Обработка ошибок в Си

Программирование на Си не обеспечивает прямой поддержки обработки ошибок, но, будучи языком системного программирования, оно предоставляет доступ на более низком уровне в форме возвращаемых значений. Большинство вызовов функций языка Си или даже `Unix` возвращают -1 или `NULL` в случае любой ошибки и устанавливают код ошибки `errno`. Использование `errno` доступно при подключении заголовочного файла `<errno.h>`

Переменная `errno` хранит целочисленный код последней ошибки. В каждом потоке существует своя локальная версия `errno`, чем и обусловливается её безопасность в многопоточной среде. Обычно `errno` реализуется в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер. При запуске программы значение `errno` равно нулю. Программист C может проверить возвращаемые значения функции и может предпринять соответствующие действия в зависимости от возвращаемого значения.

Язык программирования C предоставляет функции `perror()` и `strerror()`, которые можно использовать для отображения текстового сообщения, связанного с `errno`.
Функция `perror()` отображает строку, которую ей передают, затем двоеточие, пробел, а затем текстовое представление текущего значения `errno`.
Функция `strerror()`, возвращает указатель на текстовое представление текущего значения `errno`.

Пример программы на языке Си, обрабатывающего состояние ошибки:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {
    FILE * pf;
    int errnum;
    pf = fopen ("unexist.txt", "rb");
    if (pf == NULL) {
        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n",
strerror( errnum ));
    } else {
       fclose (pf);
    }
    return 0;
}
```

Часто во время деления любого числа программисты не проверяют, равен ли делитель нулю. Такое деление создаёт ошибку времени выполнения. Код ниже исправляет это, проверяя, равен ли делитель нулю перед делением:

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

int main() {
    int dividend = 20;
    int divisor = 0;
    int quotient;

    if( divisor == 0){
        fprintf(stderr, "Division by zero! Exiting...\n");
exit(-1);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
    exit(0);
}
```

В языке Си принято завершать программу со статусом `EXIT_SUCCESS`. Обычно `EXIT_SUCCESS` является статусом и определяется как 0. Если в программе возникла ошибка, и из-за неё придется завершить программу, то следует выйти со статусом `EXIT_FAILURE`, который определен как 1. В ОС Linux код ошибки можно проверить с помощью команды `$?`. Вызов `echo $?` возвращает код ошибки последней запущенной программы.

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

int main() {
    int dividend = 20;
    int divisor = 0;
    int quotient;

    if( divisor == 0) {
        fprintf(stderr, "Error! Exiting...\n");
        exit(EXIT_FAILURE);
    }

    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
    exit(EXIT_SUCCESS);
}

…

echo $? << 1
```

Часто в языке Си можно встретить оператор `goto` в тех местах, где имеет место ветвление с проверкой на ошибки исполнения. В таком случае переход на метку необходим тогда, когда нужно пропустить часть кода, который не должен выполниться в случае если ошибка будет выявлена. Как правило переход осуществляется с целью очистки выделенных ресурсов.

```
FILE * pf0, *pf1, *pf2;
pf0 = fopen ("error_goto.c", "rb");
if (pf0 == NULL) {perror("Perror 0"); goto exit;}
pf1 = fopen ("unexist.txt", "rb");
if (pf1 == NULL){perror("Perror 1"); goto close_zero_file;}
pf2 = fopen ("some.txt", "rb");
    if (pf2 == NULL){perror(«Perror 2");goto close_first_file;}
    // какой либо код
    fclose(pf2);
close_first_file:
    puts("close_first_file");
    fclose(pf1);
close_zero_file:
    puts("close_zero_file");
    fclose(pf0);
exit:
    puts("goto exit");
```

Assert

В языке Си и C++ часто используется макрос `assert`, позволяющий проверить значения произвольных данных в произвольном месте программы. Макрос имеет следующую сигнатуру:

```
void assert(int exp);
```

Для его использования необходимо подключить заголовок `assert.h`. В качестве `exp` передаётся некоторое выражение или переменная. Если `exp` не равен нулю, то макрос `assert` не производит никаких действий, иначе пишется сообщение об ошибке в `stderr` и происходит завершение программы. В результате тестовых запусков программы assert может быть необходим, однако в релизных версиях кода нужды в нём может и не быть. Макрос обязательно раскрывается в ветвление кода, что приведёт к дополнительной нагрузке на процессор, которую можно избежать определив макрос NDEBUG перед включением заголовка `assert.h`. В таком случае все assert-ы будут проигнорированы при компиляции.

В рамках языка Си используется заголовочный файл `assert.h`, в рамках языка C++ используется `cassert`. Пример использующий `assert`:

