

# Оглавление

[1. Разработка простейшего класса](#)

[2. Программирование с шаблонами функций](#)

[3. Программирование шаблона класса](#)

# 1. Разработка простейшего класса

Классы являются основой объектно-ориентированного программирования на C++. Это инструменты программистов для создания **новых типов данных**, более соответствующих поведению реальных объектов, чем типы, реализуемые в программах с функциями. Одно из важных понятий, составляющих основу философии объектов C++ , является то, что классы — **это типы**. Определение классов в программе расширяет систему встроенных типов C++.

«Цели программирования классов в C++:

§ связывание данных и операций в один синтаксический блок и обозначение общей принадлежности этих элементов программы.

§ устранение конфликтов имен, благодаря чему данные и функции в разных классах могут без проблем использовать одни и те же имена.

§ позволить разработчику класса (программа-сервер) управлять доступом к элементам класса извне (из программы клиента).

§ поддержка инкапсуляции, сокрытия информации, переноса обязанностей с программы-клиента на программу-сервер, создание отдельных областей ответственности, устранение необходимости координации действий между программистами, работающими над разными частями программы» [3].

☉ Применение классов как синтаксической конструкции без понимания описанных выше целей, не улучшает качество программного кода. При разработке программ нужно стремиться к этим четырем целям, если включаете классы в программу.

Большая программа на C++ получит преимущества от применения классов, если эти классы используются с вышеописанными целями и корректно. Правильно сконструированная программа C++ представляет собой комбинацию компонентов (модулей), совместно выполняющих общую задачу, но при этом независимых и отдельно сопровождаемых.

Члены класса можно разделить на информационные члены (*поля*) и функции-члены (*методы*) класса. **Поля** описывают внутреннюю структуру информации, хранящейся в объекте, который создается на основе класса. **Методы** класса описывают алгоритмы обработки этой информации.

Данные, хранящиеся в полях, описывают *состояние объекта*, созданного на основе класса. Состояние объекта изменяется с помощью методов класса.

Алгоритмы, заложенные в реализации методов класса, определяют *поведение объекта*, то есть реагирование объекта на поступающие внешние воздействия в виде входных данных.

*Управление доступом к членам класса.* Принцип инкапсуляции обеспечивается определением в классе областей доступа:

– **private**, закрытый, доступный только собственным методам;

– **public**, открытый, доступный любым функциям, интерфейс класса;

– **protected**, защищенный, доступный только собственным методам и методам производных классов (см. тему Наследование).

Члены класса, находящиеся в закрытой области (private), недоступны для использования со стороны внешнего кода. Напротив, члены класса, находящиеся в открытой секции (public), доступны для использования со стороны внешнего кода. При описании класса каждый член класса помещается в одну из перечисленных выше областей доступа.

Синтаксис описания простейшего класса

```
class <имя класса> {

    private:

        //определение закрытых членов класса

    public:

        //определение открытых членов класса

    protected:

        //определение защищенных членов класса

};
```

Порядок следования областей доступа и их количество в классе – произвольны. Служебное слово, определяющее первую область доступа, может отсутствовать. По умолчанию эта область считается private.

В закрытую (private) область обычно помещаются поля, а в открытую (public) область – методы класса, реализующие интерфейс объектов класса с внешней средой. Если какой-либо метод имеет вспомогательное значение для других методов класса, являясь подпрограммой для них, то его также следует поместить в закрытую область. Это обеспечивает логическую целостность информации.

После описания класса его имя можно использовать для описания **объектов этого типа**. В объектно-ориентированном анализе и проектировании термин **"объект"** часто применяется к набору потенциальных экземпляров с одними и теми же свойствами. Термины "переменные", "экземпляры", "экземпляры класса", "объекты класса" и "экземпляры объектов" будем использовать здесь как синонимы. Все они обозначают программную сущность, для которой на какой-то период времени при выполнении программы выделяется память (в динамической или статической области). Они принадлежат к конкретному классу памяти и подчиняются правилам областей действия имен программы.

