Шаблоны C++

В C++ в некоторых случаях обоснованно использование шаблонов программирования. Механизм шаблонов позволяет решать проблему унификации алгоритма для различных типов. В языке C++ существуют шаблоны функций и шаблоны классов.

Шаблон функции представляет собой семейство разных функций. По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными.

Шаблоны классов это обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов.

Пример необходимости применения шаблонов может быть рассмотрен на основе функции `min` . В самом простом случае для целых и действительных чисел придётся перегрузить функцию.

```
int _min( int a, int b ){
    if( a < b ){
        return a;
    }
    return b;
}

double _min( double a, double b ){
    if( a < b ){
        return a;
    }
    return b;
}
```

Однако подобная функция может пригодиться и для других типов, например если необходимо выяснить какая из двух строчек меньше. В таком случае, если алгоритм является общим для типов, с которыми приходится работать, можно определить шаблон функции. Синтаксис шаблонов начинается с ключевого слова `template`, которое сообщает компилятору, что дальше мы будем объявлять параметры шаблона:

```
template<class T>
T _min(T a, T b){
    if( a < b){
        return a;
    }
    return b;
}
```

В случае перегрузки все функции компилируются, однако в случае шаблонов пока нет вызова функции `min`, при компиляции она в бинарном коде не создается. А если объявить группу вызовов функции с переменными различных типов, то для каждого компилятор создаст свою реализацию на основе шаблона. Вызов шаблонной функции, в общем, эквивалентен вызову обыкновенной функции, однако если подставляемые параметры окажутся разных типов, то компилятор не сможет вывести реализацию шаблона.

```
cout << _min( 1, 2 ) << endl;
cout << _min( 3.1, 1.2 ) << endl;
cout << _min( 5, 2.1 ) << endl;   //ошибка
```

Подобная проблема решается явным указанием типа `T` в параметрах шаблона (используются угловые скобки).

```
cout << _min<double>( 5, 2.1 ) << endl;
```

В силу исторических причин для определения параметра типа разрешается применение вместо `class` ключевого слова `typename`. Ключевое слово typename в ходе эволюции языка `C++` появилось относительно недавно, а до этого единственным способом задания параметра типа было ключевое слово `class`. Применение `class` для определения параметра типа корректно и сегодня. Семантически в данном контексте между этими двумя способами записи нет никакой разницы. Даже в случае применения ключевого слова `class` для аргументов шаблона может быть использован любой тип.

```
template<typename T>
T _min(T a, T b){
    if( a < b){
        return a;
    }
    return b;
}
```

В качестве параметров функции `min` могут выступать примитивные типы, такие как `char`, `int`, `double` и так далее, однако в тоже время это могут быть и объекты классов. Тогда функцию нужно модифицировать, так объекты класса чаще всего принято передавать по указателю или ссылке.

```
template <class T>
const T& _min(const T& a, const T& b)
{
    return (a < b) ? a : b;
}
```

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

```
template <class T1, class T2, class T3 >
const T3& _min(const T1& a, const T2& b)
{
    …
}
```

Однако, если для объекта класса, переданного в функцию `_min` не перегружены операции сравнения, то в итоге при попытке компиляции программы будет получена ошибка.

```
class Currency
{
private:
    int count;
public:
    Currency(int arg_count):count(arg_count){}
};

…

Currency seven( 7 );
Currency twelve( 12 );

Currency less = _min ( seven, twelve );
```

Препроцессор создаст следующий экземпляр шаблона функции `_min`:

```
const Currency& _min(const Currency &a, const
Currency &b)
{
    return (a < b) ? a : b;
}
```

А затем компилятор попытается скомпилировать эту функцию, но ничего не получится, так как не ясно как обрабатывать выражение `a < b`.

Необходимо перегрузить оператор <:

```
class Currency
{
private:
    int count;
public:
    Currency(int arg_count) : count(arg_count){}
    // перегрузка оператора меньше
    friend bool operator<(const Currency &c1,
const Currency &c2)
    {
        return (c1.count < c2.count);
    }
};
```

В таком случае код выполнится верно, значение `less` будет равно семи:

```
Currency seven( 7 );
Currency twelve( 12 );

Currency less = _min ( seven, twelve );
```

В некоторых случаях шаблон функции является неэффективным или неправильным для определенного типа. В этом случае можно специализировать шаблон, — то есть написать реализацию для данного типа. Например, в случае со строками можно потребовать, чтобы функция сравнивала только количество символов.

```
template<>
const string& _min(const string& a, const string& b){
    if(a.size() < b.size()){
        return a;
    }
    return b;
}
```

