Лекция № 7.
Шаблоны C++
и стандартная библиотека шаблонов

СОДЕРЖАНИЕ

1 Шаблоны ... 3
1.1 Обобщенное программирование ... 3
1.2 Специализация шаблонов ... 4
1.3 Шаблоны классов ... 4
1.4 Спецификатор `typedef` ... 5
1.5 Понятие о метапрограммировании ... 5
2 Стандартная библиотека шаблонов ... 7
2.1 Контейнеры ... 7
2.2 Итераторы ... 9
2.2.1 Поведение итераторов при манипуляциях с контейнером ... 10
2.2.2 Цикл `for` для диапазона (range-based `for` statement) ... 11
2.2.3 Забота о производительности при использовании итераторов ... 12
2.2.4 Классификация итераторов ... 12
2.3 Алгоритмы STL. Функторы, привязка и λ-функции ... 13
2.3.1 Функторы и λ-функции ... 13
2.3.2 Привязка функций и методов ... 15
2.3.3 Алгоритмы STL ... 17
2.4 Строки ... 18
2.5 «Умные» указатели ... 18
2.5.1 Виды «умных» указателей по разделению владения ... 19
2.5.2 Проблема кольцевой зависимости ... 20
2.6 Потоки ввода-вывода ... 21
Библиографический список ... 24

1 ШАБЛОНЫ

1.1 Обобщенное программирование

Рассмотрим задачу: реализовать функцию, возвращающую наибольшее из двух чисел любого типа. Листинг 1 демонстрирует решение при помощи перегрузки.

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

Листинг 1 — Перегрузки функции нахождения наибольшего из двух чисел

Предложенное решение громоздко, а вносить в него изменения неудобно, поскольку требуется одинаково править код сразу в нескольких местах. Подобные случаи, когда алгоритмы и структуры данных не зависят от используемых типов, встречаются часто. Например, списки целых чисел (**int**), вещественных чисел (**double**) или объектов отличаются друг от друга только типом данных элементов.

Было бы желательно написать реализацию один раз, обозначив тип элемента как T, а затем пользоваться ей с подстановкой `T = int`, `T = double` и т. д. Эта возможность называется *обобщенным программированием (generic programming*) [2, глава 24], и шаблоны C++ её поддерживают в виде, представленном в листинге 2.

```
1   template<typename T>
2   T max(T a, T b) {
3       return a < b ? b : a;
4   }
5   …
6   max(1, 2);       // max<int>(int a, int b)
7   max(2.71, 3.14); // max<double>(double a, double b)
```

Листинг 2 — Шаблонная функция нахождения наибольшего из двух значений

Строка 1 содержит заголовок шаблона, начинающийся с ключевого слова **template**. В угловых скобках перечислены параметры шаблона: в данном случае параметр один, и это имя типа (**typename**[1]). Параметрами шаблона также могут выступать значения встроенных типов, таких как **int**, **bool** и т. п., пример см. в подразделе 1.5 [1, подразделы 14.1 и 14.5].

Новая функция `max<T>()` может быть вызвана с аргументами любого типа, как показано в строках 6 и 7. Для каждой подстановки `T = mun` компилятором будет сгенерирована отдельная версия функции. Исходная функция `max<T>()` называется

[1] По историческим причинам вместо **typename** в заголовке шаблона допускается использовать, и часто используют, ключевое слово **class**.

определением шаблона (template definition), а любая конкретная реализация — его *воплощением (instantiation).*

Возникает вопрос об операциях, которые допустимо производить над типом `T`: что, если тип не поддерживает сравнение оператором «<»? В этом случае компиляция окажется невозможной. Однако, пока ни в одном из воплощений шаблона ошибок такого рода не возникает, допускаются любые манипуляции над значениями типа `T`: включение в выражения, вызов у них методов, работа с динамической памятью.

1.2 Специализация шаблонов

Иногда требуется для определенных типов реализовать шаблон иначе, чем для других. Например, для строк C (`T = const char*`) реализация `max<T>()` из листинга 2 бессмысленна: будут сравнены указатели, а следует сравнивать символы. Проблема решается *специализацией шаблона (template specialization)* [1, п. п. 14.5.5, 14.7.3 и 14.8.1], как показано в листинге 3.

```
1       template<>
2       const char* max(const char* a, const char* b) {
3            return strcmp(a, b) < 0 ? b : a;
4       }
```

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

Тип `T` исключен из заголовка шаблона, поскольку заменен на конкретный. Теперь для типа `const char*` будет применяться специализированная версия шаблона, а для всех прочих типов — основная.

1.3 Шаблоны классов

Шаблонными могут быть не только функции, но и целые классы [1, п. 14.5.1], как это показано в листинге 4 на примере класса динамического массива.

```
1       template<typename T>
2       class DynamicArray {
3            T* elements;
4       public:
5            DynamicArray(const size_t size) {
6                 elements = size ? new T[size] : 0;
7            }
8            ~DynamicArray() { delete[] elements; }
9            operator T* () { return elements; }
10      };
11
```

```
12        DynamicArray<double> squares(7);
13        squares[0] = 906.01;
14        printf("Square root of %g equals %g. It all seemed harmless..",
15              squares[0], sqrt(squares[0]));
```

Листинг 4 — Шаблонный класс и его применение

Тип-параметр `T` объявляется в заголовке шаблона перед классом, как на строке 1. Конкретный тип указывается для каждого воплощения шаблона, как в строке 12. Оператор приведения к указателю на `T`, определенный в строке 9, позволяет использовать `DynamicArray<T>` как динамический массив с автоматическим освобождением памяти.

