**Шаблоны функций**

Мы обсудим, что такое шаблон функции, как его определять и использовать. Это довольно просто, и многие программисты применяют шаблоны, определённые в стандартной библиотеке, даже не понимая, с чем они работают. Только пользователи, хорошо знающие язык С++, самостоятельно определяют и применяют шаблоны функций так, как здесь описано. Поэтому данный материал следует рассматривать как переход к более сложным аспектам C++. Мы начнём с рассказа о том, что такое шаблон функции и как его определять, затем на простом примере проиллюстрируем использование шаблонов. Далее мы перейдём к темам, требующим больших знаний. Сначала посмотрим на усложнённые примеры применения шаблонов, затем подробно остановимся на выведении (deduction) их аргументов и покажем, как их можно задавать при конкретизации (instantiation) шаблона функции. После этого мы посмотрим, каким образом компилятор конкретизирует шаблоны и какие требования предъявляются в этой связи к организации наших программ, а также обсудим, как определить специализацию для такой конкретизации. Затем в данной главе будут изложены вопросы, представляющие интерес для проектировщиков шаблонов функций. Мы объясним, как можно перегружать шаблоны и как применительно к ним работает разрешение перегрузки. Мы также расскажем о разрешении имён в определениях шаблонов функций и покажем, как можно определять шаблоны в пространствах имён. Завершим развёрнутым примером.

**10.1. Определение шаблона функции**

Иногда может показаться, что сильно типизированный язык создаёт препятствия для реализации совсем простых функций. Например, хотя следующий алгоритм функции min() тривиален, сильная типизация требует, чтобы его разновидности были реализованы для всех типов, которые мы собираемся сравнивать:

int min(int a, int b)

{

return a < b ? a : b;

}

double min(double a, double b)

{

return a < b ? a : b;

}

Заманчивую альтернативу явному определению каждого экземпляра функции min() представляет использование макросов, расширяемых препроцессором:

#define min(a, b) ((a) < (b) ? (a) : (b))

Но этот подход таит в себе потенциальную опасность. Определённый выше макрос правильно работает при простых обращениях к min(), например:

min(10, 20);

min(10.0, 20.0);

но может преподнести сюрпризы в более сложных случаях: такой механизм ведёт себя не как вызов функции, он лишь выполняет текстовую подстановку аргументов. В результате значения обоих аргументов оцениваются дважды: один раз при сравнении a и b, а второй – при вычислении возвращаемого макросом результата:

#include <iostream>

#define min(a,b) ((a) < (b) ? (a) : (b))

const int size = 10;  
int ia[size];

int main() {  
 int elem\_cnt = 0;  
 int \*p = &ia[0];

// подсчитать число элементов массива  
 while (min(p++,&ia[size]) != &ia[size])  
 ++elem\_cnt;

cout << "elem\_cnt : " << elem\_cnt  
 << "\texpecting: " << size << endl;  
 return 0;  
}

На первый взгляд, эта программа подсчитывает количество элементов в массиве ia целых чисел. Но в этом случае макрос min() расширяется неверно, поскольку операция постинкремента применяется к аргументу-указателю дважды при каждой подстановке. В результате программа печатает строку, свидетельствующую о неправильных вычислениях:

elem\_cnt : 5 expecting: 10

Шаблоны функций предоставляют в наше распоряжение механизм, с помощью которого можно сохранить семантику определений и вызовов функций (инкапсуляция фрагмента кода в одном месте программы и гарантированно однократное вычисление аргументов), не принося в жертву сильную типизацию языка C++, как в случае применения макросов.  
Шаблон даёт алгоритм, используемый для автоматической генерации экземпляров функций с различными типами. Программист параметризует все или только некоторые типы в интерфейсе функции (т.е. типы формальных параметров и возвращаемого значения), оставляя её тело неизменным. Функция хорошо подходит на роль шаблона, если её реализация остается инвариантной на некотором множестве экземпляров, различающихся типами данных, как, скажем, в случае min().

Так определяется шаблон функции min():

template <class Type>

Type min2(Type a, Type b) {

return a < b ? a : b;

}

int main() {  
 // правильно: min(int, int);  
 min(10, 20);

// правильно: min(double, double);  
 min(10.0, 20.0);  
 return 0;  
}

Если вместо макроса препроцессора min() подставить в текст предыдущей программы этот шаблон, то результат будет правильным:

elem\_cnt : 10 expecting: 10

В стандартной библиотеке C++ есть шаблоны функций для многих часто используемых алгоритмов, например для min(). Здесь мы приводим собственные упрощённые версии некоторых алгоритмов из стандартной библиотеки.

Как объявление, так и определение шаблона функции всегда должны начинаться с ключевого слова **template**, за которым следует список разделённых запятыми идентификаторов, заключённый в угловые скобки '<' и '>', – список параметров шаблона, обязательно непустой. У шаблона могут быть параметры-типы, представляющие некоторый тип, и параметры-константы, представляющие фиксированное константное выражение.

Параметр-тип состоит из ключевого слова class или ключевого слова typename, за которым следует идентификатор. Эти слова всегда обозначают, что последующее имя относится к встроенному или определённому пользователем типу. Имя параметра шаблона выбирает программист. В приведённом примере мы использовали имя Type, но могли выбрать и любое другое:

template <class Glorp>

Glorp min2(Glorp a, Glorp b) {

return a < b ? a : b;

}

При конкретизации (порождении конкретного экземпляра) шаблона вместо параметра-типа подставляется фактический встроенный или определенный пользователем тип. Любой из типов int, double, char\*, vector<int> или list<double> является допустимым аргументом шаблона.  
Параметр-константа выглядит как обычное объявление. Он говорит о том, что вместо имени параметра должно быть подставлено значение константы из определения шаблона. Например, size – это параметр-константа, который представляет размер массива arr:

template <class Type, int size>

Type min(Type (&arr) [size]);

Вслед за списком параметров шаблона идет объявление или определение функции. Если не обращать внимания на присутствие параметров в виде спецификаторов типа или констант, то определение шаблона функции выглядит точно так же, как и для обычных функций:

template <class Type, int size>  
Type min(const Type (&r\_array)[size])  
{  
 /\* параметризованная функция для поиска минимального значения в массиве \*/  
 Type min\_val = r\_array[0];  
 for (int i = 1; i < size; ++i)  
 if (r\_array[i] < min\_val)  
 min\_val = r\_array[i];  
 return min\_val;  
}

В этом примере Type определяет тип значения, возвращаемого функцией min(), тип параметра r\_array и тип локальной переменной min\_val; size задает размер массива r\_array. В ходе работы программы при использовании функции min() вместо Type могут быть подставлены любые встроенные и определённые пользователем типы, а вместо size – те или иные константные выражения. (Напомним, что работать с функцией можно двояко: вызвать её или взять её адрес).

Процесс подстановки типов и значений вместо параметров называется конкретизацией шаблона. Список параметров нашей функции min() может показаться чересчур коротким. Когда параметром является массив, передаётся указатель на его первый элемент, первая же размерность фактического аргумента-массива внутри определения функции неизвестна. Чтобы обойти эту трудность, мы объявили первый параметр min() как ссылку на массив, а второй – как его размер. Недостаток подобного подхода в том, что при использовании шаблона с массивами одного и того же типа int, но разных размеров генерируются (или конкретизируются) различные экземпляры функции min().

Имя параметра разрешено употреблять внутри объявления или определения шаблона. Параметр-тип служит спецификатором типа; его можно использовать точно так же, как спецификатор любого встроенного или пользовательского типа, например, в объявлении переменных или в операциях приведения типов. Параметр-константа применяется как константное значение – там, где требуются константные выражения, например, для задания размера в объявлении массива или в качестве начального значения элемента перечисления.

// size определяет размер параметра-массива и инициализирует

// переменную типа const int

template <class Type, int size>

Type min(const Type (&r\_array)[size])

{

const int loc\_size = size;

Type loc\_array[loc\_size];

// ...

}

Если в глобальной области видимости объявлен объект, функция или тип с тем же именем, что у параметра шаблона, то глобальное имя оказывается скрытым. В следующем примере тип переменной tmp не double, а тот, что у параметра шаблона Type:

typedef double Type;

template <class Type>

Type min(Type a, Type b)

{

// tmp имеет тот же тип, что параметр шаблона Type, а не заданный

// глобальным typedef

Type tm = a < b ? a : b;

return tmp;

}

Объект или тип, объявленные внутри определения шаблона функции, не могут иметь то же имя, что и какой-то из параметров:

template <class Type>

Type min(Type a, Type b)

{

// ошибка: повторное объявление имени Type, совпадающего с именем

// параметра шаблона

typedef double Type;

Type tmp = a < b ? a : b;

return tmp;

}

Имя параметра-типа шаблона можно использовать для задания типа возвращаемого значения:

// правильно: T1 представляет тип значения, возвращаемого min(),

// а T2 и T3 – параметры-типы этой функции

template <class T1, class T2, class T3>

T1 min(T2, T3);

В одном списке параметров некоторое имя разрешается употреблять только один раз. Например, следующее определение будет помечено как ошибка компиляции:

// ошибка: неправильное повторное использование имени параметра Type

template <class Type, class Type>

Type min(Type, Type);

Однако одно и то же имя можно многократно применять внутри объявления или определения шаблона:

// правильно: повторное использование имени Type внутри шаблона

template <class Type>

Type min(Type, Type);

template <class Type>

Type max(Type, Type);

Имена параметров в объявлении и определении не обязаны совпадать. Так, все три объявления min() относятся к одному и тому же шаблону функции:

// все три объявления min() относятся к одному и тому же шаблону функции

// опережающие объявления шаблона  
template <class T> T min(T, T);  
template <class U> U min(U, U);

// фактическое определение шаблона  
template <class Type>  
 Type min(Type a, Type b) { /\* ... \*/ }

Количество появлений одного и того же параметра шаблона в списке параметров функции не ограничено. В следующем примере Type используется для представления двух разных параметров:

#include <vector>

// правильно: Type используется неоднократно в списке параметров шаблона

template <class Type>

Type sum(const vector<Type> &, Type);

Если шаблон функции имеет несколько параметров-типов, то каждому из них должно предшествовать ключевое слово class или typename:

// правильно: ключевые слова typename и class могут перемежаться

template <typename T, class U>

T minus(T\*, U);

// ошибка: должно быть <typename T, class U> или  
// <typename T, typename U>  
template <typename T, U>  
 T sum(T\*, U);

В списке параметров шаблона функции ключевые слова typename и class имеют одинаковый смысл и, следовательно, взаимозаменяемы. Любое из них может использоваться для объявления разных параметров-типов шаблона в одном и том же списке (как было продемонстрировано на примере шаблона функции minus()). Для обозначения параметра-типа более естественно, на первый взгляд, употреблять ключевое слово typename, а не class, ведь оно ясно указывает, что за ним следует имя типа. Однако это слово было добавлено в язык, как часть стандарта C++, поэтому в старых программах вы скорее всего встретите слово class (не говоря уже о том, что слово class короче, чем typename, а человек по своей природе существо ленивое).

Ключевое слово typename упрощает разбор определений шаблонов. (Мы лишь кратко остановимся на том, зачем оно понадобилось. Желающим узнать об этом подробнее рекомендуем обратиться к книге Страуструпа “Design and Evolution of C++”.)

При таком разборе компилятор должен отличать выражения-типы от тех, которые таковыми не являются; выявить это не всегда возможно. Например, если компилятор встречает в определении шаблона выражение Parm::name и если Parm – это параметр-тип, представляющий класс, то следует ли считать, что name представляет член-тип класса Parm?

template <class Parm, class U>

Parm minus(Parm\* array, U value)

{

Parm::name \* p; // это объявление указателя или умножение?

// На самом деле умножение

}

Компилятор не знает, является ли name типом, поскольку определение класса, представленного параметром Parm, недоступно до момента конкретизации шаблона. Чтобы такое определение шаблона можно было разобрать, пользователь должен подсказать компилятору, какие выражения включают типы. Для этого служит ключевое слово typename. Например, если мы хотим, чтобы выражение Parm::name в шаблоне функции minus() было именем типа и, следовательно, вся строка трактовалась как объявление указателя, то нужно модифицировать текст следующим образом:

template <class Parm, class U>

Parm minus(Parm\* array, U value)

{

typename Parm::name \* p; // теперь это объявление указателя

}

Ключевое слово typename используется также в списке параметров шаблона для указания того, что параметр является типом.

