**Шаблоны**

Хотя функции и классы являются довольно гибкими инструментами для эффективного программирования, в некоторых случаях они ограничены из-за требования C++ указывать типы всех используемых параметров.

Например, рассмотрим функцию, которая будет суммировать 2 элемента:

int sum(int a, int b) {

return a + b;

}

При вводе 2х целых чисел, она прекрасно будет работать, но что, если ввести значения другого типа? К примеру, числа типа double. В таком случае, она уже не будет работать корректно. **(1.cpp)**

#include <iostream>

using namespace std;

int sum(int a, int b) {

return a + b;

}

int main() {

int a = 5,b=3;

std::cout << sum(a, b)<<' ';

double x = 5.5, y = 3.5;

std::cout << sum(x,y)<<' ';

} *//оба раза программа выведет 8, хотя во втором случае ответ 9*

Существует несколько способов решения этой проблемы:

1. **Copy Paste**

Просто скопировать функцию и перегрузить её для работы с типом double:

double sum(double a, double b) {

return a + b;

}

Теперь всё работает как надо, однако существует и множество других типов.

Таким образом, поскольку C++ требует, чтобы мы указывали типы наших переменных, нам приходится записывать несколько версий одной и той же функции, где единственное, что меняется — это тип параметров.

Поддерживать такой код непросто как по затраченным усилиям, так и по времени. И самое важное то, что это нарушает одну из концепций эффективного программирования — сократить до минимума дублирование кода.

1. **C style**

Рассмотрим способ борьбы с нашей проблемой на примере класса Array.

Аналогично функции sum, класс массивов устроен одинаково как для int, так и для double(отличается лишь тип элементов массива). Попробуем реализовать универсальный класс способом, которым пользовались в языке C:

В файл Array.h записываем:

class Array {

Type\* data;

Type get(int i);

}

В файл main.cpp:

#define Type int *//Определяем Type как int*

#include "Array.h"

#undef Type

При компиляции создаётся class Array#Type{…} (То есть в нашем случае Arrayint. Если бы определили Type как double, было бы Arraydouble)

Несколько другой вариант - просто вовнутрь программы вставить следующий текст:

#{define DefineArray(name,Type)\

class name{\

Type data;\

public:\

Type get (int i){\

return data[i];\

}\

};

Но здесь сам код «появляется» лишь при компиляции, следовательно, могут возникать очень странные ошибки препроцессора, поэтому отлаживать такой код практически невозможно.

1. **C++ style**

Итак, мы добрались до наиболее оптимального варианта решения проблемы – шаблонов.

Шаблоны можно использовать как для функций, так и для классов.

В языке C++ **шаблоны функций** — это функции, которые служат образцом для создания других подобных функций. Главная идея — создание функций без указания точного типа(ов) некоторых или всех переменных. Для этого мы определяем функцию, указывая тип параметра шаблона, который используется вместо любого типа данных. После того, как мы создали функцию с типом параметра шаблона, мы фактически создали «трафарет функции».

При вызове шаблона функции, компилятор использует «трафарет» в качестве образца функции, заменяя тип параметра шаблона на фактический тип переменных, передаваемых в функцию.

Разберём создание шаблона функции на примере всё той же функции sum.

Для начала ещё раз рассмотрим sum для значений типа int:

int sum(int a, int b) {

return a + b;

}

Здесь мы трижды указываем тип данных: в параметрах a, b и в типе возврата функции. Для создания шаблона этой функции нам нужно заменить тип int на тип параметра шаблона функции. Поскольку в этом случае используется только один тип данных (int), то нам нужно указать только один тип параметра шаблона.

Мы можем назвать этот тип как угодно, главное, чтобы это не было зарезервированным/ключевым словом (т.е. словами типа bool, class, int, sizeof, do, if, else и т.д.). В языке C++ принято называть типы параметров шаблонов большой буквой T (сокращённо от «Type»).

Вот так будет выглядеть наша функции после замены типа на тип параметра шаблона:

T sum(T a, T b) {

return a + b;

}

Однако после этого программа пока работать не будет, ибо компилятор не знает, что такое T.

Нам необходимо выполнить объявление параметров шаблона:

template <typename T>

Работает это так: ключевое слово template сообщает компилятору, что дальше мы будем объявлять параметры шаблона, которые указываются в угловых скобках (<>).

Для создания типов параметров шаблона используются ключевые слова typename и class. В базовых случаях использования шаблонов функций разницы между typename и class нет, поэтому можно выбрать любое из двух. Если использовать ключевое слово class, то передаваемый тип параметров не обязательно должен быть классом (это может быть переменная фундаментального типа данных, указатель или что-то другое).

