Глава 4
Общие принципы

В этой главе описаны основные принципы стандартной библиотеки C++, необходимые для работы со всеми или с большинством из ее компонентов.

- Пространство имен std.
- Имена и форматы заголовочных файлов.
- Общие принципы обработки ошибок и исключений.
- Вызываемые объекты.
- Основы параллельности и многопоточности.
- Краткое описание распределителей памяти.

4.1. Пространство имен `std`

При использовании разных модулей и/или библиотек всегда существует вероятность конфликтов имен, которые возникают из-за того, что в модулях и библиотеках один и тот же идентификатор может использоваться для названия разных сущностей. В языке C++ эта проблема была решена введением *пространств имен*. Пространство имен — это определенная область видимости идентификаторов. В отличие от классов, пространства имен открыты для расширений, которые могут определяться в любых исходных текстах. Таким образом, в пространстве имен можно определять компоненты, распределенные по нескольким физическим модулям. Типичным примером такого компонента является стандартная библиотека C++, поэтому в ней используется пространство имен.

Практически все идентификаторы стандартной библиотеки C++ определяются в пространстве имен `std`. В стандарте C++11 это также относится к идентификаторам, введенным в документе TR1 и помещенным в пространство имен `std::tr1` (см. раздел 2.1).

Кроме того, в настоящее время в языке C++ есть пространство имен `posix`, которое зарезервировано, но пока не используется в стандартной библиотеке.

В следующем списке перечислены пространства имен, вложенные в пространство имен `std` в стандартной библиотеке C++:

- `std::rel_ops` (см. раздел 5.5.3);
- `std::rel_chrono` (см. раздел 5.7.1);
- `std::placeholders` (см. раздел 6.10.3);
- `std::regex_constants` (см. раздел 14.6);
- `std::this_thread` (см. раздел 18.3.7).

В соответствии с концепцией пространств имен существуют три варианта использования идентификатора из стандартной библиотеки C++.

- Явная квалификация идентификатора. Например, можно написать `std::ostream` вместо `ostream`. Полная команда вывода может иметь следующий вид:

```
std::cout << std::hex << 3.4 << std::endl:
```

- *Объявление using*. Например, следующий фрагмент программы предоставляет локальную возможность пропустить префикс `std::` для объекта `cout` и модификатор формата `endl`:

```
using std::cout;
using std::endl:
```

- В этом случае предыдущий оператор записывается так:

```
cout << std::hex << 3.4 << end!;
```

- *Директива using*. Это простейший вариант. После выполнения директивы `using` для пространства имен `std` все идентификаторы этого пространства доступны так, будто они были объявлены глобально. Например, оператор

```
using namespace std:
```

- позволяет написать

```
cout << hex << 3.4 << endl:
```

- Отметим, что в сложных программах это может привести к случайным конфликтам имен или, что еще хуже, к непредсказуемым последствиям из-за запутанных правил перегрузки. Никогда не используйте директиву `using`, если контекст неясен (например, в заголовочных файлах).

Примеры в книге довольно невелики, поэтому в завершенных примерах программ обычно используются директивы `using`.

4.2. Заголовочные файлы

В процессе стандартизации в язык C++ для всех идентификаторов было внедрено пространство имен `std`. Это изменение не имеет обратной совместимости со старыми заголовочными файлами, в которых идентификаторы стандартной библиотеки C++ объявлялись в глобальной области видимости. Кроме того, в процессе стандартизации изменились некоторые интерфейсы классов (впрочем, при этом по возможности обеспечивалась обратная совместимость). Для этой цели был разработан новый стиль имен стандартных заголовочных файлов, позволяющий разработчикам компиляторов сохранить обратную совместимость со старыми заголовочными файлами.

Определение новых имен для стандартных заголовочных файлов позволило стандартизировать расширения заголовочных файлов. Раньше использовались разные варианты расширений (например, `.h`, `.hpp` и `.hxx`). Однако решение, принятое относительно нового стандартного расширения заголовочных файлов, может показаться неожиданным — теперь стандартные заголовочные файлы вообще не имеют расширений. Таким образом,

директивы include для стандартных заголовочных файлов теперь выглядят примерно так:

```
#include <iostream>
#include <string>
```

Аналогичное правило действует для заголовочных файлов в стандарте языка C. Теперь заголовочные файлы языка C снабжаются префиксом c вместо прежнего расширения .h.

```
#include <cstdlib> // было: <stdlib.h>
#include <cstring> // было: <string.h>
```

В этих заголовочных файлах все идентификаторы объявляются в пространстве имен std.

Одно из преимуществ такого стиля именования файлов заключается в том, что он позволяет легко отличить старый заголовочный файл для функций работы со строками char* из языка C от стандартного заголовочного файла C++ для работы с классом string.

```
#include <string>  // Класс string из языка C++
#include <cstring> // Функции char* из языка C
```

Новая схема именования заголовочных файлов не означает, что файлы стандартных заголовков не имеют расширений с точки зрения операционной системы. Выполнение директивы include для стандартных заголовочных файлов зависит от реализации. Системы программирования C++ могут добавлять расширения и даже использовать встроенные объявления, не читая файл. Однако на практике большинство систем просто включает заголовок из файла, имя которого точно совпадает с именем, указанным в директиве include. В большинстве систем *стандартные* заголовочные файлы C++ просто не имеют расширений. В принципе, целесообразно указывать расширения для ваших собственных заголовочных файлов, чтобы упростить их идентификацию в файловой системе.

