Шаблоны (template). Как обычно рассмотрим ситуацию, как люди выживали до этого. Мы с вами рассматривали ряд классов основанных на конкретный тип данных.

class Safearray {

private: int \*data\_;

};

Основная задача была – скрыть данные внутри и обезопасить. Очевидно бывает нужен массив и для double и для float и для char. Пришлось бы создавать ещё такие же классы, где всё было похожим, кроме названия типа. То же самое с умными указателями.

class SharedPointer {

private: Complex \*obj;

};

То же самое, указатель умный нужен и для других классов.

Как такая задача решается в классическом Си? С помощью препроцессора, директива define. Пример:

#define Array(Type)\

class Array {\

Type\* data;\

... \

};

Вообще по синтаксису макрос должен быть вытянут в одну строку. Обратный слеш позволяет продолжать строку. Это макрос с параметром, в месте, где Type можно подставлять тип. Теперь можно написать:

Array(int); Array ai; Array(complex); Array ac; // везде внутри подставится вместо слова Type указанный конкретно вами тип. Проблема состоит в том, что если в одном файле массив одного типа, то все нормально. Но если два разных массива, то получается два класса с одним именем. Не скомпилируется из-за multiply definition. Изменим немного макрос.

#define Array(Type,Name)\

class Array#Name {\

Type\* data;\

... \

};

Синтаксис # - склеивание двух строк, если написать ArrayName то компилятор бы считал, что это один идентификатор. У макроса теперь два параметра. Теперь можно писать:

Array(int,Int); Array(complex,Complex); Дальше можно писать используя имя класса Array#Name

ArrayInt ai; ArrayComplex ac;

Это решение в стиле Си. Проблема состоит в том, что компилятор и вы видите код по-разному. Сложнее искать и отлаживать.

Теперь про С++.

template <class T>

class Array {

private: T\* data;

T& operator[](size\_t i) { return data[i]; }

};

Ключевое слово template обозначает, что перед нами шаблонный класс (или шаблонная функция). T – имя параметра шаблона. Вместо class T можно писать typename. Дальше везде, где предполагалось использование типа, пишем T. Как это использовать?

Array<int> arrayint; Важное замечание – при использовании шаблонов уже **нет раздельной компиляции**. Мы не можем предварительно скомпилировать объектный код нашего шаблонного класса и компилировать только main, потом линковать всё вместе. Компилятор при обработке main.cpp должен знать не только объявления методов шаблонного класса, но и определения, иначе никак не скомпилировать инструкции типа return data[i]; не зная типа, то есть невозможно её скомпилировать независимо, а потом во время линковки подставить типы. Таким образом если вы работаете с библиотекой шаблонов, то она поставляется в виде исходников, а не предварительно скомпилированных объектных файлов, т.к. определение необходимо в момент компиляции.

Array<double> arraydouble; Теперь при компиляции будет создано два разных класса, какие конкретно у них будут имена мы не знаем точно, зависит от компилятора. Все методы внутри шаблона inline. Какие недостатки? Увеличивается время компиляции.

Есть второй вариант синтаксиса, определение отдельно от объявления:

template <class T>

T& Array<T> :: operator[](size\_t i)

{

return data[i];

}

#include “Array\_impl.h”

Такую штуку надо писать перед каждым методом. Файл с объявлением называют Array.h, с определениями называют Array\_impl.h. Хотя это всё фикция, раздельной компиляции все равно не будет, в момент компиляции все определения подключатся к объявлениям.

Шаблонные функции.

template <typename T> // буква T совершенно точно не является идеальным именем

void swap (T& a, T& b) {

T t(a);

a=b;

b=t;

} // код меняющий местами два объекта любого типа. Какое неявное предположение делается о классах, которые используются в этой функции? У них должен быть конструктор копирования и оператор присваивания. По тексту шаблона невозможно понять, будет ли работать эта функция для конкретного класса. Процесс инстанцирования (instance – экземпляр, пример) шаблона – конкретизация шаблона. Пример:

int i = 10; int j = 20;

swap<int>(i,j); // для примитивных типов данных возможен синтаксис конструктора копирования

Поскольку компилятор довольно умная штука, можно написать swap(i,j); компилятор догадается о параметре шаблона, но можно указывать явно. Бывают случаи (могут зависеть от особенностей самой функции и вызова функции), когда компилятор не может догадаться о параметре шаблона и выдаст ошибку компиляции.