Затем называем тип параметра шаблона (обычно T).

Если же требуется несколько типов параметров шаблона, то они разделяются запятыми:

template <typename T1, typename T2>

Также стоит упомянуть о том, что типом параметра шаблона может быть и non-type параметр.

**Параметр non-type в шаблоне** — это специальный параметр шаблона, который заменяется не типом данных, а конкретным значением. Этим значением может быть: целочисленное значение или перечисление, указатель или ссылка на объект класса, указатель или ссылка на функцию, указатель или ссылка на метод класса. То есть для примера из пункта C style с массивом мы могли бы использовать не только параметр типа, но и параметр non-type – размер массива:

template <class T, int size>

А объявление такого массива будет выглядеть так:

Array<double, 5> B; *//массив размера 5 элементов типа double*

Вернёмся к нашей функции sum, теперь вся наша программа будет выглядеть следующим образом**(2.cpp)**:

#include <iostream>

using namespace std;

template <typename T>

T sum(T a, T b) {

return a + b;

}

int main() {

int a = 5, b = 3;

std::cout << sum(a, b) << ' ';

double x = 5.5, y = 3.5;

std::cout << sum(x, y) << ' ';

} //всё верно работает

Замечание: Поскольку тип аргумента функции, передаваемый в тип T, может быть классом, а классы, как правило, не рекомендуется передавать по значению, то лучше сделать параметры и возвращаемое значение нашего шаблона функции константными ссылками, то есть в конечном итоге программа будет выглядеть так**(3.cpp)**:

#include <iostream>

using namespace std;

template <typename T>

const T& sum(const T& a, const T& b) {

return a + b;

}

int main() {

int a = 5, b = 3;

std::cout << sum(a, b) << ' ';

double x = 5.5, y = 3.5;

std::cout << sum(x, y) << ' ';

}

Язык C++ не компилирует шаблоны функций напрямую. Вместо этого, когда компилятор встречает вызов шаблона функции, он копирует шаблон функции и заменяет типы параметров шаблона функции фактическими (передаваемыми) типами данных. Функция с фактическими типами данных называется экземпляром шаблона функции (или «объектом шаблона функции»).

То есть в нашей программе при вызове sum(a,b) компилятор видит, что оба числа являются целочисленными, поэтому он копирует шаблон функции и создает экземпляр шаблона sum(int, int):

const int& sum(const int& a, const int& b) {

return a + b;

}

А при вызове sum(x,y) – sum(double,double):

const double& sum(const double& a, const double& b) {

return a + b;

}

Однако же типом параметра шаблона может быть и класс. Экземпляр шаблона компилируется как обычная функция. В обычной функции любые операторы или вызовы других функций, которые используются в этой функции, должны быть определены/перегружены, или вы получите ошибку компиляции. Аналогично, любые операторы или вызовы других функций, которые присутствуют в шаблоне функции, должны быть определены/перегружены для работы с фактическими (передаваемыми) типами данных.

Рассмотрим это на примере: создадим некий класс и вызовем функцию sum для него. **(4.cpp)**

#include <iostream>

using namespace std;

template <typename T>

T sum(T a, T b) {

return a + b;

}

class Numbers {

int x;

public:

Numbers(int i) :x(i) {};

};

int main() {

Numbers five(5), one(1);

sum(five, one);

} *//VS выдаёт следующую ошибку: error C2676: бинарный "+": "T" не определяет этот оператор или преобразование к типу приемлемо к встроенному оператору*

Сообщение об ошибке указывает на то, что необходимо перегрузить оператор +. Перегрузим **(5.cpp)**:

#include <iostream>

using namespace std;

template <typename T>

T sum(T a, T b) {

return a + b;

}

class Numbers {

int x;

public:

Numbers(int i) :x(i) {};

friend int operator +(const Numbers& A, const Numbers& B) {

return (A.x + B.x);

}

};

int main() {

Numbers five(5), one(1);

sum(five, one);

}

Программа работает, но для того, чтобы вывести получившийся ответ перегрузим оператор вывода для нашего класса **(6.cpp)**:

#include <iostream>

using namespace std;

template <typename T>

T sum(T a, T b) {

return a + b;

}

class Numbers {

int x;

public:

Numbers(int i) :x(i) {};

friend int operator +(const Numbers& A, const Numbers& B) {

return (A.x + B.x);

}

friend std::ostream& operator<< (std::ostream& out, const Numbers& A)

{

out << A.x;

return out;

}

};

int main() {

Numbers five(5), one(1);

std::cout << sum(five, one);

} *//выводит верный ответ*

Итак, мы рассмотрели работу шаблонов функции. Аналогичным образом работают **шаблоны классов**.