Доступ к полям и методам объекта, описанным в открытой секции (public), осуществляется через объект или ссылку на объект с помощью операции выбора члена класса `'.'`. Если работа с объектом выполняется с помощью указателя на объект, то доступ к соответствующим членам класса осуществляется на основе указателя на член класса `'->'`:

```
//объявление объекта

<имя класса> <имя объекта>;

<имя класса> *<имя указателя объекта>;

...

//доступ к методу объекту

<имя объекта>.<имя метода>;

<имя указателя объекта> -> <имя метода>;
```

Пример 1

```
#include <iostream>

class Circle {
private:
    double x, y;
    double r;
public:
    //конструктор по умолчанию
    Circle() { x = 0; y = 0; r = 1; }
    //конструктор с параметрами
    Circle(double Vx, double Vy, double Vr) { x = Vx; y = Vy; r = Vr; }
    // метод - выводит на экран параметры окружности
    double getRadius() { return r; }
    // метод -устанавливает значение полей класса
    void setCircle(double Vx, double Vy, double Vr) { x = Vx; y = Vy; r = Vr; };
    // метод - перемещает центр, выполняя перемещение окружности
    void moveCircle(double dx, double dy) { x = x + dx; y = y + dy; }
    //метод - масштабирует, преобразование подобия с коэффициентом k
    void zoomCircle(double k) { r = r * k; }
    // деструктор - пустое тело функции, так как в составе класса нет
    // динамических полей
    ~Circle() {};
};

//...
Circle disk1, * disk2; // объекты класса
// объект, где автоматически вызван конструктор с параметрами
Circle disk3(-2.5, 4, 7.5);
// объект, где автоматически вызван конструктор по умолчанию
Circle* disk4 = new Circle;
```

На Рис.1 показан доступ к программе-серверу класса Circle программы-клиента main(). Класс имеет следующий состав: данные, функции и границу класса, отделяющего класс от того, что находится вне его. Поля-данные доступны только внутри класса, реализация методов находится внутри класса, а интерфейсы, известные клиенту, — снаружи. Если клиенту необходимы значения полей класса Circle (для масштабирования, вывода или присваивания значений полям), он использует методы (функции-члены) zoomCircle(), getRadius(), setCircle() и не обращается к полям напрямую, так как прямой доступ к данным запрещен (private).

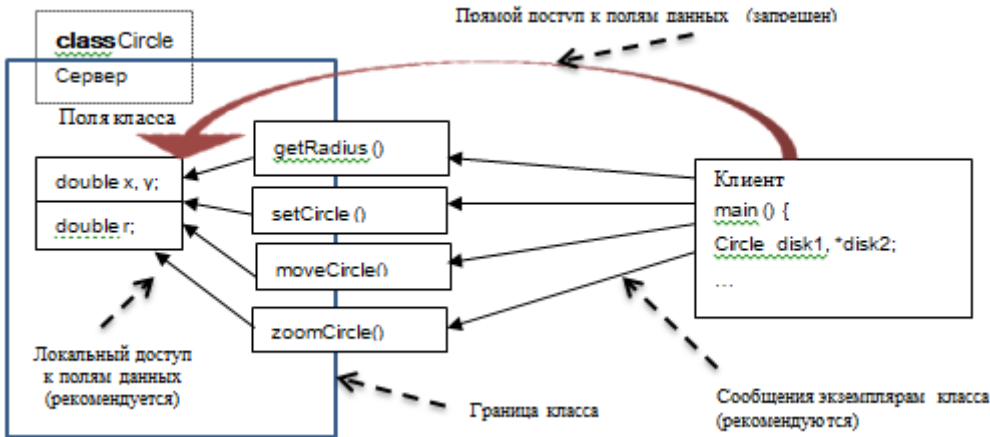


Рис.1 Разделение доступа к членам класса Circle из клиента main().