```
#include <iostream>
#include <cassert>

void print_adds(int* value)
{
  assert(value != NULL);
  std::cout << "Адрес значения в памяти = " << value << std::endl;
}

int main()
{
  int  a = 10;
  int *ptr1 = &a;
  int *ptr2 = NULL;
  print_adds(ptr1);
  print_adds(ptr2);
  return 0;
}
```

При выполнении в консоль возвращается ошибка `a.out: assert.cpp:6: void print_adds(int*): Assertion 'value != NULL' failed`. Если при компиляции определить макрос `NDEBUG` (`-DNDEBUG`), то `assert` будет проигнорирован.

Static assert

В C++11 добавили еще один тип `assert`-а — `static_assert`. В отличие от `assert`, который срабатывает во время выполнения программы, `static_assert` срабатывает во время компиляции, вызывая ошибку компилятора, если условие не является истинным. Если условие ложное, то выводится диагностическое сообщение. Вот пример использования `static_assert` для проверки размеров определенных типов данных:

```
#include <iostream>

static_assert(sizeof(long) == 8,
"long must be 8 bytes");
static_assert(sizeof(int) == 4,
"int must be 4 bytes");

int main()
{
    return 0;
}
```

Поскольку `static_assert` обрабатывается компилятором, то условная часть `static_assert` также должна обрабатываться во время компиляции. Поскольку `static_assert` не обрабатывается во время выполнения программы, то стейтменты `static_assert` могут быть размещены в любом месте кода (даже в глобальном пространстве). В C++11 диагностическое сообщение должно быть обязательно предоставлено в качестве второго параметра. В C++17 предоставление диагностического сообщения является необязательным.

Исключения

Механизм исключений появился в языке C++. Он пришел на смену стандартному способу обработки ошибок из Си. Оснований появления нового метода было достаточно много. Во-первых, возвращаемые значения не всегда понятны. Если функция возвращает -1, обозначает ли это какую-то специфическую ошибку или это корректное возвращаемое значение? Часто это бывает трудно понять, не видя перед глазами код самой функции. Во-вторых когда кода много, то многие вещи могут пойти не так, как нужно, поэтому коды возврата нужно постоянно проверять. В-третьих функции могут возвращать только одно значение, хотя иногда необходимо получить как промежуточный результат выполнения функции, так и код ошибки. Исключения лишены этих недостатков. Обработка исключений как раз и обеспечивает механизм, позволяющий отделить обработку ошибок или других исключительных обстоятельств от общего потока выполнения кода. Это предоставляет больше свободы в конкретных ситуациях, уменьшая при этом беспорядок, который вызывают коды возврата.

Исключения в языке C++ реализованы с помощью трех ключевых слов, которые работают в связке друг с другом: `throw`, `try` и `catch`.
В языке C++ оператор `throw` используется для сигнализирования о возникновении исключения или ошибки. Сигнализирование о том, что произошло исключение, называется генерацией исключения. Для генерации исключений применяется ключевое слово `throw`, а за ним указывается значение любого типа данных, которое необходимо задействовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение. Выбрасывание исключений — это лишь одна часть процесса обработки исключений. В языке C++ используется ключевое слово `try` для определения блока стейтментов. Блок `try` действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке `try`, например:

```
try
{
    throw -1; // генерация исключения с помощью throw
}
```

Блок `try` не определяет, как будет обрабатываться исключение, он просто сообщает компилятору о том, что внутри блока может случиться исключение. Фактически, обработка исключений — это работа блока или блоков `catch`. Ключевое слово `catch` используется для определения блока кода, который обрабатывает исключения определенного типа данных. Пример блока `catch`, который обрабатывает исключения типа `int`:

```
catch (int a)
{
    // Обрабатываем исключение типа int
    std::cerr << "We caught an int exception with value" << a << std::endl;
}
```

Блоки `try` и `catch` работают вместе. Блок `try` обнаруживает любые исключения, которые были выброшены в нем, и направляет их в соответствующий блок `catch` для обработки. Блок `try` должен иметь, по крайней мере, один блок `catch`, который находится сразу же за ним, но также может иметь и несколько блоков `catch`, размещенных последовательно (друг за другом).
Как только исключение было поймано блоком `try` и направлено в блок `catch` для обработки, оно считается обработанным (после выполнения кода блока `catch`), и выполнение программы возобновляется. Параметры `catch` работают так же, как и параметры функции, причем параметры одного блока `catch` могут быть доступны и в другом блоке `catch` (который находится за ним).
Исключения фундаментальных типов данных могут быть пойманы по значению (параметром блока `catch` является значение), но исключения нефундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока `catch` является константная ссылка), чтобы избежать ненужного копирования.

Как и в случае с функциями, если параметр не используется в блоке `catch`, то имя переменной можно не указывать, это предотвратит вывод предупреждений компилятора о неиспользуемых переменных:

```
catch (double) // возможно не указываем имя переменной
{
    std::cerr << "We caught an exception of type
double" << std::endl;
}
```

Программа, которая использует `throw`, `try` и несколько блоков `catch`:

```
try
{
    throw -1; // стейтмент throw
}
catch (int a)
{
    // исключения типа int, сгенерированные в блоке try, обрабатываются здесь
    std::cerr << "We caught an int exception with value: " << a <<
std::endl;
}
catch (double) // возможно не указываем имя переменной
{
    // исключения типа double, сгенерированные в блоке try, обрабатываются здесь
    std::cerr << "We caught an exception of type double" << std::endl;
}
catch (const std::string &str) // поймать исключения по константной ссылке
{
    // исключения типа std::string, сгенерированные внутри блока try, обрабатываются здесь
    std::cerr << "We caught an exception of type string" << std::endl;
}
std::cout << "Continuing our way!\n»;
```

Оператор `throw` используется для генерации исключения -1 типа `int`. Затем блок `try` обнаруживает оператор `throw` и перемещает его в соответствующий блок `catch`, который обрабатывает исключения типа `int`. Блок `catch` типа `int` и выводит соответствующее сообщение об ошибке. После обработки исключения, программа продолжает свое выполнение и выводит на экран «Continuing our way!».

В итоге можно заметить, что:
При выбрасывании исключения (оператор `throw`), точка выполнения программы немедленно переходит к ближайшему блоку `try`. Если какой-либо из обработчиков `catch`, прикрепленных к блоку `try`, обрабатывает этот тип исключения, то точка выполнения переходит в этот обработчик и, после выполнения кода блока `catch`, исключение считается обработанным. Если подходящих обработчиков `catch` не существует, то выполнение программы переходит к следующему блоку `try`. Если до конца программы не найдены соответствующие обработчики `catch`, то программа завершает свое выполнение с ошибкой исключения. Компилятор не выполняет неявные преобразования при сопоставлении исключений с блоками `catch`. Например, исключение типа `char` не будет обрабатываться блоком `catch` типа `int`, а исключение типа `int`, в свою очередь, не будет обрабатываться блоком `catch` типа `float`.

Если исключение направлено в блок `catch`, то оно считается «обработанным», даже если блок `catch` пуст. Есть три распространенные вещи, которые выполняют блоки `catch`, когда они поймали исключение:
Во-первых, блок `catch` может вывести сообщение об ошибке (либо в консоль, либо в лог-файл). Во-вторых, блок `catch` может возвратить значение или код ошибки обратно в `caller`. В-третьих, блок `catch` может сгенерировать другое исключения.
Поскольку блок `catch` не находится внутри блока `try`, то новое сгенерированное исключение будет обрабатываться следующим блоком `try`.

Сгенерированные исключения с помощью throw вовсе не обязаны находиться непосредственно в блоке `try`, благодаря такой операции, как «раскручивание стека». Это предоставляет необходимую гибкость в разделении общего потока выполнения кода программы и обработки исключений. Например:

```
double mySqrt(double a){
    // если пользователь ввел отрицательное число, то выбрасывается исключение
    if (a < 0.0)
        throw "Can not take sqrt of negative number"; // выбрасывается исключение типа const char*
    return sqrt(a);
}

int main(){
    std::cout << "Enter a number: ";
    double a;
    std::cin >> a;

    try{
        double d = mySqrt(a);
        std::cout << "The sqrt of " << a << " is " << d << '\n';
    }
    catch (const char* exception){
        std::cerr << "Error: " << exception << std::endl;
    }
}
```

При генерации исключения компилятор смотрит, можно ли сразу же обработать это исключение (для этого нужно, чтобы исключение выбрасывалось внутри блока try). Поскольку точка выполнения не находится внутри блока try, то и обработать исключение немедленно не получится. Таким образом, выполнение функции mySqrt() приостанавливается, и программа будет анализировать, может ли caller (который и вызывает mySqrt()) обработать это исключение. Если нет, то компилятор завершает выполнение caller-а и переходит на уровень выше — к caller-у, который вызывает текущего caller-а, чтобы проверить, сможет ли тот обработать исключение. И так последовательно до тех пор, пока не будет найден соответствующий обработчик исключения, или пока функция `main()` не завершит свое выполнение без обработки исключения. Этот процесс называется раскручиванием стека. Другими словами, блок `try` ловит исключения не только внутри себя, но и внутри функций, которые вызываются в этом блоке `try`.

Например:

```
void last()  // вызывается функцией three()
{
    std::cout << "Start last" << std::endl;
    std::cout << "last throwing int exception" << std::endl;
    throw -1;
    std::cout << "End last" << std::endl;
}

void three()  // вызывается функцией two()
{
    std::cout << "Start three" << std::endl;
    last();
    std::cout << "End three" << std::endl;
}
```

```
void two()  // вызывается функцией one()
{
    std::cout << "Start two" << std::endl;
    try
    {
        three();
    }
    catch(double)
    {
         std::cerr << "two caught double exception" << std::endl;
    }
    std::cout << "End two" << std::endl;
}
```

```
void one() // вызывается функцией main()
{
    std::cout << "Start one" << std::endl;
    try
    {
        two();
    }
    catch (int)
    {
         std::cerr << "one caught int exception" << std::endl;
    }
    catch (double)
    {
         std::cerr << "one caught double exception" << std::endl;
    }
    std::cout << "End one" << std::endl;
}
```

```
int main()
{
    std::cout << "Start main" << std::endl;
    try
    {
        one();
    }
    catch (int)
    {
         std::cerr << "main caught int exception" << std::endl;
    }
    std::cout << "End main" << std::endl;

    return 0;
}
```

Непосредственный `caller`, вызывающий функцию, в которой выбрасывается исключение, не обязан обрабатывать это исключение, если он этого не хочет. В примере, приведенном выше, функция `three()` не обрабатывает исключение, генерируемое функцией `last()`. Она делегирует эту ответственность на другой `caller` из стека. Если блок `try` не имеет обработчика `catch` соответствующего типа, то раскручивание стека происходит так же, как если бы этого блока `try` не было вообще. В примере, приведенном выше, функция `two()` не обрабатывает исключение, потому что у нее нет соответствующего обработчика `catch`. Когда исключение обработано, выполнение кода продолжается как обычно, начиная с конца блока `catch` (в котором это исключение было обработано). В примере, приведенном выше, функция `one()` обработала исключение, а затем продолжила свое выполнение выводом строки End one. К тому времени, когда точка выполнения возвращается обратно в функцию `main()`, исключение уже было сгенерировано и обработано. Раскручивание стека является очень полезным механизмом, так как позволяет функциям не обрабатывать исключения, если они этого не хотят. Операция раскручивания стека выполняется до тех пор, пока не будет обнаружен соответствующий блок `catch`.

В текущем примере функция `mySqrt()` выбрасывает исключение и предполагает, что его кто-то обработает. Что будет, если вызывающая функция не обернёт блоком `try` вызов функции `mySqrt()`?

```
double mySqrt(double a)
{
    // Если отрицательное число, то выбрасывается исключение типа const char*
    if (a < 0.0)
        throw "Can not take sqrt of negative number";

    return sqrt(a);
}
```

```
int main()
{
    std::cout << "Enter a number: ";
    double a;
    std::cin >> a;