Шаблон функции можно объявлять как inline или extern – как и обычную функцию. Спецификатор помещается после списка параметров, а не перед словом template.

// правильно: спецификатор после списка параметров

template <typename Type>

inline

Type min(Type, Type);

// ошибка: спецификатор inline не на месте  
 inline  
 template <typename Type>  
 Type min(Array<Type>, int);

**Упражнение 1**

Определите, какие из данных определений шаблонов функций неправильны. Исправьте ошибки.

(a) template <class T, U, class V>

void foo(T, U, V);

(b) template <class T>

T foo(int \*T);

(c) template <class T1, typename T2, class T3>

T1 foo(T2, T3);

(d) inline template <typename T>

T foo(T, unsigned int\*);

(e) template <class myT, class myT>

void foo(myT, myT);

(f) template <class T>

foo(T, T);

(g) typedef char Ctype;

template <class Ctype>

Ctype foo(Ctype a, Ctype b);

**Упражнение 2**

Какие из повторных объявлений шаблонов ошибочны? Почему?

(a) template <class Type>

Type bar(Type, Type);

template <class Type>

Type bar(Type, Type);

(b) template <class T1, class T2>

void bar(T1, T2);

template <typename C1, typename C2>

void bar(C1, C2);

**Упражнение 3**

Перепишите функцию putValues() из раздела 7.3.3 в виде шаблона. Параметризуйте его так, чтобы было два параметра шаблона (для типа элементов массива и для размера массива) и один параметр функции, являющийся ссылкой на массив. Напишите определение шаблона функции.

**10.2. Конкретизация шаблона функции**

Шаблон функции описывает, как следует строить конкретные функции, если задано множество фактических типов или значений. Процесс конструирования называется конкретизацией шаблона. Выполняется он неявно, как побочный эффект вызова или взятия адреса шаблона функции. Например, в следующей программе min() конкретизируется дважды: один раз для массива из пяти элементов типа int, а другой – для массива из шести элементов типа double:

массива из шести элементов типа double:  
// определение шаблона функции min()  
// с параметром-типом Type и параметром-константой size  
template <typename Type, int size>  
 Type min(Type (&r\_array)[size])  
{  
 Type min\_val = r\_array[0];  
 for (int i = 1; i < size; ++i)  
 if (r\_array[i] < min\_val)  
 min\_val = r\_array[i];  
 return min\_val;  
}  
// size не задан -- ok  
// size = число элементов в списке инициализации  
int ia[] = { 10, 7, 14, 3, 25 };  
double da[6] = { 10.2, 7.1, 14.5, 3.2, 25.0, 16.8 };  
#include <iostream>  
int main()  
{  
 // конкретизация min() для массива из 5 элементов типа int  
 // подставляется Type => int, size => 5  
 int i = min(ia);  
 if (i != 3)  
 cout << "??oops: integer min() failed\n";  
 else cout << "!!ok: integer min() worked\n";  
 // конкретизация min() для массива из 6 элементов типа double  
 // подставляется Type => double, size => 6  
 double d = min(da);  
 if (d != 3.2)  
 cout << "??oops: double min() failed\n";  
 else cout << "!!ok: double min() worked\n";  
 return 0;  
}

Вызов

int i = min(ia);

приводит к конкретизации следующего экземпляра функции min(), в котором Type заменено на int, а size на 5:

int min(int (&r\_array)[5])

{

int min\_val = r\_array[0];

for (int i = 1; i < 5; ++i)

if (r\_array[i] < min\_val)

min\_val = r\_array[i];

return min\_val;  
}

Аналогично вызов

double d = min(da);

конкретизирует экземпляр min(), в котором Type заменено на double, а size на 6:  
В качестве формальных параметров шаблона функции используются параметр-тип и параметр-константа. Для определения фактического типа и значения константы, которые надо подставить в шаблон, исследуются фактические аргументы, переданные при вызове функции. В нашем примере для идентификации аргументов шаблона при конкретизации используются тип ia (массив из пяти int) и da (массив из шести double). Процесс определения типов и значений аргументов шаблона по известным фактическим аргументам функции называется выведением (deduction) аргументов шаблона. (В следующем разделе мы расскажем об этом подробнее. А в разделе 10.4 речь пойдет о возможности явного задания аргументов.)

Шаблон конкретизируется либо при вызове, либо при взятии адреса функции. В следующем примере указатель pf инициализируется адресом конкретизированного экземпляра шаблона. Его аргументы определяются путем исследования типа параметра функции, на которую указывает pf:

template <typename Type, int size>

Type min(Type (&p\_array)[size]) { /\* ... \*/ }

// pf указывает на int min(int (&)[10])

int (\*pf)(int (&)[10]) = &min;

Тип pf – это указатель на функцию с параметром типа int(&)[10], который определяет тип аргумента шаблона Type и значение аргумента шаблона size при конкретизации min(). Аргумент шаблона Type будет иметь тип int, а значением аргумента шаблона size будет 10. Конкретизированная функция представляется как min(int(&)[10]), и указатель pf адресует именно ее.  
Когда берется адрес шаблона функции, контекст должен быть таким, чтобы можно было однозначно определить типы и значения аргументов шаблона. Если сделать это не удается, компилятор выдает сообщение об ошибке:

template <typename Type, int size>

Type min(Type (&r\_array)[size]) { /\* ... \*/ }

typedef int (&rai)[10];  
typedef double (&rad)[20];

void func(int (\*)(rai));  
void func(double (\*)(rad));

int main() {

// ошибка: как конкретизировать min()?  
 func(&min);  
}

Функция func() перегружена и тип её параметра не позволяет однозначно определить ни аргумент шаблона Type, ни значение аргумента шаблона size. Результатом конкретизации вызова func() может быть любая из следующих функций:

min(int (\*)(int(&)[10]))

min(double (\*)(double(&)[20]))

Поскольку однозначно определить аргументы функции func() нельзя, взятие адреса конкретизированного шаблона в таком контексте приводит к ошибке компиляции.  
Этого можно избежать, если использовать явное приведение типов для указания типа аргумента:

int main() {

// правильно: с помощью явного приведения указывается тип аргумента

func(static\_cast< double(\*)(rad) >(&min));

}

Лучше, однако, применять явное задание аргументов шаблона, как будет показано в разделе 10.4.

**10.3. Вывод аргументов шаблона А**

При вызове шаблона функции типы и значения его аргументов определяются путем исследования типов фактических аргументов функции. Этот процесс называется выводом аргументов шаблона.  
Параметром функции в шаблоне min() является ссылка на массив элементов типа Type:

template <class Type, int size>

Type min(Type (&r\_array)[size]) { /\* ... \*/ }

Для сопоставления с формальным параметром функции фактический аргумент также должен быть l-значением, представляющим тип массива. Следующий вызов ошибочен, так как pval имеет тип int\*, а не является l-значением типа “массив int”.

void f(int pval[9]) {

// ошибка: Type (&)[] != int\*

int jval = min(pval);

}

При выводе аргументов шаблона не принимается во внимание тип значения, возвращаемого конкретизированным шаблоном функции. Например, если вызов min() записан так:

double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };

int i1 = min(da);

то конкретизированный экземпляр min() имеет параметр типа “указатель на массив из восьми double” и возвращает значение типа double. Перед инициализацией i1 это значение приводится к типу int. Однако тот факт, что результат вызова min() используется для инициализации объекта типа int, не влияет на вывод аргументов шаблона.  
Чтобы процесс такого вывода завершился успешно, тип фактического аргумента функции не обязательно должен совпадать с типом соответствующего формального параметра. Допустимы три вида преобразований типа: трансформация l-значения, преобразование спецификаторов и приведение к базовому классу, конкретизированному из шаблона класса. Рассмотрим последовательно каждое из них.  
Напомним, что трансформация l-значения – это либо преобразование l-значения в r-значение, либо преобразование массива в указатель, либо преобразование функции в указатель (все они рассматривались в разделе [9.3](http://valera.asf.ru/cpp/book/c09.html)). Для иллюстрации влияния такой трансформации на вывод аргументов шаблона рассмотрим функцию min2() c одним параметром шаблона Type и двумя параметрами функции. Первый параметр min2() – это указатель на тип Type\*. size теперь не является параметром шаблона, как в определении min(), вместо этого он стал параметром функции, а его значение должно быть явно передано при вызове:

template <class Type>

// первый параметр имеет тип Type\*

Type min2(Type\* array, int size)

{

Type min\_val = array[0];

for (int i = 1; i < size; ++i)

if (array[i] < min\_val)

min\_val = array[i];

return min\_val;  
}

min2() можно вызвать, передав в качестве первого аргумента массив из четырех int, как в следующем примере:

int ai[4] = { 12, 8, 73, 45 };

int main() {  
 int size = sizeof (ai) / sizeof (ai[0]);

// правильно: преобразование массива в указатель  
 min2(ai, size);  
}

Фактический аргумент функции ai имеет тип “массив из четырех int” и не совпадает с типом соответствующего формального параметра Type\*. Однако, поскольку преобразование массива в указатель допустимо, то аргумент ai приводится к типу int\* еще до вывода аргумента шаблона Type, для которого затем выводится тип int, и шаблон конкретизирует функцию min2(int\*, int).  
Преобразование спецификаторов добавляет const или volatile к указателям (такие трансформации также рассматривались в [разделе 9.3](http://valera.asf.ru/cpp/book/c09.html)). Для иллюстрации влияния преобразования спецификаторов на вывод аргументов шаблона рассмотрим min3() с первым параметром функции типа const Type\*:

template <class Type>

// первый параметр имеет тип const Type\*

Type min3(const Type\* array, int size) {

// ...

}

min3() можно вызвать, передав int\* в качестве первого фактического аргумента, как в следующем примере:

int \*pi = &ai;

// правильно: приведение спецификаторов к типу const int\*

int i = min3(pi, 4);

Фактический аргумент функции pi имеет тип “указатель на int” и не совпадает с типом формального параметра const Type\*. Однако, поскольку преобразование спецификаторов допустимо, то он приводится к типу const int\* еще до вывода аргумента шаблона Type, для которого затем выводится тип int, и шаблон конкретизирует функцию min3(const int\*, int).  
Теперь обратимся к преобразованию в базовый класс, конкретизированный из шаблона класса. Вывод аргументов шаблона можно выполнить, если тип формального параметра функции является таким шаблоном, а фактический аргумент – базовый класс, конкретизированный из него. Чтобы проиллюстрировать такое преобразование, рассмотрим новый шаблон функции min4() с параметром типа Array<Type>&, где Array – это шаблон класса, определенный в разделе [2.5](http://valera.asf.ru/cpp/book/c02.html). (В главе 16 шаблоны классов обсуждаются во всех деталях.)

template <class Type>

class Array { /\* ... \*/ }

template <class Type>  
 Type min4(Array<Type>& array)  
{  
 Type min\_val = array[0];  
 for (int i = 1; i < array.size(); ++i)  
 if (array[i] < min\_val)  
 min\_val = array[i];

return min\_val;  
}

min4() можно вызвать, передав в качестве первого аргумента ArrayRC<int>, как показано в следующем примере. (ArrayRC – это шаблон класса, также определенный в главе [2](http://valera.asf.ru/cpp/book/c02.html); наследование классов подробно рассматривается в главах 17 и 18.)

template <class Type>

class ArrayRC : public Array<Type> { /\* ... \*/ };

int main() {  
 ArrayRC<int> ia\_rc(10);  
 min4(ia\_rc);  
}

Фактический аргумент ia\_rc имеет тип ArrayRC<int>. Он не совпадает с типом формального параметра Array<Type>&. Но одним из базовых классов для ArrayRC<int> является Array<int>, так как он конкретизирован из шаблона класса, указанного в качестве формального параметра функции. Поскольку фактический аргумент является производным классом, то его можно использовать при выводе аргументов шаблона. Таким образом, перед выводом аргумент функции ArrayRC<int> преобразуется в тип Array<int>, после чего для аргумента шаблона Type выводится тип int и конкретизируется функция min4(Array<int>&).  
В процессе вывода одного аргумента шаблона могут принимать участие несколько аргументов функции. Если параметр шаблона встречается в списке параметров функции более одного раза, то каждый выведенный тип должен точно соответствовать типу, выведенному для того же аргумента шаблона в первый раз:

template <class T> T min5(T, T) { /\* ... \*/ }

unsigned int ui;

int main() {  
 // ошибка: нельзя конкретизировать min5(unsigned int, int)  
 // должно быть: min5(unsigned int, unsigned int) или  
 // min5(int, int)  
 min5(ui, 1024);  
}

