# **§10.2 Контейнерные классы**

**Контейнерный класс** (или ещё **«класс-контейнер»**) в C++ — это **класс**, предназначенный для хранения и обработки нескольких объектов определённого типа данных (пользовательских или встроенных). Существует много разных контейнерных классов, каждый из которых имеет свои преимущества, недостатки или ограничения в использовании. Безусловно, наиболее часто используемым контейнером в программировании является **массив**. Хотя в C++ есть стандартные обычные массивы, большинство программистов используют контейнерные классы-массивы: **std::array** или **std::vector** из-за преимуществ, которые они предоставляют. В отличие от стандартных массивов, контейнерные классы-массивы имеют возможность динамического изменения своего размера, когда элементы добавляются или удаляются. Это не только делает их более удобными, чем обычные массивы, но и безопаснее.

Обычно, **функционал классов-контейнеров** в C++ следующий:

1. Создание пустого контейнера (через **конструктор**).
2. Добавление нового объекта в контейнер.
3. Удаление объекта из контейнера.
4. Просмотр количества объектов, находящихся на данный момент в контейнере.
5. Очистка контейнера от всех объектов.
6. Доступ к сохранённым объектам.
7. Сортировка объектов/элементов (опционально).

В отличие от реальной жизни, когда контейнеры могут хранить любые типы объектов, которые в них помещают, в C++ контейнеры обычно содержат только один тип данных. Например, если у вас целочисленный массив, то он может содержать только целочисленные значения. В отличие от некоторых других языков, C++ не позволяет смешивать разные типы данных внутри контейнеров. Если вам нужны контейнеры для хранения значений типов int и double, то вам придётся написать два отдельных контейнера (или использовать шаблоны, о которых мы будем говорить в следующих параграфах).

# **§10.3 Контейнерный класс-массив**

Сейчас мы напишем целочисленный класс-массив с нуля, реализуя функционал контейнеров в С++. Этот класс-массив будет типа контейнера значения, в котором будут храниться копии элементов, которые он содержит.

Наш класс ArrayInt должен отслеживать два значения: непосредственно данные и свою длину. Поскольку мы хотим, чтобы наш массив мог изменять свою длину, то нам нужно использовать динамическое выделение памяти, что означает, что мы будем использовать указатель для хранения данных:

**Листинг 11.3**

| 1 | class ArrayInt |
| --- | --- |
| 2 | { |
| 3 | private: |
| 4 | int m\_length; |
| 5 | int \*m\_data; |
| 6 | }; |

Теперь нам нужно добавить конструкторы, чтобы иметь возможность создавать объекты класса ArrayInt. Мы добавим два конструктора: первый будет создавать пустой массив, второй — массив заданного размера:

**Листинг 11.4**

| 1 | class ArrayInt |
| --- | --- |
| 2 | { |
| 3 | private: |
| 4 | int m\_length; |
| 5 | int \*m\_data; |
| 6 | public: |
| 7 | ArrayInt(): m\_length(0), m\_data(NULL) |
| 8 | { |
| 9 | } |
| 10 | ArrayInt(int length): m\_length(length) |
| 11 | { |
| 12 | assert(length >= 0); |
| 13 | if (length > 0) |
| 14 | m\_data = new int[length]; |
| 15 | else |
| 16 | m\_data = NULL; |
| 17 | } |
| 18 | }; |

Нам также потребуются функции, которые будут выполнять очистку ArrayInt. Во-первых, добавим деструктор, который будет просто освобождать любую динамически выделенную память. Во-вторых, напишем функцию erase(), которая будет выполнять очистку массива и сбрасывать его длину на 0:

**Листинг 11.5**

| 1 | ~ArrayInt() |
| --- | --- |
| 2 | { |
| 3 | delete[] m\_data; |
| 4 | // Здесь нам не нужно присваивать значение NULL для m\_data или выполнять m\_length = 0, так как объект и так будет уничтожен |
| 5 | } |
| 6 | void erase() |
| 7 | { |
| 8 | delete[] m\_data; |
| 9 | // Здесь нам нужно указать m\_data значение NULL, чтобы на выходе не было висячего указателя |
| 10 | m\_data = NULL; |
| 11 | m\_length = 0; |
| 12 | } |

Теперь **перегрузим оператор индексации []**, чтобы иметь доступ к элементам массива. Мы также должны выполнить проверку корректности передаваемого индекса, что лучше всего сделать с помощью **выражения assert()**. Также добавим **функцию доступа** для возврата длины массива:

**Листинг 11.6**

| 1 | class ArrayInt |
| --- | --- |
| 2 | { |
| 3 | private: |
| 4 | int m\_length; |
| 5 | int \*m\_data; |
| 6 | public: |
| 7 | ArrayInt(): |
| 8 | m\_length(0), m\_data(NULL) |
| 9 | { |
| 10 | } |
| 11 | ArrayInt(int length): |
| 12 | m\_length(length) |
| 13 | { |
| 14 | assert(length >= 0); |
| 15 | if (length > 0) |
| 16 | m\_data = new int[length]; |
| 17 | else |
| 18 | m\_data = NULL; |
| 19 | } |
| 20 | ~ArrayInt() |
| 21 | { |
| 22 | delete[] m\_data; |
| 23 | } |
| 24 | void erase() |
| 25 | { |
| 26 | delete[] m\_data; |
| 27 | // Указываем m\_data значение NULL, чтобы на выходе не было висячего указателя |
| 28 | m\_data = NULL; |
| 29 | m\_length = 0; |
| 30 | } |
| 31 | int& operator[](int index) |
| 32 | { |
| 33 | assert(index >= 0 && index < m\_length); |
| 34 | return m\_data[index]; |
| 35 | } |
| 36 | int getLength() { return m\_length; } |
| 37 | }; |

Теперь у нас уже есть класс ArrayInt, который мы можем использовать. Мы можем выделить массив определённого размера и использовать оператор [] для извлечения или изменения значений элементов.