Так же рассмотрим класс массивов (как и в C style).

Создаём описание класса:

template <typename T>

class Array {

size\_t mySize;

T\* myData;

public:

Array();

Array(size\_t size);

Array(Array const& a);

~Array();

Array& operator =(Array const& A);

T& get(size\_t i);

void set(size\_t i, T value);

size\_t size() const;

};

Далее определим методы – так как они все вне тела класса, то нуждаются в собственных определениях шаблона метода (т.е. template <typename T>):

template<typename T>

Array<T>::Array() {

mySize = 0;

myData = nullptr;

}

template<typename T>

Array<T>::Array(size\_t size):mySize(size),myData(new T[size]){}

template<typename T>

Array<T>::Array(Array const& A):mySize(A.mySize),myData(new T[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

template<typename T>

Array<T>::~Array() {

delete[] myData;

}

template<typename T>

Array<T>&Array<T>::operator =(const Array<T>&A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new T[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

template<typename T>

T&Array<T>::get(size\_t i) {

return myData[i];

}

template<typename T>

void Array<T>::set(size\_t i, T value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

template<typename T>

size\_t Array<T>::size()const {

return mySize;

}

Теперь, соединив всё вместе и написав пример, получим следующее: **(7.cpp)**

#include <iostream>

using namespace std;

template <typename T>

class Array {

size\_t mySize;

T\* myData;

public:

Array();

Array(size\_t size);

Array(Array const& a);

~Array();

Array& operator =(Array const& A);

T& get(size\_t i);

void set(size\_t i, T value);

size\_t size() const;

};

template<typename T>

Array<T>::Array() {

mySize = 0;

myData = nullptr;

}

template<typename T>

Array<T>::Array(size\_t size) :mySize(size), myData(new T[size]) {}

template<typename T>

Array<T>::Array(Array const& A) : mySize(A.mySize), myData(new T[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

template<typename T>

Array<T>::~Array() {

delete[] myData;

}

template<typename T>

Array<T>& Array<T>::operator =(const Array<T>& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new T[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

template<typename T>

T& Array<T>::get(size\_t i) {

return myData[i];

}

template<typename T>

void Array<T>::set(size\_t i, T value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

template<typename T>

size\_t Array<T>::size()const {

return mySize;

}

int main() {

Array<int> A(10); *//вызвали нетривиальный конструктор для значений типа int*

Array<int> X; *//вызвали тривиальный конструктор для значений типа int*

for (size\_t i = 0; i < 10; i++)

A.set(i, i);

X = A; *//присвоили значения массива A массиву X*

for (size\_t i = 0; i < X.size(); i++)

std::cout << X.get(i) << ' ';

std::cout << endl;

Array<double> B(5); *//вызвали тривиальный конструктор для значений типа double*

for (size\_t i = 0; i < 5; i++)

B.set(i, i + 0.03);

for (size\_t i = 0; i < B.size(); i++)

std::cout << B.get(i) << ' ';

std::cout << endl;

Array<char> D(10); *//вызвали тривиальный конструктор для значений типа char*

for (size\_t i = 0; i < 10; i++)

D.set(i, i+97);

for (size\_t i = 0; i < 10; i++)

std::cout << D.get(i) << ' ';

} *//программа выведет:*

*//0 1 2 3 4 5 6 7 8 9*

*//0.03 1.03 2.03 3.03 4.03*

*//a b c d e f g h i j*

Однако этим возможности нашего шаблона не ограничиваются - возможно так же создать массив с типом параметра таким же массивом. Для того, чтобы увидеть результат работы программы с таким массивом, нам необходимо перегрузить оператор вывода, что мы и сделаем. В результате получим: **(8.cpp)**

#include <iostream>

using namespace std;

template <typename T>

class Array {

size\_t mySize;

T\* myData;

public:

Array();

Array(size\_t size);

Array(Array const& a);

~Array();

Array& operator =(Array const& A);

T& get(size\_t i);

void set(size\_t i, T value);

size\_t size() const;

friend std::ostream& operator<<(std::ostream& out, const Array& A) {

for (size\_t i = 0; i < A.mySize; i++)

out << A.myData[i] << ' ';

std::cout << endl;

return out;

}

};

template<typename T>

Array<T>::Array() {

mySize = 0;

myData = nullptr;

}

template<typename T>

Array<T>::Array(size\_t size) :mySize(size), myData(new T[size]) {}

template<typename T>

Array<T>::Array(Array const& A) : mySize(A.mySize), myData(new T[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

template<typename T>

Array<T>::~Array() {

delete[] myData;