Неявная, автоматическая инициализация объектов класса осуществляется **конструктором класса**. Конструктор — это функция-член класса, которая имеет специальный синтаксис. Конструктор класса не может иметь произвольное имя, имя должно совпадать с именем класса. Заголовок конструктора не может описывать возвращаемый тип (даже void) и возвращать значения. В составе класса может быть несколько конструкторов.

Конструктор класса нельзя вызывать явно, как обычный метод. Конструктор вызывается только при создании объекта. Компилятор автоматически вызывает конструктор сразу после создания экземпляра объекта, и поэтому конструктор должен быть описан в разделе public класса.

Освобождения памяти от объекта производит **деструктор** класса, тем самым уничтожая объект класса. Деструктор вызывается непосредственно перед уничтожением объекта. Если деструктор в классе не определен программистом, то компилятор автоматически сформирует его по умолчанию и вызовет (как и конструктор по умолчанию, он ничего не делает). **Деструктор**, определяемый программистом, является методом класса, он единственный в классе. Синтаксис деструктора похож на синтаксис конструктора, но более строгий. Деструктор имеет то же имя, что и имя класса, но ему предшествует тильда (~), например ~Circle(). Деструкторы в отличие от конструкторов не могут иметь параметров. В теле метода не может присутствовать оператор return и поэтому описывать возвращаемый тип в его заголовке недопустимо.

Программист определяет в классе деструктор в том случае, если в составе полей есть динамические переменные. Эти поля нужно явно уничтожать, так как автоматически стек памяти не освобождается от них с разрушением объекта класса.

Конструктор вызывается после создания объекта, а деструктор — перед его уничтожением. Конструкторы инициализируют поля объекта после того, как они уже созданы, и распределены дополнительные ресурсы, например, динамическая память. Деструкторы лишь освобождают ресурсы, выделенные объектам во время их существования, например динамическую память, выделенную в конструкторах и других функциях.

В составе объектов C++ компилятором автоматически создается неявный указатель **this** как специальный указатель на текущий объект данного класса. Каждый метод класса получает данный указатель в качестве **неявного параметра**. Методы класса через него получают доступ к другим членам класса. Указатель **this** - это неявная локальная константа в составе объекта, имеющая тип указателя на класс, например Circle\*. Нет необходимости использовать его явно, он используется явно только в том случае, когда выходным значением для метода является текущий объект.

**Статические члены класса** - поля класса, которые представлены в единственном экземпляре для всех объектов данного типа. Они не являются частью объектов этого класса и размещаются в статической памяти. Для их описания используется служебное слово **static**. Для всех объектов, созданных на основе класса, содержащего статический член, существует только одна копия этого члена.

Примером такого статического члена является счетчик числа созданных объектов данного класса. Такой счетчик может существовать только в отрыве от всех экземпляров объектов данного класса, в то же время работать с ним обычно приходится одновременно с вызовом обычных методов, например, конструкторов объектов, поэтому описывать счетчик удобнее именно как элемент класса.

Статическими могут быть не только информационные члены класса, но и его методы. Статический метод не может использовать никакие нестатические члены класса, так как, являясь частью класса, а не объекта, он не имеет неявного параметра **this**. Поскольку статический метод не использует специфического содержимого конкретного объекта, то обращение к нему может осуществляться не только с использованием идентификатора объекта, но и с использованием идентификатора класса:

```
<имя класса>:: <имя_статической_функции> (фактические_параметры);
```

**Перегрузка операций в C++** - это описание функции, задающее еще одну интерпретацию стандартной операции.

Для перегрузки операций используется ключевое слово **operator**.

Синтаксис перегруженной операции:

```
<тип возвращаемого значения> operator <символ операции> (операнды)
{ ...};
```

Правило описания перегрузки операций: сохраняется количество аргументов, приоритеты и правила ассоциации (слева направо или справа налево) как для стандартных операций. В бинарных операциях вызвавший операцию объект считается первым операндом.