Для поддержки совместимости с языком C в стандарте сохранена поддержка “старых” стандартных заголовочных файлов языка C. При необходимости можно написать директиву

```
#include <stdlib.h>
```

В этом случае идентификаторы объявляются и в глобальной области видимости, и в пространстве имен std. Фактически эти заголовочные файлы ведут себя так, как если бы все идентификаторы были объявлены в пространстве имен std, после чего была выполнена директива using.

В стандарте нет спецификации заголовочных файлов C++ “старого” формата, таких как <iostream.h>. Таким образом, они не поддерживаются. На практике большинство поставщиков будут поддерживать их для обеспечения обратной совместимости. Отметим, что изменения в заголовках не ограничиваются введением пространства имен std. В общем, следует либо использовать старые имена заголовочных файлов, либо переключиться на новые стандартизированные имена.

4.3. Обработка ошибок и исключений

Стандартная библиотека языка C++ неоднородна. Она содержит программы из разнообразных источников, отличающихся стилями проектирования и реализации. Ярким

примером таких различий является обработка ошибок и исключений. Одни части библиотеки, например строковые классы, поддерживают подробную обработку ошибок, проверяя все возможные проблемы и генерируя исключение, если возникла ошибка. Другие компоненты, такие как стандартная библиотека шаблонов STL и массивы `valarray`, оптимизируются по быстродействию в ущерб безопасности, поэтому они редко проверяют логические ошибки и генерируют исключения только при возникновении ошибок времени выполнения.

4.3.1. Стандартные классы исключений

Все исключения, генерируемые языком или библиотекой, являются производными от базового класса `exception`. Этот класс является корнем иерархии, показанной на рис. 4.1. Стандартные классы исключений можно разделить на три категории.

1. Языковая поддержка.
2. Логические ошибки.
3. Ошибки времени выполнения.

Логических ошибок обычно удается избежать, поскольку их причины, например нарушение предусловия, находятся в самой программе. Ошибки времени выполнения, например нехватка ресурсов, вызываются причинами, внешними по отношению к программе.

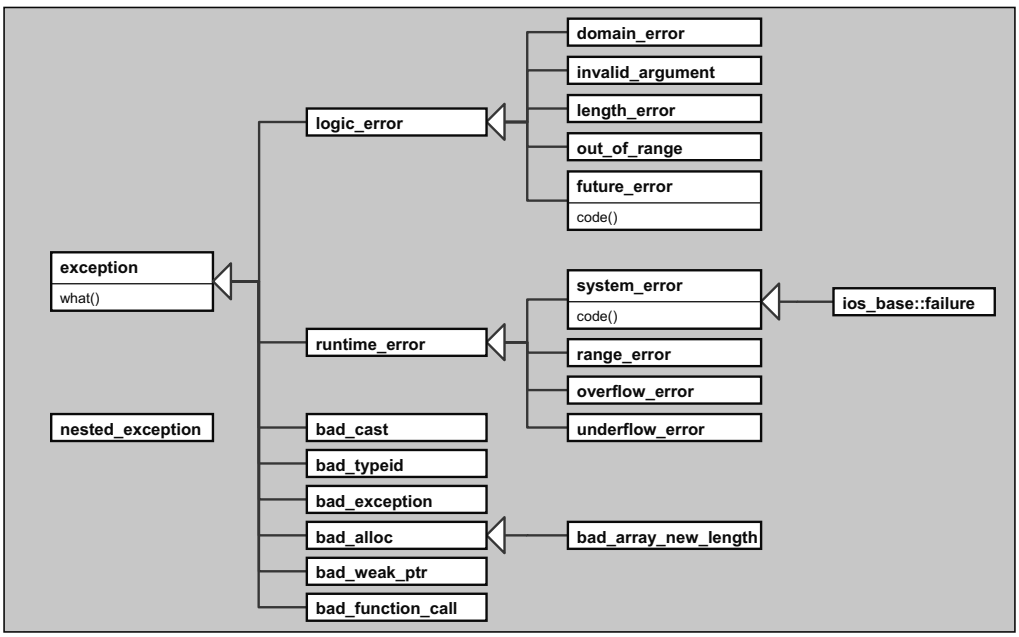

Рис. 4.1. Иерархия стандартных исключений

Классы исключений для поддержки языка

Исключения для поддержки языка используются средствами языка C++. Следовательно, было бы логичнее считать их частью базового языка, а не библиотеки. Эти исключения генерируются при неудачном выполнении некоторых операций.