    // однако, здесь нет никакого обработчика исключений
    std::cout << "The sqrt of " << a << " is " << mySqrt(a) << '\n';

    return 0;
}
```

Теперь предположим, что пользователь ввел `-5`, и `mySqrt(-5)` сгенерировало исключение. Функция `mySqrt()` не обрабатывает свои исключения самостоятельно, поэтому стек начинает раскручиваться, и точка выполнения возвращается обратно в функцию `main()`. Но, поскольку в `main()` также нет обработчика исключений, выполнение `main()` и всей программы прекращается. Когда main() завершает свое выполнение с необработанным исключением, то операционная система обычно уведомляет о том, что произошла ошибка необработанного исключения. Допустить этого ни в коем случае не следует.

В случае, если для конкретной функции не известно заранее её реализацию, однако необходимо поймать её ошибки в любом случае, существует обработчик `catch-all`. Для синтаксиса `catch-all` блока используется троеточие. Например:

```
try{
    throw 7; // выбрасывается исключение типа int
}
catch (double a)
{
    std::cout << "We caught an exception of type
double: " << a << std::endl;
}
catch (...) // обработчик catch-all
{
    std::cout << "We caught an exception of an
undetermined type!" << std::endl;
}
```

Поскольку для типа int не существует специального обработчика `catch`, то обработчик `catch-all` ловит это исключение.

Обработчик `catch-all` должен находиться последним в цепочке блоков `catch`. Это делается для того, чтобы исключения сначала могли быть пойманы обработчиками `catch`, адаптированными к конкретным типам данных (если они вообще существуют). Часто блок обработчика `catch-all` оставляют пустым. Такой обработчик ловит любые непредвиденные исключения и предотвращает раскручивание стека (и, следовательно, потенциальное завершение выполнения всей программы), однако здесь он не выполняет никакой обработки исключений.

Спецификации исключений

Спецификации исключений — это механизм объявления функций с указанием того, будет ли функция генерировать исключения (и какие именно) или нет. Это может быть полезно при определении необходимости помещения вызова функции в блок `try`.
Существуют три типа спецификации исключений, каждый из которых использует так называемый синтаксис throw (…).
Во-первых, возможно использовать пустой оператор `throw` для обозначения того, что функция не генерирует никакие исключения, которые выходят за её пределы:

```
int doSomething() throw(); // не выбрасываются исключения
```

Функция `doSomething()` все еще может генерировать исключения, только обрабатывать она должна их самостоятельно. Любая функция, объявленная с использованием `throw()` (как в вышеприведенном примере), должна немедленно прекратить выполнение программы, если она попытается сгенерировать исключение, которое приведет к раскручиванию стека.

Во-вторых, возможно использовать оператор `throw` с указанием типа исключения, которое может генерировать эта функция:

```
int doSomething() throw(double); // могут генерироваться исключения типа double
```

В-третьих, возможно использовать `throw(...)` Для обозначения того, что функция может генерировать разные типы исключений:

```
int doSomething() throw(...); // могут генерироваться любые исключения
```

Из-за плохой реализации и совместимости с компиляторами, и учитывая тот факт, что спецификации исключений больше напоминают заявления о намерениях, чем гарантии чего-либо, и то, что они плохо совместимы с шаблонами функций, и то, что большинство программистов C++ не знают о их существовании, приводит к тому, что использовать спецификации исключений не рекомендуется.

Классы-исключения

Исключения можно применять и в объектно ориентированном коде. Это может быть исключение при вызове метода объекта класса или исключение при не удачном конструировании объекта. Если конструктор не сработал, то следует сгенерировать исключение, которое сообщит, что объект не удалось создать. Создание объекта прерывается, а деструктор никогда не выполняется (также следует обратить внимание, что конструктор должен самостоятельно выполнять очистку памяти перед генерацией исключения). Исключения могут применяться например для перегрузки операторов в пользовательских классах. Например можно рассмотреть следующую перегрузку оператора индексации [] для простого целочисленного класса-массива:

```
int& ArrayInt::operator[](const int index)
{
    return m_data[index];
}
```

Эта функция отлично работает до тех пор, пока значением переменной `index` является корректный индекс массива. В такой ситуации может помочь механизм обработки ошибок. Если для этих целей использовать `assert`, то при вводе некорректного индекса программа просто прекратит выполнение, однако хотелось бы, чтобы вызывающая функция всего лишь приняла меры по противодействию обращению к не принадлежащей объекту классу памяти.

Лучшим выходом в данной ситуации будет использование механизма исключений, так как они не прерывают выполнение программы и не изменяют сигнатуры функции.

