

Языки программирования

Лекция 6

ПМИ Семестр 2

Демяненко Я.М.

2024

Virtual Methods Table (VMT) и система RTTI (Runtime Type Identification)

Полиморфизм в C++ реализуется с помощью таблиц виртуальных функций — Virtual Methods Table (VMT)

Для каждого класса, содержащего хотя бы одну виртуальную функцию, создаётся VMT, которая содержит адреса всех виртуальных функций, как этого класса, так и всех его предков.

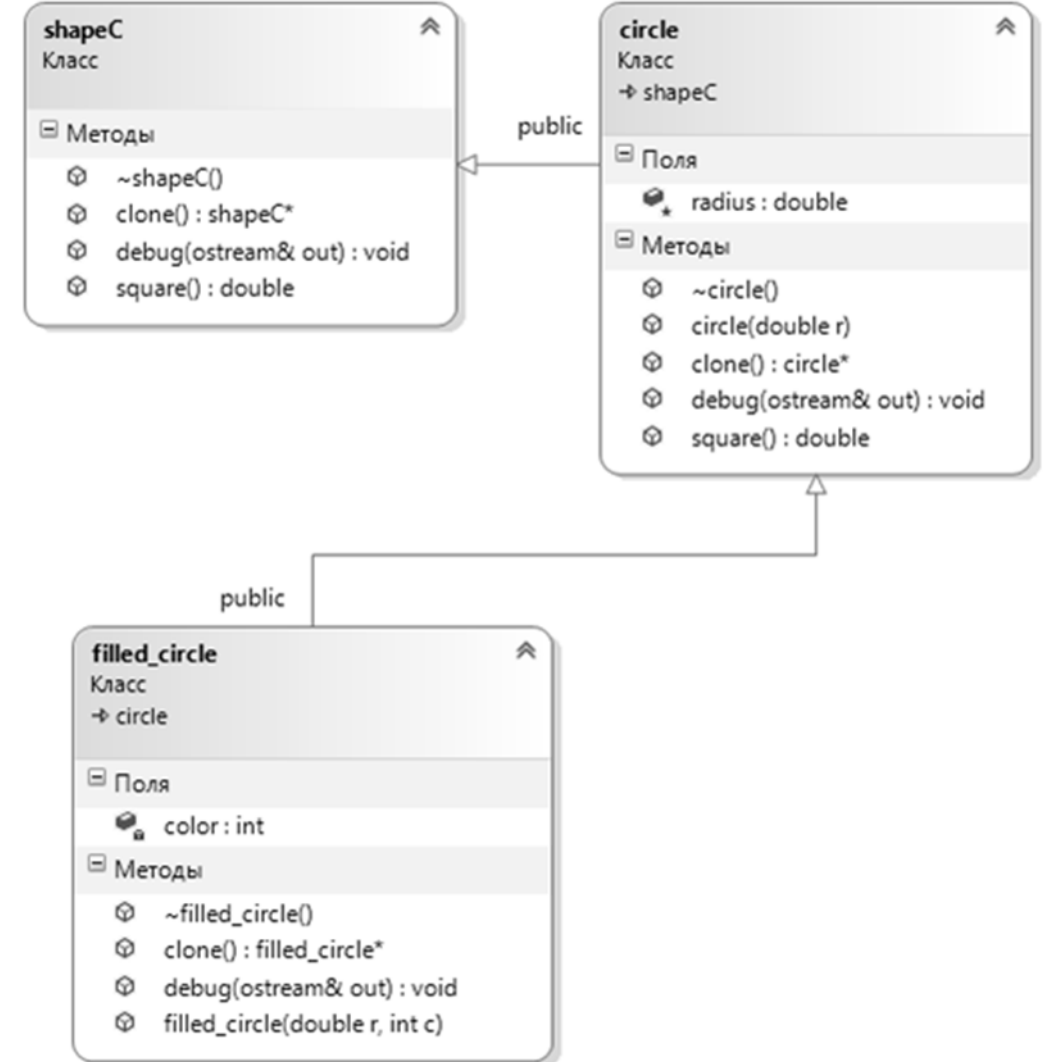
В каждом объекте такого класса появляется дополнительный указатель `vptr` на таблицу виртуальных функций.