- Исключение класса `bad_cast`, определенного в заголовке `<typeinfo>`, генерируется оператором `dynamic_cast`, если приведение типа ссылки во время выполнения завершается неудачей.
- Исключение класса `bad_typeid`, определенного в заголовке `<typeinfo>`, генерируется оператором `typeid`, предназначенным для идентификации типов во время выполнения. Если аргументом оператора `typeid` является нуль или нулевой указатель, генерируется исключение.
- Исключение класса `bad_exception`, определенного в заголовке `<exception>`, предназначено для обработки непредвиденных исключений. Оно может генерироваться функцией `unexpected()`, которая вызывается, если было сгенерировано исключение, не указанное в спецификации исключений соответствующей функции ((см. раздел 3.1.7).

Эти исключения также могут генерироваться библиотечными функциями. Например, исключение `bad_cast` может генерироваться функцией `use_facet()`, если аспект локализации (facet) в конкретной локализации (locale) недоступен (см. раздел 16.2.2).

Классы исключений для логических ошибок

Классы исключений для логических ошибок обычно являются производными от класса `logic_error`. Логическими называют ошибки, которые можно предотвратить, например, с помощью дополнительной проверки аргументов функции. Примерами логических ошибок являются нарушение логических предусловий или инварианта класса. Стандартная библиотека C++ содержит следующие классы для логических ошибок.

- Исключение класса `invalid_argument` используется для генерирования сообщений о некорректных аргументах, например, когда битовое множество (массив битов) инициализируется данными типа `char`, отличными от `'0'` и `'1'`.
- Исключение класса `length_error` используется для генерирования сообщений о попытке превысить установленные пределы, например, при добавлении к строке слишком большого количества символов.
- Исключение класса `out_of_range` используется для генерирования сообщений о том, что аргумент выходит за пределы допустимых значений, например, при использовании неправильного индекса в коллекциях наподобие массивов или строк.
- Исключение класса `domain_error` используется для генерирования сообщений об ошибке выхода за пределы области допустимых значений.
- В стандарте C++11 определен класс исключений `future_error` для генерирования сообщений о логических ошибках при использовании асинхронных системных вызовов (см. главу 18). Отметим, что ошибки времени выполнения в этих ситуациях приводят к генерированию исключения класса `system_error`.

В принципе, классы для логических ошибок определены в заголовке <stdexcept>, однако класс future_error определен в заголовке <future>.

Классы исключений для ошибок времени выполнения

Исключения, производные от класса runtime_error, используются для генерирования сообщений о событиях, не контролируемых программой, или об ошибках, которых трудно избежать. Стандартная библиотека C++ содержит следующие классы ошибок времени выполнения.

- Исключение класса range_error используется для генерирования сообщений об ошибках выхода за пределы допустимого диапазона во внутренних вычислениях. В соответствии со стандартом C++11 это исключение генерируется в стандартной библиотеке языка C++ при попытке выполнить преобразование широкой строки в строку байтов, и наоборот (см. раздел 16.4.4).
- Исключение класса overflow_error используется для генерирования сообщений об арифметическом переполнении. В стандартной библиотеке языка C++ это исключение может возникнуть, если битовое множество преобразовывается в целочисленное значение (см. раздел 12.5.1).
- Исключение класса underflow_error используется для генерирования сообщений о потере значимости при выполнении арифметических операций.
- В соответствии со стандартом C++11 исключение класса system_error используется для генерирования сообщений об ошибках, связанных с работой операционной системы. В стандартной библиотеке языка C++ это исключение может быть сгенерировано в контексте параллельной работы, например, классом thread, классами, управляющими "гонкой данных" и функцией async() (см. главу 18).
- Исключение класса bad_alloc, определенного в заголовке <new>, генерируется, если глобальный оператор new не достигает успеха, за исключением ситуаций, в которых используется версия оператора new, не генерирующая исключений (nothrow). Вероятно, это самое важное исключение времени выполнения, поскольку оно может появиться в любой момент в любой нетривиальной программе.
- В соответствии со стандартом C++11 исключение bad_array_new_length, производное от исключения bad_alloc, генерируется оператором new, если размер, переданный оператору new, меньше нуля или превышает установленный реализацией языка предел (т.е. это скорее логическая ошибка, а не ошибка времени выполнения).
- Исключение класса bad_weak_ptr, определенное в заголовке <memory>, генерируется при неудачном создании слабого указателя из разделяемого указателя (см. раздел 5.2.2).
- Исключение класса bad_function_call, определенного в заголовке <functional>, генерируется, когда объект function, играющий роль обертки для объекта, вызывается без указания целевого объекта (см. раздел 5.4.4).

Кроме того, часть библиотеки, реализующая ввод-вывод, генерирует исключение специального класса под названием `ios_base::failure`, определенного в заголовке `<ios>`. Исключение этого класса может быть сгенерировано, когда поток изменяет свое состояние из-за ошибки или достижения конца файла. В соответствии со стандартом C++11 этот класс является производным от класса `system_error`. Ранее он был непосредственным наследником класса `exception`. Точное поведение этого класса исключений описано в разделе 15.4.4.

Концептуально исключение `bad_alloc` можно рассматривать как системную ошибку. Однако по историческим причинам и из-за важности проблем при появлении ошибок, связанных с нехваткой памяти, проектировщики предпочитают генерировать исключение `bad_alloc`, а не `system_error`.