}

template<typename T>

Array<T>& Array<T>::operator =(const Array<T>& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new T[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

template<typename T>

T& Array<T>::get(size\_t i) {

return myData[i];

}

template<typename T>

void Array<T>::set(size\_t i, T value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

template<typename T>

size\_t Array<T>::size()const {

return mySize;

}

int main() {

Array<int> A(10); *//вызвали нетривиальный конструктор для значений типа int*

Array<int> X; *//вызвали тривиальный конструктор для значений типа int*

for (size\_t i = 0; i < 10; i++)

A.set(i, i);

X = A; *//присвоили значения массива A массиву X*

for (size\_t i = 0; i < X.size(); i++)

std::cout << X.get(i) << ' ';

std::cout << endl;

Array<double> B(5); *//вызвали тривиальный конструктор для значений типа double*

for (size\_t i = 0; i < 5; i++)

B.set(i, i + 0.03);

for (size\_t i = 0; i < B.size(); i++)

std::cout << B.get(i) << ' ';

std::cout << endl;

Array<char> D(10); *//вызвали тривиальный конструктор для значений типа char*

for (size\_t i = 0; i < 10; i++)

D.set(i, i+97);

for (size\_t i = 0; i < 10; i++)

std::cout << D.get(i) << ' ';

std::cout << endl;

Array<Array<int>> Y(2); *//возможно даже создать массив из массивов*

Y.set(0, A);

Y.set(1, X);

for (size\_t i = 0; i < 2; i++)

std::cout << Y.get(i);