Шаблонных параметров может быть несколько.

template <class F, class S>

struct pair {

F first;

S second; };

pair <int, double> idp; // такие структуры удобны, много где используются – ключ+значение или id+имя. Внутри одного шаблона может быть другой шаблон.

pair < int, Array< int > > obj; // если не сделать пробел, компилятор может перепутать со сдвигом.

pair < int, pair <int, double> > obj2; // сначала создастся один класс, ему присвоится какое-то имя, потом создастся другой класс, со своим именем, а потом уже создастся объект obj2.

Пример шаблонного метода с двумя параметрами.

copy.h

template <class T, class V>

bool copy (Array <T>& a, const Array <V>& b) {

if (a.size() != b.size()) return false;

else

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

a.set(i,b.get(i));

}

Array <int> ia; Array <double> da; copy (ia, da); // компилятор опять же поймёт о параметрах шаблона. Вообще говоря, double приводить к int плохо. В цикле надо написать static\_cast. Вариант copy (da, ia); без проблем сработает.

main.cpp:

#include “copy.h”

Если в main функция copy не используется, то шаблон не будет компилироваться в принципе. Шаблон компилируется только в момент конкретизации шаблона (инстанцирования). При вызове copy с параметрами шаблона уже произойдёт компиляция.

Осталось рассмотреть шаблонный метод шаблонного класса.

template <class T> class Array {

private: T\* data;

private: size\_t size;

public:

template <class V>

Array<T>& operator= (const Array<V>& objV) {

... опустим проверку this выделение памяти освобождение и прочее

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

data[i] = objV.data[i];

}

}; // если определять функцию вне объявления класса то строчка будет длиннее:

template <class T, class V>

Array<T>& Array<T> :: operator= (const Array<V>& objV) { ... }

Вызов:

Array<int> ia; Array<double> da;

da = ia; // в процессе компиляции получатся два разных класса с разными именами (аналогия, как линковщик именует перегружаемые функции по-особому), никакого доступа к приватным полям друг друга у них не будет, поэтому при вызове у da метода оператор= не получится считать значения objV.data[i]. Поэтому так делать нельзя. Нужны публичные методы size, get.

Для сокращения синтаксиса удобно использовать typedef. Вместо конструкций pair <int, Array<int> > obj; приятнее typedef Array<int> AInt; тогда pair <int, AInt> obj;

Что можно указывать в качестве параметра template? Мы рассмотрели использование типа. Можно использовать значение переменной. Это нужно, если, например, у вас есть статический массив какого-то размера int mass[k]; k будет параметром шаблона.

Специализация шаблона.

Если вы реализовали массив шаблонный, но хотите как-то по-особому работать при шаблонном параметре bool. Хочется меньше памяти использовать, работать с таким массив не как с double или int, а как-то группировать биты. Можно сделать такую специальную специализацию. В .h задаём общий случай, а потом ниже описываем специализацию для Array<bool>. Частичная специализация шаблона. Пример:

template <class T>

class DataBase

{

T\* data;

unsigned int size; … и так далее

Специализация:

template <> class DataBase <bool>

{

unsigned char\* data; // используем каждые 8 бит получается меньше памяти нужно

unsigned int size; … и т.д.

DataBase<int> dbi(9, 5); // будет использован общий случай

DataBase<bool> dbb(true, 10); // будет использована специализация шаблона

Можно накладывать ограничения на тип, передаваемый в качестве параметра шаблона.

template <typename T> class C; // only declaration

template <> class C<int> {}; // full definition

template <> class C<double> {}; // full definition

main()

{ C<int> ci; C<Complex> cc; // ошибка компиляции }

Можно сделать так, чтобы передать можно было только указатель, например.

Когда мы описываем шаблон, есть понятие шаблонные переменные (то, что мы передаём в шаблон, чтобы сделать из него настоящий объект). Что туда можно передавать?