Вообще-то, классы ошибок времени выполнения определены в заголовочном файле `<stdexcept>`, однако класс `system_error` определен в заголовочном файле `<system_error>`.

Исключения, генерируемые стандартной библиотекой

Как уже указывалось, стандартная библиотека C++ может генерировать практически любые исключения. В частности, при выделении памяти для хранилища может генерироваться исключение `bad_alloc`.

Кроме того, поскольку компоненты библиотеки могут использовать код, написанный прикладным программистом, функции библиотеки могут опосредованно генерировать любые исключения.

Реализации стандартной библиотеки могут содержать дополнительные классы исключений, определенные на одном уровне со стандартными классами или в виде производных классов. Однако использование нестандартных классов нарушает переносимость кода, так как замена реализации стандартной библиотеки требует переработки программы. Таким образом, следует всегда использовать только стандартные классы исключений.

Заголовочные файлы для классов исключений

Классы исключений определены в разных заголовках. Для того чтобы использовать их, необходимо включить следующие заголовки:

```
#include <exception>     // для классов exception и bad_exception
#include <stdexcept>     // для большинства классов логических
                         // ошибок и ошибок времени выполнения
#include <system_error>  // для системных ошибок (C++11)
#include <new>           // для ошибок, связанных с нехваткой памяти
#include <ios>           // для ошибок ввода-вывода
#include <future>        // для ошибок, связанных с асинхронным режимом
#include <typeinfo>      // для классов bad_cast и bad_typeid
```

4.3.2. Члены классов исключений

Для обработки исключений в разделе `catch` можно использовать интерфейс, предоставляемый классами исключений. Во всех таких классах предусмотрена функция `what()`; в некоторых классах также содержится функция `code()`.

Функция-член what()

Во всех стандартных классах исключений для получения дополнительной информации, помимо сведений о самом типе исключения, можно использовать только одну функцию-член — виртуальную функцию-член what(), возвращающую строку, завершающуюся нулевым байтом.

```
namespace std {
   class exception {
      public:
        virtual const char* what() const noexcept;
        ...
   };


}
```

Содержимое строки, возвращаемой функцией what(), определяется реализацией. Отметим, что эта строка, завершающаяся нулевым байтом, может быть многобайтовой, которую можно преобразовывать и отображать как строку wstring (см. раздел 13.2.1). Строка в стиле языка C, возвращаемая функцией what(), является корректной, пока не будет уничтожен объект исключения, из которого она была получена, или пока для этого объекта не будет установлено новое значение.

Коды и условия ошибок

В классах исключений system_error и future_error имеется дополнительная функция-член, позволяющая получить подробную информацию об исключении. Но, прежде чем погружаться в детали, мы должны указать на различия между кодами и условиями ошибок.

- **Коды ошибок** — это легковесные объекты, инкапсулирующие значения кодов ошибок, которые могут зависеть от реализации. Однако некоторые коды ошибок стандартизированы.
- **Условия ошибок** — это объекты, предоставляющие переносимые абстракции для описания ошибок.

В зависимости от контекста стандартная библиотека C++ иногда определяет для исключений коды ошибок, а иногда — условия ошибок.

- Класс std::errc предоставляет *условия ошибок* для исключений класса std::system_error, соответствующие номерам стандартных системных ошибок, определенных в заголовке <cerrno> или <errno.h>.
- Класс std::io_errc предоставляет *код ошибки* для исключения класса std::ios_base::failure, генерируемого потоковыми классами в соответствии со стандартом C++11 (см. раздел 15.4.4).
- Класс std::future_errc предоставляет *коды ошибок* для исключений класса std::future_error, генерируемых библиотекой параллельного программирования (см. главу 18).

В табл. 4.1 представлены значения условий ошибок, определенные в стандартной библиотеке языка C++ для исключений класса `system_error`. Они представляют собой *перечисления с ограниченной областью видимости* (см. раздел 3.1.13), поэтому следует использовать префикс `std::errc::`. Значения этих условий необходимы для того, чтобы определить соответствующее значение `errno`, определенное в заголовке `<cerrno>` или `<errno.h>`. Это *не* код ошибки; коды ошибок обычно зависят от реализации.

Таблица 4.1. Условия ошибок для исключений класса `system_error`

Условие ошибки	Значение перечисления
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	CONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE

Продолжение табл. 4.1

Условие ошибки	Значение перечисления
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLCK
no_message_available	ENODATA
no_message	ENOMSG
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device_or_address	ENXIO
no_such_device	ENODEV
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT

Окончание табл. 4.1

Условие ошибки	Значение перечисления
too_many_files_open_in_system	ENFILE
too_many_files_open	EMFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

В табл. 4.2. приведены значения кодов ошибок, определенные стандартной библиотекой языка C++ для исключений типа future_errc. Они представляют собой *перечисления с ограниченной областью видимости* (см. раздел 3.1.13), поэтому следует использовать префикс std::future_errc::.[1]

Единственный код ошибки для исключений класса ios_base::failure определен значением std::io_errc::stream.

