```
int    max(int a,    int b)    { return a < b ? b : a; }
double max(double a, double b) { return a < b ? b : a; }
// И так далее для всех числовых типов.
```

Листинг 1 — Перегрузки функции нахождения наибольшего из двух чисел

```
template<typename T>
T max(T a, T b) {
     return a < b ? b : a;
}
…
max(1, 2);       // max<int>(int a, int b)
max(2.71, 3.14); // max<double>(double a, double b)
```

Листинг 2 — Шаблонная функция нахождения наибольшего из двух значений

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

Листинг 3 — Специализация функции `max<T>()` для строк C

```
template<typename T>
class DynamicArray {
     T* elements;
public:
     DynamicArray(const size_t size) {
          elements = size ? new T[size] : 0;
     }
     ~DynamicArray() { delete[] elements; }
     operator T* () { return elements; }
};
DynamicArray<double> squares(7);
squares[0] = 906.01;
printf("Square root of %g equals %g. It all seemed harmless..",
     squares[0], sqrt(squares[0]));
```

Листинг 4 — Шаблонный класс и его применение

```
template<bool FirstIsLarger, typename T1, typename T2>
struct X { };

template<typename T1, typename T2>
struct X<true, T1, T2>  { typedef T1 Type; };

template<typename T1, typename T2>
struct X<false, T1, T2> { typedef T2 Type; };

```

```
template<typename T1, typename T2>
struct SelectLargestType {
     typedef
          typename X<sizeof(T1) > sizeof(T2), T1, T2>::Type
          Result;
};
typedef SelectLargestType<long double, double>::Result Number;
```

Листинг 5 — Выбор наиболее вместительного типа метапрограммированием

```
unsigned limit = 1000;
std::vector<unsigned> numbers;
numbers.reserve(limit - 1);
for (unsigned i = 2; i < limit; ++i)
     numbers.push_back(i);
for (size_t i = 0; i < numbers.size(); ++i) {
     size_t j = i + 1;
     while (j < numbers.size()) {
          if (numbers[j] % numbers[i]  == 0)
               numbers.erase(&numbers[j]);
          else
               ++j;
     }
}
numbers.clear();
```

Листинг 6 — Применение контейнера `std::vector` для поиска простых чисел

```
const size_t length = 42;
double container[length];
double* begin = &container[13];
double* end = &container[27];
for (auto iterator = begin; iterator != end; ++iterator) {
     printf("%g", *iterator);
}
```

Листинг 7 — Обработка массива при помощи указателей на его элементы

```
std::list<double> numbers { 1, 2, 3, 4, 5};
for (auto i = numbers.begin(); i != numbers.end(); ++i) {
     printf("%g", *i);
}
```

Листинг 8 — Применение итераторов для обработки `std::list`

```
std::list<double> container { 1, 2, 3, 4, 5};
for (auto i = container.begin(); i != container.end(); ++i) {
     if (*i % 2 == 0)
          container.erase(i);
}
```

Листинг 9 — Удаление из контейнера четных элементов (версия с ошибкой)

```
auto i = container.begin();
while (i != container.end()) {
     if (*i % 2 == 0)
          i = container.erase(i);
     else
          ++i;
}
```

Листинг 10 — Удаление из контейнера четных элементов (правильная версия)

```
int array[4] = { 1, 2, 3, 4 };
for (auto& item : array) {
     item *= item;
}
for (int item : array) {
     printf("%d", item);
}
```

Листинг 11 — Цикл for для диапазона

```
std::vector<int> numbers { 4, 2, 5, 1, 3, 6 };
std::sort(numbers.begin(), numbers.end());
std::is_sorted(numbers.begin(), numbers.end());
auto middle = numbers.begin() + numbers.size() / 2;
auto place = std::find(numbers.begin(), middle, 3);
if (place != middle)
     puts("Число 3 найдено в первой половине контейнера.");
```

Листинг 12 — Пример использования алгоритмов STL

```
bool greaterThen4(const int element) {
     return element > 4;
}
…
std::remove_if(numbers.begin(), numbers.end(), greaterThen4);
```

Листинг 13 — Использование указателя на функцию с алгоритмом STL

```
int n = rand();
std::remove_if(
     numbers.begin(), numbers.end(),
     [n] (const int element) { return element < n; });
```

Листинг 14 — Использование λ-функций с алгоритмами STL

```
std::array<double, 5> array = { 1, 2, 3, 4, 5 };
double exponent = 2;
double sum = 0.0;
std::for_each(
     array.begin(), array.end()
     [exponent, &sum] (double x) {
          const double power = pow(x, exponent);
          sum += power;
     });
```

Листинг 15 — Захват локальных переменных в λ-функции

```
int N = 42, m = 32;
auto greaterThenN = [N](auto x) -> bool {
     return std::greater(x, N);
};
std::cout << m << " больше " << N << "?"
     << greaterThenN(m) ? "Да." : "Нет." << std::endl;
```