Оба фактических аргумента функции должны иметь один и тот же тип: либо int, либо unsigned int, поскольку в шаблоне они принадлежат к одному типу T. Аргумент шаблона T, выведенный из первого аргумента функции, – это int. Аргумент же шаблона T, выведенный из второго аргумента функции, – это unsigned int. Поскольку они оказались разными, процесс вывода завершается неудачей и при конкретизации шаблона выдается сообщение об ошибке. (Избежать её можно, если явно задать аргументы шаблона при вызове функции min5(). В разделе 10.4 мы увидим, как это делается.)  
Ограничение на допустимые типы преобразований относится только к тем фактическим параметрам функции, которые принимают участие в выводе аргументов шаблона. К остальным аргументам могут применяться любые трансформации. В следующем шаблоне функции sum() есть два формальных параметра. Фактический аргумент op1 для первого параметра участвует в выводе аргумента Type шаблона, а второй фактический аргумент op2 – нет.

template <class Type>

Type sum(Type op1, int op2) { /\* ... \*/ }

Поэтому при конкретизации шаблона функции sum() его можно подвергать любым трансформациям. (Преобразования типов, применимые к фактическим аргументам функции, описываются в разделе 9.3.) Например:

int ai[] = { ... };

double dd;

int main() {

// конкретизируется sum(int, int)

sum(ai[0], dd);

}

Тип второго фактического аргумента функции dd не соответствует типу формального параметра int. Но это не мешает конкретизировать шаблон функции sum(), поскольку тип второго аргумента фиксирован и не зависит от параметров шаблона. Для этого вызова конкретизируется функция sum(int,int). Аргумент dd приводится к типу int с помощью преобразования целого типа в тип с плавающей точкой.  
Таким образом, общий алгоритм вывода аргументов шаблона можно сформулировать следующим образом:

1. По очереди исследуется каждый фактический аргумент функции, чтобы выяснить, присутствует ли в соответствующем формальном параметре какой-нибудь параметр шаблона.
2. Если параметр шаблона найден, то путем анализа типа фактического аргумента выводится соответствующий аргумент шаблона.
3. Тип фактического аргумента функции не обязан точно соответствовать типу формального параметра. Для приведения типов могут быть применены следующие преобразования:
   1. трансформации l-значения
   2. преобразования спецификаторов
   3. приведение производного класса к базовому при условии, что формальный параметр функции имеет вид T<args>& или T<args>\*, где список аргументов args содержит хотя бы один параметр шаблона.
4. Если один и тот же параметр шаблона найден в нескольких формальных параметрах функций, то аргумент шаблона, выведенный по каждому из соответствующих фактических аргументов, должен быть одним и тем же.

**Упражнение 10.4**

Назовите два типа преобразований, которые можно применять к фактическим аргументам функций, участвующим в процессе вывода аргументов шаблона.

**Упражнение 10.5**

Пусть даны следующие определения шаблонов:

template <class Type>

Type min3(const Type\* array, int size) { /\* ... \*/ }

template <class Type>

Type min5(Type p1, Type p2) { /\* ... \*/ }

Какие из приведенных ниже вызовов ошибочны? Почему?

double dobj1, dobj2;

float fobj1, fobj2;

char cobj1, cobj2;

int ai[5] = { 511, 16, 8, 63, 34 };

(a) min5(cobj2, 'c');

(b) min5(dobj1, fobj1);

(c) min3(ai, cobj1);

**10.4. Явное задание аргументов шаблона A**

В некоторых ситуациях автоматически вывести типы аргументов шаблона невозможно. Как мы видели на примере шаблона функции min5(), если процесс вывода дает два различных типа для одного и того же параметра шаблона, то компилятор сообщает об ошибке – неудачном выводе аргументов.  
В таких ситуациях приходится подавлять механизм вывода и задавать аргументы явно, указывая их с помощью заключенного в угловые скобки списка разделенных запятыми значений, который следует после имени конкретизируемого шаблона функции. Например, если мы хотим задать тип unsigned int в качестве значения аргумента шаблона T в рассмотренном выше примере использования min5(), то нужно записать вызов конкретизируемого шаблона так:

// конкретизируется min5(unsigned int, unsigned int)

min5< unsigned int >(ui, 1024);

В этом случае список аргументов шаблона <unsigned int> явно задает их типы. Поскольку аргумент шаблона теперь известен, вызов функции больше не приводит к ошибке.  
Обратите внимание, что при вызове функции min5() второй аргумент равен 1024, т.е. имеет тип int. Так как тип второго формального параметра функции при явном задании аргумента шаблона установлен в unsigned int, то второй фактический параметр функции приводится к типу unsigned int с помощью стандартного преобразования целых типов.  
В предыдущем разделе мы говорили, что в процессе вывода аргументов шаблона к фактическим аргументам функции разрешается применять только ограниченное множество преобразований типов. Трансформация int в unsigned int в это множество не входит. Но если аргументы шаблона задаются явно, выполнять вывод типов не нужно, поскольку они уже зафиксированы. Следовательно, при явном задании аргументов шаблона для приведения типов фактических аргументов функции к типам формальных параметров можно применять любые стандартные преобразования.  
  
Помимо разрешения любых преобразований фактических аргументов функции, явное задание аргументов шаблона помогает избежать и других проблем, встающих перед программистом. Рассмотрим следующую задачу. Мы хотим определить шаблон функции с именем sum() так, чтобы его конкретизация возвращала значения типа, достаточно большого для представления суммы двух значений любых двух типов, переданных в любом порядке. Как это сделать? Какой тип возвращаемого значения следует задать?

// каким должен быть тип возвращаемого значения: T или U

template <class T, class U>

??? sum(T, U);

В нашем случае нельзя использовать ни тот, ни другой параметрический тип, иначе мы неизбежно допустим ошибку:

char ch; unsigned int ui;

// ни T, ни U нельзя использовать в качестве типа возвращаемого значения  
sum(ch, ui); // правильно: U sum(T, U);  
sum(ui, ch); // правильно: T sum(T, U);

Решение заключается в том, чтобы ввести в шаблон третий параметр для обозначения типа возвращаемого значения:

// T1 не появляется в списке параметров шаблона функции

template <class T1, class T2, class T3>

T1 sum(T2, T3);

Поскольку тип возвращаемого значения может отличаться от типов аргументов функции, T1 не упоминается в списке формальных параметров. Это потенциальная проблема, так как тип T1 не может быть выведен из фактических аргументов функции. Однако, если при конкретизации sum() мы зададим аргументы шаблона явно, то избегнем сообщения компилятора о невозможности вывести T1. Например:

typedef unsigned int ui\_type;

ui\_type calc(char ch, ui\_type ui) {

// ...  
 // ошибка: невозможно вывести T1  
 ui\_type loc1 = sum(ch, ui);

// правильно: аргументы шаблона заданы явно  
 // T1 и T3 - это unsigned int, T2 - это char  
 ui\_type loc2 = sum< ui\_type, ui\_type >(ch, ui);  
}

Не хватает возможности явно задать T1, но не T2 и T3, поскольку их можно вывести из аргументов функции при вызове.  
При явном задании аргументов шаблона необходимо перечислять только те, которые не могут быть выведены автоматически. Но, как и в случае аргументов функции со значениями по умолчанию, опускать можно исключительно “хвостовые”:

// правильно: T3 - это unsigned int

// T3 выведен из типа ui

ui\_type loc3 = sum< ui\_type, char >(ch, ui);

// правильно: T2 - это char, T3 - unsigned int

// T2 и T3 выведены из типа pf

ui\_type (\*pf)(char, ui\_type) = &sum< ui\_type >;

// ошибка: опускать можно только “хвостовые” аргументы

ui\_type loc4 = sum< ui\_type, , ui\_type >(ch, ui);

Встречаются ситуации, когда невозможно вывести аргументы шаблона в контексте, где конкретизируется шаблон функции; следовательно, необходимо их явно задать. Именно выявление таких ситуаций и необходимость решить проблему послужила причиной поддержки явного задания аргументов шаблона в стандартном C++.  
В следующем примере берется адрес конкретизированной функции sum() и передается в качестве аргумента перегруженной функции manipulate(). Как мы показали в разделе 10.2, невозможно понять, как именно нужно конкретизировать sum(), если есть только списки параметров функций manipulate(). Имеется две разных функции sum(), и обе удовлетворяют условиям вызова. Следовательно, вызов manipulate() неоднозначен. Одним из способов разрешения такой неоднозначности является явное приведение типов. Однако лучше использовать явное задание аргументов шаблона: оно позволяет указать, как именно конкретизировать sum(), и, следовательно, выбрать нужный вариант перегруженной функции manipulate(). Например:

template <class T1, class T2, class T3>

T1 sum(T2 op1, T3 op2) { /\* ... \*/ }

void manipulate(int (\*pf)(int,char));  
void manipulate(double (\*pf)(float,float));

int main()  
{  
 // ошибка: какой из возможных экземпляров sum:  
 // int sum(int,char) или double sum(float, float)?

manipulate(&sum);

// берется адрес конкретизированного экземпляра  
 // double sum(float, float)  
 // вызывается: void manipulate(double (\*pf)(float, float));  
 manipulate(&sum< double, float, float >);  
}

Отметим, что явное задание аргументов шаблона следует использовать только тогда, когда это абсолютно необходимо для разрешения неоднозначности или для конкретизации шаблона функции в контексте, где вывести аргументы невозможно. Во-первых, определение типов и значений аргументов шаблона проще оставить компилятору. А во-вторых, если мы модифицируем объявления в программе, так что типы аргументов функции при вызове конкретизированного шаблона изменятся, то компилятор автоматически скорректирует вызов без нашего вмешательства. С другой стороны, если аргументы шаблона заданы явно, необходимо проверить, что они по-прежнему отвечают новым типам аргументов функции. Поэтому мы рекомендуем избегать явного задания аргументов шаблона.

**Упражнение 10.6**

Назовите две ситуации, когда использование явного задания аргументов шаблона необходимо.

**Упражнение 10.7**

Пусть дано следующее определение шаблона функции sum():

template <class T1, class T2, class T3>

T1 sum(T2, T3);

Какие из приведённых ниже вызовов ошибочны? Почему?

double dobj1, dobj2;

float fobj1, fobj2;

char cobj1, cobj2;

(a) sum(dobj1, dobj2);

(b) sum<double,double,double>(fobj1, fobj2);

(c) sum<int>(cobj1, cobj2);

(d) sum<double, ,double>(fobj2, dobj2);

**10.5. Модели компиляции шаблонов А**

Шаблон функции задает алгоритм для построения определений множества экземпляров функций. Сам шаблон не определяет никакой функции. Например, когда компилятор видит шаблон:

template <typename Type>

Type min(Type t1, Type t2)

{

return t1 < t2 ? t1 : t2;

}

он сохраняет внутреннее представление min(), но и только. Позже, когда встретится её реальное использование, скажем:

int i, j;

double dobj = min(i, j);

компилятор строит определение min() по сохраненному внутреннему представлению.  
Здесь возникает несколько вопросов. Чтобы компилятор мог конкретизировать шаблон функции, должно ли его определение быть видимо при вызове экземпляра этой функции? Например, нужно ли определению шаблона min() появиться до её конкретизации c целыми параметрами при инициализации dobj? Следует ли помещать шаблоны в заголовочные файлы, как мы поступаем с определениями встроенных (inline) функций? Или в заголовочные файлы можно помещать только объявления шаблонов, оставляя определения в файлах исходных текстов?  
Чтобы ответить на эти вопросы, нам придется объяснить принятую в C++ модель компиляции шаблонов, сформулировать требования к организации определений и объявлений шаблонов в программах. В C++ поддерживаются две таких модели: модель с включением и модель с разделением. В данном разделе описываются обе модели и объясняется их использование.