Таблица 4.2. Коды ошибок для исключений класса future_error

Код ошибки	Смысл
broken_promise	Разделяемое состояние аннулировано
future_already_retrieved	get_future() уже вызвана
promise_already_satisfied	Разделяемое состояние уже имеет значение/исключение или уже вызвано
no_state	Нет разделяемого состояния

Работа с кодами и условиями ошибок

Для кодов и условий ошибок в стандартной библиотеке языка C++ предусмотрены два разных класса: std::error_code и std::error_condition. Может создаться впечатление, что обработка ошибок представляет собой довольно запутанный процесс. Однако библиотека разработана так, чтобы программист всегда мог сравнивать коды ошибок с условиями ошибок, используя как объекты, так и значения перечислений.

Например, для любого объекта ошибки ec типа std::error_code или std::error_condition можно выполнить следующие операторы:

```
if (ec == std::errc::invalid_argument) { // проверка конкретного
                                         // условия ошибки
   ...
}
if (ec == std::future_errc::no_state) {  // проверка конкретного
                                         // кода ошибки
```

[1] Отметим, что в стандарте C++11 коды будущих ошибок явно определены значением future_errc::broken_promise и равны нулю. Но, поскольку нулевой код ошибки обычно означает отсутствие ошибок, это следует считать неправильным решением. Для того чтобы исправить эту проблему, все значения кодов для будущих ошибок определены как зависящие от реализации.

```
   ...
}
```

Таким образом, при обработке ошибок с помощью конкретных кодов или условий ошибок разница между ними несущественна.

Для работы с кодами и условиями ошибок класс `std::system_error`, а также его производный класс `std::ios_base::failure` и класс `std::_future_error` содержат дополнительную невиртуальную функцию-член `code()`, возвращающую объект класса `std::error_code`.[2]

```
namespace std {
   class system_error : public runtime_error {
      public:
         virtual const char* what() const noexcept;
         const error_code& code() const noexcept;
         ...
};

   class future_error : public logic_error {
      public:
         virtual const char* what() const noexcept;
         const error_code& code() const noexcept;
         ...
   };
}
```

В классе `error_code` предусмотрены функции-члены для получения детальной информации об ошибке.

```
namespace std {
   class error_code {
      public:


      const error_category& category() const noexcept;
      int value() const noexcept;
      string message() const;
      explicit operator bool() const noexcept;
      error_condition default_error_condition() const noexcept;
      ...
   };
}
```

Этот интерфейс основан на нескольких соображениях.

- Разные библиотеки могут использовать одни и те же целочисленные значения для разных кодов ошибок. По этой причине каждая ошибка имеет категорию и значение. Каждое значение является уникальным и имеет конкретный смысл только внутри категории.

[2] Строго говоря, эти объявления находятся в разных заголовочных файлах, а функция `what()`, хотя и не объявлена как виртуальная, наследует виртуальность из базового класса.

- Функция-член `message()` возвращает соответствующее сообщение, которое обычно является частью результата работы функции-члена `what()` для всех исключений, хотя это и не требуется.
- Операторная функция-член `operator bool()` сообщает, установлен ли код ошибки (значение `0` означает, что ошибки нет). При перехвате исключения этот оператор обычно возвращает значение `true`.
- Функция-член `default_error_condition()` возвращает соответствующий объект `error_condition`, также содержащий функции-члены `category()`, `value()`, `message()` и `operator bool()`.

```
namespace std {
   class error_condition {
      public:
         const error_category& category() const noexcept;
         int value() const noexcept;
         string message() const;
         explicit operator bool() const noexcept;
         ...
      };
   }
```

Класс `std::error_category` предоставляет следующий интерфейс:

```
namespace std {
   class error_category {
      public:
      virtual const char* name() const noexcept = 0;
      virtual string message (int ev) const = 0;
      virtual error_condition default_error_condition (int ev)
                                                      const noexcept;
      bool operator == (const error_category& rhs) const noexcept;
      bool operator != (const error_category& rhs) const noexcept;
      ...
   };
}
```

где функция `name()` возвращает имя категории, а функции `message()` и `default_error_condition()` возвращают сообщение и условие ошибки, заданное по умолчанию в соответствии с переданным значением (т.е. то, что вызывают соответствующие функции-члены класса `error_code`). Операторы == и != позволяют сравнивать категории ошибок.

В стандартной библиотеке шаблонов языка C++ определены следующие имена категорий:

- `"iostream"` — для исключений потоков ввода-вывода типа `ios_base::failure`;
- `"generic"` — для системных исключений типа `system_error`, если их значения соответствуют значениям ошибок в стандарте POSIX;
- `"system"` — для системных исключений типа `system_error`, если их значения не соответствуют значениям ошибок в стандарте POSIX;
- `"future"` — для исключений типа `future_error`.