Шаблонные классы могут иметь специализации и шаблонные функции-члены.

Шаблон класса должен целиком размещаться в одном файле (единице трансляции).

1.4 Спецификатор `typedef`

Зачастую названия типов-воплощений оказываются громоздкими. Определить тип-псевдоним для существующего можно при помощи спецификатора **`typedef`** [1, п. 7.1.3]:

`typedef` *`старое_имя новое_имя`*;

Например:

```
typedef DynamicArray<double> DoubleArray;
DoubleArray values(7);
```

Определение типа с **`typedef`** может размещаться в любой области видимости.

1.5 Понятие о метапрограммировании

Метапрограммирование (metaprogramming) означает изменение или генерацию кода программы во время компиляции. Метапрограммирование возможно реализовать при помощи шаблонов, поскольку по их воплощениям во время компиляции генерируются классы, и аргументами шаблонов выступают как типы, так и выражения [2, подраздел 28.1].

Рассмотрим типичную задачу метапрограммирования: нужно из двух типов выбрать наиболее вместительный. Например, требуется использовать для математических расчетов самый точный тип, будь то **`double`** или **`long double`**. Листинг 5 демонстрирует решение.

```
1         template<bool FirstIsLarger, typename T1, typename T2>
2         struct X { };
3
4         template<typename T1, typename T2>
5         struct X<true, T1, T2>  { typedef T1 Type; };
6
7         template<typename T1, typename T2>
8         struct X<false, T1, T2> { typedef T2 Type; };
```

```
9
10      template<typename T1, typename T2>
11      struct SelectLargestType {
12           typedef
13                typename X<sizeof(T1) > sizeof(T2), T1, T2>::Type
14                Result;
15      };
16
17      typedef SelectLargestType<long double, double>::Result Number;
```

Листинг 5 — Выбор наиболее вместительного типа метапрограммированием

В строках 1—2 объявляется пустая вспомогательная структура-шаблон `X` с тремя параметрами: логическим значением `FirstTypeIsLarger` и двумя типами, `T1` и `T2`. Строки 4—5 и 7—8 содержат 2 специализации X для обоих возможных значений первого параметра. Если `FirstTypeIsLarger == true`, псевдоним `X::Type` будет означать `T1`, иначе — `T2`.

На строке 13 шаблон структуры `X` воплощается. Первым аргументом ему передается логическое значение `sizeof(T1) > sizeof(T2)`. Если размер `T1` больше, чем размер `T2`, это выражение равно **`true`**, будет выбрана специализация `X` в строках 4—5, и псевдоним `X<…>::Type` будет означать `T1`. В противном случае, если размер `T2` не меньше размера `T1`, псевдоним `X<…>::Type` будет означать `T2`. Итого, `X<…>::Type` будет псевдонимом для наибольшего из типов `T1` и `T2`.

Оператор **`typedef`** на строке 12 переименовывает громоздкое выражение для `X<…>::Type` в `SelectLargestType<…>::Result`. Ключевое слово **`typename`** в строке 12 нужно для указания компилятору, что `X<…>::Type` является именем типа.

Наконец, строка 17 создает псевдоним `Number` для типа `SelectLargestType<long double, double>::Result`, который является псевдонимом для типа `X<true, long double, double>::Type`, который является псевдонимом для **`long double`**. Таким образом, `Number` становится псевдонимом для **`long double`**, если размер последнего больше размера **`double`**. В противном случае `Number` станет псевдонимом для **`double`**.

Шаблонное метапрограммирование — это очень сложный и очень мощный инструмент C++. С его помощью можно на этапе компиляции выполнить любые действия. Существуют шаблонные решения для проведения математических расчетов, вычисления справочных таблиц и т. п. С другой стороны, поддержка такого кода сложна, поэтому на практике злоупотребление метапрограммированием порицается. В настоящем курсе метапрограммирование не изучается, для его освоения предлагаются [3, 4].

2 СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

Стандартная библиотека шаблонов C++ (Standard Template Library, STL) содержит шаблонные классы и функции, реализующие типовые структуры данных и алгоритмы, удобные классы строк, элементы многопоточности, объектно-ориентированные средства для работы с датой и временем и т. п. В настоящей лекции производится обзор только важнейших компонент и концепций STL.

2.1 Контейнеры

Контейнерами STL (containers) называются классы, реализующие структуры данных для хранения коллекций: массивов, списков и т. п. Перечень основных контейнеров STL приведен в таблице 1. Полный перечень и описания см. в [1, раздел 23].

Таблица 1 — Основные контейнеры STL

Класс[2]	**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).

[2] Здесь и далее в разделе, где это не оговорено особо, подразумевается пространство имен `std`.

`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`.