Пример

```
class filled_circle: public circle {  
public:  
...  
void set_color(int c) {color = c;}  
...  
};
```

```
shapeC *p[4];  
filled_circle r(10, 255);  
p[1] = &r;
```

p[1]->set_color(10); // ошибка компиляции !!!



Операция `dynamic_cast`

Нужно выполнить **явное приведение типа** с помощью операции **`dynamic_cast <тип>`** для преобразования указателя на `shapeC` к указателю на `filled_circle`

```
dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

Оператор `dynamic_cast` может быть применён к указателям или ссылкам

Правила для `dynamic_cast`

Проверка корректности приведения типов для классов с виртуальными функциями производится во время выполнения программы.

Если в классе имеются виртуальные методы, то на этапе выполнения операция `dynamic_cast` пытается выполнить преобразование к указанному типу данных.

В случае преобразования указателя к типу данных, который не является фактическим типом объекта, в результате будет получен **нулевой указатель**.

При работе **со ссылками**, если преобразование невозможно, будет сгенерировано исключение **`std::bad_cast`**. Для его обработки операцию `dynamic_cast` следует поместить в блок `try/catch`.

Если **виртуальных** методов в классе и его предках **нет**, то `dynamic_cast` работать не будет. Нужно использовать `static_cast`

Обработчики событий

```
try{  
    ...  
}  
  
catch(bad_cast &e){  
    ...  
}  
  
catch(my_ex &e){  
    ...  
}  
  
catch(...){  
    ...  
}
```

В последовательности блоков catch вначале должны обрабатываться более специфичные исключения.

RTTI (Runtime Type Identification)

Операция `dynamic_cast` использует механизм динамической идентификации типа данных RTTI

RTTI основана на способности системы сообщать о динамическом типе объекта и предоставлять информацию об этом типе во время выполнения (в отличие от времени компиляции)

RTTI доступен только для классов, которые являются полиморфными, т.е. у них есть хотя бы одна виртуальная функция

`dynamic_cast` позволяет идентифицировать динамический тип переменной во время выполнения

Операция typeid() и структура type_info

typeid() используется для определения типа переменной во время выполнения.

возвращает ссылку на объект класса std::type_info, который содержит поля, позволяющие получить информацию о типе.

Этот класс содержит перегруженные операции == и !=, а также функцию name().

Проверки идентичны по результатам

Первый вариант

```
filled_circle *q = dynamic_cast<filled_circle *>(p[1]);  
if (q!=nullptr)  
    q ->set_color(10);
```

Второй вариант

```
#include <typeinfo> //требуется для использования typeid()  
...  
if (typeid(*p[1]).name() == typeid(filled_circle).name())  
    dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

Результат вызова

Результат вызова функции `typeid(*p[1]).name()` — строка «class **filled_circle**»,

Результат вызова `typeid(p[1]).name()` — строка «class **shapeC ***»

Отношение подобия

C++ рассматривает открытое наследование как отношение типа «является».

В рассмотренных примерах класс Student открыто наследует классу Person.

При этом в случае необходимости, компилятор неявно преобразует указатель или ссылку на объект класса Student в указатель или ссылку на объект класса Person

Используя закрытое наследование, можно реализовать отношение «подобен», или «as-a»

Закрытое наследование — ограничения

В противоположность открытому наследованию компиляторы в случае закрытого наследования не преобразуют указатели или ссылки на объекты производного класса в указатели или ссылки на объекты базового класса.

Кроме того, члены, наследуемые от закрытого базового класса, становятся закрытыми, даже если в базовом классе они были объявлены как защищенные или открытые.

Поэтому для объектов наследника мы не можем вызывать функции предка.

Закрытое наследование — смысл

Закрытое наследование означает «реализовано посредством...»

Делая класс Derived закрытым наследником класса Base, мы заинтересованы в **использовании уже написанного для Base кода**

Закрытое наследование означает наследование одной **только реализации, без интерфейса**.

Закрытое наследование ничего не означает в ходе проектирования программного обеспечения и **обретает смысл только на этапе реализации**.

