Лекция 5

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ В С++. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА С++

Лекция 5. Обобщенное программирование в С++. Стандартная библиотека языка С++

План

1. Понятие обобщенного программирования.
2. Шаблоны функций и классов в С++.
3. Контейнерные классы.
4. Строки и строковые потоки.
5. Алгоритмы.

1. Понятие обобщенного программирования

Рассмотрим следующий пример. Пусть необходимо разработать класс для работы с целочисленными одномерными массивами. Простейший вариант его реализации может выглядеть так:

```
class Array
{
private:
	int *data = nullptr; // Указатель на блок памяти, где буду хранится данные
	int size = 0; // Размер массива
public:
	Array(int s) // Базовый конструктор
	{
		size = s;
		data = new int[size];
	}
	Array(const Array& rhs) // Копирующий конструктор
	{
		data = new int [size = rhs.size];
		memcpy(data, rhs.data, size * sizeof(int)); // Копирование данных
	}
```

1. Понятие обобщенного программирования

```
    ~Array(void) // Декструктор
    {
        if (size) delete [] data;
    }
    void ReSize(int s) // Динамическое изменение размера массива
    {
        if (size) delete [] data;
        data = new int[size = s];
    }
    int &operator [] (int i) // Доступ к элементам массива по индексу
    {
        return data[i];
    }
    int Size(void) const // Возврат размера массива
    {
        return size;
    }
    // ...
};
```

1. Понятие обобщенного программирования

Использовать такой класс-массив можно, например, следующим образом:

```
// …

Array my_arr;
int size;

cout << "Input array size: ";
cin >> size;
my_arr.ReSize(size);
for (int i = 0; i < my_arr.Size(); i++)
    my_arr[i] = i;

// …
```

Ясно, что при необходимости класс Array можно расширить за счет новой функциональности, например, добавив в него операции сложения, вычитания, поиска минимума или максимума и т. п.

1. Понятие обобщенного программирования

Теперь представим себе, что возникала задача создать аналогичный класс для работы с одномерными массивами вещественных чисел. Ясно, что решить ее можно очень просто. Для этого достаточно создать копию предыдущего класса, обозвав ее, например, FloatArray, и поменяв объявление

```
int *data = nullptr;
```

на

```
float *data = nullptr;
```

Кроме того, нужно внести соответствующие изменения в копирующий конструктор и оператор индексирования.

Аналогичным образом можно создать классы DoubleArray, CharArray и т. д. Однако, очевидно, что такой подход является избыточным и далеко не самым оптимальным.

1. Понятие обобщенного программирования

Таким образом, возникает идея создания таких функций и классов, тип используемых данных в которых был бы параметром (шаблоном). На этой идее базируется парадигма **обобщенного программирования** (generic programming), заключающаяся в некотором описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

Суть данной парадигмы заключается в таком разделении структур данных и алгоритмов через использование абстрактных типов данных, которое бы позволило описывать алгоритм решения задачи в форме, не зависящей от конкретного типа данных. Например, функцию определения максимального значения среди своих двух аргументов с помощью псевдокода можно описать так:

```
T Max(T x, T y):
begin
    if x > y
        return x
    else
        return y
end
```

2. Шаблоны функций и классов в C++

Язык программирования C++ поддерживает парадигму обобщенного программирования.

Например, описание шаблонной функции, возвращающей максимум двух своих аргументов, в C++ можно реализовать так:

```
template <typename T> T max(T x, T y)        // T – задает обобщенный тип
{
    return (x > y) ? x : y; // Для T должна быть определена операция «>»
}
```

Вызывать эту функцию в программе можно так:

```
max<int>(10, 50);    // Максимум среди целых аргументов – 50
max<double>(10.5, -3.8);    // ... вещественных аргументов – 10.5
max<string>("abc", "def");    // ... строк – результат "def"
```

Примечание. В качестве T в приведенной функции можно использовать любой тип данных или класс, для которого определена (перегружена) операция «>».

