `self assignment`

🏅 An expression such as `s2 = s2` will delete `s2.elements` before proceeding to access the elements in the subsequent loop. Instead, the assignment should have no effect when both operands are the same object, so we need to check for this case before doing any work.

```
if (this == &rhs) {    // self-assignment check
    return *this;
}
```

6.1.4 The Law of Big Three

🏅 We have seen that the implicitly defined copy constructor and assignment operator both do a shallow copy, and that this behavior is incorrect for classes that manage a dynamic resource. Instead, we need a deep copy, which requires us to write our own custom versions of the two functions.

We also saw last time that resource management requires us to write our own custom destructor as well. It is not a coincidence that we needed to write custom versions of all three of the destructor, copy constructor, and assignment operator. The *Law of the Big Three* (also know as the *Rule of Three*) is a rule of thumb in C++ that if a custom version of any of the destructor, copy constructor, or assignment operator is required, almost certainly all three need to be custom. We refer to these three members as the *big three*.

By default, C++ provides implicit definitions of each of the big three:

- The implicitly defined destructor does no special cleanup; it is equivalent to a destructor with an empty body. Like other destructors, it does implicitly destroy the members as well as the base class, if there is one.
- The implicitly defined copy constructor does a member-by-member shallow copy.
- The implicitly defined assignment operator does a member-by-member shallow copy.

When a class manages a resource, however, the programmer must provide custom versions of the big three:

- The destructor should free the resources.
- The copy constructor should make a deep copy of the resources and shallow copy the non-resources.
- The assignment operator should:
  - Do a self-assignment check.
  - Free the old resources.

- Make a deep copy of the right-hand-side object’ s resources.
- Shallow copy the non-resources from the right-hand-side object.
- Return the left-hand-side object with `*this`.

6.1.5 Destructors and Polymorphism

🏅 We saw previously that applying the `delete` operator to a pointer follows the pointer and kills the object at the given address. If the object is of class type, its destructor is run. However, a subtle issue arises when the static and dynamic type of the object does not match: does the destructor use static or dynamic binding?

考虑如下例子：

```cpp
class Base {
  public:
    virtual void add(int x) = 0;

    ~Base() {
        cout << "Base dtor" << endl;
    }
};

class Derived : public Base {
  public:
    void add(int x) override {
        items.push_back(x);
    }

    ~Derived() {
        cout << "Derived dtor" << endl;
    }

  private:
    vector<int> items;
};

int main() {
    Base *bptr = new Derived;
    bptr->add(3);
    delete bptr;
}
```

它的输出是

```
Base dtor
```

🏅 The destructor is statically bound, so `~Base()` is invoked rather than `~Derived()`. This is problematic: even though `Derived` itself is not managing dynamic memory, its member variable `items` is. Since the destructor for `Derived` was not called, the members introduced by `Derived` were also not destructed.

The solution is to use dynamic binding for the destructor instead by declaring it as virtual.

```cpp
class Base {
  public:
    virtual void add(int x) = 0;

    virtual ~Base() {
        cout << "Base dtor" << endl;
    }
};

class Derived : public Base {
  public:
    void add(int x) override {
        items.push_back(x);
    }

    ~Derived() { // virtual implicitly inherited
        cout << "Derived dtor" << endl;
    }

  private:
    vector<int> items;
};

int main() {
    Base *bptr = new Derived;
    bptr->add(3);
    delete bptr;
}
```

现在它的输出是

```
Derived dtor
Base dtor
```

🎖️ In general, **a base class should always declare the destructor to be virtual if the class will be used polymorphically** (meaning that a base-class pointer or reference may be bound to a derived-class object). If the base class destructor has no work to do, it may be defaulted. The derived class destructor need not be explicitly defined, since “virtualness” is inherited even if the destructor is implicitly defined by the compiler.

