

# Обобщенное программирование. Шаблоны

## (параметризованные типы)

Обобщённое программирование (*generic programming*) — парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

- Обобщённое программирование = параметрический полиморфизм = статический полиморфизм. В C++ поддерживается шаблонами (template).

- Полиморфизм подтипов = динамический полиморфизм. В C++ поддерживается наследованием классов.

- Специальный полиморфизм (перегрузка функций).

**Повод:** необходимость реализовать некий новый объем кода, аналогичный уже написанному, но изменив типы данных.

**Варианты решения:**

**1. Дублирование фрагментов кода (плохо!)**

**2. Средствами языка C**

**1) Макроопределения. Недостатки:**

- целесообразно только для очень простых функций;
- отсутствует контроль типов;
- трудности при отладке;
- может сильно увеличить размер исполняемой программы

```
1. #define SP(Type) \  
struct Shared_ptr { \  
    Type *p; \  
    ...  
};
```

SP(LongString); // не будет компилироваться для разных типов,  
//т.к. название структуры Shared\_ptr будет одинаково для всех  
ТИПОВ.

```
2. #define SP(Type, Name) \  
struct Name { \  
    Type *p; \  
    ...  
};
```

SP(LongString, spLongString); //Код пишется не на языке, а макросом.  
//Следовательно, компилятор не сможет проверить код.  
//Могут возникнуть неожиданные подстановки

- Данным решением стоит пользоваться во встраиваемых системах, которые поддерживают только C.

**2) Обобщённое программирование с использованием нетипизированных указателей `void*` , например, библиотечные функции сортировки `qsort()`, двоичного поиска `bsearch()`, копирования памяти `memcpy()`.**

```
void qsort(void *base, size_t num, size_t size, int (*compare) (const void *, const void *));
```

```
void * memcpy( void * destptr, const void * srcptr, size_t num );
```

Недостатки:

- отсутствие информации о типах;
- требует аккуратной работы с отдельными байтами;
- преобразование любых указателей к `void*` существенно менее наглядно

### **3. Средствами языка C++**

**1) Переопределение функций.** Делает текст программы более наглядным, но не избавляет от необходимости повторять один и тот же алгоритм в нескольких местах.

#### **2) Шаблоны**

- Позволяют отделить общий алгоритм от его реализации применительно к конкретным типам данных.
- Сочетают преимущества однократной подготовки фрагментов программы (аналогично макрокомандам) и контроль типов, присущий переопределяемым функциям.

# Шаблоны функций

Объявление шаблона функции начинается с заголовка, состоящего из ключевого слова `template`, за которым следует список параметров

```
шаблона
template <class X>
X min (X a, X b)
{
    return a < b ? a : b;
}
```

```
// X – имя типа
...
int m = min (1, 2);
```

Экземпляр шаблона функции породит код, сгенерированный компилятором  
**(инстанцирование):**

```
int min (int a, int b)
{
    return a < b ? a : b;
}
```

```
template <class T>
    T toPower (T base, int exponent){
        T result = base;
        if (exponent==0) return (T)1;
        if (exponent<0) return (T)0;
        while (--exponent) result *= base;
        return result;
    }
```

```
int i = toPower <int>(10, 3);
```

```
int i = toPower (10, 3); // T становится типом int
```

```
long l = toPower (1000L, 4); // T становится типом long
```

```
double d = toPower (1e5, 5); //T становится типом  
double
```

```
int i = toPower (1000L, 4); // ошибка компиляции:
```

```
// используется разный тип данных
```

# Требования к фактическим параметрам шаблона

1. T result = base;	class T{
2. return (T)1;	public:
3. return (T)0;	T (const T &base);
4. result *= base;	// конструктор
5. return result;	копирования
	T (int i); //приведение int к T
	operator *= (T base);
	// ... прочие методы

- Используя классы в шаблонах функций, убедитесь в том, что вы знаете, какие действия с ними выполняются в шаблоне функции, и определены ли для класса эти действия.