Создание шаблона класса аналогично созданию шаблона функции. Компилятор копирует шаблон класса, заменяя типы параметров шаблона класса на фактические (передаваемые) типы данных, а затем компилирует эту копию.

```
template <class T>  // это шаблон класса с T вместо фактического (передаваемого) типа данных
class Array
{
    int m_length;
    T *m_data;
public:
    Array(int length)
    {
        m_data = new T[length];
        m_length = length;
    }
    ~Array()
    {
        delete[] m_data;
    }
    void Erase()
    {
        delete[] m_data;
        m_data = nullptr;
        m_length = 0;
    }

    T& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength();  // определение метода и шаблона метода getLength() далее
};
template <class T>  // метод, определенный вне тела класса, нуждается в собственном определении шаблона
int Array<T>::getLength() { return m_length; }  // имя класса - Array<T>, а не просто Array
```

Следует обратить внимание на определение функции `getLength()` вне тела класса. Каждый метод шаблона класса, объявленный вне тела класса, нуждается в собственном объявлении шаблона.

Возможно специализировать не только шаблоны функций, но и шаблоны классов. Специализация шаблона класса рассматривается компилятором как полностью отдельный и независимый класс, хоть и выделяется как обычный шаблон класса. Это означает, что возможно изменить в классе всё что угодно, включая его реализацию/методы/спецификаторы доступа и т.д. В качестве примера можно привести класс-массив, который может хранить 8 объектов:

```
template <class T>
class Repository8
{
private:
    T m_array[8];

public:
    void set(int index, const T &value)
    {
        m_array[index] = value;
    }

    const T& get(int index)
    {
        return m_array[index];
    }
};
```

Класс-массив будет работать с любым типом данных:

```
Repository8<int> intRep;
for (int count=0; count<8; ++count)
    intRep.set(count, count);
for (int count=0; count<8; ++count)
    cout << intRep.get(count) << endl;

// >>1 2 3 4 5 6 7
```

```
Repository8<bool> boolRep;
for (int count=0; count<8; ++count)
    boolRep.set(count, count % 5);
for (int count=0; count<8; ++count)
    cout << (boolRep.get(count) ? "true":"false") << endl;

// >> false true true true true false true true

cout << sizeof(boolRep) << endl; // >> 8
```

Реализация `Repository8<bool>`, на самом деле, не столь эффективна, какой она могла бы быть. Поскольку все переменные должны иметь адрес, а ЦП не может дать адрес чему-либо меньшему, чем 1 байт, то размер всех переменных должен быть не менее 1 байта. В таком случае переменная типа `bool` будет занимать в 8 раз больше.

Можно специализировать шаблон класса для работы с определенным типом данных:

```
template <>
class Repository8<bool>
{
    unsigned char data;
public:
    Repository8() : data(0) {}

    void set(int index, bool value)
    {
        unsigned char mask = 1 << index;
        if (value)
            data |= mask;
        else
            data &= ~mask;
    }

    bool get(int index)
    {
        unsigned char mask = 1 << index;
        return (data & mask) != 0;
    }
};
```

Специализация шаблона Repository8<bool> теперь занимает не 8 байт а 1 байт. Весь функционал Repository8 при этом сохранился.

```
Repository8<int> intRep;
for (int count=0; count<8; ++count)
    intRep.set(count, count);
for (int count=0; count<8; ++count)
    cout << intRep.get(count) << endl;

// >>1 2 3 4 5 6 7
```

```
Repository8<bool> boolRep;
for (int count=0; count<8; ++count)
    boolRep.set(count, count % 5);
for (int count=0; count<8; ++count)
    cout << (boolRep.get(count) ? "true":"false") << endl;

// >> false true true true true false true true

cout << sizeof(boolRep) << endl; // >> 1
```

Для шаблонов функций явная специализация шаблона и обычная перегрузка не шаблонной функцией — это два альтернативных способа задания частной версии функции. Рекомендуется для этой цели пользоваться именно вторым вариантом — перегрузкой. Явная специализация предназначена в первую очередь для шаблонов классов, а не функций. Можно привести следующий пример:

```
// 1: базовый шаблон
template <class T> void foo(T);
// 2: еще один, перегруженный базовый шаблон
template <class T> void foo(T *);
// 3: явная специализация для базового шаблона
template <> void foo(int *);

...

int *p;
foo(p);  // вызывается вариант 3
```