Листинг 16 — Использование λ-функции как переменной

```
class Bank {
     std::list<Account> accounts;
     void makeDeposit(const Account& account, unsigned amount) {
          account.deposit(amount);
     }
     void sendGifts(unsigned amount) {
          using std::placeholders::_1;
          // Способ № 1:
          std::for_each(
               accounts.begin(), accounts.end(),
               std::bind(&Account::deposit, _1, amount));
          // Способ № 2:
          std::for_each(
               accounts.begin(), accounts.end(),
               std::bind(&Bank::makeDeposit, this, _1, amount));
     }
};
```

Листинг 17 — Применение `std::bind` для привязки метода.

```
const char* vulgarHyphen = " – ";
size_t length = strlen(vulgarHyphen);
std::string properDash = "\u00A0\u2013 ";
std::string input;
std::cin >> input;
do {
     auto index = input.find(vulgarHyphen))
     if (index == std::string::npos)
          break;
     input.replace(index, length, properDash);
} while (true);
puts(input.c_str());
```

Листинг 18 — Работа с классом `std::string` для замены дефисов на тире

```
std::list< std::shared_ptr<Account> > accounts;
accounts.push_back(
     std::shared_ptr(
          new DebitAccount(1234, "Джон Шепард", 0.0)));
accounts.push_back(
     std::make_shared<CreditAccount>(5678, "Тали'Зора", 0.0));
for (auto *account : accounts)
     account->print();
```

Листинг 19 — Пример использования «умных» указателей.

```
std::string fileName;
std::getline(std::cin, fileName);
std::vector<double> numbers;
while (std::cin.good()) {
     double value;
     std:cin >> value;
     numbers.push_back(value);
}
std::ofstream output(fileName);
std::copy(
     numbers.begin(), numbers.end(),
     std::ostream_iterator<double>(output, " "));
std::cout << "Задача завершена!" << std::endl;
```

Листинг 20 — Пример работы с библиотекой потоков ввода-вывода STL

```
std::ostream& operator<<
     (std::ostream& stream, const Account& account) {
     stream << "На счете " << account.getID() << " находится "
            << account.getBalance() << " у. е." << std::endl;
     return stream;
}
```

Листинг 21 — Перегрузка оператора вывода в поток для класса `Account`

Таблица 1 — Основные контейнеры STL

Класс	Структура данных	Примечания
Последовательные (sequential)		
`array`	статический массив	Размер задается пользователем на этапе компиляции как параметр шаблона.
`dynarray`	динамический массив	Размер задается пользователем во время выполнения.
`vector`	динамический массив с расширенной функциональностью	Доступны операции вставки, удаления и поиска элементов. Объем занимаемой памяти регулируется автоматически. Доступ по индексу и размер за O(1), вставка и удаление за O(N).
`list`	двусвязный список	Доступ по индексу и размер за O(N), вставка и удаление элементов за O(1).
`forward_list`	односвязный список	Сложность операций как у `list`. Доступен проход по списку только в одном направлении.
`dequeue`	двунаправленная очередь (дек)	Подобна `vector`, но сложность операций вставки и удаления элементов с концов — O(1).
Ассоциативные (associative)		
`map`	ассоциативный массив	Доступ по уникальному ключу за O(log N). Может также рассматриваться как упорядоченная по ключу коллекция пар (ключ, значение).
`set`	упорядоченное множество	Проверка вхождения элемента за O(log N). Может также рассматриваться как упорядоченная коллекция уникальных значений.
Адаптеры к контейнерам (container adaptors)		
`stack`	стек (список LIFO)	Реализуются на основе последовательного контейнера, по умолчанию — `deque` (задается параметром шаблона).
`queue`	очередь (список FIFO)	
`priotity_queue`	очередь с приоритетами	Реализуется подобно `stack` и `queue`, но по умолчанию — на основе `vector`.

Таблица 2 — Некоторые алгоритмы STL

Алгоритм	Операция	Примечания
`for_each`	перебор	Проходит по всем элементам контейнера и передает их функтору.
`find`	поиск одного элемента	Возможен поиск конкретного элемента

		или удовлетворяющего предикату
`remove`	удаление элемента	Возможно удаление конкретного элемента или удовлетворяющего предикату
`count`	подсчет количества элементов	Возможен подсчет всех элементов контейнера или удовлетворяющих предикату.
`copy`	копирование	Возможно копирование всех элементов диапазона или удовлетворяющих предикату.
`move`	перемещение	Аналогично копированию. Имеется отдельная версия `std::move()`, возвращающая rvalue-ссылку на аргумент.
`swap`	обмен	Допускается как обмен всех элементов контейнеров, так и просто значениями двух переменных.
`transform`	проекция (преобразование элементов контейнера)	В выходной контейнер записываются элементы входного, преобразованные функтором.
`accumulate`	свертка (подсчет суммы и т. п. элементов контейнера)	Вычисления ведутся функтором, которому передается промежуточный итог и очередной элемент контейнера. Есть стандартный функтор `std::plus` для сложения и т. п. Расположен в `<numeric>`.
`sort`	сортировка	Алгоритм выбирается автоматически.