2. Шаблоны функций и классов в C++

Аналогичным образом в C++ создаются и шаблонные (template) классы. В общем виде они описываются следующим образом:

```
template <typename T> class <Имя_класса> {};
```

Например, шаблонный класс, описывающий одномерный массив, можно определить так:

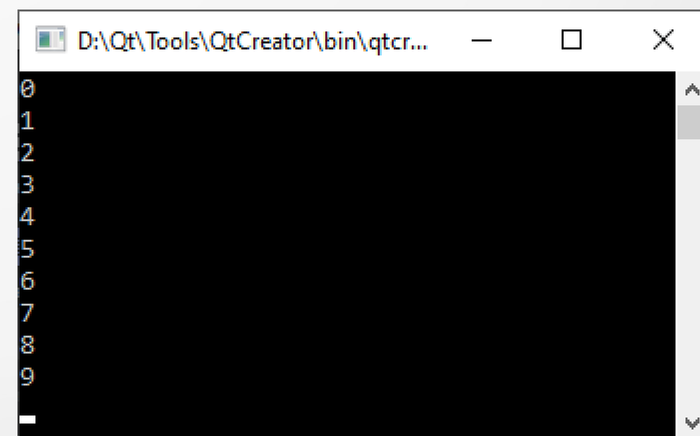
```
template <typename T> class Array
{
private:
    T *data = nullptr; // Указатель на данные
    int size = 0; // Размер массива
public:
  Array(int s) // Базовый конструктор
  {
    data = new T[size = s];
  }
  // ...
  T &operator [] (int); // Доступ к элементам массива по индексу
  // ...
};
```

2. Шаблоны функций и классов в C++

Использовать такой класс можно, например, следующим образом:

```
// …

int main()
{
    Array<double> arr(10);

    for (int i = 0; i < arr.Size(); i++)
    {
        arr[i] = i;
        cout << arr[i] << endl;
    }
    return 0;
}
```

Результат работы программы



3. Контейнерные классы

Под **контейнером** (**контейнерным классом**) понимается класс, предназначенный для хранения множества (коллекции) объектов определенного типа.

В **STL** (Standard Template Library) – стандартной библиотеке шаблонов С++ содержится большое количество контейнеров, реализующих различные структуры данных.

Рассмотрим самые популярные среди них;

- **std::vector** – динамический массив произвольного доступа с автоматическим изменением размера при добавлении или удалении элемента:
- **std::list** – двунаправленный связанный список;
- **std::stack** – контейнер, реализующий функции стека (LIFO);
- **std::queue** – контейнер, реализующий функции очереди (FIFO);
- **std::map** – коллекция, сохраняющая отсортированные однозначные пары вида <ключ, значение>;
- **std::set** – упорядоченное множество уникальных (неповторяющихся) значений;

3. Контейнерные классы

Примеры применения вектора std::vector

```
#include <vector> // Подключение соответствующего заголовка

using namespace std;

vector<double> vec{1.5, 2.3, 4}; // Объявление с инициализацией
vec.resize(10); // Изменение размера
vec.push_back(3.14); // Добавление нового элемента в конец вектора
vec.at(5) = -100; // Доступ по индексу
vec[6] = -1.4; // …
for (auto it : vec) // Итерация по всем элементам vec (it – текущий элемент)
	cout << it << endl;
for (int i = 0; i < vec.size(); i++) // Еще один способ итерации...
	cin >> vec[i];
```

3. Контейнерные классы

Примеры применения списка std::list

```
#include <list>

std::list<int> lst{1, 3, 2}; // Объявление с инициализацией
lst.size(); // Количество элементов списка
lst.push_back(5); // Добавление нового элемента в конец списка
lst.pop_back(); // Удаление последнего элемента из списка
for (auto it: lst) // Итерация по всем элементам
	std::cout << it << std::endl;
for (auto it = lst.begin(); it != lst.end(); it++) // Еще один способ итерации
	std::cout << *it << std::endl;
lst.sort(); // Сортировка
```

3. Контейнерные классы

Примеры применения стека std::stack

```
#include <stack>


using namespace std;

stack<int> stk; // Объявление (инициализация не поддерживается)
stk.size(); // Количество элементов в стеке
stk.push(-10); // Поместить элемент в стек
stk.pop(); // «Вытолкнуть» элемент из стека
while (!stk.empty()) // Пример работы со всеми всеми элементами стека
{
	cout << stk.top() << endl;
	stk.pop();
}
```