Пример работы с контейнером приведен в листинге 6.

```
1       unsigned limit = 1000;
2       std::vector<unsigned> numbers;
3       numbers.reserve(limit - 1);
4       for (unsigned i = 2; i < limit; ++i)
5            numbers.push_back(i);
6       for (size_t i = 0; i < numbers.size(); ++i) {
7            size_t j = i + 1;
8            while (j < numbers.size()) {
9                 if (numbers[j] % numbers[i]  == 0)
10                     numbers.erase(&numbers[j]);
11                else
12                     ++j;
13           }
14      }
15      numbers.clear();
```

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

В строке 2 объявляется переменная `numbers` — контейнер для беззнаковых чисел. В строке 3 в контейнере резервируется место под `limit` элементов, чтобы при вставке в контейнер (строка 5) тому не приходилось выделять все большую область памяти. Количество элементов в любом контейнере STL можно получить методом `size()`, как в строках 6 и 8. Контейнер `std::vector` индексируется[3] как простой массив (строка 9). Удалять элементы из `std::vector` можно методом `erase()`, имея адрес, как показано в строке 10. Так как элементы, стоявшие после удаляемого, сдвинутся при этом на освободившуюся позицию, переходить к следующему элементу (`++j`) после удаления не требуется. В строке 15 контейнер очищается; при этом занимаемая память не освобождается, а это будет сделано автоматически в деструкторе `std::vector`.

[3] На практике рекомендуется вместо индексации оператором `[]` пользоваться методом `at()`, возбуждающим исключение `std::out_of_range` при обращении по недопустимому индексу.

2.2 Итераторы

Рассмотрим обработку массива при помощи указателей на его элементы (листинг 7).

```
1       const size_t length = 42;
2       double container[length];
3       double* begin = &container[13];
4       double* end = &container[27];
5       for (auto iterator = begin; iterator != end; ++iterator) {
6            printf("%g", *iterator);
7       }
```

Листинг 7 — Обработка массива при помощи указателей на его элементы

Обрабатывается *диапазон (range)* элементов массива с 13-го по 26-й (то есть, левая граница `begin` включается, а правая `end` — нет).

В цикле обработки потребовались операции:

1) Разыменование (*), при помощи которого можно получить элемент, на который указывает `iterator` в текущий момент (говорят: «где, или на котором, находится `iterator`»).
2) Перемещение к следующему элементу оператором инкремента (`++`).
3) Сравнение указателей друг с другом, чтобы определить достижение конца диапазона обрабатываемых элементов.

Заметим теперь, что цикл обработки не зависит от того, что обрабатывается массив, и что тип `iterator`, `begin` и `end` — указатель. Вместо них можно было бы использовать объекты с перегруженными операторами, которые обеспечивали бы операции 1) — 3), например, для связанного списка. Такие объекты называются *итераторами (iterator)* и широко применяются в STL. Итераторы позволяют реализовывать алгоритмы над контейнерами без привязки к типу контейнера [2, подраздел 33.1]. В свою очередь, контейнеры STL позволяют получить итераторы для своего обхода. Листинг 8 демонстрирует получение итераторов для связанного списка `std::list` и обработку элементов с их помощью.

```
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`

Цикл обработки, по существу, не отличается от случая для массива. Метод `begin()` у контейнеров STL возвращает итератор, «находящийся» *на начальном* элементе, а метод `end()` — итератор *за последним* элементом. Использование **`auto`** для вывода типа

итератора весьма удобно, поскольку полное наименование типа итератора `i` громоздко: `std::list<double>::iterator`.

2.2.1 Поведение итераторов при манипуляциях с контейнером

Важным аспектом любого контейнера является указание, после каких операций над контейнером ранее полученные итераторы становятся *непригодными к использованию (invalid)*.

Пусть, например, контейнер хранит информацию в динамическом массиве `data` размером `capacity`, в котором заняты данными `size` элементов; в качестве итераторов используются указатели на элементы `data`. Вставим новый элемент «3» в середину контейнера, сдвинув правую часть массива на одну позицию, как показано на рис. 1.

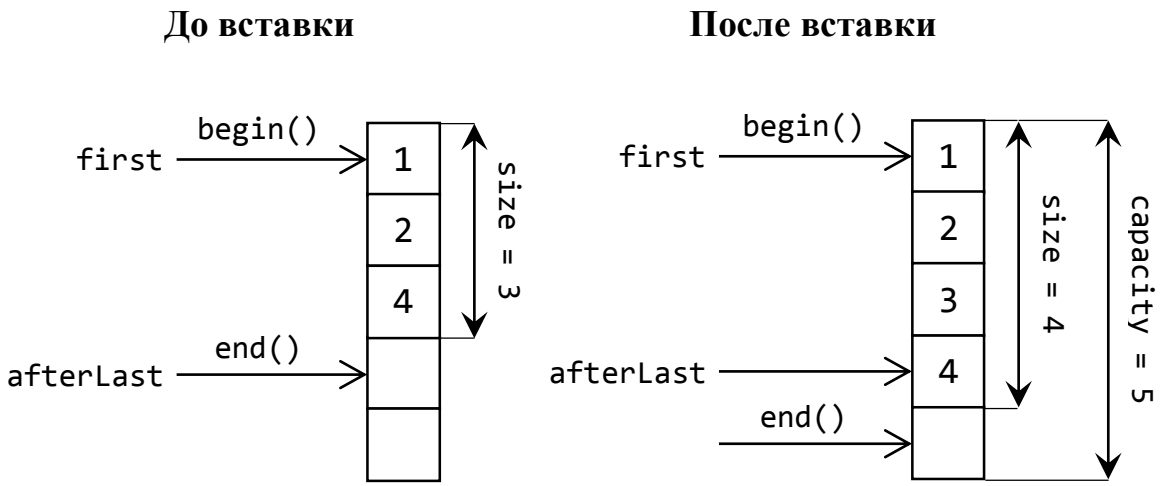

Рисунок 1 — Поведение итераторов при вставке элемента в контейнер

Итератор `afterLast` был получен до вставки методом `end()` и указывал за последний элемент. После вставки этот итератор не изменился (адрес тот же), но указывает он на последний элемент, поэтому итератор `afterLast` больше непригоден (invalid) для использования как граница массива. Напротив, итератор `first` остался пригодным (valid) к использованию.

В более сложных случаях непригодными могут стать все итераторы. Например, если бы не хватило `capacity` для размещения нового элемента, потребовалось бы выделить новую область памяти под данные, а старую освободить. Указатели на элементы старой области памяти потеряли бы при этом смысл, став непригодными. Поведение итераторов при различных манипуляциях с контейнерами закреплено стандартом C++ в описаниях методов классов-контейнеров.

Листинг 9 содержит пример работы с итераторами, при которой необходимо учитывать их пригодность после удаления элемента из контейнера.

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

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

Допущена типичная ошибка: операция `erase()` делает непригодным к использованию переданный её итератор, поэтому переход к элементу, следующему за удаленным, как `++i`, неправомочен. Правильный способ показан в листинге 10.

