# Лекция 2

# Контейнеры и шаблоны

Шаблоны C++ предоставляют возможность повторно использовать исходные тексты (в отличие от функций, которые позволяют многократного использовать объектный код). Шаблоны принадлежат к числу инструментов обобщенного программирования.   
Стандартная библиотека *С++* включает в себя библиотеку шаблонов контейнерных классов, которую часто называют STL. Это не совсем точно, но мы будем для сокращения записи использовать это название.  
 Контейнер – это объект, который предназначен для хранения других объектов.

## 1.1 Контейнеры

Потребность в контейнерах по-настоящему возникает при создании объектов в куче оператором *new* и их уничтожении оператором *delete*. В общем случае во время написания программы неизвестно, сколько объектов вам потребуется. Например, информационная система программируется так, чтобы поддерживать множество объектов, но только пользователь (во время выполнения программы) определяет, какие и сколько объектов ему понадобится.

Альтернативное решение, которое использовалось в языке С, – это работа с огромным глобальным массивом объектов.   
В C++ выбран другой подход: когда вам потребуется объект, вы создаете его при помощи оператора new и сохраняете указатель в контейнере. Затем указатель извлекается из контейнера, и программа делает с ним что-нибудь полезное. В этом случае создаются только те объекты, которые действительно используются в программе. Этот подход позволяет создавать наиболее гибкие и универсальные программы.

Итак, в типичной ситуации контейнер содержит указатели на объекты.   
Программа создает экземпляры объектов при помощи оператора new, сохраняет полученные указатели в контейнере и извлекает их позднее, когда с объектом нужно выполнить операцию. При использовании контейнеров из библиотеки нужно иметь ввиду, что выделение динамической памяти (вызов оператора new) может выполняться как методом контейнера, так и программой, использующей контейнер. Перед использованием контейнера важно установить, кто должен отвечать за освобождение памяти: контейнер или программа. Это необходимо сделать для предотвращения «замусоривания» памяти.

## 1.2 Общие сведения о шаблонах

А теперь о проблеме, которая возникает при создании библиотеки контейнеров. Предположим, что мы создали контейнер-массив IntArray для хранения целых чисел; но нам может потребоваться массив для хранения геометрических фигур, самолетов, растений или еще чего-нибудь. Каждый раз переделывать исходные тексты? Вряд ли это можно назвать разумным решением, тем более в языке, ориентированном на многократное использование кода. В данной ситуации C++ предлагает использовать шаблоны.  
Шаблоны – это средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию). Шаблоны C++ позволяют *определить семейство функций или классов*, которые могут работать с *различными типами данных*.

## 1.3 Синтаксис шаблонов

Ключевое слово template сообщает компилятору, что в следующем за ним определении класса используется один или несколько заранее неизвестных типов. В тот момент, когда компилятор генерирует код класса на основании шаблона, эти типы должны быть известны, чтобы компилятор мог подставить их на место параметров шаблона.   
Для демонстрации синтаксиса рассмотрим небольшой пример, в котором создается массив с автоматической проверкой границ:

//L2:TemplateArray.срр

#include <iostream>

#include <iomanip>

using namespace std;

template<class T, int size=20>

class Array {

T Arr[size]; //значение size не хранится внутри создаваемого с помощью шаблона //класса, но все функции класса могут использовать его

// как целочисленную константу.

public:

T& operator[](int index);//перегрузка оператора индексации

};

//Определение функции

template<class T, int size>

T& Array<T,size>::operator[](int index) {

if(index >= 0 && index < size)

return Arr[index];

cout<<"Index out of range"<<endl;

system("pause");

exit(-1);

}

int main() {

Array<int> iArr;

Array<float,15> fArr;

for(int i = 0; i < 10; i++) {

iArr[i] = i \* i ;

fArr[i] = float(i) \* 1.5;

}

for(int j = 0; j < 10; j++)

cout <<setw(3)<< j << ": "<<setw(3) << iArr[j]

<< "; " <<setw(5)<< fArr[j] << endl;

system("pause");

}

Аргументы шаблонов не ограничиваются классами; в них также можно передавать встроенные в С++ типы. Значения этих аргументов превращаются в константы времени компиляции для указанного экземпляра шаблона. Таким аргументам даже можно присваивать значения по умолчанию (int size=20).

