Часть I

Основы

Эта часть книги знакомит читателя с общими концепциями и языковыми средствами шаблонов C++. Она начинается с обсуждения основных задач и концепций на примерах шаблонов функций и шаблонов классов. В последующих главах рассматриваются некоторые дополнительные фундаментальные приемы работы с шаблонами, в частности параметры шаблонов, не являющиеся типами, ключевое слово `typename` и шаблоны-члены. В завершение приведены некоторые распространенные приемы применения шаблонов на практике.

Данное введение в шаблоны частично использовано Николаи Джосаттисом (Nicolai M. Josuttis) в его книге *Object-Oriented Programming in C++*, опубликованной издательством John Wiley and Sons Ltd, ISBN 0-470-84399-3. Эта книга представляет собой учебное пособие, в котором дается описание всех возможностей языка C++ и его стандартной библиотеки, а также их практического применения.

Зачем нужны шаблоны

В C++ можно объявлять переменные, функции и большинство других видов объектов, используя конкретные типы. Однако в основном код для обработки объектов различных типов выглядит практически одинаково. Это особенно справедливо, если для разных типов данных требуется реализовать алгоритмы наподобие *быстрой сортировки* либо способы обработки таких структур данных, как связанный список или двоичное дерево. В таких случаях код одинаков для всех используемых типов объектов.

В общем случае (если язык программирования не поддерживает специальных средств для решения подобных задач) у программиста имеются только следующие альтернативы:

1. Можно вновь и вновь реализовывать один и тот же алгоритм для каждого типа данных.
2. Можно написать общий код для обобщенного базового типа, такого, как `Object` или `void*`.
3. Можно использовать специальные препроцессоры.

Если говорить о конкретных языках (таких, как C, Java или подобных им), то читателю, возможно, уже приходилось проделывать подобные действия. Однако каждый из описанных выше подходов имеет свои недостатки.

1. Каждый раз заново реализуя один и тот же алгоритм, мы, по сути, снова и снова изобретаем велосипед. Мы делаем одни и те же ошибки и, чтобы не наделать их еще больше, стараемся избегать более сложных, но зато и более эффективных алгоритмов.
2. Если мы пишем обобщенный код для общего базового класса, то теряем при этом преимущество проверки типов. Кроме того, в разных ситуациях может потребоваться порождение от различных базовых классов, что еще более затрудняет поддержку кода.
3. При использовании специального препроцессора, например препроцессора C/C++, теряется преимущество форматирования исходного кода. Код заменяется некоторым “тупым механизмом замены текста”, который не имеет представления ни об области видимости, ни о типах.

Шаблоны обеспечивают решение данной проблемы, лишенное недостатков, присущих рассмотренным способам. Шаблон представляет собой функцию или класс, реализованные для одного или нескольких типов данных, которые не известны в момент написания кода. При использовании шаблона в качестве аргументов ему явно или неявно передаются конкретные типы данных. Поскольку шаблоны являются средствами языка, для них обеспечивается полная поддержка проверки типов и областей видимости.

Шаблоны получили широкое применение в современном программировании. Например, практически весь код в стандартной библиотеке C++ состоит из шаблонов. Библиотека обеспечивает алгоритмы для сортировки объектов и значений определенного типа, структуры данных (так называемые *классы контейнеров*) для управления элементами конкретного типа, строки, для которых тип символа является параметризованным, и т.п. Однако это еще не все: шаблоны позволяют также параметризовать способы обработки данных, оптимизировать код и параметризовать информацию. Как это делается, описано в последующих главах. А пока начнем с самых простых шаблонов.

Глава 2

Шаблоны функций

Данная глава знакомит читателя с шаблонами функций. Шаблоны функций — это параметризованные функции; таким образом, шаблон функции представляет целое семейство функций.

2.1. Первое знакомство с шаблонами функций

Шаблон функции — это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов; другими словами, шаблон функции представляет семейство функций. Шаблон очень похож на обычную функцию, разница только в том, что некоторые элементы этой функции не определены и являются параметризованными. Чтобы проиллюстрировать сказанное выше, рассмотрим небольшой пример.