}  *//программа выведет:*

*//0 1 2 3 4 5 6 7 8 9*

*//0.03 1.03 2.03 3.03 4.03*

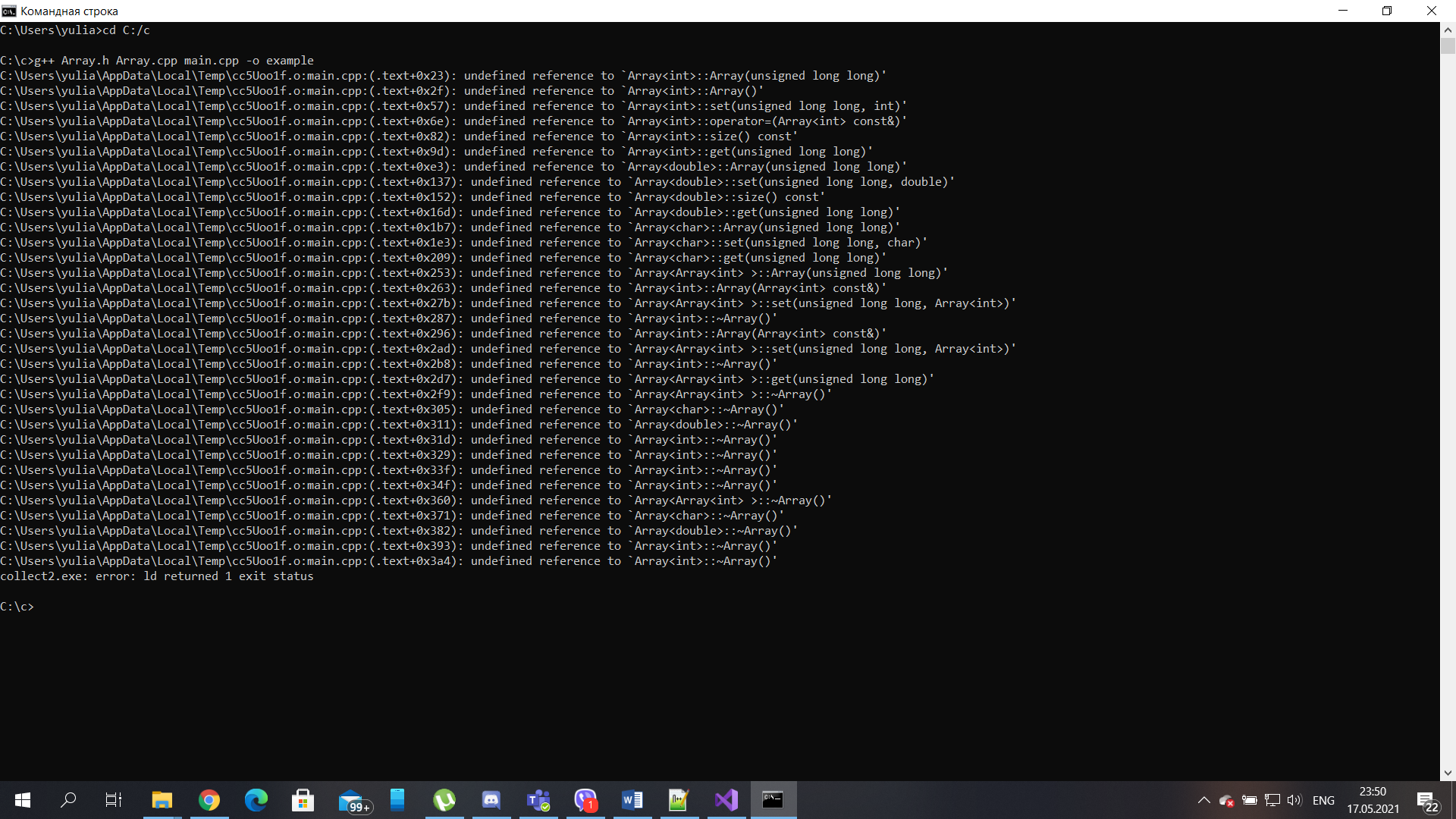
*//a b c d e f g h i j*

*//0 1 2 3 4 5 6 7 8 9*

*//0 1 2 3 4 5 6 7 8 9*

Шаблоны классов работают точно так же, как и шаблоны функций: компилятор копирует шаблон класса, заменяя типы параметров шаблона класса на фактические (передаваемые) типы данных, а затем компилирует эту копию.

Важно заметить некоторую **особенность компиляции шаблонов**. Шаблон не является ни классом, ни функцией — это трафарет, используемый для создания классов или функций. Таким образом, шаблоны работают не так, как обычные функции или классы. Работая с обычными классами мы часто помещаем определение класса в заголовочный файл, а определения методов этого класса в отдельный файл .cpp с аналогичным именем. Таким образом, фактическое определение класса компилируется как отдельный файл внутри проекта. Однако с шаблонами всё происходит несколько иначе. Рассмотрим на примере:

Запишем описание класса в файл Array.h, реализацию методов в Array.cpp, а основную часть программы в main.cpp. При сборке программы компилятор выдаст следующее: 

Это происходит потому, что язык C++ компилирует файлы по отдельности. Поэтому для использования шаблона компилятор должен видеть как определение шаблона (а не только объявление), так и тип шаблона, применяемый для создания экземпляра шаблона.

Когда заголовочный файл Array.h подключается в main.cpp, то определение шаблона класса копируется в этот файл. В main.cpp компилятор видит, что нам нужны четыре экземпляра шаблона класса: Array<int>, Array<double>, Array<char> и Array<Array<int>> он создаст их, а затем скомпилирует весь этот код как часть файла main.cpp. Однако, когда дело дойдет до компиляции Array.cpp (отдельным файлом), компилятор забудет, что мы использовали Array<int>, Array<double>, Array<char> и Array<Array<int>> в main.cpp и не создаст экземпляры шаблонов функций, которые нам нужны для выполнения программы.

Эту проблему можно решить несколькими способами.

Самый простой вариант — поместить код из Array.cpp в Array.h ниже класса. Таким образом, когда мы будем подключать Array.h, весь код шаблона класса (полное объявление и определение как класса, так и его методов) будет находиться в одном месте. Плюс этого способа — простота. Минус — если шаблон класса используется во многих местах, то мы получим много локальных копий шаблона класса, что увеличит время компиляции и линкинга файлов. Рекомендуется использовать это решение до тех пор, пока время компиляции или линкинга не является проблемой.

Другое решение — использовать подход 3-х файлов: определение шаблона класса хранится в заголовочном файле, определения методов шаблона класса хранятся в отдельном файле .cpp, а в третьем файле содержатся все необходимые нам экземпляры шаблона класса.

Всё это будет выглядеть следующим образом: **(Array)**

**(Array.h)**

#ifndef ARRAY\_H

#define ARRAY\_H

#include <iostream>

template <typename T>

class Array {

size\_t mySize;

T\* myData;

public:

Array();

Array(size\_t size);

Array(Array const& a);

~Array();

Array& operator =(Array const& A);

T& get(size\_t i);

void set(size\_t i, T value);

size\_t size() const;

friend std::ostream& operator<<(std::ostream& out, const Array& A) {

for (size\_t i = 0; i < A.mySize; i++)

out << A.myData[i] << ' ';

std::cout << '\n';

return out;

}

};

#endif

**(Array.cpp)**

#include "Array.h"

#include <iostream>

template<typename T>

Array<T>::Array() {

mySize = 0;

myData = nullptr;