Класс Array выглядит вполне обычно, если не считать строку   
 template<class Т>

Эта строка сообщает, что Т является подставляемым параметром, который отражает имя типа. Внутри определения класса идентификатор Т используется везде, где обычно указывается конкретный тип, хранимый в контейнере.   
 В Array присваивание и выборка элементов осуществляются одной функцией — перегруженной операторной функцией operator[ ]. Оператор возвращает ссылку (адрес переменной), поэтому результат может находиться по обе стороны от оператора присваивания (то есть быть как *l-value* и как *r- value* значением). Если индекс выходит за границы массива, класс выводит сообщение об ошибке.   
 Функция main() показывает, как легко создаются контейнеры Array для хранения объектов различных типов. Встречая следующие определения, компилятор дважды расширяет шаблон (также говорят: создает экземпляр шаблона) Array и создает два новых сгенерированных класса, которые можно условно обозначить Array\_int и Array\_float (разные компиляторы используют разные схемы формирования имен, но имена должны быть различными). Эти имена скрыты от программиста, и он не использует их в программе.

Array<int> iArr;   
Array<float> fArr;

Эти классы работают точно так же, как при ручной подстановке типов, если не считать того, что они автоматически создаются компилятором при определении объектов iArr и fArr. Повторяющиеся определения классов либо обходятся компилятором, либо объединяются компоновщиком.

Следует иметь в виду, что все функции-члены шаблона класса также должны быть шаблонами функций. Это требование накладывает на функции определенные ограничения. Например, используемые в функции операторы должны быть определены для всех типов данных, которые являются параметрами шаблона.

## 1.4 Заголовочные файлы

Все объявления и определения, относящиеся к шаблону, лучше разместить в заголовочном файле. На первый взгляд кажется, что это противоречит стандартному правилу «Не размещай в заголовочных файлах ничего, что связано с выделением памяти» (из-за ошибок повторного определения на стадии компиляции), однако определения шаблонов занимают особое место. Любая конструкция с префиксом *template<...>* означает, что компилятор не выделяет память немедленно, а ждет специального распоряжения (то есть создания экземпляра шаблона), при этом в компиляторе и компоновщике реализован механизм исключения множественных определений идентичного шаблона. По этой причине в заголовочный файл почти всегда включаются полные объявление и определение шаблона (это делается для простоты использования).   
Некоторые программисты считают, что вынесение всего исходного текста реализации в заголовочный файл нежелательно, поскольку дает возможность изменять (или просто красть) их программный код после покупки библиотеки. Такие опасения имеют основания, но здесь стоит спросить себя: что именно вы продаете, продукт или услугу? Если продукт, то вам следует приложить все усилия к его защите; вероятно, не стоит распространять исходные тексты, а лишь откомпилированный код. Но многие клиенты рассматривают программы как услугу, даже более того, как подписку на услугу. Он предпочитает, чтобы вы сами сопровождали свою программу, а он будет использовать ее в своей работе.

## 1.5 Знакомство с итераторами

Итератором называется объект, который позволяет перебирать содержимое контейнера и в любой момент времени указывает на один из его элементов. Итераторы обеспечивают стандартный механизм обращения к элементам независимо от того, поддерживает ли контейнер прямой доступ к элементам. Итераторы чаще всего используются с контейнерными классами, они занимают важное место в архитектуре стандартных контейнеров C++ и их практическом применении.   
 Во многих отношениях итератор может рассматриваться как «умный» указатель. И действительно, итераторы поддерживают многие операции, характерные для указателей. Но в отличие от указателей, при проектировании итераторов особое значение придавалось вопросам безопасности, поэтому неприятности типа выхода за пределы массива с ними случаются гораздо реже (а если и случаются, то выявляются гораздо быстрее).

Хотя на первый взгляд перебор сводится к простому «увеличению указателя», на самом деле выполняются более сложные действия. В этом заключается ключевая особенность итераторов: они абстрагируют сложный процесс перемещения между элементами контейнера в нечто, напоминающее использование индекса для доступа к элементам массива.

Стандартная библиотека предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. Контейнеры в стандартной библиотеке C++ реализованы как шаблоны классов. При объявлении переменной типа контейнера указывается тип элементов, которые будет содержать контейнер. Контейнеры могут создаваться с использованием списков для инициализации значений элементов (как при объявлении массивов). В отличии от массивов они содержат функции-члены для добавления и удаления элементов и выполнения других операций.

Перебор элементов в контейнере и доступ к отдельным элементам выполняется с помощью [итератор](https://msdn.microsoft.com/ru-ru/library/28f7db1d.aspx)ов.   
 *Итераторы для всех контейнеров STL имеют общий интерфейс*, но каждый контейнер определяет собственные специализированные итераторы. Рассмотрим последовательный контейнер ***vector*** и ассоциативный контейнер ***map.***

## 1.6 Последовательные контейнеры

Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов.

Контейнер ***vector*** ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер ***vector*** является наиболее предпочтительным последовательным контейнером для большинства областей применения. Если вы сомневаетесь в выборе вида последовательного контейнера, начните с использования вектора. 