2.1.1. Определение шаблона

Ниже приведен шаблон функции, возвращающей большее из двух значений.

```
// basics/max.hpp
template <typename T>
inline T const& max(T const& a, T const& b)
{
    // Если a < b, возвращаем b, иначе a
    return a < b ? b : a;
}
```

Определение шаблона задает семейство функций, возвращающих большее из двух значений; эти значения передаются функции как ее параметры `a` и `b`. Тип этих параметров оставлен не определенным и задается как *параметр шаблона* `T`. Как можно видеть из приведенного примера, параметры шаблонов необходимо объявлять, используя следующий синтаксис:

`template <` *`разделенный_запятыми_список_параметров`* `>`

В нашем примере список параметров задан как `typename T`. Обратите внимание, что в качестве скобок используются символы “меньше” и “больше”. Ключевое слово

typename задает так называемый *параметр типа*. Это наиболее распространенный вид параметров шаблонов в программах на C++, хотя возможны и другие параметры, которые рассматриваются в книге несколько позже (см. главу 4, “Параметры шаблонов, не являющиеся типами”).

В данном примере параметр типа обозначен как T. В качестве имени параметра можно использовать любой идентификатор, но обычно по соглашению используется именно T. Параметр типа представляет произвольный тип, который определяется при вызове функции. Можно использовать любой тип (это может быть один из базовых типов, класс и т.п.), который допускает применение операций, задействованных в шаблоне. В нашем случае тип T должен поддерживать оператор <, поскольку он используется в теле функции для сравнения a и b.

В силу исторических причин для определения параметра типа разрешается применение вместо typename ключевого слова class. 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. Поэтому эквивалентным способом определения шаблона max() является следующий:

```
// basics/max.hpp
template <class T>
inline T const& max(T const& a, T const& b)
{
    // Если a < b, возвращаем b, иначе a
    return a < b ? b : a;
}
```

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. Однако, поскольку ключевое слово class может ввести в заблуждение (вместо T можно подставлять не только тип, являющийся классом), в данном контексте следует отдавать предпочтение использованию ключевого слова typename. Отметим, что в отличие от объявлений типа класса, ключевое слово struct при объявлении параметров типа вместо typename использовать нельзя.

2.1.2. Использование шаблонов

В приведенном ниже фрагменте кода иллюстрируется применение шаблона функции max().

```
// basics/max.cpp
#include <iostream>
#include <string>
#include <max.hpp>

int main()
```

```
{
    int i = 42;
    std::cout << "max(7,i):   " << ::max(7,i) << std::end1;
    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << std::end1;

    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1,s2) << std::end1;
}
```

В этой программе `max()` вызывается трижды: для двух значений типа `int`, для двух `double` и для двух `std::string`. Каждый раз вычисляется большее значение. В результате программа выводит следующую информацию:

```
max(7,i):   42
max(f1,f2): 3.4
max(s1,s2): mathematics
```

Вы обратили внимание на то, что в примере каждый вызов шаблона `max()` предваряется двумя двоеточиями — `::`? Делается это вовсе не потому, что `max()` находится в глобальном пространстве имен. Причина здесь другая: в стандартной библиотеке тоже есть шаблон `std::max()`, который может быть вызван при определенных обстоятельствах или способен привести к неоднозначности[1].

Обычно шаблоны не компилируются в какой-то один объект, способный обрабатывать любой тип данных. Вместо этого из шаблона генерируются различные объекты для каждого типа, для которого применяется шаблон[2].

Таким образом, `max()` компилируется отдельно для каждого из упомянутых типов. Например, для первого вызова `max()`

```
int i = 42;
... max(7,i) ...
```

используется шаблон функции, в котором в качестве параметра шаблона `T` указан тип `int`. Таким образом, он имеет семантику вызова следующего кода:

```
inline int const& max(int const& a, int const& b)
{
    // Если a < b, то возвращаем b, иначе a
```

[1] Например, если один тип аргумента определен в пространстве имен `std` (например, `string`), тогда в соответствии с правилами поиска имен C++ будут найдены оба шаблона — как глобальный, так и `std::max()`.