Альтернатива композиции

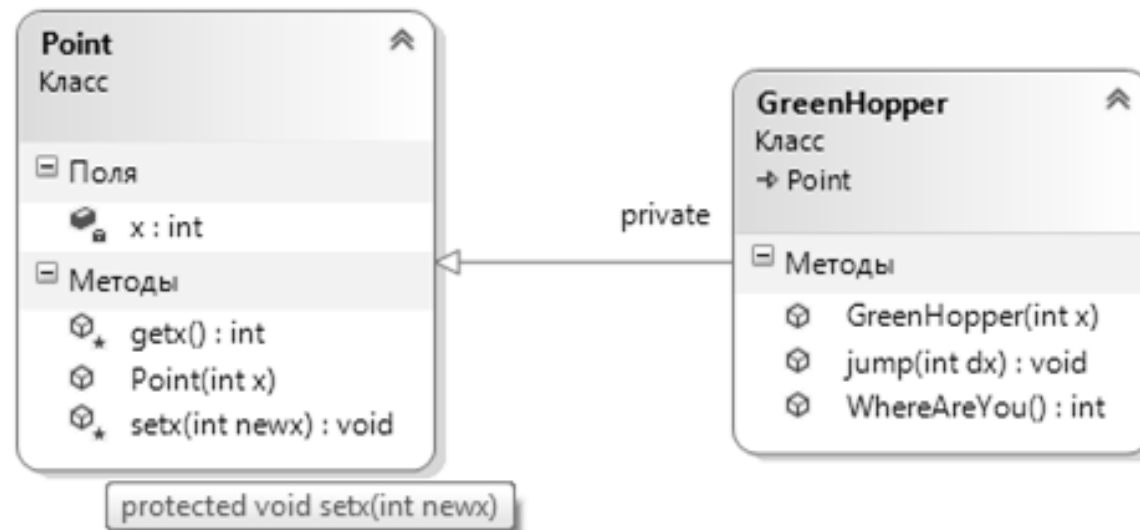
Закрытое наследование — это исключительно прием реализации, который является альтернативой включению (композиции).

Например, класс `Derived` может не наследовать, а содержать объект класса `Base`.

Пример. Использовать класс Point при реализации класса GreenHopper «исполнитель Кузнечик»

Кузнечик может перемещаться по числовой оси с помощью команды **jump(N)**, где N — любое целое число. Кроме того, он может сообщать о своём положении на числовой оси с помощью команды **WhereAreYou()**. Кузнечик выполнил программу из 50 пар команд: `jump(5)`, `jump(-3)`. На какую одну команду можно заменить эту программу, чтобы Кузнечик оказался в той же точке, что и после выполнения программы.

```
class Point {
private:
    int x;
public:
    Point(int x = 0) : x(x){}
protected:
    void setx(int newx) {
        x = newx;
    }
    int getx() {
        return x;
    }
};
```