Тем не менее, есть ещё несколько вещей, которые мы не можем выполнить с нашим ArrayInt. Это автоматическое изменение размера массива, добавление/удаление элементов, сортировка элементов.

Во-первых, давайте реализуем возможность массива изменять свой размер. Напишем две разные функции для этого. Первая функция: reallocate(), при изменении размера массива будет уничтожать все существующие элементы — это быстро. Вторая функция: resize(), при изменении размера массива будет сохранять все существующие элементы — это медленно.

**Листинг 11.7**

| 1 | void reallocate(int newLength) |
| --- | --- |
| 2 | { |
| 3 | // Удаляем все существующие элементы внутри массива |
| 4 | erase(); |
| 5 | // Если наш массив должен быть пустым, то выполняем возврат здесь |
| 6 | if (newLength <= 0) |
| 7 | return; |
| 8 | // Дальше нам нужно выделить новые элементы |
| 9 | m\_data = new int[newLength]; |
| 10 | m\_length = newLength; |
| 11 | } |
| 12 |  |
| 13 | void resize(int newLength) |
| 14 | { |
| 15 | // Если массив уже нужной длины, то выполняем return |
| 16 | if (newLength == m\_length) |
| 17 | return; |
| 18 | // Если нужно сделать массив пустым, то делаем это и затем выполняем return |
| 19 | if (newLength <= 0) |
| 20 | { |
| 21 | erase(); |
| 22 | return; |
| 23 | } |
| 24 | // Теперь мы можем предположить, что newLength состоит хотя бы из одного элемента. Выполняется следующий алгоритм: |
| 25 | // 1. Выделяем память под новый массив. |
| 26 | // 2. Копируем элементы из существующего массива в наш, только что выделенный, массив. |
| 27 | // 3. Уничтожаем старый массив и присваиваем m\_data адрес нового массива. |
| 28 | // Выделяем новый массив |
| 29 | int \*data = new int[newLength]; |
| 30 | // Затем нам нужно разобраться с количеством копируемых элементов в новый массив |
| 31 | // Нам нужно скопировать столько элементов, сколько их есть в меньшем из массивов |
| 32 | if (m\_length > 0) |
| 33 | { |
| 34 | int elementsToCopy = (newLength > m\_length) ? m\_length : newLength; |
| 35 | // Поочерёдно копируем элементы |
| 36 | for (int index=0; index < elementsToCopy; ++index) |
| 37 | data[index] = m\_data[index]; |
| 38 | } |
| 39 | // Удаляем старый массив, так как он нам уже не нужен |
| 40 | delete[] m\_data; |
| 41 | // И используем вместо старого массива новый! Обратите внимание, m\_data указывает на тот же адрес, на который указывает наш новый динамически выделенный массив. |
| 42 | // Поскольку данные были динамически выделенные, то они не будут уничтожены, когда выйдут из области видимости |
| 43 | m\_data = data; |
| 44 | m\_length = newLength; |
| 45 | } |

Реализуем возможность добавления/удаления элементов из контейнера.

**Листинг 11.8**

| 1 | void insertBefore(int value, int index) |
| --- | --- |
| 2 | { |
| 3 | // Создаём новый массив на один элемент больше старого массива |
| 4 | int \*data = new int[m\_length+1]; |
| 5 | // Копируем все элементы до index-а добавляемого элемента |
| 6 | for (int before=0; before < index; ++before) |
| 7 | data[before] = m\_data[before]; |
| 8 | // Вставляем наш новый элемент в наш новый массив |
| 9 | data [index] = value; |
| 10 | // Копируем все значения после вставляемого элемента |
| 11 | for (int after=index; after < m\_length; ++after) |
| 12 | data[after+1] = m\_data[after]; |
| 13 | // Удаляем старый массив и используем вместо него новый массив |
| 14 | delete[] m\_data; |
| 15 | m\_data = data; |
| 16 | ++m\_length; |
| 17 | } |
| 18 | void remove(int index) |
| 19 | { |
| 20 | // Если это последний элемент массива, то делаем массив пустым и выполняем return |
| 21 | if (m\_length == 1) |
| 22 | { |
| 23 | erase(); |
| 24 | return; |
| 25 | } |
| 26 | // Cоздаём новый массив на один элемент меньше нашего старого массива |
| 27 | int \*data = new int[m\_length-1]; |
| 28 | // Копируем все элементы до index-а удаляемого элемента |
| 29 | for (int before=0; before < index; ++before) |
| 30 | data[before] = m\_data[before]; |
| 31 | // Копируем все значения после удаляемого элемента |
| 32 | for (int after=index+1; after < m\_length; ++after ) |
| 33 | data[after-1] = m\_data[after]; |
| 34 | // Удаляем старый массив и используем вместо него новый массив |
| 35 | delete[] m\_data; |
| 36 | m\_data = data; |
| 37 | --m\_length; |
| 38 | } |
| 39 | // Несколько дополнительных функций просто для удобства |
| 40 | void insertAtBeginning(int value) { insertBefore(value, 0); } |
| 41 | void insertAtEnd(int value) { insertBefore(value, m\_length); } |

Хотя написание контейнерных классов может быть несколько сложным, но хорошая новость заключается в том, что вам их нужно написать только один раз. Как только контейнерный класс работает, вы можете его повторно использовать где-угодно без каких-либо дополнительных действий/усилий по части программирования.

Также стоит отметить, что, хотя наш контейнерный класс ArrayInt содержит фундаментальный тип данных (int), мы также могли бы легко использовать и пользовательский тип данных.