[2] Альтернативный способ — “один объект на все случаи жизни” — также имеет право на существование, но на практике встречается крайне редко. Все правила языка основываются на предположении, что генерируются различные объекты.

```
    return a < b ? b : a;
}
```

Процесс подстановки конкретных типов вместо параметров шаблона называется *инстанцированием шаблона* (*instantiation*). Его результатом является *экземпляр* шаблона. К сожалению, термины *инстанцирование* (*instantiation*) и *экземпляр* (*instance*) в объектно-ориентированном программировании применяются и в другом контексте, а именно для конкретного объекта класса. Однако, поскольку наша книга посвящена шаблонам, этот термин будет использоваться применительно к шаблонам, если специально не оговорено другое.

Отметим, что для запуска процесса инстанцирования достаточно просто использовать шаблон функции. Специально требовать от компилятора инстанцирования шаблона не нужно.

Аналогично, другие вызовы `max()` инстанцируют шаблон `max` для `double` и `std::string` точно так же, как они создавались бы в случае отдельного объявления и применения:

```
const double& max(double const&, double const&);
const std::string& max(std::string const&,
                       std::string const&);
```

Попытка инстанцировать шаблон для типа, который не поддерживает все используемые в шаблоне операции, приведет к ошибке компиляции, например:

```
std::complex<float> c1,c2; // complex не поддерживает
                           // оператор <
...
max(c1,c2);                // ОШИБКА компиляции
```

Таким образом, шаблоны компилируются дважды.

1. Без инстанцирования; код самого шаблона проверяется на правильность синтаксиса. Выявляются синтаксические ошибки, например пропущенные точки с запятой.
2. Во время инстанцирования код шаблона проверяется на корректность всех вызовов. Выявляются некорректные вызовы, в частности неподдерживаемые вызовы функций.

Здесь проявляется важная проблема, связанная с обработкой шаблонов: если применение шаблона функции предполагает инстанцирование, то компилятору в определенный момент потребуется полное определение этого шаблона. Это отличается от обычных функций, когда для компиляции достаточно их объявления. Методы решения этой проблемы обсуждаются в главе 6, “Применение шаблонов на практике”. А пока возьмем на вооружение простейший способ: реализуем каждый шаблон в заголовочном файле с использованием встраиваемых функций.

2.2. Вывод аргументов

При вызове шаблона функции (например, `max()`) с какими-либо аргументами параметры шаблона определяются передаваемыми в функцию аргументами. Если в качестве параметров

типа T const& передается два значения int, компилятор делает вывод, что вместо T следует подставить int. Заметим, что автоматическое преобразование типов в шаблонах не допускается. Должно быть точное соответствие для каждого параметра типа, например:

```
template <typename T>
inline T const& max(T const& a, T const& b);
...
max(4,7);    // ВЕРНО: T — int для обоих аргументов
max(4,4.2);  // ОШИБКА: первый T — int, второй — double
```

Существует несколько способов исправить эту ошибку.

1. Привести оба аргумента к одному типу:
   `max(static_cast<double>(4),4.2); //ВЕРНО`
2. Указать тип T явно:
   `max<double>(4,4.2); //ВЕРНО`
3. Указать, что параметры могут иметь различные типы.

Эти вопросы рассматривается в следующем разделе более подробно.

2.3. Параметры шаблонов

Существуют два вида параметров шаблонов функций.

1. *Параметры шаблона*, которые объявляются в угловых скобках перед именем шаблона функции:
   `template <typename T>  // T является параметром шаблона`
2. *Параметры вызова*, которые объявляются в круглых скобках после имени шаблона функции:
   `max(T const& a, T const& b); // a и b - параметры вызова`

Количество задаваемых параметров неограниченно. Однако в шаблонах функций (в отличие от шаблонов классов) нельзя использовать аргументы шаблона по умолчанию[3].

Например, можно определить шаблон max() для двух различных типов данных.