3. Контейнерные классы

Примеры применения очереди std::queue

```
#include <queue>


using namespace std;

queue<float> qe; // Объявление очереди
qe.size(); // Количество элементов
qe.push(-1.0f); // Добавить новый элемент в конец очереди
qe.pop(); // Удалить первый элемент из очереди
qe.front(); // Первый элемент
qe.back(); // Последний элемент
while (!qe.empty()) // Обработка всех элементов очереди
{
	cout << qe.front() << endl;
	qe.pop();
}
```

3. Контейнерные классы

Примеры применения контейнера std::map

```
#include <map>


using namespace std;

map<string, int> id{{"Li", 1}, {"Trump", 2}}; // Объявление и инициализация
id.insert({"Buch", 3}); // Добавление нового значения
id["Buch"] = 4; // Изменение значения по ключу
id["Smith"] = 5; // Добавление (изменение) значения по ключу
for (auto it : id) // Итерация по контейнеру
    cout << it.first << ' ' << it.second << endl;
```

3. Контейнерные классы

Примеры применения контейнера std::set

```
#include <set>


using namespace std;

set<char> s{'a', 'b'}; // Объявление и инициализация
s.size(); // Количество элементов в контейнере
s.insert('H'); // Добавление элемента в множество
s.erase('a'); // Удаление элемента из множества
s.clear(); // Очистка контейнера
for (auto it : s) // Итерация по множеству
      cout << it << endl;
```

4. Строки и строковые потоки

Для работы со строками в стандартной библиотеке C++ реализованы классы **std::string** (стандартная строка однобайтовых символов в формате **ACSII**) и **std::wstring** (строка двухбайтовых символов в формате **Unicode**).

Примеры применения строки std::string

```
#include <string>

using namespace std;

string str = "Hello"; // Объявление с инициализацией
cout << str.length() << endl; // Длина строки
str.push_back('!'); // Добавление в конец символа
str += "?"; // Еще один способ добавления символа или строки
cout << str << endl; // Вывод строки в поток
str.replace(5, 1, "?"); // Замена символа в строке, начиная с 6-ой позиции…
cout << str[0] << endl; // Получение символа по индексу
cout << str.c_str() << endl; // Преобразование в формат строки языка C
```

4. Строки и строковые потоки

Примеры применения Unicode-строки std::wstring

```
#include <xstring>
#include <locale> // wcout – вариант потока для работы с Unicode-строками


using namespace std;

wstring ws = L"Hello"; // Объявление с инициализацией
cout << ws.length() << endl; // Длина строки
wcout << ws << endl; // Вывод в Unicode-поток
ws += L"!"; // Конкатенация строк
ws.replace(5, 1, L"?"); // Замена подстроки…
wcout << ws[0] << endl; // Получение символа по индексу
ws.c_str(); // Возврат указателя на строку в формате языка C
ws.substr(1, 3); // Возврат подстроки
ws.find(L"ll"); // Поиск подстроки
```

4. Строки и строковые потоки

В стандартной библиотеке C++ реализованы специальные строковые потоки, объединяющие возможности стандартных потоков и класса std::string. Чаще всего на практике используют класс **std::stringstream**, реализующий операции ввода-вывода в памяти (в строке). Он наследует от std::istream и std::ostream все их операции по работе с потоками и, в частности, операции чтения-записи («>>» и «<<»).

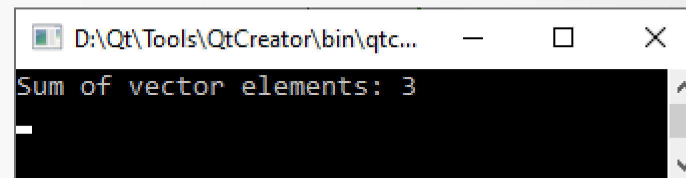
Рассмотрим следующий пример:

```
#include <sstream>

// …
int a, b;
string str;
stringstream ss; // Объявление строкового потока

// Запись в него данных
ss << "Hello!" << ' ' << 10 << ' ' << 20;
// Вывод содержимого строкового потока на экран
cout << ss.str() << endl; // Метод str() возвращает данные в виде строки
// Считывание данных из строкового потока
ss >> str >> a >> b;
```

5. Алгоритмы

Для работы с контейнерными классами в стандартной библиотеке C++ реализовано множество различных алгоритмов, описанных в заголовочном файле algorithm. Рассмотрим некоторые из них.