```
int& ArrayInt::operator[](const int index)
{
    if (index < 0 || index >= getLength())
        throw index;

    return m_data[index];
}
```

Теперь, если пользователь передаст недопустимый `index`, `operator[]` сгенерирует исключение типа `int`.
Одной из основных проблем использования фундаментальных типов данных (например, типа `int`) в качестве типов исключений является то, что они, по своей сути, являются неопределенными. Еще более серьезной проблемой является неоднозначность того, что означает исключение, когда в блоке `try` имеется несколько стейтментов или вызовов функций.

Например:

```
// используется перегрузка operator[] для ArrayInt
try
{
    int *value = new int(array[index1] +
array[index2]);
}
catch (int value)
{
    // какие исключения отловит этот catch?
}
```

В этом примере, блок catch поймает исключение типа `int`, что он сообщит? Был ли передаваемый `index` недопустим? Может оператор `+` вызвал целочисленное переполнение или может оператор new не сработал из-за нехватки памяти? Хотя возможно генерировать исключения типа `const char*`, которые будут указывать причину проблемы. Одним из способов решения этой проблемы является использование классов-исключений. Класс-Исключение — это обычный класс, который выбрасывается в качестве исключения.

Например можно ввести класс ArrayException, который возьмёт на себя роль хранителя данных об ошибке.

```
class ArrayException
{
    std::string m_error;
public:
    ArrayException(std::string error) :
m_error(error){
    }

    const char* getError() {
        return m_error.c_str();
    }
};
```

В таком случае можно будет генерировать исключения с помощью ArrayException:

```
class ArrayInt
{
    int *m_data;
    const size_t length;
public:
    ArrayInt( size_t m_length ) : m_data( new int[m_length]), length(m_length){}

    size_t getLength(){ return length;}

    int& operator[](const int index){
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

    ~ArrayInt(){
        delete[] m_data;
    }

};

int main()
{
    ArrayInt array;

    try
    {
        int value = array[7];
    }
    catch (ArrayException &exception)
    {
        std::cerr << "An array exception occurred (" << exception.getError() << ")\n";
    }
}
```

Используя такой класс, возможно генерировать исключение, возвращающее описание возникшей проблемы, это даст точно понять, что именно пошло не так. И, поскольку исключение `ArrayException` имеет уникальный тип, то с помощью цепочки из catch его можно отделить от остальных и обрабатывать соответствующим образом. Обратите внимание, в обработчиках исключений объекты класса-исключения принимать нужно по ссылке, а не по значению. Это предотвратит создание копии исключения компилятором, что является затратной операцией (особенно в случае, когда исключение является объектом класса), и предотвратит обрезку объектов при работе с дочерними классами-исключениями. Передачу по адресу лучше не использовать, если нет на это веских причин.

Так как возможно выбрасывать объекты классов в качестве исключений, а классы могут быть получены из других классов, то нужно учитывать, что произойдет, если будут использованы унаследованные классы в качестве исключений. Оказывается, обработчики могут обрабатывать исключения не только одного определенного класса, но и исключения дочерних ему классов. Например:

```
struct Parent {
    Parent() {}
};

struct Child: public Parent {
    Child() {}
};

int main()
{
    try{
        throw Child();
    }
    catch (Parent &parent){
        std::cerr << "caught Parent" << std::endl;
    }
    catch (Child &child)
    {
        std::cerr << "caught Child" << std::endl;
    }
}
```

Здесь выбрасывается исключение типа `Child`. Однако, результат выполнения данной программы:

```
>> caught Parent
```

Дочерние классы могут быть пойманы обработчиком родительского класса. Поскольку `Child` является дочерним классу `Parent`, то из этого следует, что `Child` «является» `Parent` («является» — тип отношений). Во-вторых, когда C++ пытается найти обработчик для выброшенного исключения, он делает это последовательно. Первое, что он проверяет — подходит ли обработчик исключений класса `Parent` для исключений класса `Child`. Поскольку `Child` «является» Parent, то блок `catch` для объектов класса `Parent` подходит и, соответственно, выполняется. В этом случае блок `catch` для объектов класса Child никогда не выполнится.

Чтобы этот пример работал по-другому, нам нужно изменить порядок последовательности блоков `catch`:

```
struct Parent {
    Parent() {}
};

struct Child: public Parent {
    Child() {}
};

int main()
{
    try{
        throw Child();
    }
    catch (Child &child)
    {
        std::cerr << "caught Child" << std::endl;
    }
    catch (Parent &parent){
        std::cerr << "caught Parent" << std::endl;
    }
}
```

…

>> caught Child

Таким образом, обработчик `Child` будет ловить и обрабатывать исключения класса `Child`. Исключения класса `Parent` не соответствуют обработчику `Child` (`Child` «является» `Parent`, но `Parent` «не является» `Child`) и, соответственно, будут обрабатываться только обработчиком `Parent`. То есть можно сформулировать правило: обработчики исключений дочерних классов должны находиться перед обработчиками исключений родительского класса.

std::exception

Многие классы и операторы из Стандартной библиотеки C++ выбрасывают классы-исключения при сбое. Например, оператор new и `std::string` могут выбрасывать `std::bad_alloc` при нехватке памяти. Неудачное динамическое приведение типов с помощью оператора `dynamic_cast` выбрасывает исключение `std::bad_cast` и т.д. Начиная с C++14, существует больше 20 классов-исключений, которые могут быть выброшены, а в C++17 их еще больше. Однако все эти классы-исключения являются дочерними классу `std::exception`. `std::exception` — это небольшой интерфейсный класс, который используется в качестве родительского класса для любого исключения, которое выбрасывается в Стандартной библиотеке C++.

В большинстве случаев, если исключение выбрасывается Стандартной библиотекой C++, то не имеет большого значения, было ли это неудачное выделение, конвертирование или что-либо другое. Достаточно знать, что произошло что-то катастрофическое, из-за чего в программе произошел сбой. Благодаря `std::exception` возможно настроить обработчик исключений типа `std::exception`, который будет ловить и обрабатывать как `std::exception`, так и все (20+) дочерние ему классы-исключения. Например:

```
int main()
{
    try
    {
        std::string s;
        s.resize(-1);  // генерируется исключение std::bad_alloc
    }
    // этот обработчик ловит std::exception и все дочерние ему классы-исключения
    catch (std::exception &exception)
    {
      std::cerr << "Standard exception: " <<
exception.what() << std::endl;
    }
}
```

В `std::exception` есть виртуальный метод `what()`, который возвращает строку C-style с описанием исключения. Большинство дочерних классов переопределяют функцию `what()`, изменяя это сообщение. Обратите внимание, эта строка C-style предназначена для использования только в качестве описания. Иногда необходимо обрабатывать определенный тип исключений несколько иначе, нежели остальные типы исключений. В таком случае возможно добавить обработчик исключений для этого конкретного типа, а все остальные исключения «перенаправить» в родительский обработчик. Например:

```
try
{
    // здесь должен находиться код, использующий Стандартную библиотеку C++
}
// этот обработчик ловит std::bad_alloc и все дочерние ему классы-исключения
catch (std::bad_alloc &exception){
    std::cerr << "You ran out of memory!" << '\n';
}
// этот обработчик ловит std::exception и все дочерние ему классы-исключения
catch (std::exception &exception){
    std::cerr << "Standard exception: " << exception.what()
<< std::endl;
}
```

Не стоит генерировать `std::exception` напрямую, и разработчики также должны придерживаться этого правила. Однако, возможно генерировать исключения других классов из Стандартной библиотеки C++, если они адекватно отражают потребности. Например std::runtime_error (находится в заголовочном файле stdexcept) является популярным выбором, так как имеет общее имя, а конструктор принимает настраиваемое сообщение:

```
try
{
    throw std::runtime_error("Bad things happened");
}
// этот обработчик ловит std::exception и все дочерние ему классы-исключения
catch (std::exception &exception)
{
    std::cerr << "Standard exception: " << exception.what() << std::endl;
}
```

Кроме всего прочего возможно определить свои собственные классы-исключения, дочерние классу `std::exception`, и переопределить виртуальный константный метод `what()`. Например:

```
class ArrayException: public std::exception
{
    std::string m_error;
public:
    ArrayException(std::string error):
 m_error(error)
    {}
    // возвращает std::string в качестве константной строки C-style
    // const char* what() const { return m_error.c_str(); } // до C++11
    const char* what() const noexcept {
        return m_error.c_str();
    } // C++11 и выше
};
```

В C++11 к виртуальной функции `what()` добавили спецификатор `noexcept` (который означает, что функция обещает не выбрасывать исключения самостоятельно). Следовательно, в C++11 и в более новых версиях переопределение метода `what()` также должно иметь спецификатор `noexcept`.
Можно выбирать, необходимо ли создавать свои собственные классы-исключения, использовать классы-исключения из Стандартной библиотеки C++ или писать классы-исключения, дочерние `std::exception`, всё зависит от целей.

std::optional

`std::optional` — вспомогательный тип, добавленный в C++17. По сути это обёртка для типа и флаг, который показывает, инициализировано ли значение или нет. Такая обёртка выразительно представляет объект, который может быть пустым. `std::optional` был добавлен в C++17 из `boost::optional`, где был доступен многие годы. Начиная с C++17 библиотека boost для использования `optional` не требуется, можно написать `#include <optional>` для использования этого типа. Для `std::optional` не нужно отдельно выделять память, так как он является частью словарных типов C++.

Для опциональных типов может использоваться псевдоним, хранящий пустое значение `std::nullopt`. В примере ниже можно видеть, что из функции возвращается `std::string`, полученный из `mName`, который сразу же оборачивает в `std::optional`. Если значение недоступно, то функция просто вернёт `std::nullopt`:

```
std::optional<std::string> FindName()
{
    if (name_available)
        return { mName };
    return std::nullopt; // то же самое, как если вернуть просто { };
}

// Использование
std::optional<std::string> optName = FindName();
if (optName){
    std::cout << *optName << std::endl;
}else{
    std::cout << "Опциональная переменная пуста!" << std::endl;
}
```

Существуют несколько вариантов создания std::optional переменных:

```
// пустая переменная
std::optional<int> oEmpty;
std::optional<float> oFloat = std::nullopt;

// прямое определение
std::optional<int> oInt(10);
std::optional oIntDeduced(10);

// make_optional
auto oDouble = std::make_optional(3.0);
auto oComplex = make_optional<std::complex<double>>(3.0, 4.0);

// in_place
std::optional<std::complex<double>> oInPlace{std::in_place,
3.0, 4.0};

// вызвать vector с прямой инициализацией {1, 2, 3}
std::optional<std::vector<int>> oVec(std::in_place, {1, 2,
3});

// копирование/присваивание
auto oIntCopy = oInt;
```

Для опциональных типов существует операция получения значения. Возможно несколько вариаций, например использовать `operator*()` и `operator->()` так же, как в итераторах. Если объект не содержит реального значения, то поведение не определено. Также возможно использовать метод `value()`, который возвращает значение или бросает исключение `std::bad_optional_access` или использовать метод `value_or(default)`, который возвращает значение, если доступно, или же возвращает `default`. Чтобы проверить, есть ли реальное значение в объекте, вы можете использовать метод `has_value()` или просто проверить объект с помощью `if (optional) {...}`, так как у опционального типа перегружен оператор приведения к `bool`.

```
// с помощью operator*()
std::optional<int> oint = 10;
std::cout<< "oint " << *opt1 << std::endl;

// с помощью value()
std::optional<std::string> ostr("hello");

try
{
    std::cout << "ostr " << ostr.value() << std::endl;
}
catch (const std::bad_optional_access& e)
{
    std::cout << e.what() << std::endl;
}

// с помощью value_or()
std::optional<double> odouble; // пустой
std::cout<< "odouble " << odouble.value_or(10.0) <<
std::endl;
```

Если уже существует опциональный объект, возможно поменять его значение с помощью методов `emplace, reset, swap` и `assign`. Если происходит присвоение объекту `std::nullopt`, то у реального объекта, который хранится в опциональном, будет вызван деструктор. Например:

```
class CarBrand{
    std::string mName;
public:
    explicit CarBrand(const std::string& str) : mName(str){
        std::cout << "CarBrand(\""<< mName << "\")" << std::endl;
    }
    ~CarBrand() {
        std::cout << "~CarBrand(\""<< mName << "\")" << std::endl;
    }
};

int main()
{
    std::optional<CarBrand> oEmpty;
    oEmpty.emplace("Mercedes"); // emplace
    oEmpty.emplace("Audi"); // вызовется ~Mercedes и создастся Audi
    oEmpty.reset(); // произойдёт обнуление вызовется ~Audi
    // oEmpty = std::nullopt; // то же самое
    oEmpty.emplace("Opel"); // присвоить новое значение
    oEmpty = CarBrand("Toyota");
}
```