## 2. Программирование с шаблонами функций

Многие алгоритмы не зависят от типов данных, с которыми они работают, классический пример – сортировка. Передача типа данных как параметра функции называется *параметрическим полиморфизмом*. В C++ средством параметризации являются шаблоны функций и шаблоны классов.

С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя.

Для описания шаблонов используется ключевое слово **template**, вслед за которым указываются аргументы (формальные параметры шаблона), заключенные в угловые скобки. Формальные параметры шаблона перечисляются через запятую, и могут быть как именами объектов, так и параметрическими именами типов (встроенных или пользовательских). Параметр-тип описывается с помощью служебного слова **class** или служебного слова **typename**.

Синтаксис описания шаблоны функции:

**template** < **typename** | **class** тип параметра >

*заголовок функции* { /\* тело функции \*/ },

где тип параметра – это идентификатор типа.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только базовым типом, но и пользовательским типом, например:

**template** < **class** A, **class** B, **int** i > void f( ) { ... }

При обращении к функции-шаблону после имени функции в угловых скобках указываются фактические параметры шаблона - имена реальных типов или значения объектов:

*имя функции* < фактические параметры типов > (*фактические параметры*);

Пример 1. Функция, сортирующая методом выбора массив *b* из *n* элементов любого типа (параметр *Type*), в виде шаблона описана так:

```
template <class Type>
void sort(Type* b, int n) {
    Type temp;
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++) if (b[j] < b[i])
        {
            temp = b[i]; b[i] = b[j]; b[j] = temp;
        }
}
```

Программа-клиент, вызывающая эту функцию-шаблон, будет иметь вид:

```
int main()
{
    const int n = 7;
    int i, x[n];
    for (i = 0; i < n; i++) std::cin >> x[i];

    // Сортировка целочисленного массива
    sort<int>(x, n);

    for (i = 0; i < n; i++) std::cout << x[i] << '\t';
    std::cout << std::endl;

    double a[] = { 0.22, 117, -0.08, 0.21, 42.5 };

    // Сортировка массива вещественных чисел
    sort<double>(a, 5);

    for (i = 0; i < n; i++) std::cout << a[i] << '\t';
}
```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется **инстанцированием** шаблона (instantiation), а результат является **порожденной функцией**. Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

☺ Использование шаблонов сокращает текст программы, но не сокращает программный код. В программе реально будет сгенерировано столько порожденных функций, сколько имеется вызовов функций с разными наборами фактических параметров шаблона.

Типы параметров при вызове функции-шаблона могут быть заданы явно и неявно. При вызове с неявными параметрами компилятор будет определять тип по фактическим параметрам функции. Это называется **выведением типов аргументов шаблона**. Выведение типов аргументов возможно при условии, что список фактических параметров вызова функции однозначно идентифицирует список параметров шаблона.

Пример 2. Рассмотрим пример неявного задания параметров шаблона при вызове:

```
template <class X, class Y, class Z> void fun(Y, Z);

void main() {

    fun<int, char*, double> ("Work_2", 7.0); // типы заданы явно

    fun<int, char*>(" Work_2", 7.0);        // Z определяется как double

    fun<int>(" Work_2", 7.0);               // Y как char*, а Z - как double

    fun(" Work_2", 7.0);                    // ошибка: X определить невозможно
}
```

Так как допускается параметризовать шаблоны не только именами типов, но и объектами, то аргумент *n* можно указать в виде параметра шаблона:

```
template <class Type, int n>

void sort(Type *b);
```

Тогда вызов `sort` будет выглядеть соответственно:

```
sort<double, 5>(a);
```

Если шаблонный алгоритм является неудовлетворительным для конкретного типа аргументов или неприменим к ним, то можно описать обычную функцию, список типов аргументов и возвращаемого значения которой соответствуют объявлению шаблона. Такая, перегружающая шаблон, функция, называется **специализацией шаблонной функции**.