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
    return a < b ? b : a;
}
...
max(4,4.2)    // ВЕРНО, однако тип возвращаемого
```

[3] Это ограничение является главным образом результатом проблем исторического характера в развитии шаблонов функций. Для реализации такой возможности в современных компиляторах C++ технических препятствий не существует, и в будущем задание параметров шаблона по умолчанию, вполне вероятно, станет возможным (см. раздел 13.3).

```
// значения определяется типом первого
// аргумента
```

Казалось бы, неплохо иметь возможность передавать шаблону `max()` два параметра вызова различных типов, но этот способ имеет свои недостатки. Проблема заключается в том, что мы должны объявить тип возвращаемого значения. Если для этого использовать один из типов параметров, аргумент для другого параметра должен конвертироваться в этот же тип, независимо от того, что именно хотел бы получить вызвавший этот шаблон программист. В C++ нет возможности задать выбор "наиболее мощного типа" (хотя такую возможность можно обеспечить с помощью определенных трюков при программировании шаблонов — см. раздел 15.2.4, стр. 298). Таким образом, в зависимости от порядка аргументов при вызове можно получить наибольшее из значений `42` и `66.66` и как `double 66.66`, и как `int 66`. Еще один недостаток заключается в том, что при конвертировании типа второго параметра в тип возвращаемого значения создается новый локальный временный объект, а это означает, что возврат результата по ссылке невозможен[4]. Поэтому в нашем примере тип возвращаемого значения должен быть `T1`, а не `T1 const&`.

Поскольку типы параметров вызова конструируются из параметров шаблона, параметры шаблона и параметры вызова обычно взаимосвязаны. Эта концепция называется *выводом аргументов шаблона функции (function template argument deduction)* и обеспечивает возможность вызывать шаблонную функцию так же, как и обычную.

Однако, как уже упоминалось ранее, можно явно инстанцировать шаблон для конкретных типов.

```
template <typename T>
inline T const& max(T const& a, T const& b);
...
max<double>(4,4.2); // Инстанцирование для T,
                    // представляющего собой double
```

В тех случаях, когда связь между параметрами шаблона и параметрами вызова отсутствует или когда невозможно определить параметры шаблона, аргумент шаблона в его вызове следует задавать явно. Например, можно ввести третий тип аргумента шаблона, который задает тип значения, возвращаемого функцией.

```
template <typename T1, typename T2, typename RT>
inline RT max(T1 const& a, T2 const& b);
```

Однако вывод аргументов шаблона не работает с возвращаемыми типами[5], а `RT` среди типов параметров вызова функции отсутствует. Следовательно, для определения `RT`

---

[4] Нельзя возвращать значения по ссылке, если они являются локальными для функции, поскольку при этом возвращается нечто, уже не существующее (после того как программа покинет область видимости данной функции).

[5] Вывод можно рассматривать как часть распознавания имени функции по типам ее параметров — процесс, который не использует тип возвращаемого значения. Единственным исключением является тип возвращаемого значения оператора-члена преобразования типов.

обычный вывод применить нельзя, а значит, список аргументов шаблона нужно задавать явно, например:

```
template <typename T1, typename T2, typename RT>
inline RT max(T1 const& a, T2 const& b);
...
max<int,double,double>(4,4.2)  // ВЕРНО, но утомительно
```

До сих пор рассматривались случаи, когда все аргументы шаблона функции либо задавались явно, либо явно не задавался ни один из них. Существует еще один подход: явно задается только первый аргумент, а остальные определяются при помощи вывода. Общее правило можно сформулировать так: следует явно задавать все типы аргументов, которые нельзя определить неявно. Таким образом, если в нашем примере изменить порядок следования параметров шаблона, то при вызове потребуется указать только тип возвращаемого значения.