## 1.7 Ассоциативные контейнеры

В ассоциативных контейнерах элементы вставляются в предварительно определенном порядке — например, с сортировкой по возрастанию.   
 Контейнер ***map***, который часто называют словарем, хранит в узлах пару "ключ-значение". Ключ является уникальным и используется для упорядочивания последовательности, а значение связано с ключом. Например, ***map*** может хранить пары, ключом в которых являются заголовки столбцов в таблице (тип ключа – string), а значением является структура (тип struct), которая описывает тип данных в столбце и их максимальный размер в байтах.  
Для ***map*** перегружена операция индексации. Она использует ключ в качестве индекса, а возвращает значение (адрес объекта). Поэтому её можно использовать как в правой, так и в левой части оператора присваивания, как в обыкновенном массиве.

Для хранения данных в контейнерах *map* обычно используется бинарное дерево (объект класса *tree*).

Внутренняя реализация контейнера ***map*** может быть и другой (она скрыта от пользователя). При работе с контейнерами пользователю доступен только его интерфейс, который реализует операции с графом типа бинарное дерево.

Бинарное дерево — это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве размещаются узлы с ключами, имеющими значения, меньшие, чем значение ключа данного узла, а в правом поддереве размещаются узлы с ключами, имеющими значения, большие, чем значение ключа данного узла.

При создании дерева создается один пустой узел, который будет корнем дерева.   
Элементы добавляются в дерево согласно приведенному выше правилу.   
Запишем в бинарное дерево элементы массива   
int arr[]={7, 3, 11, 1 ,0, 2, 5, 9, 6, 4, 13, 14, 8, 10, 12};

В этом случае ключ элемента одновременно является и его значением.

Мы получим приведенное на рисунке дерево.

7  
3 11

1 5 9 13

0 2 4 6 8 10 12 14

При записи данных в дерево выполняется их сортировка.  
Обход дерева выполняется с помощью итератора.

# 2. Использование контейнеров map и vector для хранения таблиц в макете СУБД

//файл DBTableTxt.h

//============ "Описание класса DBTableTxt - таблицы макета СУБД"==============

//--------DBType-перечисление типов полей таблиц БД------

enum DBType {

NoType, // 0

Int32, // 1

Double, // 2

String, // 3

Date // 4

};

enum Condition{Undefined,Equal,NotEqual,Less,Greater,LessOrEqual,GreaterOrEqual};

const int LENGTH = 24;//длина имени таблицы и имени столбца.

//ColumnDesc - описание данных в столбце таблицы

struct ColumnDesc {

char colName[LENGTH];//имя столбца

DBType colType;//тип данных в столбце таблицы

int length; //максимальное число символов, допустимое

//для представления данных в столбце

};

struct Strip{//полоса распечатки таблицы (используется в PrintTable())

int nField;//число полей в полосе

int\* fieldWidth;//ширина полей в полосе (массив)

};

typedef map<string, void\*> Row;//тип записей в строках таблицы

typedef map<string, ColumnDesc> Header;//тип заголовка

void\* GetValue(string value, string columnName,Header hdr); //возвращает значение //переменной value, преобразованное из типа

//string в тип colType, имя которого указано в заголовке

//столбца columnName таблицы.

void\* SetValue(string value, string columnName,Header hdr);

bool comparator(DBType type,void \*obj1,Condition condition,void \*obj);

int GetLength(ColumnDesc colDesc);

//------------------- класс DBTableTxt ----------------------

class DBTableTxt{

Header columnHeaders;

string tableName;

string primaryKey;

vector<Row> data;

string fileName;

public:

DBTableTxt(){}

DBTableTxt(string tabName);

DBTableTxt(string tabName,Header hdr,string primKey);

const char\* TypeName(DBType type);

~DBTableTxt(){}

vector<int> IndexOfRecord(void\* keyValue,string keyColumnName);

string valueToString(Row& row,string columnName);

void ReadDBTable(string fileName);//fileName=path+tableName+".txt"

void PrintTable(int screenWidth);

void WriteDBTable(string fileName);//fileName=path+tableName +".txt"

int GetSize();

DBType GetType(char\* columnName); );//возвращает код типа данных в // столбце columnName

Row operator[](int ind);

void SetFileName(string path);

void SetTableName(string tabName);

void SetPrimaryKey(string key);

string GetFileName();

string GetTableName();

string GetPrimaryKey(){return primaryKey;}

Header GetHeader();

void SetHeader(Header& hdr);

Row CreateRow();//создать строку таблицы

Row GetRow(int index);

void AddRow(Row row,int index);//добавить строку

//Макет распечатки таблицы:

//screenWidth - ширина экрана (входной параметр)

//nStrip - число полос в распечатке (выходной параметр)