```
1       auto i = container.begin();
2       while (i != container.end()) {
3            if (*i % 2 == 0)
4                 i = container.erase(i);
5            else
6                 ++i;
7       }
```

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

Метод `std::list::erase()` возвращает итератор, соответствующий элементу за удаленным, поэтому итератор `i` всегда остается пригодным к использованию, а также не происходит пропуска элементов, поскольку `++i` выполняется, только если удаление не производилось.

2.2.2 Цикл `for` для диапазона (range-based `for` statement)

Стандартом C++11 введен новый вид цикла **`for`**, основанный на итераторах и удобный для перебора контейнеров. Он называется *range-based **`for`** statement* [1, п. 6.5.4] и применяется, как показано в листинге 11.

```
1       int array[4] = { 1, 2, 3, 4 };
2       for (auto& item : array) {
3            item *= item;
4       }
5       for (int item : array) {
6            printf("%d", item);
7       }
```

Листинг 11 — Цикл `for` для диапазона

Первый цикл (строки 2—4) оперирует `item` как ссылкой на очередной элемент, поэтому допускается изменять `item`. Второй цикл (строки 5—7) оперирует `item` как копией очередного элемента, поэтому изменить с её помощью массив нельзя.

2.2.3 Забота о производительности при использовании итераторов

Производительность операций над итераторами различается между контейнерами. Например, для связанного списка вычисление размера имеет линейную сложность. При использовании итераторов следует руководствоваться такими оптимизациями:

а) По возможности получать и сохранять результат `end()` или размер коллекции методом `size()` один раз перед циклом, если они остаются пригодными на всем протяжении работы цикла.

б) Использовать префиксную форму инкремента (`++i`), чтобы не копировать старое значение итератора в качестве результата инкремента.

2.2.4 Классификация итераторов

К итераторам предъявляются неодинаковые требования. Например, если коллекция чисел задана диапазоном, для ее печати нужно только передвигаться вперед от начального итератора до конечного и считывать (но не изменять) их значения. При сортировке же требуется, по меньшей мере, изменять значения, полученные через итераторы. Таким образом, к итератором можно предъявлять наборы требований, которые зачастую типовые.

Принято [1, подраздел 24.2] различать итераторы:

а) По результату разыменования:

1) входных данных (input), если результат разыменования предназначен только для чтения, и его нельзя изменить;

2) выходных данных (output), если возможно изменить результат разыменования, в частности, присвоить ему значение.

б) По направлению движения:

1) поступательные (forward), для которых доступен только оператор `++`;

2) возвратные (backward), для которых доступен только оператор `--`;

3) двунаправленные (bidirectional), для которых доступны и `++`, и `--`.

в) Произвольного доступа (random access), если из данного итератора можно получить другой, прибавив к нему целое число (или воспользоваться индексацией).

Вообще говоря, итератор может и не быть связан с контейнером. Например, если оператор инкремента всегда возвращает `*this`, сравнение итератора с любым другим имеет результат `false`, а результат разыменования вычисляется динамически, получается итератор по бесконечной последовательности (случайных чисел, чисел Фибоначчи и т. п.).

2.3 Алгоритмы STL. Функторы, привязка и λ-функции

Над контейнерами существует много типовых операций: поиск элемента, сортировка, удаление элементов по условию и т. п. STL предлагает их реализацию в виде функций, называемых алгоритмами и расположенных в заголовочном файле `<algorithm>`. Пример использования алгоритмов приведен в листинге 12.

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

Листинг 12 — Пример использования алгоритмов STL

В строке 2 контейнер упорядочивается по возрастанию, а в строке 3 проверяется, что это действительно сделано [1, п. 25.4.1]. Можно видеть, что алгоритмы обычно принимают контейнер в виде пары итераторов. Этим можно воспользоваться, чтобы обрабатывать любые части контейнеров: в строке 4 формируется итератор `middle`, соответствующий середине контейнера. В строке 5 происходит поиск элемента «3», причем не во всем контейнере, а от его начала `begin()` до середины `middle` (не включая `*middle`). Алгоритм `std::find()` возвращает итератор-конец диапазона, если элемент не найден, что характерно для многих алгоритмов поиска [1, п. 25.2.5].

2.3.1 Функторы и λ-функции

Некоторые алгоритмы связаны с выполнением произвольных действий. Например, алгоритм `std::remove_if()` удаляет элементы, удовлетворяющие условию [1, п. 25.3.8]. Возникает вопрос: как передать алгоритму это условие? Условие используется этим алгоритмом как функция, которой передается элемент, и если возвращаемое значение — **`true`**, он удаляется из контейнера. Можно воспользоваться указателем на функцию, как это показано в листинге 13.