Для каждой категории предусмотрены глобальные функции, возвращающие категорию[3].

```
const error_category& generic_category() noexcept; // в <system_errror>
const error_category& system_category() noexcept;  // в <system_error>
const error_category& iostream_category();         // в <ios>
const error_category& future_category() noexcept;  // в <future>
```

Таким образом, из объекта кода ошибки `e` можно вызвать функцию-член, чтобы определить, является ли он кодом сбоя ввода-вывода.

```
if (e.code().category() == std::iostream_category())
```

Следующая программа демонстрирует, как с помощью обобщенной функции можно обрабатывать (в данном случае выводить на экран) разные исключения:

```
// util/exception.hpp
#include <exception>
#include <system_error>
#include <future>
#include <iostream>

template <typename T>
void processCodeException (const T& e)
{
   using namespace std;
   auto c = e.code();
   cerr << "- category: " << c.category().name() << endl;
   cerr << "- value: " << c.value() << endl;
   cerr << "- msg: " << c.message() << endl;
   cerr << "- def category: "
        << c.default_error_condition().category().name() << endl;
   cerr << "- def value: "
        << c.default_error_condition().value() << endl;
   cerr << "- def msg: "
        << c.default_error_condition().message() << endl;
}

void processException()
{
   using namespace std;
   try {
       throw; // повторно генерируем исключение,
              // чтобы обработать его здесь
   }
   catch (const ios_base::failure& e) {
       cerr << "I/O EXCEPTION: " << e.what() << endl;
       processCodeException(e);
   }
   catch (const system_error& e) {
      cerr << "SYSTEM EXCEPTION: " << e.what() << endl;
```

[3] Вероятно, по недосмотру функция-член `iostream_category()` объявлена без ключевого слова `noexcept`.

```
      processCodeException(e);
   }
   catch (const future_error& e) {
      cerr << "FUTURE EXCEPTION: " << e.what() << endl;
      processCodeException(e);
   }
   catch (const bad_alloc& e) {
      cerr << "BAD ALLOC EXCEPTION: " << e.what() << endl;
   }
   catch (const exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
   }
   catch (...) {
      cerr << "EXCEPTION (unknown)" << endl;
   }
}
```

Это позволяет обработать исключения следующим образом:

```
try {
   ...
}
catch (...) {
   processException();
}
```

Другие члены

Остальные члены классов стандартных исключений создают, копируют, присваивают и уничтожают объекты исключений.

Отметим, что кроме функций-членов `what()` и `code()` ни в одном классе стандартных исключений нет дополнительного члена, описывающего вид исключения. Например, нет переносимого способа выяснить контекст исключения или неправильный индекс при ошибке выхода за пределы диапазона. Таким образом, единственным переносимым способом определить вид исключения является вывод сообщения, возвращаемого функцией `what()`.

```
try {
   ...
}
catch (const std::exception& error) {
   // выводим на печать сообщение об ошибке, определенное реализацией
   std::cerr << error.what() << std::endl;
   ...
}
```

Единственной возможной альтернативой может быть интерпретация точного типа исключения. Например, если было сгенерировано исключение `bad_alloc`, программа может попытаться получить больше памяти.

4.3.3. Передача исключений с помощью класса exception_ptr

В соответствии со стандартом C++11 стандартная библиотека языка C++ обеспечивает возможность хранить исключения в объектах типа exception_ptr, чтобы обработать их позднее или в другом контексте.

```
#include <exception>

std::exception_ptr eptr; // объект для хранения исключений (или nullptr)

void foo ()
{
   try {
       throw ...;
   }
   catch (...) {
       eptr = std::current_exception(); // сохраняем исключения
                                        // для дальнейшей обработки
   }
}

void bar ()
{
   if (eptr != nullptr) {
       std::rethrow_exception(eptr);   // обрабатываем сохраненное
                                       // исключение
   }
}
```

Функция-член current_exception() возвращает объект класса exception_ptr, ссылающийся на обрабатываемое исключение. Значение, возвращаемое функцией-членом current_exception(), является корректным, пока на него ссылается объект класса exception_ptr. Функция-член rethrow_exception() повторно генерирует сохраненное исключение, чтобы функция-член bar() работала так, будто в ней функция-член foo() сгенерировала первичное исключение.

Эта функциональная возможность особенно полезна при передаче исключения между потоками (см. раздел 18.2.1).

4.3.4. Генерирование стандартных исключений

Программист может генерировать стандартные исключения в своих библиотеках и программах. Все классы стандартных ошибок, включая логические ошибки и ошибки времени выполнения, содержащие интерфейсную функцию-член what(), имеют конструктор только с аргументом std::string и (начиная со стандарта C++11) const char*. Значение, переданное в объект этого класса, превращается в описание, возвращаемое функцией what(). Например, класс logic_error определен следующим образом:

```
namespace std {
   class logic_error : public exception {
     public:
```

```
        explicit logic_error (const string& whatString);
        explicit logic_error (const char* whatString); // since C++11
        ...
    };
}
```

Класс `std::system_error` предусматривает возможность создания объекта исключения с помощью передачи кода ошибки, строки для функции `what()` и необязательной категории.

```
namespace std {
    class system_error : public runtime_error {
      public:
         system_error (error_code ec, const string& what_arg);
         system_error (error_code ec, const char* what_arg);
         system_error (error_code ec);
         system_error (int ev, const error_category& ecat,
                       const string& what_arg);
         system_error (int ev, const error_category& ecat,
                       const char* what_arg);
         ...
    };
}
```

Для создания объекта `error_code` предусмотрены вспомогательные функции-члены `make_error_code()`, получающие только значение кода ошибки.