}

template<typename T>

Array<T>::Array(size\_t size) :mySize(size), myData(new T[size]) {}

template<typename T>

Array<T>::Array(Array const& A) : mySize(A.mySize), myData(new T[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

template<typename T>

Array<T>::~Array() {

delete[] myData;

}

template<typename T>

Array<T>& Array<T>::operator =(const Array<T>& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new T[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

template<typename T>

T& Array<T>::get(size\_t i) {

return myData[i];

}

template<typename T>

void Array<T>::set(size\_t i, T value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

template<typename T>

size\_t Array<T>::size()const {

return mySize;

}

**(main.cpp)**

#include <iostream>

#include "Array.h"

#include "templates.cpp"

using namespace std;

int main() {

Array<int> A(10);

Array<int> X;

for (size\_t i = 0; i < 10; i++)

A.set(i, i);

X = A;

for (size\_t i = 0; i < X.size(); i++)

std::cout << X.get(i) << ' ';

std::cout << '\n';

Array<double> B(5);

for (size\_t i = 0; i < 5; i++)

B.set(i, i + 0.03);

for (size\_t i = 0; i < B.size(); i++)

std::cout << B.get(i) << ' ';

std::cout << '\n';

Array<char> D(10);

for (size\_t i = 0; i < 10; i++)

D.set(i, i + 97);

for (size\_t i = 0; i < 10; i++)

std::cout << D.get(i) << ' ';

std::cout << '\n';

Array<Array<int>> Y(2);

Y.set(0, A);

Y.set(1, X);

for (size\_t i = 0; i < 2; i++)

std::cout << Y.get(i);

}

**(templates.cpp)**

#include "Array.h"

#include "Array.cpp"

template class Array<int>; *// явно создаем экземпляр шаблона класса Array<int>*

template class Array<double>; *// явно создаем экземпляр шаблона класса Array<double>*

template class Array<char>; *//явно создаем экземпляр шаблона класса Array<char>*

template class Array<Array<int>>; *// явно создаем экземпляр шаблона класса Array<Array<int>>*

Часть template class заставит компилятор явно создать указанные экземпляры шаблона класса. В нашем случае компилятор создаст Array<int>, Array<double>, Array<char> и Array<Array<int>> внутри templates.cpp. Поскольку templates.cpp находится внутри нашего проекта, то он скомпилируется и удачно свяжется с другими файлами (пройдет линкинг).

Этот метод более эффективен, но всегда требует создания/поддержки третьего файла (templates.cpp) для каждой из программ (проектов) отдельно.

Рассмотрим также важную особенность шаблонов – **специализацию**.

Специализация бывает явной(полной) и частичной, для функций и для классов. Разберём всё по порядку – начнём с **полной специализации шаблона функции**. Предположим, мы хотим получать из массива элементы типа double в два раза больше, чем они есть на самом деле.

Проведём полную специализацию функции get. Записываем экземпляр шаблона функции (если функция является методом класса, то делаем это за пределами класса), указывая нужный нам тип данных.

template <>

double& Array<double>::get(size\_t i){

double x = 2 \* myData[i];

return x;

}

Часть template <> сообщает компилятору, что это шаблон функции, но без параметров (так как в этом случае мы явно указываем нужный нам тип данных).

Когда компилятору нужно будет создать экземпляр Array<double>::get(size\_t i)), он увидит, что мы уже явно определили эту функцию, и поэтому он будет использовать именно этот экземпляр, а не копировать общую для всех типов данных версию шаблона функции get(size\_t i).

Пример работы программы **(9.cpp)**

Итак, далее рассмотрим **явную специализацию класса**.  
Предположим, мы хотим создать массив размера 8 из переменных типа bool. Та программа, которая написана сейчас **(8\_bool.cpp)** сработает действительно правильно, однако реализация Array<bool>, на самом деле, не столь эффективна, какой она могла бы быть. Поскольку все переменные должны иметь адрес, а центральный процессор не может дать адрес чему-либо меньшему, чем 1 байт, то размер всех переменных должен быть не менее 1 байта. Следовательно, каждая переменная типа bool занимает целый байт, хотя технически ей нужен только 1 бит для хранения значения true или false. Таким образом, переменная типа bool — это 1 бит полезной информации и 7 бит потраченного впустую места. Получается, что класс Array<bool>, который имеет 8 переменных типа bool, фактически работает только с 1 байтом данных, а остальные 7 байтов тратятся зря.  
Этого можно избежать, используя явную специализацию шаблона класса.

Специализация шаблона класса рассматривается компилятором как полностью отдельный и независимый класс, хоть и выделяется как обычный шаблон класса. Это означает, что мы можем изменить в классе всё что угодно, включая его реализацию/методы/спецификаторы доступа и т.д.