```
1       bool greaterThen4(const int element) {
2            return element > 4;
3       }
4       …
5       std::remove_if(numbers.begin(), numbers.end(), greaterThen4);
```

Листинг 13 — Использование указателя на функцию с алгоритмом STL

Главный недостаток решения очевиден: из места вызова `std::remove_if()` нельзя передать условию никаких данных, если только не пользоваться глобальными

переменными, что «засорило» бы окружающий код. Даже с применением шаблонов не избежать написания новых функций, аналогичных `greaterThen4()`, для удаления элементов, больших 5, 6 и т. д.

Желательно было бы иметь возможность:

1) Определять функции, передаваемые алгоритмам, прямо по месту вызова;
2) Захватывать (to capture) часть локальных переменных в такую функцию.

Это называется созданием замыканий[4] (closures), или λ-функций (λ-functions), что на практике принято сокращать до лямбд (lambdas). Листинг 14 демонстрирует использование λ-функций для удаления элементов, меньших локальной переменной `n`.

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

Листинг 14 — Использование λ-функций с алгоритмами STL

Выражение в строке 4 и есть выражение λ-функции, включающее 3 части [1, п. 5.1.2]:

1) *Список захвата (lambda capture)* в квадратных скобках, в данном случае это значение локальной переменной `n`.
2) Список параметров в круглых скобках, в данном случае аргумент единственный — элемент контейнера.
3) Тело λ-функции в фигурных скобках, в данном случае — возврат результата сравнения элемента с захваченным значением.

Захват переменных с места определения λ-функции (говорят: «из контекста») может быть выполнен двумя способами.

а) По значению, когда тело λ-функции оперирует копиями захваченных переменных, а сами они не изменяются. Копии также неизменяемы. Этот способ действует по умолчанию; можно добиться того же, предваряя имена захватываемых переменных знаком равенства (`=`).

б) По ссылке, когда тело λ-функции оперирует ссылками на захваченные переменные, изменяя их. Этот способ используется, если имя переменной предваряется символом амперсанда (`&`).

В листинге 15 приведен пример захвата локальных переменных различными способами для подсчета суммы квадратов.

```
1       std::array<double, 5> array = { 1, 2, 3, 4, 5 };
2       double exponent = 2;
```

[4] Данные термины исторически заимствованы из функционального исчисления (lambda calculus). Строго говоря, замыкания и λ-функции — разные понятия, но здесь для рассмотрения это не существенно.

```
3       double sum = 0.0;
4       std::for_each(
5            array.begin(), array.end()
6            [exponent, &sum] (double x) {
7                 const double power = pow(x, exponent);
8                 sum += power;
9            });
```

Листинг 15 — Захват локальных переменных в λ-функции

После вызова `std::for_each()` в переменной `sum` будет накоплена сумма элементов массива, возведенных в степень `exponent`, поскольку `sum` была захвачена по ссылке. Алгоритм `std::for_each()` выполняет проход по диапазону и совершает над каждым элементом указанное действие [1, п. 25.2.4].

Чтобы захватить все используемые в λ-функции переменные по значению или по ссылке, вместо переменных в списке захвата следует оставить просто знак `=` или `&` соответственно. Захват **`this`** всегда должен выполняться по ссылке.

Технически объявление λ-функции соответствует объявлению анонимного класса, который содержит все захваченные значения и ссылки в качестве полей, а также реализует оператор `()`. Такие объекты называются *функторами (function objects),* и до введения в C++11 λ-функций те же задачи решались написанием функторов вручную. Коль скоро λ-функция является объектом, можно присвоить её переменной, чтобы использовать повторно; при этом потребуется указать тип возвращаемого значения после списка параметров через `->`, как показано в листинге 16.