**10.5.1. Модель компиляции с включением**

Согласно этой модели мы включаем определение шаблона в каждый файл, где этот шаблон конкретизируется. Обычно оно помещается в заголовочный файл, как и для встроенных функций. Именно такой моделью мы пользуемся в нашей книге. Например:

// model1.h

// модель с включением:

// определения шаблонов помещаются в заголовочный файл

template <typename Type>  
 Type min(Type t1, Type t2) {

return t1 < t2 ? t1 : t2;  
}

Этот заголовочный файл включается в каждый файл, где конкретизируется функция min():

// определения шаблонов включены раньше

// используется конкретизация шаблона

#include "model1.h"

int i, j;  
double dobj = min(i, j);

Заголовочный файл можно включить в несколько файлов с исходными текстами программы. Означает ли это, что компилятор конкретизирует экземпляр функции min() с целыми параметрами в каждом файле, где имеется обращение к ней? Нет. Программа должна вести себя так, словно min() с целыми параметрами определена только один раз. Где и когда в действительности конкретизируется шаблон функции, оставляется на усмотрение разработчика компилятора. Нам достаточно знать, что где-то в программе нужная функция min() была конкретизирована. (Как мы покажем далее, с помощью явного объявления конкретизации можно указать, где и когда оно должно быть выполнено. Такие объявления желательно использовать на поздних стадиях разработки продукта для улучшения производительности.)  
Решение включать определения шаблонов функций в заголовочные файлы не всегда удачно. Тело шаблона описывает детали реализации, которые пользователям не интересны или которые мы хотели бы от них скрыть. В действительности, если определение шаблона велико, то количество кода в заголовочном файле может превысить разумные пределы. Кроме того, многократная компиляция одного и того же определения при обработке разных файлов увеличивает общее время компиляции программы. Отделить объявления шаблонов функций от их определений позволяет модель компиляции с разделением. Посмотрим, как её можно использовать

**10.5.2. Модель компиляции с разделением**

Согласно этой модели объявления шаблонов функций помещаются в заголовочный файл, а определения – в файл с исходным текстом программы, т.е. объявления и определения шаблонов организованы так же, как в случае с невстроенными (non-inline) функциями. Например:

// model2.h

// модель с разделением

// сюда помещается только объявление шаблона

template <typename Type> Type min(Type t1, Type t2);

// model2.C  
// определение шаблона  
export template <typename Type>  
 Type min(Type t1, Type t2) { /\* ... \*/ }

Программа, которая конкретизирует шаблон функции min(), должна предварительно включить этот заголовочный файл:

// user.C

#include "model2.h"

int i, j;  
double d = min (i, j); // правильно: здесь производится конкретизация

Хотя определение шаблона функции min() не видно в файле user.c, конкретизацию min(int,int) произвести можно. Но для этого шаблон min() должен быть определен специальным образом. Вы уже заметили, как именно? Если вы внимательно посмотрите на файл model2.c, то увидите, что определению шаблона функции min() предшествует ключевое слово export. Таким образом, шаблон min() становится экспортируемым. Слово export говорит компилятору, что данное определение шаблона может понадобиться для конкретизации функций в других файлах. В таком случае компилятор должен гарантировать, что это определение будет доступно во время конкретизации.  
  
Для объявления экспортируемого шаблона перед ключевым словом template в его определении надо поместить слово export. Если шаблон экспортируется, то его разрешается конкретизировать в любом исходном файле программы – для этого нужно лишь объявить его перед использованием. Если слово export перед определением опущено, то компилятор может и не конкретизировать экземпляр функции min() с целыми параметрами и нам не удастся связать программу.  
  
Обратите внимание, что в некоторых реализациях это ключевое слово не нужно, поскольку поддерживается расширение языка, согласно которому неэкспортированный шаблон функции может встречаться только в одном исходном файле, при этом экземпляры такого шаблона в других файлах конкретизируются правильно. Однако подобное поведение не соответствует стандарту, который требует, чтобы пользователь всегда помечал определения шаблонов функций как экспортируемые, если объявление шаблона видно в исходном файле до его конкретизации.  
Ключевое слово export в объявлении шаблона, находящемся в заголовочном файле, можно опустить. Так, в объявлении min() в файле model2.h этого слова нет.  
Шаблон функции должен быть определен как экспортируемый только один раз во всей программе. К сожалению, поскольку компилятор обрабатывает файлы один за другим, он обычно не замечает, что шаблон определен как экспортируемый в нескольких исходных файлах. В результате подобного недосмотра может произойти следующее:

* при редактировании связей возникает ошибка, показывающая, что шаблон функции определен более, чем в одном файле;
* компилятор несколько раз конкретизирует шаблон функции с одним и тем же множеством аргументов, что приводит к ошибке повторного определения функции при связывании программы;
* компилятор может конкретизировать шаблон с помощью одного из его экспортированных определений, игнорируя все остальные.

Нельзя с уверенностью утверждать, что наличие в программе нескольких экспортируемых определений шаблона функции обязательно вызовет ошибку. При организации программы надо быть внимательным и следить за тем, чтобы подобные определения размещались только в одном исходном файле.  
Модель с разделением позволяет отделить интерфейс шаблонов функций от его реализации и организовать программу так, что интерфейсы всех шаблонов помещаются в заголовочные файлы, а реализации – в файлы с исходным текстом. Однако не все компиляторы поддерживают такую модель, а те, которые поддерживают, не всегда делают это правильно: модель с разделением требует более изощренной среды программирования, которая доступна не во всех реализациях C++. (В другой нашей книге, “Inside C++ Object Model”, описан механизм конкретизации шаблонов, поддержанный в одной из реализаций C++, а именно в компиляторе Edison Design Group.)  
Поскольку приводимые нами примеры работы с шаблонами невелики и поскольку мы хотим, чтобы они компилировались максимально большим числом компиляторов, мы ограничились использованием модели с включением.

**10.5.3. Явные объявления конкретизации**

При использовании модели с включением определение шаблона функций включается в каждый исходный файл, где встречается конкретизация этого шаблона. Мы отмечали, что, хотя неизвестно, где и когда понадобится шаблон функции, программа должна вести себя так, как будто экземпляр шаблона для данного множества аргументов конкретизирован ровно один раз. В действительности некоторые компиляторы (особенно старые) конкретизируют шаблон функции с данным множеством аргументов шаблона неоднократно. В рамках этой модели для использования на этапе сборки или на одной из предшествующих ей стадий выбирается один из конкретизированных экземпляров, а остальные игнорируются.  
Результат работы программы не зависит от того, сколько раз конкретизировался шаблон: в конечном итоге используется лишь один экземпляр. Но если приложение состоит из большого числа файлов, то время компиляции приложения заметно возрастает.  
Подобные проблемы, характерные для старых компиляторов, затрудняли использование шаблонов. Поэтому в стандарте C++ введено понятие явного объявления конкретизации, помогающее программисту управлять моментом, когда конкретизация происходит.  
В явном объявлении конкретизации за ключевым словом template идет объявление шаблона функции, в котором его аргументы указаны явно. Рассмотрим шаблон sum(int\*, int):

template <typename Type>

Type sum(Type op1, Type op2) { /\* ... \*/ }

// явное объявление конкретизации  
 template int\* sum< int\* >(int\*, int);

Здесь в качестве аргумента явно задается int\*. Явное объявление конкретизации с одним и тем же множеством аргументов шаблона может встречаться в программе не более одного раза.  
Определение шаблона функции должно находиться в том же файле, где и явное объявление конкретизации. Если же его не видно, то явное объявление приводит к ошибке:

#include <vector>

template <typename Type>  
 Type sum(Type op1, int op2); // только объявление

// определяем typedef для vector< int >  
 typedef vector< int > VI;

// ошибка: sum() не определен  
 template VI sum< VI >(VI , int);

Если в некотором исходном файле встречается явное объявление конкретизации, то что произойдет в других файлах, где используется такая же конкретизация шаблона функции? Как сказать компилятору, что явное объявление находится в другом файле и что при использовании в этом файле шаблон конкретизировать не надо?  
Явные объявления конкретизации используются в сочетании с опцией компилятора, которая подавляет неявную конкретизацию шаблонов. Название опции в разных компиляторах различно. Например, в VisualAge for C++ для Windows версии 3.5 фирмы IBM эта опция называется /ft-. Если приложение компилируется с данной опцией, то компилятор предполагает, что шаблоны будут конкретизироваться явно, и не выполняет автоматической конкретизации.  
Разумеется, если мы не включили в программу явного объявления конкретизации для некоторого шаблона, но задали опцию /ft-, то при сборке произойдет ошибка из-за того, что функция не была конкретизирована.

**Упражнение 10.8**

Назовите две модели компиляции шаблонов, поддерживаемые в C++. Объясните, как организуются определения шаблонов функций в каждой модели.

**Упражнение 10.9**

Пусть дано следующее определение шаблона функции sum():

template <typename Type>

Type sum(Type op1, char op2);

Как записать явное объявление конкретизации этого шаблона с аргументом типа string?

**10.6. Явная специализация шаблона А**

Не всегда удается написать шаблон функции, который годился бы для всех возможных типов, с которыми он может быть конкретизирован. В некоторых случаях имеется специальная информация о типе, позволяющая написать более эффективную функцию, чем конкретизированная по шаблону. А иногда общее определение, предоставляемое шаблоном, для некоторого типа просто не работает. Рассмотрим, например, следующее определение шаблона функции max():

// обобщенное определение шаблона

template <class T>

T max(T t1, T t2) {

return (t1 > t2 ? t1 : t2);

}

Когда этот шаблон конкретизируется с аргументом типа const char\*, то обобщенное определение оказывается семантически некорректным, если мы интерпретируем каждый аргумент как строку символов в смысле языка C, а не как указатель на символ. В этом случае необходимо предоставить специализированное определение для конкретизации шаблона.  
*Явное определение специализации* – это такое определение, в котором за ключевым словом template следует пара угловых скобок <>, а за ними – определение специализированного шаблона. Здесь указывается имя шаблона, аргументы, для которых он специализируется, список параметров функции и её тело. В следующем примере для max(const char\*, const char\*) определена явная специализация:

#include <cstring>

// явная специализация для const char\*:  
// имеет приоритет над конкретизацией шаблона  
// по обобщенному определению

typedef const char \*PCC;  
template<> PCC max< PCC >(PCC s1, PCC s2) {  
 return (strcmp(s1, s2) > 0 ? s1 : s2);

Поскольку имеется явная специализация, шаблон не будет конкретизирован с типом const char\* при вызове в программе функции max(const char\*, const char\*). При любом обращении к max() с двумя аргументами типа const char\* работает специализированное определение. Для любых других обращений функция сначала конкретизируется по обобщенному определению шаблона, а затем вызывается. Вот как это выглядит:

#include <iostream>

// здесь должно быть определение шаблона функции max()   
// и его специализации для аргументов const char\*

int main() {  
 // вызов конкретизированной функции: int max< int >(int, int);  
 int i = max(10, 5);

// вызов явной специализации:  
 // const char\* max< const char\* >(const char\*, const char\*);  
 const char \*p = max("hello", "world");

cout << "i: " << i << " p: " <<

p << endl;  
 return 0;  
}

Можно объявлять явную специализацию шаблона функции, не определяя ее. Например, для функции max(const char\*, const char\*) она объявляется так:

// объявление явной специализации шаблона функции

template< > PCC max< PCC >(PCC, PCC);

При объявлении или определении явной специализации шаблона функции нельзя опускать слово template и следующую за ним пару скобок <>. Кроме того, в объявлении специализации обязательно должен быть список параметров функции:

// ошибка: неправильные объявления специализации

// отсутствует template<>  
PCC max< PCC >(PCC, PCC);

// отсутствует список параметров  
template<> PCC max< PCC >;

Однако здесь можно опускать задание аргументов шаблона, если они выводятся из формальных параметров функции:

// правильно: аргумент шаблона const char\* выводится из типов параметров

template<> PCC max(PCC, PCC);

В следующем примере шаблон функции sum() явно специализирован:

template <class T1, class T2, class T3>

T1 sum(T2 op1, T3 op2);

// объявления явных специализаций

// ошибка: аргумент шаблона для T1 не может быть выведен;  
// он должен быть задан явно

template<> double sum(float, float);

// правильно: аргумент для T1 задан явно,  
// T2 и T3 выводятся и оказываются равными float

template<> double sum<double>(float, float);

// правильно: все аргументы заданы явно

template<> int sum<int,char>(char, char);

Пропуск части template<> в объявлении явной специализации не всегда является ошибкой. Например:

// обобщенное определение шаблона

template <class T>

T max(T t1, T t2) { /\* ... \*/ }

// правильно: обычное объявление функции  
const char\* max(const char\*, const char\*);