1) Тип std::vector<int> vi;

2) Целое число

template<size\_t size> class BitSet {

char m[size/8]; };

Если мне нужно сохранить множество бит, с помощью логических операций в один байт можно положить 8 бит. Хочется задать размер массива, это можно делать с помощью шаблона. Использование: BitSet<16> bs; Есть похожий пример:

class Model {

const int N=5;

int board[N][N]; };

Так делать нельзя, константу можно проинициализировать только в конструкторе. Надо было либо использовать #define, либо использовать динамический массив. А ещё можно с помощью шаблона сделать то же самое, размер поля будет параметром шаблона.

3) Указатель / ссылка на глобальную переменную:

const double g\_mars = 3,7;

const double g\_earth = 9.8;

template <const double & g> class World {

... }; Использование: World<g\_mars> mars; World<g\_earth> earth; // Смысл – есть класс, описывающий физику какого-то мира. Способом № 2 это сделать нельзя, т.к. тут double. В итоге здесь получится два разных класса. Между ними не будет работать приведение типа. Можно так сделать, чтобы человек, использующий эти объекты не мог ошибиться, передав в какую-то функцию объект не того мира. Синтаксис редкий, глобальные переменные – вообще плохо, так лучше не делать. 4) Неинстанцированный шаблон (instance – экземпляр класса, т.е. здесь имеется в виду не экземпляр класса):

template <class T, template <class V> > V<T> create\_container(size\_t size) {

V<T> t; ... использование size ... return t; }

Если я хочу сделать такую функцию, которая в зависимости от переданных параметров вернёт либо vector, либо list. Например функция, создающая контейнер и наполняющая его какими-то числами, при этом тип контейнера хочется передать как параметр. То есть внутри должен стоять тип, у которого тоже есть шаблонный параметр. Вызов:

create\_container <int, std::vector>(100); // Вместо V подставится шаблонный класс, в него подставится тип T (int здесь). В STL как раз stack / queue можно делать на list, так и на vector или deque. Там применяются как раз такие механизмы. Передаётся тип и неинстацированный шаблон.

Специализация шаблона. Мы описывали SafeArray, говорили, что хорошо бы его сделать шаблонным, чтобы не описывать для каждого типа, который мы захотим хранить внутри.

template <class T> class SafeArray {

... }; Однако я понимаю, что для некоторых типов я могу сделать более эффективную версию. И если будут создавать объект на таком типе, то поведение будет специальное. Синтаксис такой:

template <> class SafeArray <bool> {

... }; Здесь можно уже описать более эффективно, хранить по 8 значений в каждом байте в массиве, вместо 8 байт. Если вызовут SafeArray<int> ar; то применится общий шаблон, если SafeArray<bool> arr2; то вызовется вариант, который работает с битами. Ещё пример:

template <size\_t n> class Points {

int arr[n];

}; Пусть это будут точки на прямой. Для какого случая стоит написать отдельную специализацию, чтобы она выглядела поизящнее? Для одной точки, делать массив не очень хорошо в таком случае. Специализируем:

template <> class Points<1> {

int p; ... }; Специализация – указание специальных случаев. Посмотрим ещё пару примеров:

template <class T> class MyArray <T\*> { ... };

template <class T> class MyArray <MyArray <T> > { ... };

Такая специализация говорит, что можно делать MyArray <int\*> ar; - будет работать первая специализация, второй вариант – двумерные массивы. Если без специализации, то вариант только по-старинке организовывать циклы с new и указателями. А здесь можно MyArray <MyArray<int> > mass; Подобного механизма для шаблонных функций нет. Как сделать? Там можно сделать перегрузку, чтобы функция работала по-разному для разных типов. Пример:

template <class T> void sort (Array<T> & m)

template <class T> void sort (T& t) { } – для какого-то конкретного типа

Не забываем, что шаблоны – это умные #define на этапе компиляции, позволяющие избежать много ошибок.

Параметры по умолчанию. Синтаксис:

template <class T=string> class V {

T name; ... }; // То есть если этот шаблон инстанцировать V<> obj; то поле имя будет string по умолчанию. Идея какая? Кто-то использует мой класс, описывающий человека. На STL кто-то захочет использовать string, кто-то может все-таки массив char захочет.

Если template <class T1, class T2=string > class A { ... }; то можно писать A <int> a; Важно, что типы со значением по умолчанию должны идти в конце.