# Шаблоны функций с несколькими аргументами

```
// Шаблон функции  
поиска // в массиве  
template <class atype>  
int find(atype* array,  
        atype value, int size) {  
    for(int j = 0; j < size; j++)  
        if(array[j] == value)  
            return j;  
    return -1;  
}
```

```
int intArr[] = { 1, 3, 5, 7 };  
int in = 3;  
float fl = 5.0;
```

```
int value = find(intArr, in,4);
```

```
int value = find(intArr, fl, 4); //  
ошибка! Аргументы  
// шаблона должны  
быть // согласованы
```

# Отождествление типов

## аргументов

```
template <class T>
    T max (T a, T b)
    {
        return a > b ? a : b;
    }
...
int i = max (1, 2);
double d = max (1.2, 3.4);
// Однако, если
// аргументы
// различных типов, то
// вызов max()
// приведет к ошибке.
```

1. Приведение типов

```
int i = max ((int)'a', 100);
```

2. Явное объявление версии экземпляра шаблона

```
int max (int, int);
```

```
int j = max ('a', 100);
```

3. template <class T1, class T2>

```
T1 max (T1 a, T2 b)
```

```
{
```

```
    return a > (T1)b ? a : (T1)b;
```

```
}
```

```
max ('a', 100); //char max (char, int);
```

```
max (100, 'a'); //int max (int, char);
```

# Шаблоны классов

```
// класс, хранящий
// пару значений
template <class T>
class Pair
{
    T a, b;
public:
    Pair (T t1, T t2);
    T Max();
    T Min ();
    int isEqual ();
};
```

```
template <class T>
    T Pair <T>::Min()
    {
        return a < b ? a : b;
    }
```

// Если бы Pair был  
обычным классом , а  
не шаблоном:

```
T Pair::Min()
{
    return a < b ? a : b;
}
```

- Полное описание шаблона должно быть известно до его использования. Нельзя разбить объявление и реализацию на .cpp и .h файлы, **реализация должна быть известна и находиться в заголовочном файле.**

```
template <class T>
```

```
Pair <T>::Pair (T t1, T t2) :  
    a(t1), b(t2) {}
```

```
template <class T>
```

```
T Pair <T>::Max()  
    {return a>b ? a : b;}
```

```
template <class T>
```

```
int Pair <T>::isEqual(){  
    if (a==b) return 1;  
    return 0;}
```

Чтобы создать экземпляр класса Pair для некоторого классового типа, например для класса X, этот класс должен содержать следующие общедоступные функции:

```
X (X &);  
// конструктор копирования  
int operator == (X);  
int operator < (X);
```

# Параметризация числовыми параметрами

- Возможность задания числовых параметров позволяет, например, создавать объекты типов "Вектор из 20 целых", "Вектор из 1000 целых" или "Вектор из 10 переменных типа double".

```
template <class T, int n> class Vector
{
public:
    Vector();
    ~Vector() {delete[] coord;}
    void newCoord (T x);
    T Max ();
    T Min();
    int isEqual();
private:
    T *coord;
    int current;
};
```

```
//конструктор
    template <class T, int n>
    Vector <T, n>::Vector():
    {
        coord = new T[n];
        current = 0;
    }
```

```
template <class T, int n>
T Vector <T, n>::Max():
{
    T result (coord[0]);
    for (int i=0; i<n; i++)
        if (result < coord[i])
            result = coord[i];
}
```

# Шаблонные методы

```
template <class T>
struct Array{
    template<class V>
    Array<T>& operator=
        (Array<V> const & m);
};
template<class T>
template<class V>
Array<T>& Array<T>::
    operator =
        (Array const & m) { ...
}
```

```
Array<int> m;
Array<double> d;
d = m;
/*Просто запись Array
внутри класса означает
Array с уже
подставленным
параметром. Вне класса
это не действует*/
template <class T>
void sort(Array<T>& m) { ... }
sort(d); /*в sort передан
массив из double */
```