Однако эта инструкция не является специализацией шаблона функции. Здесь просто объявляется обычная функция с типом возвращаемого значения и списком параметров, которые соответствуют полученным при конкретизации шаблона. Объявление обычной функции, являющееся конкретизацией шаблона, не считается ошибкой.  
Так почему бы просто не объявить обычную функцию? Как было показано в разделе 10.3, для преобразования фактического аргумента функции, конкретизированной по шаблону, в соответствующий формальный параметр в случае, когда этот аргумент принимает участие в выводе аргумента шаблона, может быть применено лишь ограниченное множество преобразований типов. Точно так же обстоит дело и в ситуации, когда шаблон функции специализируется явно: к фактическим аргументам функции при этом тоже применимо лишь ограниченное множество преобразований. Явные специализации не помогают обойти соответствующие ограничения. Если мы хотим выйти за их пределы, то должны определить обычную функцию вместо специализации шаблона. (В разделе 10.8 этот вопрос рассматривается более подробно; там же показано, как работает разрешение перегруженной функции для вызова, который соответствует как обычной функции, так и экземпляру, конкретизированному из шаблона.)  
Явную специализацию можно объявлять даже тогда, когда специализируемый шаблон объявлен, но не определен. В предыдущем примере шаблон функции sum() лишь объявлен к моменту специализации. Хотя определение шаблона не обязательно, объявление все же требуется. То, что sum() – шаблон, должно быть известно до того, как это имя может быть специализировано.  
Такое объявление должно быть видимо до его использования в исходном файле. Например:

#include <iostream>

#include <cstring>

// обобщенное определение шаблона  
template <class T>  
 T max(T t1, T t2) { /\* ... \*/ }

int main() {  
 // конкретизация функции  
 // const char\* max< const char\* >(const char\*, const char\*);  
 const char \*p = max("hello", "world");

cout << "p: " << p << endl;  
 return 0;  
}

// некорректная программа: явная специализация const char \*:  
// имеет приоритет над обобщенным определением шаблона  
typedef const char \*PCC;  
template<> PCC max< PCC >(PCC s1, PCC s2) { /\* ... \*/ }

В предыдущем примере конкретизация max(const char\*, const char\*) предшествует объявлению явной специализации. Поэтому компилятор имеет право предположить, что функция должна быть конкретизирована по обобщенному определению шаблона. Однако в программе не может одновременно существовать явная специализация и экземпляр, конкретизированный по тому же шаблону с тем же множеством аргументов. Когда в исходном файле после конкретизации встречается явная специализация max(const char\*, const char\*), компилятор выдает сообщение об ошибке.  
Если программа состоит из нескольких файлов, то объявление явной специализации шаблона должно быть видимо в каждом файле, в котором она используется. Не разрешается в одних файлах конкретизировать шаблон функции по обобщенному определению, а в других специализировать с тем же множеством аргументов. Рассмотрим следующий пример:

// --------- max.h -------

// обобщенное определение шаблона

template <class Type>

Type max(Type t1, Type t2) { /\* ... \*/ }

// --------- File1.C -------  
#include <iostream>  
#include "max.h"  
void another();

int main() {  
 // конкретизация функции  
 // const char\* max< const char\* >(const char\*, const char\*);  
 const char \*p = max("hello", "world");

cout << "p: " << p << endl;  
 another();

return 0;  
}

// --------- File2.C -------  
#include <iostream>  
#include <cstring>  
#include "max.h"

// явная специализация шаблона для const char\*  
typedef const char \*PCC;

template<> PCC max< PCC >(PCC s1, PCC s2) { /\* ... \*/ }

void another() {

// явная специализация  
 // const char\* max< const char\* >(const char\*, const char\*);  
 const char \*p = max("hi", "again");

cout << " p: " << p << endl;

return 0;  
}

Эта программа состоит из двух файлов. В файле File1.C нет объявления явной специализации max(const char\*, const char\*). Вместо этого шаблон функции конкретизируется из обобщенного определения. В файле File2.C объявлена явная специализация, и при обращении к max("hi", "again") именно она и вызывается. Поскольку в одной и той же программе функция max(const char\*, const char\*) то конкретизируется по шаблону, то специализируется явно, компилятор считает программу некорректной. Для исправления этого объявление явной специализации шаблона должно предшествовать вызову функции max(const char\*, const char\*) в файле File1.C.  
Чтобы избежать таких ошибок и гарантировать, что объявление явной специализации шаблона max(const char\*, const char\*) внесено в каждый файл, где используется шаблон функции max() с аргументами типа const char\*, это объявление следует поместить в заголовочный файл "max.h" и включать его во все исходные файлы, в которых используется шаблон max():

// --------- max.h -------

// обобщенное определение шаблона

template <class Type>

Type max(Type t1, Type t2) { /\* ... \*/ }

// объявление явной специализации шаблона для const char\*  
typedef const char \*PCC;  
template<> PCC max< PCC >(PCC s1, PCC s2);

// --------- File1.C -------  
#include <iostream>  
#include "max.h"  
void another();

int main() {  
 // специализация  
 // const char\* max< const char\* >(const char\*, const char\*);  
 const char \*p = max("hello", "world");

// ....  
}

**Упражнение 10.10**

Определите шаблон функции count() для подсчета числа появлений некоторого значения в массиве. Напишите вызывающую программу. Последовательно передайте в ней массив значений типа double, int и сhar. Напишите специализированный экземпляр шаблона count() для обработки строк.

**10.7. Перегрузка шаблонов функций А**

Шаблон функции может быть перегружен. В следующем примере есть три перегруженных объявления для шаблона min():

// определение шаблона класса Array

// (см. [раздел 2.4](http://valera.asf.ru/cpp/book/c02.html))

template <typename Type>  
 class Array(/\* ... \*/ };

// три объявления шаблона функции min()

template <typename Type>  
 Type min(const Array<Type>&, int); // #1

template <typename Type>  
 Type min(const Type\*, int); // #2

template <typename Type>  
 Type min(Type, Type); // #3

Следующее определение main() иллюстрирует, как могут вызываться три объявленных таким образом функции:

#include <cmath>

int main()  
{  
 Array<int> iA(1024); // конкретизация класса  
 int ia[1024];

// Type == int; min(const Array<int>&, int)  
 int ival0 = min(iA, 1024);

// Type == int; min(const int\*, int)  
 int ival1 = min(ia, 1024);

// Type == double; min(double, double)  
 double dval0 = min(sqrt(iA[0]), sqrt(ia[0]));

return 0;  
}

Разумеется, тот факт, что три перегруженных шаблона функции успешно объявлены, не означает, что они могут быть также успешно вызваны. Такие шаблоны могут приводить к неоднозначности при вызове конкретизированного шаблона. Например, для следующего определения шаблона min5()

template <typename T>

int min5(T, T) { /\* ... \*/ }

функция не конкретизируется по шаблону, если min5() вызывается с аргументами разных типов; при этом процесс вывода заканчивается с ошибкой, поскольку из фактических аргументов функции выводятся два разных типа для T.

int i;

unsigned int ui;

// правильно: для T выведен тип int  
min5(1024, i);

// вывод аргументов шаблона заканчивается с ошибкой:  
// для T можно вывести два разных типа  
min5 (i, ui);

Для разрешения второго вызова можно было бы перегрузить min5(), допустив два различных типа аргументов:

template <typename T, typename U>

int min5(T, U);

При следующем обращении производится конкретизация этого шаблона функции:

// правильно: int min5(int, usigned int)

min5(i, ui);

К сожалению, теперь стал неоднозначным предыдущий вызов:

// ошибка: неоднозначность: две возможных конкретизации

// из min5(T, T) и min5(T, U)

min5(1024, i);

Второе объявление min5() допускает наличие у функции аргументов различных типов, но не требует этого. В нашем случае и T, и U типа int. Оба объявления шаблонов могут быть конкретизированы вызовом, в котором два аргумента функции имеют один и тот же тип. Единственный способ указать, какой шаблон более предпочтителен, устранив тем самым неоднозначность, – явно задать его аргументы. (О явном задании аргументов шаблона см. раздел 10.4.) Например:

// правильно: конкретизация из min5(T, U)

min5<int, int>(1024, i);

Однако в этом случае мы можем обойтись без перегрузки шаблона функции. Поскольку шаблон min5(T,U) подходит для всех вызовов, для которых подходит min5(T,T), то одного объявления min5(T,U) вполне достаточно, а объявление min5(T,T) можно удалить. Мы уже говорили в главе 9, что, хотя перегрузка допускается, при проектировании таких функций надо быть внимательным и использовать её только при необходимости. Те же соображения применимы и к определению перегруженных шаблонов.  
В некоторых ситуациях неоднозначности при вызове не возникает, хотя по шаблону можно конкретизировать две разных функции. Если имеются следующие два шаблона для функции sum(), то предпочтение будет отдано первому даже тогда, когда конкретизированы могут быть оба:

template <typename Type>

Type sum(Type\*, int);

template <typename Type>  
 Type sum(Type, int);

int ia[1024];

// Type == int ; sum<int>(int\*, int); или  
// Type == int\*; sum<int\*>(int\*, int); ??  
int ival1 = sum<int>(ia, 1024);

Как это ни удивительно, такой вызов не приводит к неоднозначности. Шаблон конкретизируется из первого определения, так как выбирается наиболее специализированное определение. Поэтому для аргумента Type принимается int, а не int\*.  
Для того чтобы один шаблон был более специализирован, чем другой, оба они должны иметь одни и те же имя и число параметров, а для параметров разных типов, как, скажем, T\* и T в предыдущем примере, параметр в одном шаблоне должен быть способен принять более широкое множество фактических аргументов, чем соответствующий параметр в другом. Например, для шаблона sum(Type\*, int) вместо первого формального параметра функции разрешается подставлять только фактические аргументы типа “указатель”. В то же время в шаблоне sum(Type, int) первому формальному параметру могут соответствовать фактические аргументы любого типа. Первый шаблон sum(Type\*, int) допускает более узкое множество аргументов, чем второй, т.е. он более специализирован, а следовательно, он и конкретизируется при вызове функции.

**10.8. Разрешение перегрузки при конкретизации A**

В предыдущем разделе мы видели, что шаблон функции может быть перегружен. Кроме того, допускается использование одного и того же имени для шаблона и обычной функции:

// шаблон функции

template <class Type>

Type sum(Type, int) { /\* ... \*/ }

// обычная функция (не шаблон)

double sum(double, double);

Когда программа обращается к sum(), вызов разрешается либо в пользу конкретизированного экземпляра шаблона, либо в пользу обычной функции – это зависит от того, какая функция лучше соответствует фактическим аргументам. (Для решения такой проблемы применяется процесс разрешения перегрузки, описанный в главе [9](http://valera.asf.ru/cpp/book/c09.html).) Рассмотрим следующий пример:

void calc(int ii, double dd) {

// что будет вызвано: конкретизированный экземпляр шаблона

// или обычная функция?

sum(dd, ii);

}

Будет ли при обращении к sum(dd,ii) вызвана функция, конкретизированная из шаблона, или обычная функция? Чтобы ответить на этот вопрос, выполним по шагам процедуру разрешения перегрузки. Первый шаг заключается в построении множества функций-кандидатов состоящего из одноименных вызванной функций, объявления которых видны в точке вызова.  
Если существует шаблон функции и на основе фактических аргументов вызова из него может быть конкретизирована функция, то она будет являться кандидатом. Так ли это на самом деле, зависит от результата процесса вывода аргументов шаблона. (Этот процесс описан в разделе 10.3.) В предыдущем примере для вывода значения аргумента Type шаблона используется фактический аргумент функции dd. Тип выведенного аргумента оказывается равным double, и к множеству функций-кандидатов добавляется функция sum(double, int). Таким образом, для данного вызова имеются два кандидата: конкретизированная из шаблона функция sum(double, int) и обычная функция sum(double, double).  
  
После того как функции, конкретизированные из шаблона, включены в множество кандидатов, процесс вывода аргументов шаблона продолжается как обычно.  
Второй шаг процедуры разрешения перегрузки заключается в выборе устоявших функций из множества кандидатов. Напомним, что устоявшей называется функция, для которой существуют преобразования типов, приводящие каждый фактический аргумент функции к типу соответствующего формального параметра. (В разделе [9.3](http://valera.asf.ru/cpp/book/c09.html)описаны преобразования типов, применимые к фактическим аргументам функции.) Нужные трансформации существуют как для конкретизированной функции sum(double, int), так и для обычной функции sum(double, double). Следовательно, обе они являются устоявшими.  
Проведем ранжирование преобразований типов, примененных к фактическим аргументам для выбора наилучшей из устоявших функций. В нашем примере оно происходит следующим образом:

Для конкретизированной из шаблона функции sum(double, int):

* для первого фактического аргумента как сам этот аргумент, так и формальный параметр имеют тип double, т.е. мы видим точное соответствие;
* для второго фактического аргумента как сам аргумент, так и формальный параметр имеют тип int, т.е. снова точное соответствие.