По правилам языка C++ в процессе `overload resolution` участвуют только базовые шаблоны `1` и `2` (но не специализация `3`). Побеждает шаблон `2`. Только после этого во внимание принимаются возможные специализации этого шаблона. В результате выбирается явная специализация `3` и вызывается вариант `3`. Если поменять местами объявления `2` и `3`, то поведение программы изменится.

```
// 1: базовый шаблон
template <class T> void foo(T);
// 3: явная специализация для базового шаблона
template <> void foo(int *);
// 2: еще один, перегруженный базовый шаблон
template <class T> void foo(T *);


...

int *p;
foo(p);  // вызывается вариант 2
```

Теперь, при таком порядке объявления, явная специализация `3` привязывается уже к базовому шаблону `1`, а не `2`. Далее в процессе `overload resolution`, как и прежде, участвуют только базовые шаблоны `1` и `2`. Побеждает, как и прежде, шаблон `2`. Но теперь явная специализация `3` уже не принимается во внимание и не выбирается, ибо она теперь "принадлежит" шаблону `1`. В результате вызывается вариант `2`. Перестановка объявлений поменяла семантику кода. Чтобы этого не происходило, рекомендуется для шаблонов функций применять механизм перегрузки, а не явной специализации. Это позволит частным версиям функции участвовать в процессе `overload resolution` сразу и самостоятельно.

Параметры по умолчанию

В языке C++ существует понятие параметра по умолчанию. Это параметры функции, которые имеют определенное значение. Если же пользователь передает значение, то это значение используется вместо значения по умолчанию.

```
void printValues(int a, int b=5)
{
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}

int main()
{
    printValues(1);  // в качестве b будет использоваться значение по умолчанию 5
    //>>a: 1
    //>>b: 5
    printValues(6, 7);  // в качестве b будет использоваться значение, предоставляемое пользователем 7
    //>>a: 6
    //>>b: 7
}
```

В первом вызове функции не передается аргумент для `b`, поэтому функция использует значение по умолчанию — `5`. Во втором вызове передаем значение для `b`, поэтому оно используется вместо параметра по умолчанию. Также функция может иметь несколько параметров по умолчанию:

```
void printValues(int a=10, int b=11, int c=12)
{
    cout << "Values: " << a << " " << b << " " << c <<
endl;
}

…

printValues(3, 4, 5); //>>Values: 3 4 5
printValues(3, 4); //>>Values: 3 4 12
printValues(3); //>>Values: 3 11 12
printValues(); //>>Values: 10 11 12
```

Все параметры по умолчанию в прототипе или в определении функции должны находиться справа. Появление следующей функции в коде вызовет ошибку:

```
// не допустимая функция
void printValues(int a=10, int b=11, int c)
{
    cout << "Values: " << a << " " << b << " " <<
c << endl;
}

// допустимая функция с двумя обязательными и одним
необязательным параметрами
void printValues(int a, int b, int c=12)
{
    cout << "Values: " << a << " " << b << " " <<
c << endl;
}
```

Для шаблонных классов и функций существует также понятие аргументов шаблона по умолчанию. Можно использовать такие аргументы по аналогии с аргументами по умолчанию обычных функций. Например `std::vector` из `STL`:

```
template<class Type, class Allocator =
std::allocator<Type>>
class vector;
```

Класс вектора стандартной библиотеки C++ — это шаблон класса для контейнеров последовательностей. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Параметр `Type` - это тип данных элементов, сохраняемых в векторе. `Allocator` - это тип, представляющий объект, управляющий памятью. Этот аргумент является необязательным, и значением по умолчанию является `std::allocator<Type>`.

Псевдонимы

Язык C++ позволяет использовать псевдонимы для типов и использовать их вместо фактического имени типа.
Переопределение возможно с помощью ключевого слова `typedef`:

```
typedef short studentID_t;
```

`typedef` также позволяет изменить базовый тип объекта без внесения изменений в большое количество кода. Например, если для хранения идентификационного номера учащегося используется тип `short`, однако потом потребовалось использовать тип `long`, то необходимо лишь заменить тип в объявлении `typedef`.
Также `typedef` позволяет скрывать специфические для определенных операционных систем детали.