# Шаблонный

## конструктор:

```
template<class T>
struct Array {
    template<class V>
    Array(Array<V> a) { ... }
}; /* предполагается, что
    есть неявное
    приведение типа V к
    T*/
```

*Если шаблонная функция (или метод  
шаблонного класса) не вызывается, то она и не  
копируется!*

*\*Виртуальная функция не может быть  
шаблонной.*

## Конструктор

копирования,

конструктор по

умолчанию не могут

быть шаблонными:

```
template<class M>
```

```
Array() {...} // нельзя!
```

```
Array<int> ??? a;
```

```
// непонятно, куда
```

```
вставить вторую пару
```

```
<>.
```

# Специализация шаблона

```
template <class T>
struct Array { ... };

template<>
struct Array<bool>
{
...
//отдельная
реализация только
для bool
};
```

Реализация класса точки в различных измерениях (1D, 2D, 3D...):

```
template <size_t dimention>
struct point;
```

```
template<>
struct point<1>{
... //определение
};
```

Теперь возможно создать точку только для 1 измерения



# Частичная специализация

Массив указателей:

```
template<class T>  
struct Array<T*> { ... };
```

// T- указатель на  
какой-либо тип

Массив массивов:

```
template<class T>  
struct Array<Array<T> >  
{ ... };
```

//T - массив Array

```
template<int N>
```

```
struct fact{
```

```
    static const int v =
```

```
        fact<n-1>::v * n;
```

```
};
```

```
template<>
```

```
struct fact<0>{
```

```
    static const int v = 1;
```

```
}
```

fact<10>::v; //Значение  
факториала будет посчитано  
на этапе компиляции!! (Для  
каждой реализации шаблона  
статические данные

# Различия между шаблоном класса и функциями

Для функций отсутствуют частичные специализации. Но их можно заменить перегрузкой функций:

```
template<class T>
void sort(Array<T>& m)
{...};
```

```
template<class T>
void sort(T& t) {...};
```

Шаблон класса может иметь параметры по умолчанию

```
template<class T = string>
struct V{ ... };
```

/\* В запись вида V<> по умолчанию будет подставлен string \*/

```
template<class A, class B = A>
struct V{ ... };
```

```
V<int>
```

// вместо A и B будет int

# Typedef

Чтобы избежать громоздких записей имен типов вида

```
Array<pair<Array<int>, string> > m;
```

СТОИТ ИСПОЛЬЗОВАТЬ typedef:

```
typedef Array<int> AInt;
```

```
Array<pair<AInt, string> > m;
```

```
typedef int* PI;
```

```
typedef const PI CPI; //CPI будет константным  
указателем, т.к. известно, что PI - указатель. А при  
использовании #define в CPI было бы const int *, т.е.  
указатель на const int.
```

# Наследование в шаблонах классов

```
template <class T>
class Trio: public Pair <T>{
    T c;
public:
    Trio (T t1, T t2, T t3);
    ...
};
```

```
template <class T>
Trio<T>::Trio (T t1, T t2, T t3): Pair <T> (t1, t2), c(t3) {}
/* вызов родительского конструктора также
сопровождается передачей типа T в качестве
параметра*/
```

- Базовый класс для шаблона может быть как шаблонным, так и обычным классом. Обычный класс может быть порожден от реализованного шаблона.

- Нельзя использовать указатель на базовый шаблонный класс для получения доступа к методам производных классов: типы, полученные даже из одного и того же шаблона, всегда являются разными.

- В описание шаблона классов можно включать дружественные функции. Если функция-друг не использует спецификатор шаблона, то она считается универсальной для всех экземпляров шаблона. Если же в прототипе функции-друга содержится шаблон параметров, то эта функция будет дружественной только для того класса, экземпляр которого создается.