Для обычной функции sum(double, double):

* для первого фактического аргумента как сам этот аргумент, так и формальный параметр имеют тип double – точное соответствие;
* для второго фактического аргумента сам этот аргумент имеет тип int, а формальный параметр – тип double, т.е. необходимо стандартное преобразование между целым и плавающим типами.

Если рассматривать только первый аргумент, то обе функции одинаково хороши. Однако для второго аргумента конкретизированная из шаблона функция лучше. Поэтому наиболее подходящей (лучшей из устоявших) считается функция sum(double, int).  
Функция, конкретизированная из шаблона, включается в множество кандидатов только тогда, когда процесс вывода аргументов завершается успешно. Неудачное завершение в данном случае не является ошибкой, но кандидатом функция считаться не будет. Предположим, что шаблон функции sum() объявлен следующим образом:

// шаблон функции

template <class T>

int sum(T\*, int) { ... }

Для описанного вызова функции вывод аргументов шаблона будет неудачным, так как фактический аргумент типа double не может соответствовать формальному параметру типа T\*. Поскольку для данного вызова и данного шаблона конкретизировать функцию невозможно, в множество кандидатов ничего не добавляется, т.е. единственным его элементом останется обычная функция sum(double, double). Именно она вызывается при обращении, и её второй фактический аргумент приводится к типу double.  
А если вывод аргументов шаблона завершается удачно, но для них есть явная специализация? Тогда именно она, а не функция, конкретизированная из обобщенного шаблона, попадает в множество кандидатов. Например:

// определение шаблона функции

template <class Type> Type sum(Type, int) { /\* ... \*/ }

// явная специализация для Type == double  
template<> double sum<double>(double,int);

// обычная функция  
double sum(double, double);

void manip(int ii, double dd) {  
 // вызывается явная специализация шаблона sum<double>()  
 sum(dd, ii);  
}

При обращении к sum() внутри manip() в процессе вывода аргументов шаблона обнаруживается, что функция sum(double,int), конкретизированная из обобщенного шаблона, должна быть добавлена к множеству кандидатов. Но для нее имеется явная специализация, которая и становится кандидатом. На более поздних стадиях анализа выясняется, что эта специализация дает наилучшее соответствие фактическим аргументам вызова, так что разрешение перегрузки завершается в её пользу.  
Явные специализации шаблона не включаются в множество кандидатов автоматически. Лишь в том случае, когда вывод аргументов завершается успешно, компилятор будет рассматривать явные специализации данного шаблона:

// определение шаблона функции

template <class Type>

Type min(Type, Type) { /\* ... \*/ }

// явная специализация для Type == double  
template<> double min<double>(double, double);

void manip(int ii, double dd) {  
 // ошибка: вывод аргументов шаблона неудачен,  
 // нет функций-кандидатов для данного вызова  
 min(dd, ii);  
}

Шаблон функции min() специализирован для аргумента double. Однако эта специализация не попадает в множество функций-кандидатов. Процесс вывода для вызова min() завершился неудачно, поскольку аргументы шаблона, выведенные для Type на основе разных фактических аргументов функции, оказались различными: для первого аргумента выводится тип double, а для второго – int. Поскольку вывести аргументы не удалось, в множество кандидатов никакая функция не добавляется, и специализация min(double, double) игнорируется. Так как других функций-кандидатов нет, вызов считается ошибочным.  
Как отмечалось в разделе 10.6, тип возвращаемого значения и список формальных параметров обычной функции может точно соответствовать аналогичным атрибутам функции, конкретизированной из шаблона. В следующем примере min(int,int) – это обычная функция, а не специализация шаблона min(), поскольку, как вы, вероятно, помните, объявление специализации должно начинаться с template<>:

// объявление шаблона функции

template <class T>

T min(T, T);

// обычная функция min(int,int)  
int min(int, int) { }

Вызов может точно соответствовать как обычной функции, так и функции, конкретизированной из шаблона. В следующем примере оба аргумента в min(ai[0],99) имеют тип int. Для этого вызова есть две устоявших функции: обычная min(int,int) и конкретизированная из шаблона функция с тем же типом возвращаемого значения и списком параметров:

int ai[4] = { 22, 33, 44, 55 };

int main() {

// вызывается обычная функция min(int, int)

min(ai[0], 99);

}

Однако такой вызов не является неоднозначным. Обычной функции, если она существует, всегда отдается предпочтение, поскольку она реализована явно, так что перегрузка разрешается в пользу обычной функции min(int,int).  
Если перегрузка разрешилась таким образом, то изменений уже не будет: если позже обнаружится, что в программе нет определения этой функции, компилятор не станет конкретизировать её тело из шаблона. Вместо этого на этапе сборки мы получим ошибку. В следующем примере программа вызывает, но не определяет обычную функцию min(int,int), и редактор связей выдает сообщение об ошибке:

// шаблон функции

template <class T>

T min(T, T) { ... }

// это обычная функция, не определенная в программе  
int min(int, int);

int ai[4] = { 22, 33, 44, 55 };  
int main() {  
 // ошибка сборки: min(int, int) не определена  
 min(ai[0], 99);  
}

Зачем определять обычную функцию, если её тип возвращаемого значения и список параметров соответствуют функции, конкретизированной из шаблона? Вспомните, что при вызове конкретизированной функции к её фактическим аргументам в ходе вывода аргументов шаблона можно применять только ограниченное множество преобразований. Если же объявлена обычная функция, то для приведения типов аргументов допустимы любые трансформации, так как типы формальных параметров обычной функции фиксированы. Рассмотрим пример, показывающий, зачем может потребоваться объявить обычную функцию.  
Предположим, что мы хотим определить специализацию шаблона функции min<int>(int,int). Нужно, чтобы именно эта функция вызывалась при обращении к min() с аргументами любых целых типов, пусть даже неодинаковых. Из-за ограничений, наложенных на преобразования типов, при передаче фактических аргументов разных типов функция min<int>(int,int) не будет конкретизирована из шаблона. Мы могли бы заставить компилятор выполнить конкретизацию, явно задав аргументы шаблона, однако решение, при котором не требуется модифицировать каждый вызов, предпочтительнее. Определив обычную функцию, мы добьемся того, что программа будет вызывать специальную версию min(int,int) для любых фактических аргументов целых типов без явного указания аргументов шаблона:

// определение шаблона функции

template <class Type>

Type min(Type t1, Type t2) { ... }

int ai[4] = { 22, 33, 44, 55 };  
short ss = 88;

void call\_instantiation() {  
 // ошибка: для этого вызова нет функции-кандидата  
 min(ai[0], ss);  
}

// обычная функция  
int min(int a1, int a2) {  
 min<int>(a1, a2);  
}

int main() {  
 call\_instantiation() {  
 // вызывается обычная функция  
 min(ai[0], ss);  
 }

Для вызова min(ai[0],ss) из call\_instantiation нет ни одной функции-кандидата. Попытка сгенерировать её из шаблона min() провалится, поскольку для аргумента шаблона Type из фактических аргументов функции выводятся два разных значения. Следовательно, такой вызов ошибочен. Однако при обращении к min(ai[0],ss) внутри main() видимо объявление обычной функции min(int, int). Тип первого фактического аргумента этой функции точно соответствует типу формального параметра, а второй аргумент может быть преобразован в тип формального параметра с помощью расширения типа. Поскольку для второго вызова устояла только данная функция, то она и вызывается.  
Разобравшись с разрешением перегрузки функций, конкретизированных из шаблонов, специализацией шаблонов функций и обычных функций с тем же именем, подытожим все, что мы об этом рассказали:

1. Построить множество функций-кандидатов.  
   Рассматриваются шаблоны функций с тем же именем, что и вызванная. Если аргументы шаблона выведены из фактических аргументов функции успешно, то в множество функций-кандидатов включается либо конкретизированный шаблон, либо специализация шаблона для выведенных аргументов, если она существует.
2. Построить множество устоявших функций (см. [раздел 9.3](http://valera.asf.ru/cpp/book/c09.html)).  
   В множестве функций-кандидатов остаются только функции, которые можно вызвать с данными фактическими аргументами.
3. Ранжировать преобразования типов (см. [раздел 9.3](http://valera.asf.ru/cpp/book/c09.html)).
   1. Если есть только одна функция, вызвать именно ее.
   2. Если вызов неоднозначен, удалить из множества устоявших функции, конкретизированные из шаблонов.
4. Разрешить перегрузку, рассматривая среди всех устоявших только обычные функции (см. раздел 9.3).
   1. Если есть только одна функция, вызвать именно ее.
   2. В противном случае вызов неоднозначен.

Проиллюстрируем эти шаги на примере. Предположим, есть два объявления – шаблона функции и обычной функции. Оба принимают аргументы типа double:

template <class Type>

Type max(Type, Type) { ... }

// обычная функция  
double max(double, double);

А вот три вызова max(). Можете ли вы сказать, какая функция будет вызвана в каждом случае?

int main() {

int ival;

double dval;

float fd;

// ival, dval и fd присваиваются значения

max(0, ival);  
 max(0.25, dval);  
 max(0, fd);  
}

Рассмотрим последовательно все три вызова:

1. max(0,ival). Оба аргумента имеют тип int. Для вызова есть два кандидата: конкретизированная из шаблона функция max(int, int) и обычная функция max(double, double). Конкретизированная функция точно соответствует фактическим аргументам, поэтому она и вызывается;
2. max(0.25,double). Оба аргумента имеют тип double. Для вызова есть два кандидата: конкретизированная из шаблона max(double, double) и обычная max(double, double). Вызов неоднозначен, поскольку точно соответствует обеим функциям. Правило 3b говорит, что в таком случае выбирается обычная функция;.
3. max(0,fd). Аргументы имеют тип int и float соответственно. Для вызова существует только один кандидат: обычная функция max(double, double). Вывод аргументов шаблона заканчивается неудачей, так как значения типа Type, выведенные из разных фактических аргументов функции, различны. Поэтому в множество кандидатов конкретизированная из шаблона функция не попадает. Обычная же функция устояла, поскольку существуют преобразования типов фактических аргументов в типы формальных параметров; она и выбирается. Если бы обычная функция не была объявлена, вызов закончился бы ошибкой.

А если бы мы определили еще одну обычную функцию для max()? Например:

template <class T> T max(T, T) { ... }

// две обычные функции

char max(char, char);

double max(double, double);

Будет ли в таком случае третий вызов разрешен по-другому? Да.

int main() {

float fd;

// в пользу какой функции разрешается вызов?  
 max(0, fd);  
}

Правило 3b говорит, что, поскольку вызов неоднозначен, следует рассматривать только обычные функции. Ни одна из них не считается наилучшей из устоявших, так как преобразования типов фактических аргументов одинаково плохи: в обоих случаях для установления соответствия требуется стандартная трансформация. Таким образом, вызов неоднозначен, и компилятор сообщает об ошибке.

**Упражнение 10.11**

Вернемся к представленному ранее примеру:

template <class Type>

Type max(Type, Type) { ... }

double max(double, double);  
int main() {  
 int ival;  
 double dval;  
 float fd;

max(0, ival);  
 max(0.25, dval);  
 max(0, fd);  
}

Добавим в множество объявлений в глобальной области видимости следующую специализацию шаблона функции:

template <> char max<char>\* char, char) { ... }

Составьте список кандидатов и устоявших функций для каждого вызова max() внутри main().  
Предположим, что в main() добавлен следующий вызов:

int main() {

// ...

max(0, 'j');

}

В пользу какой функции он будет разрешен? Почему?

**Упражнение 10.12**

Предположим, что есть следующее множество определений и специализаций шаблонов, а также объявления переменных и функций:

int i; unsigned int ui;

char str[24]; int ia[24];

template <class T> T calc(T\*, int);

template <class T> T calc(T, T);

template<> chat calc(char\*. int);

double calc(double, double);

Выясните, какая функция или конкретизированный шаблон вызывается в каждом из показанных ниже случаев. Для каждого вызова перечислите функции-кандидаты и устоявшие функции; объясните, какая из устоявших функций будет наилучшей.