```
1       int N = 42, m = 32;
2       auto greaterThenN = [N](auto x) -> bool {
3            return std::greater(x, N);
4       };
5       std::cout << m << " больше " << N << "?"
6            << greaterThenN(m) ? "Да." : "Нет." << std::endl;
```

Листинг 16 — Использование λ-функции как переменной

STL предоставляет ряд стандартных функторов для простых операций, например, `std::is_even()` для проверки, что аргумент четный, `std::greater()` для проверки, что первый аргумент больше второго, и т. п. [1, подраздел 20.10].

2.3.2 Привязка функций и методов

Часто требуется создавать функторы двух простых типов:

а) вызов метода конкретного объекта;

б) вызов функции (или метода), когда часть аргументов фиксирована, а часть — аргументы, передаваемые функтору.

Разумеется, можно захватывать объект и значения фиксированных аргументов, но так как это типовая задача, для её решения в STL предусмотрена специальная возможность — *привязка функций (binding),* пример которой представлен в листинге 17.

```
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

Объект класса `Bank` хранит список счетов в виде `std::list`. Имеется метод `sendGifts()` для зачисления на все счета заданной суммы `amount`. Привязка осуществляется функцией `std::bind()`: ей передается адрес метода, который реализует действие, и указатель на объект, у которого данный метод будет вызываться [1, п. 20.10.9]. Используя привязку, можно решить задачу двумя способами.

Способ № 1 заключается в том, чтобы вызывать метод `deposit()` с аргументом `amount` у очередного элемента. В качестве объекта в этом случае передается *местозаполнитель (placeholder)* `_1` из пространства имен `std::placeholders`. При вызове функтора _1 будет заменен на первый из аргументов, переданных функтору, то есть на очередной объект (аналогично для `_2` и т. д.). Далее в списке аргументов `std::bind()` следуют значения параметров вызываемого метода, в данном случае — значение суммы средств `amount`.

Способ № 2 состоит в привязке метода `makeDeposit()` данного объекта (**`this`**). Этому методу понадобится передавать 2 параметра: первый аргумент функтора `_1`, то есть очередной счет, и сумму средств `amount`.

Функции STL для привязки сосредоточены в заголовочном файле `<functional>`.

2.3.3 Алгоритмы STL

Имеется около 60-и алгоритмов STL [1, раздел 25, подраздел 26.7], каждый из которых обладает несколькими перегрузками или семейством подобных (например, `std::remove` и `std::remove_if`). Важнейшие алгоритмы перечислены в таблице 2.

Таблица 2 — Некоторые алгоритмы STL

Алгоритм	Операция	Примечания
`for_each`	перебор	Проходит по всем элементам контейнера (см. ниже) и передает их функтору.
`find`	поиск одного элемента	Возможен поиск конкретного элемента или удовлетворяющего предикату (см. ниже).
`remove`		
`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.

Примечания о терминологии:

1) Контейнеры задаются диапазонами `[начало; конец)`, если данные из контейнера поступают, или итератором-началом, если данные в контейнер записываются.
2) *Предикат (predicate)* — это функтор-условие, возвращаемый тип которого — **`bool`**, а параметры зависят от алгоритма, и обычно это очередной элемент при переборе.

3) *Компаратор (comparer)* — это функтор, принимающий 2 параметра, значения которых необходимо сравнить и вернуть положительное число, если первый больше, отрицательное — если меньше, и 0, если они равны.

2.4 Строки

STL предусмотрены классы для символьных строк в заголовочном файле `<string>`. Основным является класс `std::string` (символы — **`char`**) и несколько подобных ему, в которых символы представляются более длинными кодами [1, подраздел 21.4]. Рассмотрим пример работы с `std::string`, приведенный в листинге 18.

```
1       const char* vulgarHyphen = " – ";
2       size_t length = strlen(vulgarHyphen);
3       std::string properDash = "\u00A0\u2013 ";
4       std::string input;
5       std::cin >> input;
6       do {
7            auto index = input.find(vulgarHyphen))
8            if (index == std::string::npos)
9                 break;
10           input.replace(index, length, properDash);
11      } while (true);
12      puts(input.c_str());
```

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

Программа выполняет замену дефисов, обрамленных пробелами, на длинные тире («—», код U+2013) с неразрывным пробелом (код U+00A0). В строке 3 демонстрируется создание объекта-строки по заданной строке C. На строке 7 применяется метод `find()` для поиска подстроки; он возвращает `std::string::npos`, если подстрока не найдена. Метод `replace()`, вызываемый на строке 10, в строке `input` заменяет `length` символов, начиная с позиции `index`, на строке `properDash`. В строке 12 производится печать функцией `puts()`; для получения из `std::string` строки C используется метод `c_str()`.

Класс `std::string` является также полноценным контейнером (символов), поэтому имеет итераторы, и может быть обработан алгоритмами STL.

2.5 «Умные» указатели

При работе с динамической памятью требуется бороться с её утечками, для чего необходимо отслеживать владение динамически размещенными данными. Эффективным решением этой задачи являются *«умные» указатели (smart pointers)* [2, подраздел 34.3]. Так называют объекты, способные автоматически удалять объекты, ставшие в программе

ненужными, и наоборот, продлевающие время жизни объектов до тех пор, пока на них продолжают ссылаться иные. Листинг 19 демонстрирует применение «умных» указателей.