for_each() – по порядку применяет заданную функцию к каждому элементу контейнера в указанном диапазоне.




```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> vec{3, 4, 2, 9, -15};
    auto sum = 0;

    for_each(vec.begin(), vec.end(), [&](int &it){ sum += it; });
    std::cout << "Sum of vector elements: " << sum << ' ' << std::endl;
    return 0;
}
```

5. Алгоритмы

В приведенном выше примере для каждого элемента вектора, находящегося в диапазоне от std::vector.begin() до std::vector.end(), выполняется выражение, заданное лямбда-функцией:

```
[&](int &it){ sum += it; }
```

Лямбда-функция (**лямбда-выражение**) это способ описания анонимных (безымянных) функциональных выражений, которые могут быть встроены в различные конструкции языка C++.

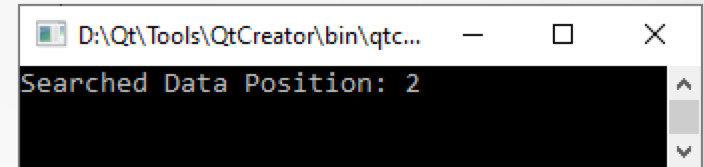
Например, вывод всех элементов вектора можно реализовать следующим образом:

```
for_each(vec.begin(), vec.end(), [](int it){ std::cout << it << ' '; });
```

Отметим, что если лямбда-функция описана как []…, то в ее теле недоступны никакие внешние переменные, если – [&]…, то ей доступны по ссылке все переменные из внешнего контекста, если – [=]…, то ей доступны внешние переменные по значению.

5. Алгоритмы

search() – ищет первое вхождение заданной последовательности элементов в указанный диапазон значений.
Например:

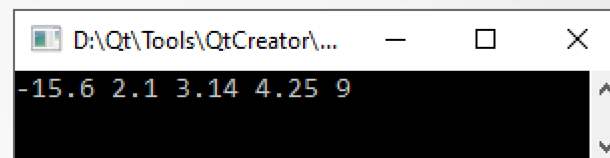



```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> data{3, 4, 2, 9, -15},
                     find{2, 9};
    auto ret = search(data.begin(), data.end(), find.begin(), find.end());

    if (ret != data.end())
        std::cout << "Searched Data Position: " << (ret - data.begin()) << std::endl;
    else
        std::cout <<  "Data not found" << std::endl;
    return 0;
}
```

5. Алгоритмы

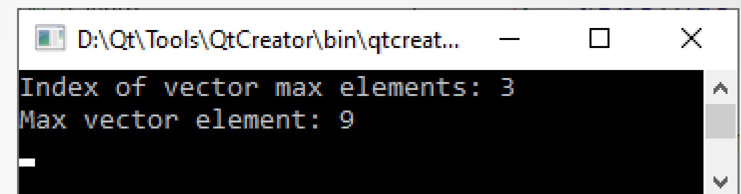
sort() – выполняет сортировку элементов в заданному диапазоне в порядке возрастания.
Например:

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<double> vec{3.14, 4.25, 2.1, 9, -15.6};

    sort(vec.begin(), vec.end());
    for_each(vec.begin(), vec.end(), [](float it) { std::cout << it << ' '; });
    std::cout << std::endl;
    return 0;
}
```



5. Алгоритмы

max_element() – находит наибольший элемент в заданном диапазоне. Например:




```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    vector<int> vec{3, 4, 2, 9, -15};
    auto index = max_element(vec.begin(), vec.end());

    cout << "Index of vector max elements: " << index - vec.begin() << ' ' << endl;
    cout << "Max vector element: " << vec.at(index - vec.begin()) << ' ' << endl;
    return 0;
}
```

5. Алгоритмы

Кроме того, в библиотеке алгоритмов есть еще такие часто используемые функции:

- **min_element()** – поиск наименьшего элемента в заданном диапазоне;
- **minmax_element()** – поиск наименьшего и наибольшего элементов в заданном диапазоне;
- **max()** – возвращает наибольший из двух аргументов;
- **min()** – возвращает наименьший из двух элементов;
- **minmax()** – возвращает большее и меньшее из двух элементов;
- **find()** – находит первый элемент, удовлетворяющий определенным критериям;
- **count()** – возвращает количество элементов, удовлетворяющих определенным критериям;
- etc