//strips[nStrip] – массив из значений ширины  
 // столбцов в полосе (выходные параметры)

void CreateTableMaket(Strip\* &strips, int &nStrip, int screenWidth);

friend void ReadDBTable1(DBTableTxt& tab,string fileName);   
 //дружественная функция, используемая для разработки своей функции //чтения таблицы

};

## 2.1 UML-диаграмма класса DBTableTxt



## 2.2 Чтение заголовка в columnHeaders.

Рассмотрим фрагмент кода, который поможет вам считать заголовок таблицы из файла БД в объект DBTableTxt при разработке метода ReadDBTable1().   
*Предупреждение.* Прежде чем вставлять приведенный ниже код в программу, нужно разобраться в нем и внести необходимые дополнения.

vector<ColumnDesc> strArray;//буфер для чтения строки заголовка таблицы

char \*token, \*next\_token;  
fin.getline(line, 200); //чтение заголовка в буфер line  
 next\_token=line;  
 //цикл по словам (лексемам) в строке  
 ColumnDesc colHdr;

while( (token = strtok\_s( next\_token, delims, &next\_token)) != NULL )

{

strcpy\_s(colHdr.colName,token);

token = strtok\_s( next\_token, delims, &next\_token);

colHdr.colType=GetType(token);

token = strtok\_s( next\_token, delims, &next\_token);

colHdr.length=atoi(token);

strArray.push\_back(colHdr); //глубокое копирование

}

Header hdr;

for (unsigned int j = 0; j < strArray.size() ; j ++)

{

hdr[strArray[j].colName]=strArray[j];// !!! использование операции индексации   
// для записи . Если столбца colName в заголовке нет, то он создается и затем ему //присваивается значение

}

SetHeader(hdr);//

## 2.3 Чтение строк данных.

//читаем строки в line (до EOF) и записываем их в table.data

data.clear();//вызов метода контейнера vector (для обеспечения записи в начало //необходимо при повторном чтении)

while (fin.getline(line, 200)) {

Row row=\*(new Row());//формирование строки таблицы:  
 //память выделяется только под корневой узел map

int j = 0;

token = strtok\_s( line, delims, &next\_token);

//цикл по столбцам (словам) в строке

while(token)

{

string value=token;

//добавление поля в строку с преобразованием типа

//strArray[j] - заголовок столбца в заголовке (шапке) таблицы

row[strArray[j].colName]=GetValue(value,strArray[j].colName,columnHeaders);

j=j++;//индекс следующего столбца в векторе strArray

token = strtok\_s( next\_token, delims, &next\_token);

}

data.push\_back(row);//добавить строку данных в таблицу

## 2.4 Пример использования итераторов при записи таблицы в файл

ofstream fout;//открытие файлового потока на вывод

Header::iterator iter,iterWhile;

//Запись имени таблицы и имени столбца primaryKey (первая строка файла).

fout<<tableName<<'|'<<primaryKey<<endl;

//Запись заголовка таблицы (вторая строка файла).

//В соответствии с форматом файла, после последнего члена

//строки вместо '|' нужно вставить перевод строки.

iter=columnHeaders.begin();

iterWhile=iter;//чтобы не пытаться печатать в конце цикла строку iter++, которой нет.

// Из-за отсутствия операций iter+1 и iter<, используем сравнение на равенство с //++iterWhile

while(++iterWhile!=columnHeaders.end())

{

fout<< iter->second.colName<<"|"<< TypeName(iter->second.colType)<<

"|"<<iter->second.length<<"|";

++iter;//переход к заголовку следующего столбца таблицы

}

В программировании часто возникает необходимость обрабатывать два объекта как один. Такая задача стала особенно актуальной, когда в синтаксис программирования были внедрены [*ассоциативные контейнеры*](http://proginfo.ru/map/) с их парой: название поля –> значение поля. Причём типы обоих значений в паре могут быть совершенно разными – в этом и состоит особая ценность рассматриваемой [структуры](http://proginfo.ru/struct/) *pair* и на это необходимо обратить внимание.

В узле заголовка (тип Header) хранится пара *pair<string,ColumnDesc>*, которая представляет реализацию шаблона структуры *pair* из стандартной библиотеки С++. Структура *pair* содержит два поля с именами *first* и *second*.   
*string* - это тип поля *first*, а тип поля *second – ColumnDesc* . Поле *first* используется в качестве ключа, а поле *second* – в качестве значения.   
В типе *Header* класса *DBTableTxt* ключом является поле *colName* структуры типа *ColumnDesc* (тип char\* имени столбца *colName* по умолчанию преобразуется в тип string), а значением – переменная типа ColumnDesc.   
Итератор указывает на переменную типа *pair<string,ColumnDesc>*.