С одним и тем же именем функции можно написать несколько шаблонов и перегруженных обычных функций. Алгоритм выбора перегруженной функции с учетом шаблонов является обобщением правил выбора перегруженной функции.

1. Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.
2. Если могут быть два шаблона функции и один из них более специализирован, то на следующих этапах рассматривается только он (порядок специализаций описан далее).
3. Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона функции были определены путем вывода по типам фактических параметров вызова функции, то при дальнейшем поиске

оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **Точного** отождествления.

4. Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.
5. Так же, как и при поиске оптимально отождествляемой функции для обычных функций, если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Использование шаблонов позволяет создавать универсальные алгоритмы, без привязки к конкретным типам.

При разработке унифицированных функций-шаблонов возникает проблема логических условных вычислений для проверки, поиска и др. Это проблема разрешается путем применения функций-предикатов, как аргументов функций-шаблонов. **Функция-предикат** - это функция, где возвращаемым результатом будет параметр логического типа `bool`.

Пример 3. Вычислить произведение положительных элементов массива заданного типа.

```
template <typename T, int size >
T ProductOfArray(T* mas, bool(*predicate) (T))
{
    T p = 1;
    for (int i = 0; i < size; i++)
        if ((*predicate)(mas[i])) p *= mas[i];
    return p;
}

bool polInt(int k) { return k > 0; }    // фактическая функция- предикат
bool polFloat(float k) { return k > 0; }

int main(int argc, char* argv[])
{
    int M[5] = { -1,2,4,-7,0 };
    float X[7] = { 2.3, -5.4, 7.2, 0, -11.3, 4.1, -3.3 };

    cout << ProductOfArray<int, 5>(M, polInt);    //применение предиката
    cout << ProductOfArray<float, 7>(X, polFloat);
    return 0;
}
```

Использование указателей функций как аргументов в описании параметров функции делает код очень гибким в применении. Это не обязательно предикатные функции, это могут быть любые вычисления. В Примере 3 можно оператор вычисления произведения заменить на любое другое вычисление (сумма, количество, выражение), определив этот оператор как функцию.



### 3. Программирование шаблона класса

## Синтаксис определения шаблонного класса

Язык C++ поддерживает метод повторного использования структуры класса. Этот инструмент называется шаблонный класс. Вместо класса с фиксированным типом компонентов, создается класс, где тип компонентов интерпретируется как параметр класса.

**Шаблоны классов** предоставляют возможность аналогичную возможностям шаблонов функций, позволяя создавать параметризованные классы. Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра.

Синтаксис описания шаблона:

```
template <class имя_параметра, ... >

    class имя {

        private:

            ...

        public:

            ...

    };

```

Каждый параметр шаблона в угловых скобках представляет собой "заполнитель" для типа. Параметризованный класс (родовой класс) может иметь любое количество параметров типа. Несколько шаблонных параметров в списке разделяются запятыми.

```
template <class T1, class T2, class T3> //три параметра типа

    class Triple;                // объявление класса

```

Как и в других классах C++ , зарезервированное слово **class** в списке параметров имеет иное значение, чем в других контекстах. Оно показывает, что идентификатор, расположенный за ним, является "заполнителем" фактического типа. Этот тип не обязательно должен быть классом. Он также может быть любым встроенным типом. Кроме того, допускается использование конкретных типов параметров выражений.

```
template <class Type, int size>  // тип и значение

    class Array;

```

Это подобно параметрам функций — в момент реализации класса Array значение указанного типа (в данном случае int) должно предоставляться клиентской программой.

В качестве параметров могут использоваться типы, шаблоны, переменные. Область действия параметра шаблона – от точки описания до конца шаблона, поэтому параметр можно использовать для описания следующих за ним, например:

```
template <class T, T* p, class U = T>

    class X {

        ...

    };

```

В данном примере типу *U* по умолчанию определен тип *T*, а указатель имеет ранее определенный тип *T*.