(a) cslc(str, 24); (d) calc(i, ui);

(b) calc(is, 24); (e) calc(ia, ui);

(c) calc(ia[0], 1); (f) calc(&i, i);

**10.9. Разрешение имен в определениях шаблонов А**

Внутри определения шаблона смысл некоторых конструкций может различаться в зависимости от конкретизации, тогда как смысл других всегда остается неизменным. Главную роль играет наличие в конструкции формального параметра шаблона:

template <typename Type>  
 Type min(Type\* array, int size)  
{  
 Type min\_val = array[0];  
 for (int i = 1; i < size; ++i)  
 if (array[i] < min\_val)  
 min\_val = array[i];  
 print("Minimum value found: ");  
 print(min\_val);  
 return min\_val;  
}

В функции min() типы переменных array и min\_val зависят от фактического типа, которым будет заменен Type при конкретизации шаблона, тогда как тип переменной size останется int при любом типе параметра шаблона. Следовательно, типы array и min\_val в разных конкретизациях различны. Поэтому мы говорим, что типы этих переменных зависят от параметра шаблона, тогда как тип size от него не зависит.  
  
Так как тип min\_val неизвестен, то неизвестна и операция, которая будет использоваться при появлении min\_val в выражении. Например, какая функция print() будет вызвана при обращении print(min\_val)? С типом аргумента int? Или float? Будет ли вызов ошибочным, поскольку не существует функции, которая может быть вызвана с аргументом того же типа, что и min\_val? Принимая все это во внимание, мы говорим, что и вызов print(min\_val) зависит от параметра шаблона.  
Такие вопросы не возникают для тех конструкций внутри min(), которые не зависят от параметров шаблона. Например, всегда известно, какая функция должна быть вызвана для print("Minimum value found: "). Это функция печати строк символов. В данном случае print() остается одной и той же при любой конкретизации шаблона, то есть не зависит от его параметров.  
  
В главе [7](http://valera.asf.ru/cpp/book/c07.html)мы видели, что в C++ функция должна быть объявлена до её вызова. Нужно ли объявлять функцию, вызываемую внутри шаблона, до того, как компилятор увидит его определение? Должны ли мы объявить функцию print() в предыдущем примере до определения шаблона min()? Ответ зависит от особенностей имени, на которое мы ссылаемся. Конструкцию, не зависящую от параметров шаблона, следует объявить перед её использованием в шаблоне. Представленное выше определение шаблона функции min() некорректно. Поскольку вызов

print("Minimum value found: ");

не зависит от параметров шаблона, то функция print() для печати строк символов должна быть объявлена до использования. Чтобы исправить эту ошибку, можно поместить объявление print() перед определением min():

// ---- primer.h ----

// это объявление необходимо:  
// внутри min() вызывается print(const char \*)  
void print(const char \*);

template <typename Type>  
 Type min(Type\* array, int size) {

// ...

print("Minimum value found: ");  
 print(min\_val);

return min\_val;

}

С другой стороны, объявление функции print(), используемой для печати min\_val, пока не нужно, так как еще неизвестно, какую конкретно функцию надо искать. Мы не знаем, какая функция print() будет вызвана при обращении print(min\_val), пока тип min\_val не станет известным.  
Когда же должна быть объявлена функция print(), вызываемая при обращении print(min\_val)? До конкретизации шаблона. Например:

#include <primer.h>

void print(int);

int ai[4] = {12, 8, 73, 45 };

int main() {  
 int size = sizeof(ai) / sizeof(int);  
 // конкретизируется min(int\*, int)  
 min(&ai[0], size);  
}

main() вызывает конкретизированную из шаблона функцию min(int\*,int). В этой реализации Type заменено int, и тип переменной min\_val, следовательно, равен int. Поэтому при обращении print(min\_val) вызывается функция с аргументом типа int. Именно тогда, когда конкретизируется min(int\*,int), становится известно, что при втором вызове аргумент print() имеет тип int. В этот момент такая функция должна быть видима. Если бы функция print(int) не была объявлена до конкретизации min(int\*,int), то компилятор выдал бы сообщение об ошибке.  
Поэтому разрешение имен в определении шаблона происходит в два этапа. Сначала разрешаются имена, не зависящие от его параметров, а затем, при конкретизации, – имена, зависящие от параметров.  
  
Но зачем нужны два шага? Почему бы, например, не разрешать все имена при конкретизации?  
Если вы проектируете шаблон функции, то, вероятно, хотели бы сохранить контроль над тем, когда разрешаются имена в его определении. Предположим, что шаблон min() – это часть библиотеки, в которой определены и другие шаблоны и функции. Желательно, чтобы реализации min() по возможности использовали другие компоненты нашей же библиотеки. В предыдущем примере интерфейс библиотеки определен в заголовочном файле <primer.h>. Как объявление функции print(const char\*), так и определение функции min() являются частями интерфейса. Мы хотим, чтобы конкретизации шаблона min() пользовались функцией print() из нашей библиотеки. Первый этап разрешения имени это гарантирует. Если имя, использованное в определении шаблона, не зависит от его параметров, то оно обязательно будет относиться к компоненту внутри библиотеки, т.е. к тому объявлению, которое включено в один пакет с этим определением в заголовочном файле <primer.h>.  
На самом деле автор шаблона должен позаботиться о том, чтобы были объявлены все имена, использованные в определениях и не зависящие от параметров. Если этого нет, то определение шаблона вызовет ошибку. При конкретизации шаблона компилятор её не исправляет:

// ---- primer.h ----

template <typename Type>

Type min(Type\* array, int size)

{

Type min\_val = array[0];

// ...

// ошибка: функция print(const char\*) не найдена

print("Minimum value found: ");

// правильно: зависит от параметра шаблона  
 print(min\_val);  
 // ...  
 }

// ---- user.C ----

#include <primer.h>

// это объявление print(const char\*) игнорируется  
void print(const char\*);  
void print(int);

int ai[4] = {12, 8, 73, 45 };

int main() {  
 int size = sizeof(ai) / sizeof(int);  
 // конкретизируется min(int\*, int)  
 min(&ai[0], size);  
}

Объявление функции print(const char\*) в файле user.C невидимо в том месте, где появляется определение шаблона. Однако оно видимо там, где конкретизируется шаблон min(int\*,int), но это объявление не рассматривается при компиляции вызова print("Minimum value found: "), так как последний не зависит от параметров шаблона. Если некоторая конструкция в определении шаблона не зависит от его параметров, то имена разрешаются в контексте самого определения, и результат разрешения в дальнейшем не пересматривается. Поэтому на программиста возлагается ответственность за то, чтобы объявления имен, встречающихся в определении, были включены в интерфейс библиотеки вместе с шаблоном.  
А теперь предположим, что библиотека была написана кем-то другим, а мы её пользователи, которым доступен интерфейс, определенный в заголовочном файле <primer.h>. Иногда нужно, чтобы объекты и функции, определенные в нашей программе, учитывались при конкретизации шаблона из библиотеки. Допустим, мы определили в своей программе класс SmallInt и хотели бы конкретизировать функцию min() из библиотеки <primer.h> для получения минимального значения в массиве объектов типа SmallInt.  
При конкретизации шаблона min() для массива объектов типа SmallInt вместо аргумента шаблона Type подставляется тип SmallInt. Следовательно, min\_val в конкретизированной функции min() имеет тот же тип. Тогда как разрешится вызов функции print(min\_val)?

// ---- user.h ----

class SmallInt { /\* ... \*/ }

void print(const SmallInt &);

// ---- user.C ----  
#include <primer.h>  
#include "user.h"

SmallInt asi[4];

int main() {  
 // задать значения элементов массива asi

// конкретизируется min(SmallInt\*, int)  
 // int size = sizeof(asi) / sizeof(SmallInt);  
 min(&asi[0], size);  
}

Это нормально: мы хотим, чтобы учитывалась именно наша функция print(const SmallInt &). Рассмотрения функций, определенных в библиотеке <primer.h>, недостаточно. Второй шаг разрешения имени гарантирует, что если имя, использованное в определении, зависит от параметров шаблона, то принимаются во внимание имена, объявленные в контексте конкретизации. Поэтому можно быть уверенным, что функции, умеющие манипулировать объектами типа SmallInt, попадут в поле зрения компилятора при анализе шаблона, которому в качестве аргумента передан тип SmallInt.  
Место в программе, где происходит конкретизация шаблона, называется точкой конкретизации. Знание этой точки важно потому, что она определяет, какие объявления учитывает компилятор для имен, зависящих от параметров шаблона. Такая точка всегда находится в области видимости пространства имен и следует за функцией, внутри которой произошла конкретизация. Например, точка конкретизации min(SmallInt\*,int) расположена сразу после функции main() в области видимости пространства имен:

// ...

int main() {

// ...

// использование min(SmallInt\*,int)

min(&asi[0], size);

}

// точка конкретизации min(SmallInt\*,int)

// как будто объявление конкретизированной функции выглядит так:

SmallInt min(SmallInt\* array, int size)

{ /\* ... \*/ }

Но что, если конкретизация шаблона случается в одном исходном файле несколько раз? Где тогда будет точка конкретизации? Вы можете спросить: “А какая, собственно, разница?” В нашем примере для SmallInt разница есть, поскольку объявление функции print(const SmallInt &) должно появиться перед точкой конкретизации min(SmallInt\*,int):

#include <primer.h>

void another();

SmallInt asi[4];

int main() {  
 // задать значения элементов массива asi  
 int size = sizeof(asi) / sizeof(SmallInt);  
 min(&asi[0], size);

another();  
 // ...  
}  
// точка конкретизации здесь?

void another() {  
 int size = sizeof(asi) / sizeof(SmallInt);  
 min(&asi[0], size);  
}  
// или здесь?

В действительности точка конкретизации находится после определения каждой функции, в которой используется конкретизированный экземпляр. Компилятор может выбрать любую из этих точек, чтобы конкретизировать в ней шаблон. Отсюда следует, что при организации кода программы надо быть внимательным и помещать все объявления, необходимые для разрешения имен, зависящих от параметров некоторого шаблона, перед первой точкой. Поэтому разумно поместить их в заголовочный файл, который включается перед любой возможной конкретизацией шаблона:

#include <primer.h>

// user.h содержит объявления, необходимые при конкретизации

#include "user.h"

void another();

SmallInt asi[4];

int main() {  
 // ...  
}

// первая точка конкретизации min(SmallInt\*,int)

void another() {  
 // ...  
}   
// вторая точка конкретизации min(SmallInt\*,int)

А если конкретизация шаблона происходит в нескольких файлах? Например, что будет, если функция another() находится в другом файле, нежели main()? Тогда точка конкретизации есть в каждом файле, где используется конкретизированная из шаблона функция. Компилятор свободен в выборе любой из них, так что нам снова придется проявить аккуратность и включить файл "user.h" во все исходные файлы, где используются конкретизированные функции. Тем самым гарантируется, что реализация min(SmallInt\*,int) будет ссылаться именно на нашу функцию print(const SmallInt &) вне зависимости от того, какую из точек конкретизации выберет компилятор.

**Упражнение 10.13**

Назовите два шага разрешения имени в определениях шаблона. Объясните, каким образом первый шаг отвечает потребностям разработчика библиотеки, а второй обеспечивает гибкость, необходимую пользователям шаблонов.

**Упражнение 10.14**

На какие объявления ссылаются имена display и SIZE в реализации max(LongDouble\*,SIZE)?