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

Листинг 19 — Пример использования «умных» указателей

Контейнер `accounts`, объявляемый на строке 1 — это связанный список, в узлах которого хранятся «умные» указатели на объекты-счета `std::shared_ptr<Account>`. В строках 2—4 и 5—6 в список добавляются два «умных» указателя. Первый из них создается вызовом конструктора на строке 3. Второй получен на строке 6 функцией `std::make_shared<T>()`, которая создает в динамической памяти объект заданного класса, передав его конструктору указанные параметры, и возвращает «умный» указатель на него [1, параграф 20.9.2.2.6]. Строка 8 демонстрирует, что «умные» указатели можно использовать прозрачно, как будто это простые указатели на объекты.

Программа в листинге 19 полностью свободна от утечек памяти, так как деструктор списка вызывает деструкторы элементов-«умных» указателей, а те, в свою очередь, вызывают деструкторы объектов-счетов и освобождают память под них.

«Умные» указатели не предназначены для работы с памятью, выделенной через **`new[]`**, потому что освобождают её по умолчанию через **`delete`** (что настраиваемо). Это не является проблемой, поскольку для управления коллекциями объектов в STL предусмотрены контейнеры.

2.5.1 Виды «умных» указателей по разделению владения

В случае, когда несколько «умных» указателей относятся к одним и тем же данным, возникает проблема *разделяемого владения* (*shared ownership*) этими данными. Способы её решения приведены в таблице 3.

Таблица 3 — Способы решения проблемы разделяемого владения

Решение и класс в STL	**Описание**	**Преимущества**	**Недостатки**
Запрет разделяемого владения	Динамически размещенными данными владеет только один объект, который либо удаляет	Простота, полное решение проблемы.	Риск уничтожения данных при копировании

(unique_ptr, [1, п. 20.9.1])	их в своем деструкторе, либо передает владение другому «умному» указателю.		«умного» указателя. Иногда разделяемое владение нужно.
Применение «слабых» ссылок (weak_ptr, [1, п. 20.9.3])	Один из «умных» указателей владеет динамически размещенными данными и может их уничтожить, остальные указатели — «слабые» (weak), и могут только проверить, что данные еще не удалены.	Нет риска создания кольцевой зависимости (см. ниже).	Данные могут быть удалены, даже когда «слабый» указатель еще существует.
Подсчет ссылок (shared_ptr, [1, п. 20.9.2])	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.

2.5.2 Проблема кольцевой зависимости

Проблема создания кольцевой зависимости возникает при использовании решения std::shared_ptr и заключается в бесконечной рекурсии, как это показано на рис. 2.

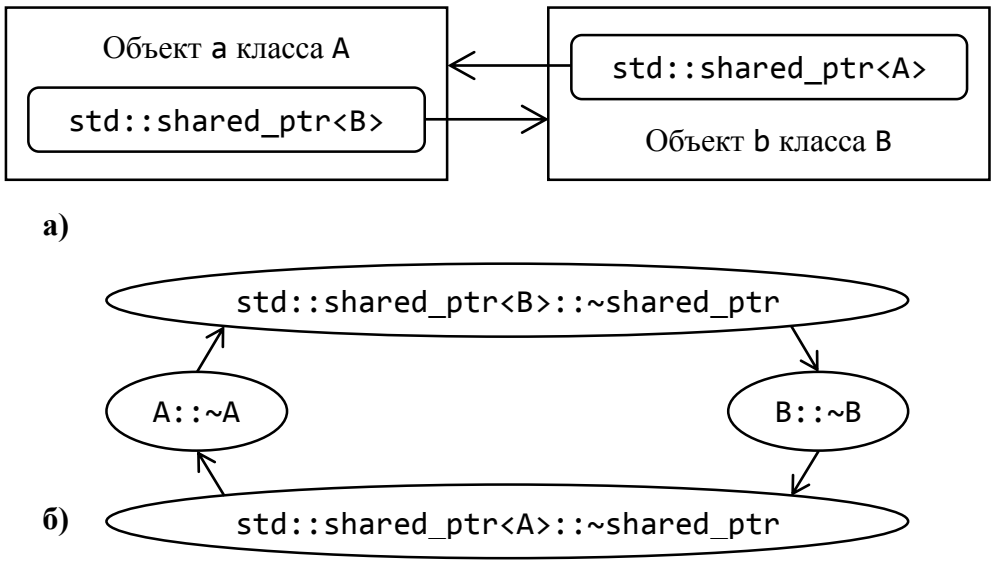

Рисунок 2 — Проблема кольцевой зависимости: объекты и указатели в памяти (а) и диаграмма вызовов деструкторов (б)

Пусть объект a содержит «умный» указатель на объект b, а объект b — на объект a. Тогда в деструкторе объекта a будет уничтожен «умный» указатель на объект b. Деструктор «умного» указателя выполнит уничтожение объекта b с вызовом деструктора последнего.

В деструкторе объекта `b` будет уничтожен «умный» указатель на объект `a`. Деструктор этого «умного» указателя выполнит уничтожение объекта `a`, при этом деструктор объекта `a` будет вызван повторно. Таким образом, возникает бесконечная рекурсия.