Вот так будет выглядеть класс Array<bool>:

template<>

class Array<bool> {

size\_t mySize;

char\* myData;

public:

Array() { *//тривиальный конструктор не изменится*

mySize = 0;

myData = nullptr;

}

Array(size\_t size) :mySize(size), myData(new char[(size + 7) / 8]) {}

Array(Array const& A) : mySize(A.mySize), myData(new char[(mySize + 7) / 8]) {

for (size\_t i = 0; i < (mySize + 7) / 8; i++)

myData[i] = A.myData[i];

}

~Array() { *//деструктор не изменится*

delete[] myData;

}

Array<bool>& operator =(Array<bool> const& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new char[(mySize + 7) / 8];

for (size\_t i = 0; i < (mySize + 7) / 8; i++)

myData[i] = A.myData[i];

return \*this;

}

bool get(size\_t i) {

char temp = myData[i / 8];

return (temp >> (i % 8)) % 2; *// >> - битовый сдвиг т.е. 10011011 >> 1 = 01001101*

}

void set(size\_t i, bool value) {

if ((i >= 0) && (i < mySize)) {

if ((myData[i/8] >> (i % 8)) % 2 == value) return; *// проверяем, не имеет ли этот бит уже нужное нам значение*

unsigned char mask = 1 << (i % 8); *// создаём маску с единицей на нужном нам месте, т.е. если i=3, mask=00001000*

if (value) myData[i/8] |= mask; *// если нам необходимо значение true, то проводим логическое сложение маски и элемента массива myData*

else myData[i/8] &= ~mask; *// если нам необходимо false, проводим логическое умножение с отрицанием маски (т.е. 11110111 для примера маски представленного выше)*

}

}

size\_t size() const { *//не изменится*

return mySize;

}

};

Убедится в том, что всё работает так же правильно, можно в **10.cpp**.

Следующим пунктом в теме шаблонов является **частичная специализация**.

**Частичная специализация шаблона** позволяет выполнить специализацию шаблона класса (но не функции!), где некоторые (но не все) параметры шаблона явно определены. Рассмотрим её на примере класса Array с двумя типами параметров: типом элементов массива и non-type параметром – размером массива (Array <typename T, size\_t size>).

Добавим в наш код 8.cpp параметр size и функцию вывода элементов массива (вместо оператора вывода) print:

template <typename T, size\_t size>

void print(Array<T, size>& A){

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i) << ' ';

std::cout << endl;

}

И попробуем в main записать следующее: **(11.cpp)**

int main() {

Array<char,14> D;

char\* d = new char[14];

strcpy\_s(d, 14, "Hello, world!"); *// подключаем в начале библиотеку cstring для использования преобразования строки в массив типа char*

for (size\_t i = 0; i < 14; i++)

D.set(i,d[i]);

print(D);

}

Программа выведет «H e l l o , w o r l d !» - с пробелами, так как в функции print у нас указано выводит пробел после каждого элемента массива. Именно здесь нам пригодится частичная специализация.

Почему не подойдёт полная специализация? Проблема с полной специализацией шаблона заключается в том, что все параметры шаблона должны быть явно определены. Поэтому мы можем создать лишь специализацию конкретного размера массива (Array<char, 14>) и уже для других размеров она работать не будет.

Вот так будет выглядеть шаблон функции print(), который принимает частично специализированный шаблон класса Array:

template <size\_t size>

void print(Array<char, size>& A) {

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i);

std::cout << endl;

}

Вся программа будет выглядеть так: **(12.cpp)**

#include <iostream>

#include <cstring>

using namespace std;

template <typename T, size\_t size>

class Array {

size\_t mySize = size;

T\* myData;

public:

Array();

Array(Array const& a);

~Array();

Array& operator =(const Array& A);

T& get(size\_t i);

void set(size\_t i, T value);

};

template <typename T, size\_t size>

Array<T, size>::Array() {

myData = new T[size];

}

template <typename T, size\_t size>

Array<T, size>::Array(Array const& A) : mySize(A.mySize), myData(new T[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

template <typename T, size\_t size>

Array<T, size>::~Array() {

delete[] myData;

}

template <typename T, size\_t size>

Array<T, size>& Array<T, size>::operator =(const Array<T, size>& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new T[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

template <typename T, size\_t size>

T& Array<T, size>::get(size\_t i) {

return myData[i];

}

template <typename T, size\_t size>

void Array<T, size>::set(size\_t i, T value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

template <typename T, size\_t size>

void print(Array<T, size>& A){

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i) << ' ';

std::cout << endl;