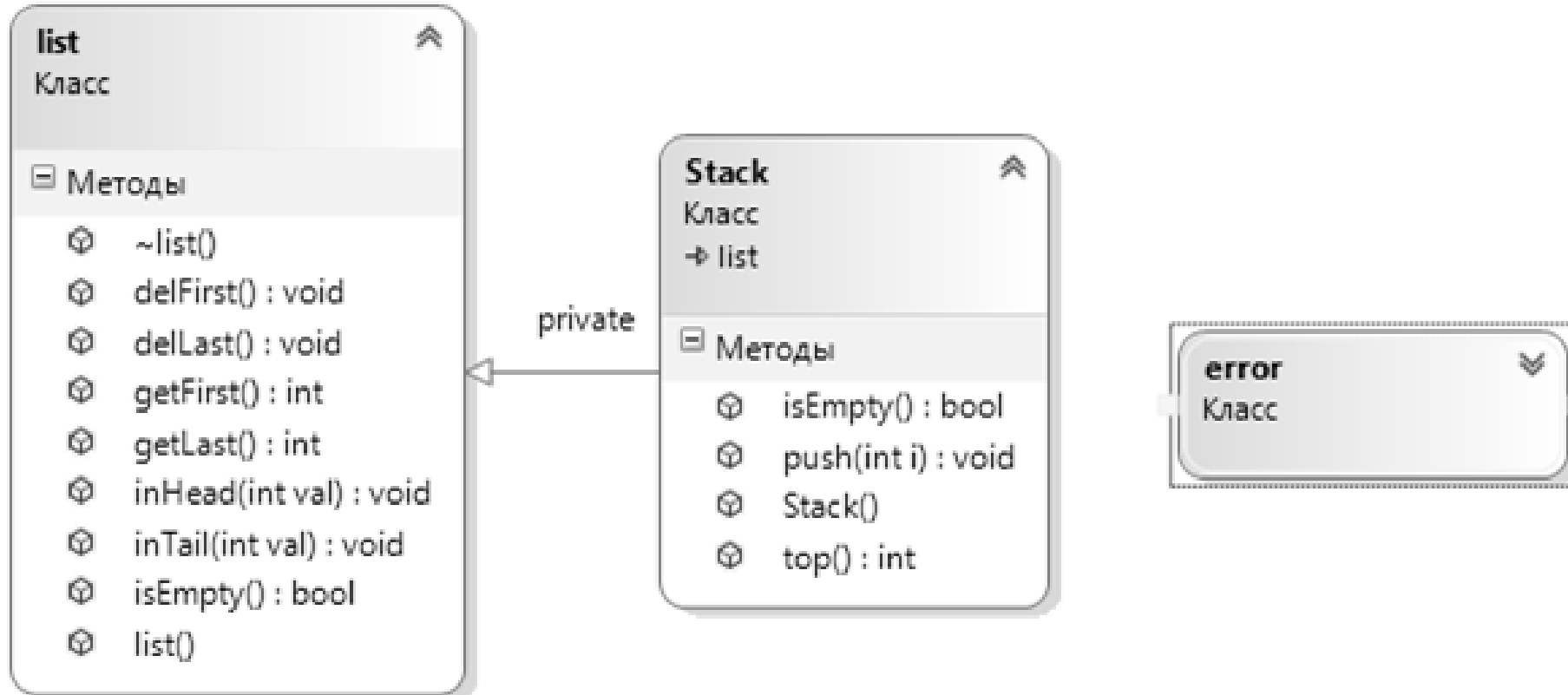
Использование открытого наследования невозможно вследствие требования, что кузнечик должен понимать только две команды `jump(N)` и `WhereAreYou()`. Использование включения не позволяет обращаться к функциям `setx(N)` и `getx()`, поскольку они объявлены как `protected`.

Решение задачи

```
class GreenHopper : private Point {
public:
    GreenHopper(int x=0) :Point(x) {}
    void jump(int dx) {
        setx(getx()+dx);
    }
    int WhereAreYou() {
        return getx();
    }
};
```

```
int main() {
    GreenHopper G;
    //cout<<G.getx();// ошибка доступа
    int t = G.WhereAreYou();
    cout << t << endl;
    for (int i = 0; i < 50; ++i) {
        G.jump(5);
        G.jump(-3);
    }
    cout << "jump(" << G.WhereAreYou() - t << ")"<<endl;
    return 0;
}
```


Пример. Используя готовую реализацию класса «список целых чисел», создать класс «стек целых чисел»



class list

```
class error{  
};
```

```
class list{  
private:  
    // внутренняя организация класса может быть разной  
public:  
    list();  
    ~list();  
    bool isEmpty() const;  
    void inHead(int val);  
    void inTail(int val);  
    int getFirst()const throw (error);  
    int getLast()const throw (error);  
    void delFirst()throw (error);  
    void delLast()throw (error);  
    //в полной реализации могут быть еще функции  
};
```

class Stack

```
class Stack: private list{
public:
    Stack(): list(){}
    bool isEmpty () {
        return list::isEmpty();
    }
    void push (int i) {
        inHead(i);
    }
    int top () const throw (error){
        return getFirst();
    }
    int pop () throw (error) {
        int res=getFirst();
        delFirst();
        return res;
    }
};
```

```
int main() {
    Stack p;
    for (int i=0;i<10;i++)
        p.push(i);
    while (!p.isEmpty())
        cout<<p.pop()<<' ';
    return 0;
}
```

Принципы SOLID

- SOLID (single responsibility, open–closed, Liskov substitution, interface segregation и dependency inversion) в программировании — мнемонический акроним, введённый Майклом Фэзерсом (Michael Feathers) для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали 5 основных принципов объектно-ориентированного программирования и проектирования.
- Это стандарт программирования, который все разработчики должны хорошо понимать, чтобы избежать создания плохой архитектуры.
- Этот стандарт широко используется в ООП.
- Если применять его правильно, он делает **код более расширяемым, логичным и читабельным**

Принципы SOLID

S	Принцип <u>единственной ответственности</u> (single responsibility principle). Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.
O	Принцип <u>открытости/закрытости</u> (open-closed principle)«программные сущности ... должны быть открыты для расширения, но закрыты для модификации».
L	Принцип <u>подстановки Лисков</u> (Liskov substitution principle)«объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы».
I	Принцип <u>разделения интерфейса</u> (interface segregation principle)«много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения».
D	Принцип <u>инверсии зависимостей</u> (dependency inversion principle)«Зависимость на Абстракциях. Нет зависимости на что-то конкретное».