Класс `std::ios_base::failure` имеет конструктор, получающий строку, которая является аргументом функции `what()`, и (начиная со стандарта C++11) необязательный объект `error_code`. Класс `std::future_error` имеет лишь конструктор, получающий один объект класса `error_code`.

Таким образом, генерирование стандартного исключения не представляет затруднений.

```
throw std::out_of_range ("out_of_range (somewhere, somehow)");

throw
    std::system_error (std::make_error_code(std::errc::invalid_argument),
                       "argument ... is not valid");
```

Отметим, что нельзя генерировать исключения базового класса `exception` и любые исключения, предназначенные для языковой поддержки (`bad_cast`, `bad_typeid`, `bad_exception`).

4.3.5. Наследование классов стандартных исключений

Другая возможность использования классов стандартных исключений в своем коде заключается в определении специального класса исключений, производного прямо или косвенно от класса `exception`. Для этого следует убедиться в том, что механизмы функции `what()` или `code()` действительно работают, что вполне возможно, потому что функция-член `what()` является виртуальной. В качестве примера см. класс `Stack` в разделе 12.1.3.

4.4. Вызываемые объекты

В разных местах стандартной библиотеки языка C++ используется термин *вызываемый объект* (callable object), обозначающий объекты, которые так или иначе можно использовать для вызова некоей функциональности, например:

- функция, получающая дополнительные аргументы в виде аргументов функции;
- указатель на функцию-член, которая вызывается для объекта, передаваемого как первый дополнительный аргумент (должен быть ссылкой или указателем), и получает остальные аргументы как параметры функции-члена;
- функциональный объект (оператор `()` передаваемого объекта), которому передаются дополнительные аргументы;
- лямбда-функция (см. раздел 3.1.10), которая, строго говоря, является разновидностью функционального объекта.

Например:

```
void func (int x, int y);

auto l = [] (int x, int y) {
     ...
};

class C {
   public:
      void operator () (int x, int y) const;
      void memfunc (int x, int y) const;
   };

int main()
{
   C c;
   std::shared_ptr<C> sp(new C);

   // Функция bind() использует вызываемые
   // объекты для связывания аргументов:
   std::bind(func,77,33)();           // вызывает: func(77,33)
   std::bind(l,77,33)();              // вызывает: l(77,33)
   std::bind(C(),77,33)();            // вызывает: C::operator()(77,33)
   std::bind(&C::memfunc,c,77,33)();  // вызывает: c.memfunc(77,33)
   std::bind(&C::memfunc,sp,77,33)(); // вызывает: sp->memfunc(77,33)

   // Функция async() использует вызываемые объекты
   // для запуска задач (в фоновом режиме):
   std::async(func,42,77);            // вызывает: func(42,77)
   std::async(l,42,77);               // вызывает: l(42,77)
   std::async(c,42,77);               // вызывает: c.operator()(42,77)
   std::async(&C::memfunc,&c,42,77);  // вызывает: c.memfunc(42,77)
   std::async(&C::memfunc,sp,42,77);  // вызывает: sp->memfunc(42,77)
}
```

Для передачи объекта, для которого вызывается функция-член, можно использовать даже интеллектуальные указатели (см. раздел 5.2). Функция `std::bind()` описана в разделе 10.2.2, а функция `std::async()` — в разделе 18.1.

Для объявления *вызываемых объектов* в общем случае можно использовать класс `std::function<>` (см. раздел 5.4.4).

4.5. Параллельное программирование и многопоточность

До появления стандарта C++11 в языке C++ и его стандартной библиотеке не было поддержки параллельного программирования, хотя реализации могли предоставлять определенные гарантии. С появлением стандарта C++11 ситуация изменилась. В ядро языка и в библиотеку включены усовершенствования для поддержки параллельного программирования.

Например, для параллельной работы в ядро языка внесено несколько новшеств.

- Теперь существует модель памяти, гарантирующая, что обновления двух разных объектов двумя разными потоками происходят независимо друг от друга. До стандарта C++11 не было гарантии, что запись переменной типа `char` в одном потоке не повлияет на запись *другой* переменной типа `char` в другом потоке (см. раздел *"The memory model"* в работе [Stroustrup:C++0x]).
- Появилось новое ключевое слово `thread_local`, необходимое для определения переменных и объектов, зависящих от потока.

В библиотеку внесены следующие изменения:

- определенные гарантии безопасности потоков;
- вспомогательные классы и функции для параллельного программирования (запуск и синхронизация нескольких потоков).

Эти вспомогательные классы и функции рассматриваются в главе 18. Хотя гарантии, предоставляемые этими классами и функциями, обсуждаются на всем протяжении книги, здесь целесообразно сделать их краткий обзор.