}

template <size\_t size> *// наша частичная специализация*

void print(Array<char, size>& A) {

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i);

std::cout << endl;

}

int main() {

Array<char,14> D;

char\* d = new char[14];

strcpy\_s(d, 14, "Hello, world!");

for (size\_t i = 0; i < 14; i++)

D.set(i,d[i]);

print(D);

Array<char, 16> B;

char\* b = new char[16];

strcpy\_s(b, 16, "Hello everybody");

for (size\_t i = 0; i < 16; i++)

B.set(i, b[i]);

print(B);

Array<double, 5> A;

for (size\_t i = 0; i < 5; i++)

A.set(i, i + 0.03);

print(A);

} *//программа выведет (для массивов типа char без пробелов, для double - с пробелами ):*

*//Hello, world!*

*//Hello everybody*

*//0.03 1.03 2.03 3.03 4.03*

Теперь предположим, наша функция print является методом класса. Мы так же хотим провести для неё частичную специализацию. Она будет называться **частичная специализация шаблона метода**. Однако частично специализировать шаблон функции запрещено, поэтому что-то вроде:

template <int size>

void Array<char, size>::print()

{

for (int i = 0; i < size; i++)

std::cout << A.get(i);

std::cout << endl;

}

работать не будет.

Решение в данной ситуации будет частичная специализация шаблона всего класса: **(13.cpp)**

template<size\_t size>

class Array<char, size> {

size\_t mySize = size;

char\* myData;

public:

Array() {

myData = new char[size];

}

Array(Array const& A) : mySize(A.mySize), myData(new char[A.mySize]) {

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

}

~Array() {

delete[] myData;

}

Array& operator =(const Array& A) {

if (myData) delete[] myData;

mySize = A.mySize;

myData = new char[mySize];

for (size\_t i = 0; i < mySize; i++)

myData[i] = A.myData[i];

return \*this;

}

char& get(size\_t i) {

return myData[i];

}

void set(size\_t i, char value) {

if ((i >= 0) && (i < mySize))

myData[i] = value;

}

void print(Array& A) {

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i);

std::cout << endl;}

};

И хотя работает всё верно, у нас теперь куча дублированного кода из Array<T, size> в Array<char, size>.

Для решения этой проблемы стоит поговорить о **наследовании** от шаблонного класса.

Множества дублированного кода можно избежать с помощью общего родительского шаблона класса Array для шаблона класса myArray и шаблона класса myArray с частичной специализацией.

Надо понимать, что сам по себе шаблон не является классом – это трафарет, из которого могут быть сгенерированы классы. То есть Array не класс, а вот Array<int, 5>, Array<double, 10>, Array<char, 15> и т.д. – классы, при том совершенно разные, не имеющие ничего общего, кроме того, что они были сгенерированы из одного и того же шаблона класса.

Из-за того, что Array не является классом, мы не можем создать класс-наследник myArray от него. Притом можно отнаследоваться от классов Array<int, 5>, Array<double, 10>, Array<char, 15> и т.д., причём даже от нескольких одновременно, ибо это по сути разные классы:

class myArray:

public Array<int, 5>,  
 public Array<double, 10>  
{…};

В этом случае при определении myArray необходимо указать, из какого класса следует выводить производные. Это обязательно независимо от того, генерируются ли эти классы из шаблона или нет. Два объекта одного класса просто не могут иметь разных иерархий наследования.

Однако нам необходим не класс, а так же, как и Array, шаблон класса, и вот это уже возможно следующим образом:

template<class T, size\_t size>

class myArray: public Array<T,size>

{…};

Вся программа будет выглядеть так: **(14.cpp)**

class Array { *// общий родительский шаблон*

protected:

size\_t mySize = size; *// меняем спецификатор доступа с private на protected, чтобы возможно было использовать их наследниками*

T\* myData;

public:

Array();

Array(Array const& A);

~Array();

Array& operator =(const Array& A);

T& get(size\_t i);

void set(size\_t i, T value);

void print(Array& A);

};

. . . *// реализация методов*

template<class T, size\_t size>

class myArray: public Array<T,size> { *// создаём шаблон-наследник*

public:

myArray(){}

};

template <size\_t size>

class myArray<char, size> : public Array<char, size> *// создаём ещё один шаблон-наследник, но уже с частичной специализацией*

{

public:

virtual void print(myArray& A){

for (size\_t i = 0; i < size; i++)

std::cout << A.get(i);

std::cout << endl;

}

};

На этом основные моменты изучения шаблонов в C++ заканчиваются. Стоит отметить, что шаблоны значительно облегчают написание кода и сокращают его размер - служат для увеличения эффективности программирования.