Пример 1. Шаблон класса «стек». Тип параметра обозначается именем Type, определенным программистом.



```

4  #include <iostream>
5  #include <string>
6  #include <math.h>
7  using namespace std;
8
9  template <class Type>
10 class Stack
11 {
12 private:
13     Type* items;
14     int size;
15     int top;
16 public:
17     Stack(int sz);           //конструктор
18     Stack();                 // конструктор по умолчанию
19     Stack(const Stack<Type>& L); //конструктор копирования
20     ~Stack();                // деструктор
21 // основные операции
22     void push(Type element);
23     Type pop();
24     Type peek();
25     int Top() { return top + 1; };
26     bool isEmpty();
27     // перегрузка операции сложения - конкатенации двух стеков
28     Stack<Type>& operator + (Stack<Type>& L);
29     // перегрузка операции >
30     bool operator > (Stack<Type>& L);
31     Type* info() { return items; };
32 };
33

```

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, то его заголовок – это заголовок шаблона-функции.

```

//----- конструктор -----
template <class Type>
Stack<Type>::Stack(int sz)
{
    size = sz;
    top = -1;
    items = new Type[size];
}
//----- конструктор копирования (операция присваивания =)-----
template <class Type>
Stack<Type>::Stack(const Stack<Type>& L)
{
    items = new Type[L.size];
    size = L.size;
    top = L.top;
    for (int i = 0; i <= top; i++)
        items[i] = L.items[i];
}
//----- конструктор по умолчанию -----
template <class Type>
Stack<Type>::Stack()
{
    size = 100;
    top = -1;
    items = new Type[size];
}
//-----деструктор-----
template <class Type>
Stack<Type>::~~Stack()
{
    delete[] items;
}
//-----вывод вершины стека-----
template <class Type>
Type Stack<Type>::peek()
{
    return items[top];
}

```

```

//-----вывод вершины стека-----
template <class Type>
Type Stack<Type>::peek()
{
    return items[top];
}
//-----вставка значение в стек-----
template <class Type>
void Stack<Type>::push(Type element)
{
    if (top == size - 1) return;
    items[++top] = element;
}
//-----удаление значения из стека-----
template <class Type>
Type Stack<Type>::pop()
{
    if (!isEmpty()) return items[top--];
}
//----метод - предикат, проверка стека на пустоту-----

template <class Type>
bool Stack<Type>::isEmpty()
{
    return (top == -1);
}
//---- перегруженная операция сложения двух стеков-----
template <class Type>
Stack<Type>& Stack<Type>::operator + (Stack<Type>& L)
{
    int i = 0;
    Stack<Type>* S = new Stack<Type>(size + L.size);

    for (; i <= top; i++)
        S->push(items[i]);

    for (i = 0; i <= L.top; i++)
        S->push(L.items[i]);

    return *S;
}
//-----перегруженная операция сравнения двух стеков-----
template <class Type>
bool Stack<Type>::operator > (Stack<Type>& L)
{
    return (top > L.top);
}

```

Создание объекта шаблонного класса называется реализацией (instantiation). Оно подобно созданию объекта любого класса в C++. Когда клиентская программа приписывает значения конкретному объекту шаблонного класса, она должна предоставить для каждого шаблонного параметра аргумент фактического типа. Шаблон Stack не является классом, это всего лишь шаблон. Он не поддерживает прямое создание объектов Stack. Объект Stack без указания фактического типа при вызове функции является ошибкой.

Фактический тип определяется при реализации шаблона как имя типа в угловых скобках, которые добавляются к имени шаблонного класса. Имя объекта задается таким же образом, как и для обычных классов. Параметры конструкторов используются там, где необходимо.

Stack<int> iS(50); // стек целых длиной 50 значений

Stack<char> cS(200); // стек символов длиной в 200 символов

Рассмотрим использование шаблона стека на разных типах данных от простых базовых типов до более сложных объектных типов.