Общие гарантии параллельного программирования в стандартной библиотеке C++

Рассмотрим общие ограничения, касающиеся поддержки параллельного программирования и многопоточности в стандартной библиотеке C++ в соответствии со стандартом C++11.

- В общем случае совместное использование библиотечного объекта несколькими потоками — при условии, что хотя бы один поток модифицирует этот объект, — может привести к непредсказуемым последствиям. Процитируем стандарт: "*Модификация объекта стандартного библиотечного типа, совместно используемого потоками, порождает риск возникновения непредсказуемых последствий, если*

объекты этого типа не определены явно как разделяемые без гонки данных, или если пользователь не предусмотрел блокировочный механизм".

- Особенно рискованная ситуация возникает, когда объект создается в одном потоке, а используется в другом. Аналогично непредсказуемые последствия могут возникнуть при уничтожении объекта в одном потоке, в то время как он продолжает использоваться в другом. Отметим, что это относится даже к объектам, предназначенным для синхронизации потоков.

Наиболее важные места, в которых *поддерживается* параллельный доступ к библиотечным объектам, описаны ниже.

- Для контейнеров STL (см. главу 7) и адаптеров контейнеров (см. главу 12) предусмотрены следующие гарантии.
  - Разрешается параллельный доступ только для чтения. Это явно подразумевает вызов неконстантных функций-членов `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()` и, за исключением ассоциативных контейнеров, оператора `[]` и доступа с помощью итераторов, если они не модифицируют контейнеры.
  - Разрешается параллельный доступ к *разным элементам* одного и того же контейнера (за исключением класса `vector<bool>`). Таким образом, разные потоки могут параллельно читать и/или записывать разные элементы одного и того же контейнера. Например, каждый поток может что-нибудь обрабатывать и сохранять результат в "своем" элементе совместно используемого вектора.
- При форматированном вводе и выводе в стандартный поток, синхронизированном с механизмом ввода-вывода из языка C (см. раздел 15.14.1), возможен параллельный доступ, хотя он и может привести к чередующимся символам. По умолчанию это относится к объектам `std::cin`, `std::cout` и `std::cerr`. Однако для строковых и файловых потоков, а также буферов потоков параллельный доступ приводит к непредсказуемым результатам.
- Параллельные вызовы функций `atexit()` и `at_quick_exit()` (см. раздел 5.8.2) синхронизированы. Это относится и к функциям `set_new_handler()`, `set_unexpected()`, `set_terminate()` и их `get_`-аналогам. Функция `getenv()` также синхронизирована.
- Для всех функций-членов распределителя памяти, заданного по умолчанию (см. главу 19), за исключением деструкторов, параллельный доступ является синхронизированным.

Кроме того, стандартная библиотека языка C++ гарантирует, что в ней нет скрытых побочных эффектов, нарушающих параллельный доступ к разным объектам. Таким образом, стандартная библиотека языка C++

- не обращается к досягаемым объектам, которые не требуются для конкретной операции;
- не может скрытно вводить совместно используемые статические объекты без синхронизации;

- позволяет реализациям распараллеливать операции только при условии, что при этом не возникают заметные побочные эффекты. Тем не менее не следует забывать о фактах, изложенных в разделе 18.4.2.

4.6. Распределители памяти

В некоторых частях стандартной библиотеки языка C++ используются специальные объекты для выделения и освобождения памяти, которые называются *распределителями памяти* (allocators). Они представляют собой определенную модель памяти и используются как абстракции, преобразующие запросы на выделение памяти в физическую операцию ее выделения. Одновременное использование разных объектов распределителя памяти позволяет использовать в своей программе разные модели памяти.

Изначально распределители памяти появились в библиотеке STL для решения тяжелой проблемы, связанной с разными типами указателей для разных моделей памяти (например, `near`, `far` и `huge`) на машинах PC. В настоящее время на основе распределителей памяти разработаны технические решения, не требующие изменения интерфейса и использующие разные модели памяти, такие как совместно используемая память, сбор мусора, объектно-ориентированные базы данных. Однако эти решения возникли относительно недавно и еще не получили широкого распространения (вероятно, со временем ситуация изменится).

В стандартной библиотеке C++ определен *распределитель памяти по умолчанию*.

```
namespace std {
   template <class T>
   class allocator;
}
```

Распределитель памяти по умолчанию используется во всех тех ситуациях, когда распределитель может передаваться в качестве аргумента. Он вызывает стандартные механизмы выделения и освобождения памяти, т.е. операторы `new` и `delete`. Однако, когда и как эти операторы должны вызываться, остается неопределенным. Таким образом, конкретная реализация распределителя памяти по умолчанию может, например, выполнять внутреннее кеширование выделяемой памяти.

В большинстве программ используется распределитель памяти по умолчанию, но некоторые библиотеки предоставляют специальные распределители памяти, которые просто передаются в виде аргументов. Необходимость самостоятельно программировать распределители памяти возникает очень редко. На практике обычно вполне достаточно использовать распределитель памяти по умолчанию. Мы отложим обсуждение распределителей памяти до главы 19, в которой рассматриваются не только распределители памяти, но и их интерфейсы.