所以，这样也是没问题的：

```cpp
class Base {
  public:
    virtual void add(int x) = 0;

    virtual ~Base() = default; // use implicitly defined dtor, but make it virtual
};

class Derived : public Base {
  public:
    void add(int x) override {
        items.push_back(x);
    }
    // implicitly defined dtor inherits virtual
  private:
    vector<int> items;
};

int main() {
    Base *bptr = new Derived;
    bptr->add(3);
    delete bptr;
}
```

7. Iterators

遍历容器内的元素，是要知道容器内部是实现细节的，是数组还是链表，但外部的代码不应该依赖这些具体的实现细节，所以我们要提供另外的接口用以迭代容器（这里主要了解迭代序列容器）。

一般来讲，迭代序列容器有两种模式：`traversal by index`，`traversal by pointer`。如果容器底层支持随机访问，比如数组，我们可以通过重载`[]`来实现`traversal by index`。否则，我们只能选择`traversal by pointer`，但使用原生的指针肯定是不行的，因为这需要知道容器的底层实现细节。所以我们需要的是一种类似于指针的对象，它能指向容器的第一个元素，每次能向前移动一个元素，当超出容器时就停止。我们把这种对象称为迭代器，这种迭代模式称为`traveral by iterator`。

🏅 【There are many kinds of iterators in C++.】

An iterator is a class-type object that has the same interface as a pointer. To traverse with an iterator, it must provide the following operations:

- dereference (prefix `*`)
- increment (prefix `++`)
- equality checks (`==` and `!=`)

In addition, the container itself must provide two member functions:

- `begin()` returns an iterator to the start of the sequence
- `end()` returns a “past-the-end” iterator that represents a position that is past the last element of the sequence.

8. The `typename` keyword

考虑如下函数模板：

```cpp
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Returns the element that this iterator points to.
template <typename T>
typename List<T>::Iterator &List<T>::Iterator::operator++() {
    assert(node_ptr); // check whether this is a past-the-end iterator
    node_ptr = node_ptr->next;
    return *this;
}
```

The return type of `operator++()` is a reference to `List<T>::Iterator`; `Iterator` is a member of a template that is *dependent* on the template parameter `T`. C++ requires the `typename` keyword before a dependent name that refers to a type, so that the compiler knows that it is a type and not a value. reason

The following illustrates when the `typename` keyword is required:

```cpp
template <typename U>
void func() {
    IntList list;                    // not a qualified type
    IntList::Iterator it1;           // outer class does not depend on template
                                     // parameter U
    List<int>::Iterator it2;         // outer class does not depend on template
                                     // parameter U
    typename List<U>::Iterator it3;  // outer class depends on template parameter U
    int capacity =
        SortedSet<U>::MAX_CAPACITY;  // member is not a type
}
```

An *unqualified* name, one without the scope-resolution operator, never needs the `typename` keyword. In a *qualified* name, if the outer type does not depend on a template parameter, then no `typename` keyword is required. If the outer type does depend on a template parameter, then the `typename` keyword is required when the inner name refers to a type. If the inner name does not refer to a type, the `typename` keyword is erroneous to use, since it explicitly tells the compiler that the inner name is a type.

9. Friend Declarations

在嵌套类中，内部类的私有成员是不可以被外部类访问的，为了解决这个问题，我们可以使用友元声明。

A class gives an outside entity access to the class’ s private members by using friend declaration.

比如，

```cpp
template <typename T>
class List {
  ... public : class Iterator {
        friend class List;
        ...
    };
};
```

在 `Iterator` 类中声明友元后， `List` 就可以访问 `Iterator` 的私有成员。

> 🏅 A friend declaration can appear anywhere directly within a class body, and it tells the compiler that the given entity is allowed to access private members. The entity may be a function or a type; for instance, it is common practice to define the insertion operator as a friend.

```cpp
class Card {
  ... private : std::string rank;
    std::string suit;

    friend std::ostream &operator<<(std::ostream &os, const Card &card);
};

std::ostream &operator<<(std::ostream &os, const Card &card) {
    return os << card.rank << " of " << card.suit;
}
```

> 🏅 Friendship is given, not taken. For instance, class `C` may declare `F` as a friend:

```cpp
friend class F;
```

> 🏅 This allows `F` to access the private members of `C`, but it does not allow `C` to access the private members of `F`.