Шаблоны

- **Шаблоны классов**, так же как и шаблоны функций, являются трафаретами, по которым компилятор создает **шаблонные классы** и **шаблонные методы**
- Шаблоны классов часто называют **параметризованными типами**, так как они имеют один или большее количество параметров типа, определяющих настройку шаблона класса на специфический тип данных при создании объекта класса

«**Stack** для данных типа **int**»

«Stack для данных типа **double**»

«Stack для данных типа **Time**»

Шаблоны классов

```
template< class T >  
class A {  
...  
public:  
    A();  
    void f( T data );  
    void g( void );  
};
```

```
template< class T >  
void A<T>::f( T data ) {  
    ...  
}  
  
template< class T >  
void A<T>::g( void ) {  
    ...  
}
```

Шаблон класса Stack с реализацией на основе массива

```
template <typename T>
class Stack {
private:
    T * start;          //указатель на стек
    int head;          //положение вершины стека
    int size; //размер стека
public:
    //конструктор с параметром по умолчанию для размера стека
    Stack(int = 10);
    ~Stack() { delete[] start; } //деструктор
    void push(const T&); //добавление элемента в стек
    T pop(); //извлечение элемента из стека
    T top(); //просмотр элемента на вершине стека
    bool isEmpty() const { //true, если стек пустой
        return head == -1;
    }
    bool isFull() const { //true, если стек полон
        return head == size - 1;
    }
    class StackERR {...};
};
```

```
class stackEmpty {
};

class stackFull {
    ...
};
```



```
//конструктор
template <typename T>
Stack<T>::Stack(int n) {
    //определение «разумного» размера стека
    size = n>0 && n<1000 ? n : 10;
    start = new T[size];
    head = -1;
}

// добавление объекта в стек
//в случае переполнения стека выбрасывается исключение
template <typename T>
void Stack<T>::push(const T& x) {
    if (isFull()) throw stackFull();
    start[++head] = x;
}
```

```
//извлечение элемента из стека
template <typename T>
T Stack<T>::pop() {
    if (isEmpty()) throw stackEmpty();
    T x = start[head--];
    return x;
}

// просмотр элемента на вершине стека
template <typename T>
T Stack<T>::top() {
    if (isEmpty()) throw stackEmpty();
    T x = start[head];
    return x;
}
```

```
int main() {  
    Stack<int> intStack(5);  
    try{  
        for (int i=0; i<5; i++) intStack.push(10-i);  
        while (!intStack.isEmpty())      cout<<intStack.pop()<<' '  
        cout<<endl;  
        cout<<intStack.top(); // для проверки исключения  
    }  
    catch(stackFull e) {  
        cout<<"ERRROR="<<e.getCode()<<endl;  
    }  
    catch(Stack<int>::StackERR e) {  
        cout<<"ERRROR="<<e.getCode()<<endl;  
    }  
    return 0;  
}
```

Советы

Хорошей идеей является написать и протестировать конкретный класс, а затем преобразовать его в шаблон. Таким образом, можно решить многие проблемы проектирования и обнаружить большую часть ошибок кода в тексте конкретного класса.

Аналогично шаблонам функций, все шаблоны классов и структур должны быть помещены **в заголовочные файлы**.

Для шаблонов классов в отличие от просто классов **выносить реализации** их членов-функций в отдельный *.cpp файл **нельзя**.

Это связано с тем, что **шаблоны в языке С++ не компилируются**, потому что шаблон представляет собой не фрагмент программного кода, а лишь **инструкции для построения кода**.

Инстанцирование шаблона

Для шаблонов классов и структур код генерируется в момент определения объекта.

Причём генерируются только те члены-функции класса, которые используются.