Таблица 3 — Способы решения проблемы разделяемого владения

Решение и класс в STL	**Описание**	**Преимущества**	**Недостатки**
Запрет разделяемого владения (`unique_ptr`)	Динамически размещенными данными владеет только один объект, который либо удаляет их в своем деструкторе, либо передает владение другому «умному» указателю.	Простота, полное решение проблемы.	Риск уничтожения данных при копировании «умного» указателя. Иногда разделяемое владение нужно.
Применение «слабых» ссылок (`weak_ptr`)	Один из «умных» указателей владеет динамически размещенными данными и может их уничтожить, остальные указатели — «слабые» (weak), и могут	Нет риска создания кольцевой зависимости (см. ниже).	Данные могут быть удалены, даже когда «слабый» указатель еще существует.

	только проверить, что данные еще не удалены.		
Подсчет ссылок (`shared_ptr`)	Данные не уничтожаются, пока остается хотя бы один «умный» указатель из владеющих ими. Число владельцев подсчитывается автоматически.	Удобство использования, интеллектуальность.	Риск создания кольцевой зависимости. Накладные расходы.

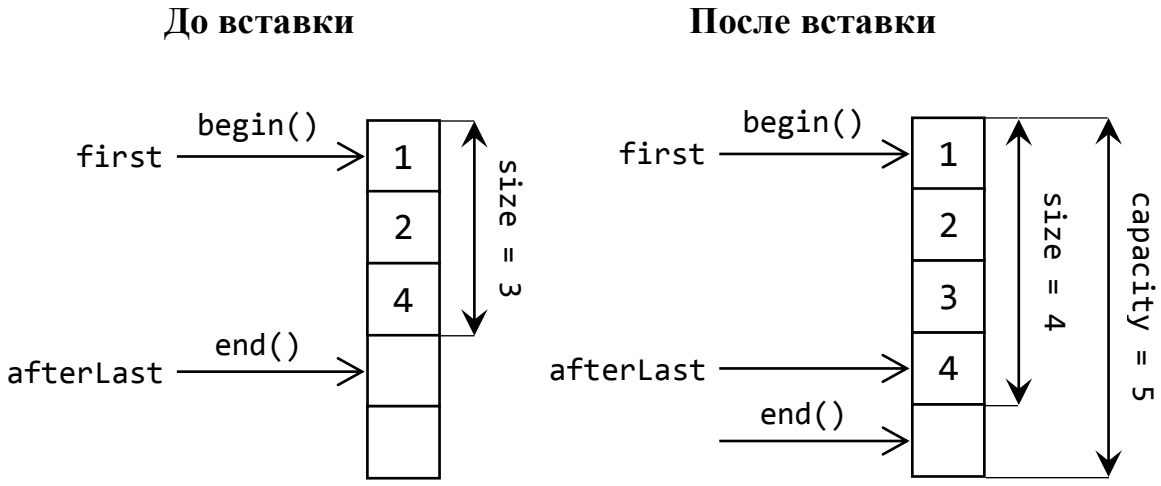

Рисунок 1 — Поведение итераторов при вставке элемента в контейнер

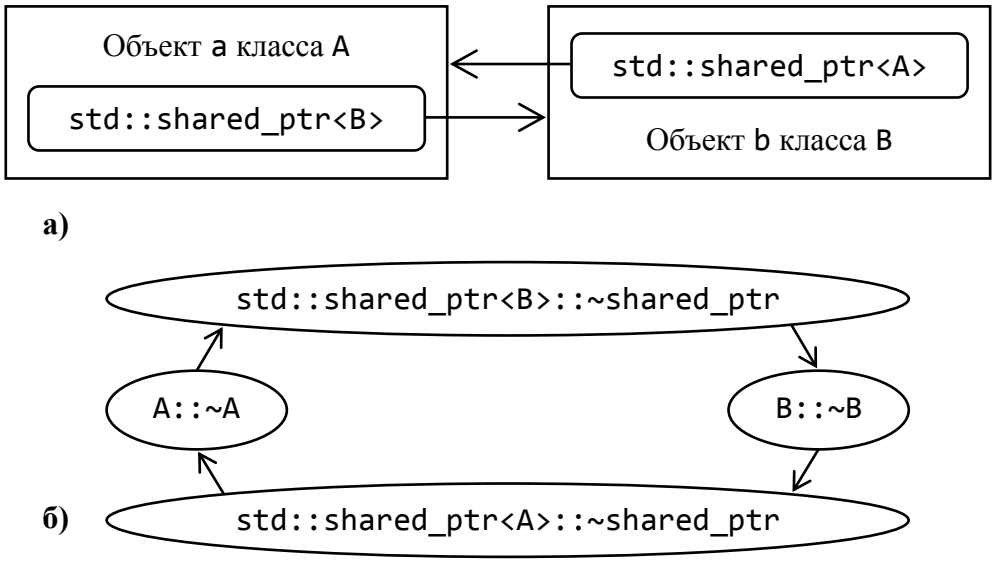

Рисунок 2 — Проблема кольцевой зависимости: объекты и указатели в памяти (а) и диаграмма вызовов деструкторов (б)