Избавиться от кольцевой зависимости можно, сделав один из «умных» указателей «слабым» (`std::weak_ptr`). Так часто поступают в иерархических структурах: элемент верхнего уровня (родитель) содержит «сильные» ссылки на дочерние, а те, в свою очередь, — «слабые» ссылки на родителя. Например, банковский счет может содержать «сильные» указатели на карты, а карты — «слабые» указатели на счет, тогда уничтожение карты не повлечет закрытия счета, но закрытие счета аннулирует все карты. На практике следует иметь в виду, что кольцо зависимости может быть длиннее, чем показано в примере.

2.6 Потоки ввода-вывода

STL предоставляет объектно-ориентированную потоковую библиотеку ввода-вывода. *Поток (stream)* — это последовательность, из которой данные можно извлекать или помещать в нее. Например, поток стандартного ввода состоит из символов, которые помещаются в него операционной системой и извлекаются программой. Бьярн Страуструп исторически является популяризатором потоков ввода-вывода как в C++, так и в целом, поэтому фундаментально изложение их концепции в [2, глава 38]. По сравнению с потоками-файлами стандартной библиотеки языка C обеспечиваются преимущества:

1) Типобезопасность (type safety) — отсутствие риска повреждения памяти из-за неправильного указания типа данных и их размера. Например, команды считывания нескольких байт и записи их по указанному адресу используется команда считывания целого числа.

2) Автоматическое управление ресурсами (resource management) — отсутствие утечек ресурсов (памяти, незакрытых файловых дескрипторов) за счет их освобождения в деструкторах объектов-потоков.

3) Удобство использования за счет применения механизмов C++ — возможность, например, сделать считывание и запись объектов собственного класса в поток такой же, как и встроенных типов.

4) Интеграция в STL — удобный ввод и вывод строк `std::string` и контейнеров.

К недостаткам потоков STL можно отнести пониженную производительность и громоздкость форматирования, реализуемого средствами заголовочного файла `<iomanip>` [1, п. п. 27.7.4—27.7.5].

Классы потоков STL расположены в файлах <iostream>, <fstream>, <sstream> и других, а их описание расположено в [1, раздел 27].

Листинг 20 демонстрирует получение чисел со стандартного ввода и запись их в указанный файл через пробел.

```
1       std::string fileName;
2       std::getline(std::cin, fileName);
3       std::vector<double> numbers;
4       while (std::cin.good()) {
5            double value;
6            std:cin >> value;
7            numbers.push_back(value);
8       }
9       std::ofstream output(fileName);
10      std::copy(
11           numbers.begin(), numbers.end(),
12           std::ostream_iterator<double>(output, " "));
13      std::cout << "Задача завершена!" << std::endl;
```

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

В строке 2 из потока стандартного ввода std::cin считывается строка-имя файла.

В строке 6, заключенной в цикл, из std::cin считываются числа при помощи перегруженного оператора >>. Условие цикла демонстрирует важную особенность потоков STL — способность реагировать на ошибки и восстанавливаться после них. Поток считается находящимся в «хорошем» состоянии, если ошибок не происходило, и в этом случае метод good() возвращает **true**. При возникновении ошибки возвращаемое значение было бы false, но это можно было бы обнаружить, обработать ошибку и вновь привести поток в нормальное состояние.

На строке 9 объявляется объект класса std::ofstream, предназначенного для вывода данных в указанный аргументом конструктора файл. При окончании времени жизни данного объекта файл будет автоматически закрыт.

Строки 10—12 демонстрируют интеграцию потоков ввода-вывода с контейнерами и алгоритмами STL. Имеется возможность создать особый итератор класса std::ostream_iterator, который будет представлять (для алгоритмов) поток вывода как контейнер, в который можно скопировать данные; на самом же деле произойдет не копирование элементов, а их запись в поток.

В строке 13 на стандартный вывод печатается сообщение. В данной инструкции следует обратить внимание на вывод в поток специального значения std::endl, что служит двум целям. Во-первых, печатаются символы перехода на следующую строку.

Во-вторых, производится сброс буфера потока: если в целях оптимизации[5] поток не печатал символы на экран, а накапливал их во внутреннем буфере, после вывода `std::endl` печать будет гарантированно произведена, а буфер потока очистится.

Распространенной задачей является вывод объектов пользовательских классов в потоки STL. Например, может быть желательно печатать объект-банковский счет в `std::cout`. Решением данной задачи является перегрузка оператора `<<` для этого случая, как показано в листинге 21.

```
1       std::ostream& operator<<
2            (std::ostream& stream, const Account& account) {
3            stream << "На счете " << account.getID() << " находится "
4                   << account.getBalance() << " y. e." << std::endl;
5            return stream;
6       }
```

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

Часто делают такую перегрузку другом (`friend`) класса, чтобы выполнять вывод информации, недоступной извне объекта.

[5] Работа с устройствами ввода-вывода существенно медленнее работы с оперативной памятью.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. ISO/IEC 14882:2011 Programming Languages — C ++. — Введен в 2011 г. — 1338 с.
2. Stroustrup, Bjarne. The C++ Programming Language (Fourth Edition) / Bjarne Stroustrup. — Addison-Wesley. — 2013 г. — 1347 с.
3. Дэвид Вандевурд, Николаи М. Джосаттис. Шаблоны C++. Справочник разработчика / C++ Templates: The Complete Guide. — М.: Вильямс. — 2008 г. — 544 с.
4. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. — Addison-Wesley Professional. — 2001 г. — 352 с.