Подстановка конкретного типа в шаблон приводит к созданию шаблонного класса, т.е. к инстанцированию шаблона (template instantiation).

Аналогично функция инстанцируется (генерируется, конкретизируется) из шаблона функции и аргумента шаблона.

Процесс порождения функции или класса из шаблона называется инстанцированием

Количество инстанций зависит от количества используемых типов.

Специализация шаблона

Версия шаблона для конкретного аргумента шаблона называется специализацией.

Только специализации шаблонов содержат настоящий код.

Генерация версий шаблона для набора аргументов шаблона является задачей компилятора, а не программиста.

По умолчанию, шаблон предоставляет единственное определение, которое должно использоваться для всех аргументов шаблона.

Явная специализация шаблона

Явная специализация позволяет обеспечить **альтернативные реализации** шаблона.

Выделяют специализации, определяемые пользователями, или просто пользовательские специализации.

Например, если для специфического типа данных нужен класс, который не соответствует общему шаблону класса, можно явно определить его, отменив тем самым действие шаблона для этого типа.

Так шаблон класса `Stack` может использоваться практически для любого типа.

Однако может возникнуть необходимость создать специфический класс `Stack` для некоторого типа, например, `Specific`.

Для этого нужно просто создать новый класс с именем `Stack<Specific>`

```
Stack<Specific>{  
...//определение специфического стека  
}
```

Ошибки, связанные с использованием конкретных параметров шаблона

```
template <typename T>  
T min (T x, T y) { return x<y? x:y; }
```

```
struct A { int a; };
```

```
A obj1, obj2;  
min( obj1, obj2 ); // ошибка
```

```
friend inline bool operator< ( const A& a1, const A& a2 ) {  
    return a1.a < a2.a;  
}  
min( obj1, obj2 ); // исправлено
```

Шаблоны и обобщения

В C++ в результате компиляции шаблона получается исполняемый код инстанцированных функций, структур и классов.

В .NET в результате компиляции обобщения создается исполняемый код самого обобщения, т.е в .NET можно создать dll с обобщенным классом.

Виды параметров шаблонов

```
template< class T1, // параметр-тип  
         typename T2, // параметр-тип  
         int I, // нетиповой параметр (обычного типа)  
         T1 DefaultValue, // нетиповой параметр (обычного типа)  
         template< class > class T3, // параметр-шаблон  
         class Character = char // параметр по умолчанию
```

>

Нетиповой параметр (параметр обычных типов)

```
template < int ArrayLength, typename SomeValueType >  
class SomeClass {  
    SomeValueType SomeValue;  
    SomeValueType SomeArray[ ArrayLength ];  
    ...  
};
```

```
SomeClass < 20, int > SomeVariable;  
SomeClass < 30, double > SomeVariable2;
```

Шаблон для класса StackParam с нетиповым параметром, аналога шаблона класса Stack

```
template <typename T, int elements>
class StackParam {
private:
    T start [elements]; // массив для размещения данных стека
    int head;           // положение вершины стека
    int size; // размер стека
public:
    StackParam ():size(elements),head(-1){} //конструктор
    ~ StackParam () {} //деструктор
    void push(const T&); //добавление элемента в стек
    T pop(); //извлечение элемента из стека
    T top(); //просмотр элемента на вершине стека
    bool isEmpty() const { //true, если стек пустой
        return head == -1;
    }
    bool isFull() const { //true, если стек полон
        return head == size - 1;
    }
};
```

Изменения в определении шаблонов членов-функций класса

```
// добавление объекта в стек
//в случае успеха возвращается true, в противном случае false
template <typename T, int elements>
void StackParam<T,elements>::push(const T& a) {
    if (isFull()) throw stackFull();
    start[++head] = a; //элемент помещается в стек
}
```

```
//выталкивание элемента из стека
template <typename T, int elements>
T StackParam<T, elements>::pop() {
    if (isEmpty()) throw stackEmpty();
    T a = start[head--]; //элемент выталкивается из стека
    return a;
}
```

...

Параметры-шаблоны

```
template< class Type, template< class > class Container >
class CrossReferences {
    Container< Type > mems;
    Container< Type* > refs;
    /* ... */
};
```

```
CrossReferences< Date, vector > cr1;
CrossReferences< string, set > cr2;
```