// ---- exercise.h ----

void display(const void\*);

typedef unsigned int SIZE;

template <typename Type>  
 Type max(Type\* array, SIZE size)  
{  
 Type max\_val = array[0];  
 for (SIZE i = 1; i < size; ++i)  
 if (array[i] > max\_val)  
 max\_val = array[i];

display("Maximum value found: ");  
 display(max\_val);

return max\_val;  
}

// ---- user.h ----  
class LongDouble { /\* ... \*/ };  
void display(const LongDouble &);  
void display(const char \*);  
typedef int SIZE;

// ---- user.C ----  
#include <exercize.h>  
#include "user.h"

LongDouble ad[7];

int main() {  
 // задать значения элементов массива ad

// конкретизируется max(LongDouble\*, SIZE)  
 SIZE size = sizeof(ad) / sizeof(LongDouble);

max(&ad[0], size);  
}

**10.10. Пространства имен и шаблоны функций А**

Как и любое другое глобальное определение, шаблон функции может быть помещен в пространство имен (см. обсуждение пространств имен в разделах 8.5 и 8.6). Мы получили бы ту же семантику, если бы определили шаблон в глобальной области видимости, скрыв его имя внутри пространства имен. При использовании вне этого пространства необходимо либо квалифицировать имя шаблона именем пространства имен, либо использовать using-объявление:

// ---- primer.h ----

namespace cplusplus\_primer {

// определение шаблона скрыто в пространстве имен

template <class Type>

Type min(Type\* array, int size) { /\* ... \*/ }

}

// ---- user.C ----  
#include <primer.h>  
int ai[4] = { 12, 8, 73, 45 };

int main() {  
 int size = sizeof(ai) / sizeof(ai[0]);

// ошибка: функция min() не найдена  
 min(&ai[0], size);

using cplusplus\_primer::min; // using-объявление  
 // правильно: относится к min() в пространстве имен cplusplus\_primer  
 min(&ai[0], size);  
}

Что произойдет, если наша программа использует шаблон, определенный в пространстве имен, и мы хотим предоставить для него специализацию? (Явные специализации шаблонов рассматривались в разделе 10.6.) Допустим, мы хотим использовать шаблон min(), определенный в cplusplus\_primer, для нахождения минимального значения в массиве объектов типа SmallInt. Однако мы осознаем, что имеющееся определение шаблона не вполне подходит, поскольку сравнение в нем выглядит так:

if (array[i] < min\_val)

В этой инструкции два объекта класса SmallInt сравниваются с помощью оператора <. Но этот оператор неприменим к объектам, если только не перегружен в классе SmallInt (мы покажем, как определять перегруженные операторы в главе 15). Предположим, что мы хотели бы определить специализацию шаблона min(), чтобы она пользовалась функцией compareLess() для сравнения двух подобных объектов. Вот её объявление:

// функция сравнения объектов SmallInt

// возвращает true, если parm1 меньше parm2

bool compareLess(const SmallInt &parm1, const SmallInt &parm2);

Как должно выглядеть определение этой функции? Чтобы ответить на этот вопрос, необходимо познакомиться с определением класса SmallInt более подробно. Данный класс позволяет определять объекты, которые хранят тот же диапазон значений, что и 8-разрядный тип unsigned char, т.е. от 0 до 255. Дополнительная функциональность состоит в том, что класс перехватывает ошибки переполнения и потери значимости. Во всем остальном он должен вести себя точно так же, как unsigned char. Определение SmallInt выглядит следующим образом:

class SmallInt {

public:

SmallInt(int ival) : value(ival) {}

friend bool compareLess(const SmallInt &, const SmallInt &);

private:

int value; // член

};

В этом классе есть один закрытый член value, в котором хранится значение объекта типа SmallInt. Класс также содержит конструктор с параметром ival:

// конструктор класса SmallInt

SmallInt(int ival) : value(ival) {}

Его единственное назначение – инициализировать член класса value значением ival.  
Вот теперь можно ответить на ранее поставленный вопрос: как должна быть определена функция compareLess()? Она будет сравнивать члены value переданных ей аргументов типа SmallInt:

// возвращает true, если parm1 меньше parm2

bool compareLess(const SmallInt &parm1, const SmallInt &parm2) {

return parm1.value < parm2.value;

}

Заметим, однако, что член value является закрытым. Как может глобальная функция обратиться к закрытому члену, не нарушив инкапсуляции класса SmallInt и не вызвав тем самым ошибку компиляции? Если вы посмотрите на определение класса SmallInt, то заметите, что глобальная функция compareLess() объявлена как дружественная (friend). Если функция объявлена таким образом, то ей доступны закрытые члены класса. (Друзья классов рассматриваются в разделе 15.2.)  
Теперь мы готовы определить специализацию шаблона min(). Она следующим образом использует функцию compareLess().

// специализация min() для массива объектов SmallInt

template<> SmallInt min<smallInt>(SmallInt\* array, int size)

{

SmallInt min\_val = array[0];

for (int i = 1; i < size; ++i)

// при сравнении используется функция compareLess()

if (compareLess(array[i], min\_val))

min\_val = array[i];

print("Minimum value found: ");  
 print(min\_val);

return min\_val;  
}

Где мы должны объявить эту специализацию? Предположим, что здесь:

// ---- primer.h ----

namespace cplusplus\_primer {

// определение шаблона скрыто в пространстве имен

template <class Type>

Type min(Type\* array, int size) { /\* ... \*/ }

}

// ---- user.h ----  
class SmallInt { /\* ... \*/ };

void print(const SmallInt &);  
bool compareLess(const SmallInt &, const SmallInt &);

// ---- user.C ----  
#include <primer.h>  
#include "user.h"

// ошибка: это не специализация для cplusplus\_primer::min()  
template<> SmallInt min<smallInt>(SmallInt\* array, int size)  
{ /\* ... \*/ }  
// ...

К сожалению, этот код не работает. Явная специализация шаблона функции должна быть объявлена в том пространстве имен, где определен порождающий шаблон. Поэтому мы обязаны определить специализацию min() в пространстве cplusplus\_primer. В нашей программе это можно сделать двумя способами.  
Напомним, что определения пространства имен не обязательно непрерывны. Мы можем повторно открыть пространство имен cplusplus\_primer для добавления специализации:

// ---- user.C ----

#include <primer.h>

#include "user.h"

namespace cplusplus\_primer {  
 // специализация для cplusplus\_primer::min()  
 template<> SmallInt min<smallInt>(SmallInt\* array, int size)  
 { /\* ... \*/ }  
}  
SmallInt asi[4];

int main() {  
 // задать значения элементов массива asi с помощью функции-члена set()

using cplusplus\_primer::min; // using-объявление  
 int size = sizeof(asi) / sizeof(SmallInt);  
 // конкретизируется min(SmallInt\*,int)  
 min(&asi[0], size);  
}

Можно определить специализацию так, как мы определяем любой другой член пространства имен вне определения самого пространства: квалифицировав имя члена именем объемлющего пространства.

// ---- user.C ----

#include <primer.h>

#include "user.h"

// специализация для cplusplus\_primer::min()  
// имя специализации квалифицируется  
namespace {  
 template<> SmallInt cplusplus\_primer::  
 min<smallInt>(SmallInt\* array, int size)  
 { /\* ... \*/ }  
 // ...

Если вы, пользуясь библиотекой, содержащей определения шаблонов, захотите написать их специализации, то должны будете удостовериться, что их определения помещены в то же пространство имен, что и определения исходных шаблонов.

**Упражнение 10.15**

Поместим содержимое заголовочного файла <exercise.h> из упражнения 10.14 в пространство имен cplusplus\_primer. Как надо изменить функцию main(), чтобы она могла конкретизировать шаблон max(), находящийся в cplusplus\_primer?

**Упражнение 10.16**

Снова обращаясь к упражнению 10.14, предположим, что содержимое заголовочного файла <exercise.h> помещено в пространство имен cplusplus\_primer. Допустим, мы хотим специализировать шаблон функции max() для массивов объектов класса LongDouble. Нужно, чтобы специализация шаблона использовала функцию compareGreater() для сравнения двух объектов класса LongDouble, объявленную как:

// функция сравнения объектов класса LongDouble

// возвращает true, если parm1 больше parm2

bool compareGreater(const LongDouble &parm1,

const LongDouble &parm2);

Определение класса LongDouble выглядит следующим образом:

class LongDouble {

public:

LongDouble(double dval) : value(ival) {}

friend bool compareGreater(const LongDouble &,

const LongDouble &);

private:

double value;

};

Напишите определение функции compareGreater() и специализацию max(), в которой эта функция используется. Напишите также функцию main(), которая задает элементы массива ad, а затем вызывает специализацию max(), доставляющую его максимальный элемент. Значения, которыми инициализируется массив ad, должны быть получены чтением из стандартного ввода cin.

**10.11. Пример шаблона функции**

В этом разделе приводится пример, показывающий, как можно определять и использовать шаблоны функций. Здесь определяется шаблон sort(), который затем применяется для сортировки элементов массива. Сам массив представлен шаблоном класса Array (см. раздел [2.5](http://valera.asf.ru/cpp/book/c02.html)). Таким образом, шаблоном sort() можно пользоваться для сортировки массивов элементов любого типа.  
  
В главе [6](http://valera.asf.ru/cpp/book/c06.html) мы видели, что в стандартной библиотеке C++ определен контейнерный тип vector, который ведет себя во многом аналогично типу Array. В главе 12 рассматриваются обобщенные алгоритмы, способные манипулировать контейнерами, описанными в главе [6](http://valera.asf.ru/cpp/book/c06.html). Один из таких алгоритмов, sort(), служит для сортировки содержимого вектора. В этом разделе мы определим собственный “обобщенный алгоритм sort()” для манипулирования классом Array, упрощенной версии алгоритма из стандартной библиотеки C++.  
Шаблон функции sort() для шаблона класса Array определен следующим образом:

template <class elemType>

void sort(Array<elemType> &array, int low, int high) {

if (low < high) {  
 int lo = low;  
 int hi = high + 1;  
 elemType elem = array[lo];

for (;;) {  
 while (min(array[++lo], elem) != elem && lo < high) ;  
 while (min(array[--hi], elem) == elem && hi > low) ;

if (lo < hi)  
 swap(array, lo, hi);  
 else break;  
 }

swap(array, low, hi);  
 sort(array, low, hi-1);  
 sort(array, hi+1, high);  
 }  
 }

В sort() используются две вспомогательные функции: min() и swap(). Обе они должны определяться как шаблоны, чтобы иметь возможность обрабатывать любые типы фактических аргументов, с которыми может быть конкретизирован шаблон sort(). min() определена как шаблон функции для поиска минимального из двух значений любого типа:

template <class Type>

Type min(Type a, Type b) {

return a < b ? a : b;

}

swap() – шаблон функции для перестановки двух элементов массива любого типа:

template <class elemType>

void swap(Array<elemType> &array, int i, int j)

{

elemType tmp = array[ i ];

array[ i ] = array[ j ];

array[ j ] = tmp;

}

Убедиться в том, что функция sort() действительно работает, можно с помощью отображения содержимого массива после сортировки. Поскольку функция display() должна обрабатывать любой массив, конкретизированный из шаблона класса Array, её тоже следует определить как шаблон:

#include <iostream>

template <class elemType>  
 void display(Array<elemType> &array)  
{ //формат отображения: < 0 1 2 3 4 5 >

cout << "< ";  
 for (int ix = 0; ix < array.size(); ++ix)  
 cout << array[ix] << " ";  
 cout << ">\n";  
}

В этом примере мы пользуемся моделью компиляции с включением и помещаем шаблоны всех функций в заголовочный файл Array.h вслед за объявлением шаблона класса Array.  
Следующий шаг – написание функции для тестирования этих шаблонов. В sort() поочередно передаются массивы элементов типа double, типа int и массив строк. Вот текст программы:

#include <iostream>  
#include <string>  
#include "Array.h"

double da[10] = {  
 26.7, 5.7, 37.7, 1.7, 61.7, 11.7, 59.7,  
 15.7, 48.7, 19.7 };

int ia[16] = {  
 503, 87, 512, 61, 908, 170, 897, 275, 653,  
 426, 154, 509, 612, 677, 765, 703 };

string sa[11] = {  
 "a", "heavy", "snow", "was", "falling", "when",  
 "they", "left", "the", "police", "station" };

int main() {  
 // вызвать конструктор для инициализации arrd  
 Array<double> arrd(da, sizeof(da)/sizeof(da[0]));

// вызвать конструктор для инициализации arri  
 Array<int> arri(ia, sizeof(ia)/sizeof(ia[0]));

// вызвать конструктор для инициализации arrs  
 Array<string> arrs(sa, sizeof(sa)/sizeof(sa[0]));  
 cout << "sort array of doubles (size == "  
 << arrd.size() << ")" << endl;  
 sort(arrd, 0, arrd.size()-1);  
 display(arrd);  
 cout << "sort array of ints (size == "  
 << arri.size() << ")" << endl;  
 sort(arri, 0, arri.size()-1);  
 display(arri);  
 cout << "sort array of strings (size == "  
 << arrs.size() << ")" << endl;  
 sort(arrs, 0, arrs.size()-1);

display(arrs);  
 return 0;  
}

Если скомпилировать и запустить программу, то она напечатает следующее (эти строки искусственно разбиты на небольшие части):

sort array of doubles (size == 10)

< 1.7 5.7 11.7 14.9 15.7 19.7 26.7

37.7 48.7 59.7 61.7 >

sort array of ints (size == 16)

< 61 87 154 170 275 426 503 509 512

612 653 677 703 765 897 908 >

sort array of strings (size == 11)

< "a" "falling" "heavy" "left" "police" "snow"

"station" "the" "they" "was" "when" >

В числе обобщенных алгоритмов, имеющихся в стандартной библиотеке C++ (и в главе 12), вы найдете также функции min() и swap(). В главе 12 мы покажем, как их использовать.