Также псевдонимы возможно объявить начиная со стандарта C++11 с помощью ключевого слова `using`:

```
using studentID_t = short;
```

Так же переименование с помощью using возможно и для шаблонов. Например `std::vector` - это шаблон класса, у него есть два параметра. Первый параметр шаблона обязательный, второй получает значение по умолчанию, в случае если ничего не указано.

```
template<class T, class Allocator =
std::allocator<T>>
class vector;
```

С помощью `using` можно получить шаблон `dync_vector`, имеющий всего один обязательный тип:

```
template <class T>
using dync_vector = std::vector<T>;
```

Рекурсивное определение шаблонов

C++ допускает рекурсивное определение шаблонов.

```
template<unsigned long N>
struct binary{
    static unsigned const value = binary<N/10>::value <<
1 | N % 10;
};
```

```
template<>
struct binary<0>{
    static unsigned const value = 0;
};
```

…

```
unsigned const one = binary<1>::value;
unsigned const two = binary<10>::value;
unsigned const seven = binary<111>::value;
```

Аналогичный пример с числами Фибоначчи:

```
template<unsigned long N>
struct fibonacci{
    static unsigned const value = fibonacci<N-2>::value +
fibonacci<N-1>::value;
};

template<>
struct fibonacci<1>{
    static unsigned const value = 1;
};
```

```
template<>
struct fibonacci<0>{
    static unsigned const value = 0;
};
…

fibonacci<1>::value; // 1
fibonacci<2>::value; // 1
fibonacci<3>::value; // 2
fibonacci<4>::value; // 3
```

Шаблоны в C++ являются средствами метапрограммирования и реализуют полиморфизм времени компиляции. В примере приведен код, реализующий двоичные нотации. Десятичное число, записанное в виде двоичного числа (для корректной работы требуется, чтобы оно состояло из нулей и единичек) передается в шаблон, который в свою очередь строит ещё один шаблон. Такой процесс продолжается рекурсивно, пока тип не сократиться до известной специализации 0. Например код `binary<111>::value` в начале будет пытаться конкретизировать шаблон с параметром `111`, потом с параметром `11`, потом с параметром `1`, потом с параметром `0` и при сложении получится десятичное число `7`. Весь процесс рекурсии исполняется компилятором, очевидно влиять на скорость исполнения программы такие нотации не будут.

Функции с переменным количеством параметров

В C++ существует возможность передавать в функцию с переменное кол-во параметров. Функция принимает аргументы исходя из соглашении о вызовах (calling convention). Чтобы унифицировать передачу различного числа аргументов чаще всего используется макрос `va_arg` из `stdarg.h`, данный пример также актуален для языка Си:

```
void printArgs(size_t k, ...) {
    va_list ap;
    va_start(ap, k);

    while (k--)
        printf("%d ", va_arg(ap, int));

    va_end(ap);
}

…

printArgs(5, 1, 2, 3, 4, 5); // >> 1 2 3 4 5
```

Первый аргумент обязательный, он содержит число параметров.
`va_list` - специальный тип, который используется для извлечения дополнительных параметров функции.
Макрос `void va_start(va_list ap, paramN)` - данный макрос инициализирует `ap` для извлечения дополнительных аргументов, которые идут после переменной `paramN`. Параметр не должен быть объявлена как `register`, не может иметь типа массива или указателя на функцию.
Макрос `void va_end(va_list ap)` - необходим для нормального завершения работы функции, необходим после вызова макроса `va_start`.
Макрос `void va_copy(va_list dest, va_list src)` - копирует `src` в `dest`. Поддерживается начиная со стандарта C++11.

Использование `va_arg` накладывает некоторые ограничения на типы, передаваемых в функцию значений. В самом простом случае это один и тот же тип, в более сложных необходимо передать дополнительные параметры, например:

```
printf("%d %f %lf", 1, 2.0f, 3.0);
```

Вариативные шаблоны

Начиная с версии C++11 возможно объявить вариативный шаблон с пакетом параметров, переменного размера. Например, можно написать функцию, которая принимает произвольное число параметров произвольных типов и возвращает их сумму. Если суммирование для какой-нибудь пары типов не определено, то можно получить ошибку компиляции.

```
template <class Arg>
Arg sum(Arg arg) { return arg; }

template <class First, class... Other>
auto sum(First first, Other... other)
{
  return first + sum(other…);
}
```

Для определения шаблона с переменным числом параметров необходимо указать пакет типов в `template`-заголовке:

```
template <class... Types>
```

`Types` можно использовать в списке параметров функции:

```
template <class... Types>
void f(Types... parameters)
```

Теперь `parameters` — это пакет параметров, типы которых задаются “параллельным” пакетом `Types`. Пакет параметров может быть только в функции-шаблоне, и ему обязан соответствовать пакет типов. Внутри функции можно использовать как пакет типов, так и пакет параметров. Для этого после имени пакета ставится многоточие …, которое интерпретируется компилятором как раскрытие пакета в список (через запятую) в указанном месте. Из-за такого минимализма возможных действий с пакетами обычным способом определения функций-шаблонов с переменным числом параметров является рекурсивное определение.