Программа – клиент для класса стек, реализованного на типе int и double представлена ниже.

```

int main()
{
    Stack<int>* stk2; stk2 = new Stack<int>(10);
    stk2->push(56);   stk2->push(67);
    cout << stk2->peek() << endl;

    Stack<double> stk1(20);           //обрабатывает конструктор с параметрами
    stk1.push(23.5); stk1.push(-34.5);
    cout << stk1.Top() << endl;

    Stack<double> stk5 = stk1;         //обрабатывает конструктор копирования
    Stack<double> st;                  //обрабатывает конструктор по умолчанию
    stk5.push(45.1);
    st = stk1 + stk5;                  //операция сложения для стека
    st.push(56.1);
    double* m = st.info();
    for (int i = 0; i < st.Top(); i++) cout << m[i] << endl;
    cout << st.Top() << endl;

    if (stk1 > st) cout << " stk1 -размер стека больше";
    else cout << " stk1 -размер стека меньше ";
}

```

Более сложная будет программа – клиент для класса стек, реализованного на объектном типе `Point`. Описание объектного типа потребует перегрузку операции вывода, в общих случаях требуется перегрузить также операцию сравнения для поиска и разработать методы обработки полей объекта.

```
class Point {
private:
    int x, y;

    // перегрузку операции помещения в поток вывода объявляем дружественной
    // классу, она должна иметь доступ к закрытым полям класса x и y
    friend ostream& operator <<(ostream& out, const Point& p);

public:
    Point() { } // конструктор по умолчанию: пустой
    Point(const Point& p) // конструктор копирования
    {
        x = p.x; y = p.y;
    }
    void set(int a, int b) // установить координаты Point
    {
        x = a; y = b;
    }
};

// перегрузка операции помещения в поток вывода
ostream& operator << (ostream& out, const Point& p)
{
    out << "(" << p.x << "," << p.y << ")"; return out;
}

int main()
{
    Point data[5]; // объект Point
    Stack<Point> s(5); // объект Stack
    data[0].set(1, 2); data[3].set(7, 8); data[1].set(3, 4);
    data[2].set(5, 6); data[4].set(9, 0);
    int n = sizeof(data) / sizeof(Point);
    cout << "Initial data: ";
    for (int j = 0; j < n; j++) { cout << data[j] << " "; } cout << endl;
    for (int i = 0; i < n; i++) { s.push(data[i]); }
    cout << "Inversed data: ";
    while (!s.isEmpty()) cout << s.pop() << " "; cout << endl;
}
```

C++ поддерживает концепцию специализации при работе с параметрами типа, которые требуют специальной обработки. Для каждого специального класса должен предусматриваться отдельный **специализированный шаблон класса**. Синтаксис описания специализации представляет собой объединение синтаксиса для самого шаблонного класса (в списке параметров шаблона) и инициализации шаблона в клиентской программе (в списке фактических типов).

После описания шаблона класса, помещается полное описание специализированного класса, при этом требуется заново определить все его методы.

```
template <class T> // удалить класс T из скобок

class Array {...};

// добавить <char*> к имени класса

template <> // пустой список параметров шаблона

class Array<char*>{ ... }; // список фактических типов
```

В методах специализации шаблонов описывается, что следует выполнить для конкретного типа. Специализация шаблона реализуется с использованием того же синтаксиса, что и для объекта шаблонного класса. Объявление специализированного объекта шаблона:

```
Array<char*> a1;
```

Каждая версия класса или функции, создаваемая по шаблону, содержит одинаковый базовый алгоритм. При этом эффективность этого алгоритма в зависимости от типа данных может сильно меняться. Если для какого-либо типа существует более эффективный алгоритм (программный код) можно предусмотреть специальную реализацию отдельных методов, т. е. **специализировать метод**.

```
template <class Data>

void List<Data>::print() { ... };

//специализированное описание в описании класса

void List<char*>::print({ ... };
```

Если создан объект

то вызван будет метод специализированный.

List <char\*> Str;