```
template <typename RT, typename T1, typename T2>
inline RT max(T1 const& a, T2 const& b);
...
max<double>(4,4.2)   // ВЕРНО: тип возвращаемого
                     // значения — double
```

В данном примере при вызове `max<double>` значение `RT` явно задается как `double`, а типы параметров `T1` и `T2` определяются путем вывода из переданных аргументов как `int` и `double`.

Заметим, что все рассмотренные выше модифицированные версии `max()` не обеспечивают сколько-нибудь значительных преимуществ — ведь ничто не мешает для версии с одним параметром явно указать тип параметра (и тип возвращаемого значения) для случая передачи аргументов различных типов. Поэтому лучше не усложнять себе жизнь и остановиться на версии `max()` с одним параметром (именно так мы и будем поступать в следующих разделах при обсуждении других вопросов, касающихся шаблонов).

Процесс вывода более подробно описан в главе 11, “Вывод аргументов шаблонов”.

2.4. Перегрузка шаблонов функций

Шаблоны могут быть перегружены точно так же, как и обычные функции. Другими словами, могут иметься различные определения функций с одним и тем же именем, и при вызове функции с этим именем компилятор C++ примет решение о том, какую из функций-кандидаток следует вызвать. Правила принятия такого решения достаточно сложны даже без использования шаблонов. В этом разделе рассматривается перегрузка при участии шаблонов. Читателям, не знакомым с основными правилами перегрузки без шаблонов, рекомендуем обратиться к приложению Б, “Разрешение перегрузки”, где дается достаточно подробный обзор правил перегрузки функций.

Приведенная ниже небольшая программа иллюстрирует перегрузку шаблона функции.

```
// basics/max2.cpp
// Большее из двух целочисленных значений
inline int const& max(int const& a, int const& b)
{
    return a < b ? b : a;
}

// Большее из двух значений произвольного типа
template <typename T>
inline T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}

// Большее из трех значений произвольного типа
template <typename T>
inline T const& max(T const& a, T const& b, T const& c)
{
    return max(max(a,b),c);
}

int main();
{
    ::max(7, 42, 68);   // Вызов шаблона для трех аргументов
    ::max(7.0, 42.0);   // Вызов max<double> (вывод
                        // аргументов)
    ::max('a', 'b');    // Вызов max<char> (вывод аргументов)
    ::max(7, 42);       // Вызов функции, не являющейся
                        // шаблоном, для двух целочисленных
                        // аргументов
    ::max<>(7, 42);     // Вызов max<int> (вывод аргументов)
    ::max<double>(7,42);// Вызов max<double> (без вывода
                        // аргументов)
    ::max('a', 42.7);   // Вызов функции, не являющейся
                        // шаблоном, для двух целых значений
}
```

Как видно из данного примера, нешаблонная функция может вполне мирно сосуществовать с одноименным шаблоном функции, который может быть инстанцирован с тем же типом. При прочих равных условиях процесс распознавания имени функции по типам ее параметров обычно отдает предпочтение нешаблонным версиям, а не тем, которые генерируются на основе шаблонов. В соответствии с этим правилом в четвертом вызове `max()` инстанцирование шаблона не состоится.

```
max(7,42);     // При наличии двух int будет вызвана
               // функция, не являющаяся шаблоном
```

Но если на базе шаблона возможно сгенерировать функцию, которая для данного вызова подходит лучше, то выбор будет сделан в пользу шаблона. Это можно продемонстрировать на примерах второго и третьего вызовов `max()`.

```
max(7.0,42.0);    // Вызов max<double> (вывод
                  // аргументов)
max('a', 'b');    // Вызов max<char> (вывод
                  // аргументов)
```

Можно указать пустой список аргументов шаблона. Такой синтаксис определяет, что вызов можно выполнить только при помощи шаблона, но все параметры шаблона должны определяться на основе аргументов вызова.

```
max<>(7,42);   // Вызов max<int> (вывод аргументов)
```

Поскольку автоматическое преобразование типов для шаблонов невозможно, но вполне применимо для обычных функций, для последнего вызова используется не являющаяся шаблоном функция (при этом и `'a'`, и `42.7` конвертируются в `int`).

```
max('a', 42.7);    // Различные типы аргументов допустимы
                   // только в функции, не являющейся
                   // шаблоном
```

Приведем еще более полезный пример: перегрузка шаблона функции, вычисляющей наибольшее значение для указателей и обычных строк в С-стиле.

```
// basics/max3.cpp
#include <iostream>
#include <cstring>
#include <string>

// Наибольшее из двух значений произвольных типов
template <typename T>
inline T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
// Наибольший из двух указателей
template <typename T>
inline T* const& max(T* const& a, T* const& b)
{
    return *a < *b ? b : a;
}
// Наибольшая из двух С-строк
inline char const* const& max(char const* const& a,
                              char const* const& b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}
int main()
```

```
{
    int a = 7;
    int b = 42;
    ::max(a,b);   // max() для двух значений int

    std::string s="hey";
    std::string t="you";
    ::max(s,t); // max() для двух значений std::string

    int* p1 = &b;
    int* p2 = &a;

    ::max(p1,p2); // max() для двух указателей

    char const* s1 = "David";
    char const* s2 = "Nico";
    ::max(s1,s2); // max() для двух C-строк
}
```

Заметим, что аргументы для всех перегруженных реализаций передаются по ссылке. В общем случае при перегрузке шаблонов функций лучше не вносить изменений больше, чем это необходимо. Изменения следует ограничить числом параметров или числом явно задаваемых параметров шаблона, так как в противном случае возможны неожиданные эффекты. Например, если мы перегружаем шаблон `max()`, которому передаются аргументы по ссылке, для двух C-строк, передаваемых по значению, то для вычисления наибольшей из трех C-строк мы не сможем использовать версию с тремя аргументами.

```
// basics/max3a.cpp
#include <iostream>
#include <cstring>
#include <string>

// Наибольшее из двух значений произвольного типа
// (передача по ссылке)
template <typename T>
inline T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
// Наибольшая из двух C-строк
// (передача по значению)
inline char const* max(char const* a, char const* b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}
// Наибольшее из трех значений произвольного типа
// (передача по ссылке)
```

```
template <typename T>
inline T const& max(T const& a, T const& b, T const& c)
{
    return max(max(a,b),c); // ОШИБКА, если в max(a,b)
                            // используется передача
                            // по значению

int main()
{
    ::max(7,42,68);   // ВЕРНО

    const char* s1 = "frederic";
    const char* s2 = "anica";
    const char* s3 = "lucas";
    ::max(s1,s2,s3))   // ОШИБКА
}
```

Проблема заключается в том, что если мы вызываем `max()` для трех С-строк, то инструкция

```
return max(max(a,b),c);
```

становится некорректной. Это происходит потому, что в `max()` для С-строк создается новая временная локальная переменная, которую функция возвращает по ссылке.

Здесь приведен только один пример кода, который вследствие нюансов правил перегрузки функций работает иначе, чем можно было бы ожидать. Например, может иметь (а может и не иметь) значение то, что не все перегруженные функции являются видимыми в момент вызова соответствующей функции. Так, определение версии `max()` с тремя аргументами для `int` при отсутствии объявления специализированной двухаргументной версии `max()` для `int` приводит к тому, что в трехаргументной версии используется двухаргументный шаблон.

```
// basics/max4.cpp
// Наибольшее из двух значений произвольного типа
template <typename T>
inline T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}

// Наибольшее из трех значений произвольного типа
template <typename T>
inline T const& max(T const& a, T const& b, T const& c)
{
    return max(max(a,b),c); // Используется шаблонная
}                           // версия даже для значений
           // int, поскольку объявление функции для
           // двух int располагается позже данного
```

```
// Максимальное из двух целочисленных значений
inline int const& max(int const& a, int const& b)
{
    return a < b ? b : a;
}
```

Более подробно этот вопрос будет рассмотрен в разделе 9.2, стр. 145, а пока в качестве правила примем следующее: объявления всех перегруженных версий функции следует помещать перед ее вызовом.

2.5. Резюме

- Шаблоны функций определяют семейство функций для разных аргументов шаблона.
- При передаче аргументов шаблона происходит инстанцирование шаблонов функций для данных типов аргументов.
- Параметры шаблонов можно задавать явно.
- Шаблоны функций можно перегружать.
- При перегрузке шаблонов функций следует ограничивать вносимые изменения явным указанием параметров шаблона.
- Следует убедиться, что все перегруженные версии шаблонов функций размещены в программе до вызовов соответствующих функций.