Powody wprowadzenia obsługi wyjątków

- Podczas wykonania programu mogą wystąpić przypadki, które nie zostały przewidziane przez programistę
  - Np. użytkownik wprowadził złe dane
  - Kontakt z urządzeniem zewnętrznym został przerwany
- W celu uniknięcia przerwania pracy programu konieczne jest zaimplementowanie obsługi błędów
  - Jedną z możliwości jest obsługa błędów poprzez znaczniki statusów (używane w C)
  - W C++ natomiast wprowadzono nowy znacznie ogólniejszy mechanizm pozwalające na obsługę wyjątków

Obsługa wyjątków

- Wyjątek nie zawsze oznacza błąd
  - Błąd jest niejako podzbiorem wyjątków
- Sytuacją wyjątkową (wyjątkiem) może być wszystko co my programiści za to uznamy
- Obsługa wyjątków stanowi nowy sposób obsługi błędów i sytuacji nazwijmy to niecodziennych
  - Należy stosować kiedy tylko jest to możliwe
- Stanowi wbudowaną własność języka
- Umożliwia obsługę wyjątków w każdym ich znaczeniu za pomocą mechanizmu niezależnego od zasadniczego przepływu sterowania w programie

Obsługa błędów poprzez znaczniki statusów (C)

- Odbywa się poprzez kontrolę wartości zwracanych przez funkcję i wywoływaniu procedur obsługujących błędy
- Błędy są wykrywane i obsługiwane przez kod programu
  - Nie ma różnicy między zwykłym przepływem sterowania programem, a obsługą błędów
    - Standardowy przepływ jest wymieszany z blokami obsługi błędów
  - Wystąpienie błędu sygnalizowane jest jakąś specjalną wartością zwracaną
    - Pojawią się problemy kiedy funkcja jako legalną wartość może zwrócić zbiór pełny (np. wszystkie liczby typu `int` lub znaki, itp.)

Języki obiektowe (C++)

- W takich językach wiele operacji w ogóle nie zwraca żadnej wartości, czyli nie ma możliwości zwrócenia wartości sygnalizującej błąd
  - Np. tworzenie nowych obiektów (wywoływany jest konstruktor)
  - Wykrycie błędu to nie jeden problem, istotne jest również poprawne jego obsłużenie
- Istnieje potrzeba wbudowania mechanizmu, który pozwoliłby na oddzielenie wykrywania błędów od ich obsługi oraz umożliwiał przekazywanie informacji w inny sposób niż parametry zwracane
- Właśnie obsługa wyjątków daje takie możliwości

Koncepcja obsługi wyjątków

- Wyjątki przetwarzane są w języku C++ w następujący sposób
  - Jeżeli niespodziewana sytuacja wystąpi wewnątrz funkcji to zostanie to zakomunikowane za pomocą specjalnej instrukcji
    - Powoduje to przełączenie z normalnego trybu wykonywanie programu do obsługi wyjątków
    - W trybie tym opuszczane są wszystkie wywołane dotąd funkcje lub bloki, aż zostanie napotkany kod obsługi danego wyjątku
  - Dla poszczególnych instrukcji programu można definiować sposób działania jeśli pojawi się wyjątek

Słowa kluczowe służące obsłudze wyjątków

- **`try`** – służy określeniu zakresu instrukcji programu, w których wyjątki są przechwytywane i wysyłane do bloku obsługi błędów
- **`throw`** – umożliwia wyrzucenie obiektu wyjątku do programu
  - Powoduje przełączenie trybu pracy z normalnego do obsługi wyjątków
- **`catch`** – stosowane w celu przyjęcia obiektu wyjątku, a następnie jego obsługi
  - Zdefiniowany zakres wykonuję się podczas opuszczenia normalnego trybu pracy programu

Blok `try` i `catch`

- Wchodząc w programie do obszaru ryzykownego powinniśmy uprzedzić o tym kompilator
  - ```
    try {
            //ryzykowne instrukcje
    }
    catch(…)
    { /*reakcja na wyjątki */ }
    ```
- Wszystko co znajduje się w bloku `try` jest chronione, nawet wywołanie innych funkcji łącznie z bibliotecznymi
  - Niezależnie jak „głęboko" zostanie wyrzucony wyjątek
    ```

Instrukcja `throw`

- Jeżeli dzieje się coś niespodziewanego używamy instrukcji **`throw`**
  - **`throw obiekt;`**
- Możemy wyobrazić sobie dwie sytuacje
  - Rzucamy obiekt, który sam w sobie jest informacją o rodzaju sytuacji wyjątkowej
  - Rzucamy obiekt, który w sobie zawiera dodatkowe informacje o danej sytuacji wyjątkowej
- Różnica jest tylko widoczna od strony obsługi wyjątków, natomiast od strony sygnalizacji żadnej różnicy nie ma

Blok `catch`

- W bloku **`catch`** umieszczamy procedury obsługi wyjątku (-ów)
- Blok **`catch`** może tylko wystąpić bezpośrednio po bloku **`try`** lub innym bloku **`catch`**
- Bloków **`catch`** może być więcej, gdyż mogą one łapać obiekty różnych typów
  - Wtedy każdy blok **`catch`** przystosowany jest od złapania jednego konkretnego typu obiektu
  - Możliwe jest także umieszczenie takiego bloku **`catch`**, który złapie wszystkie wyjątki niezależnie od typu obiektu jak został wyrzucony
- Przykład cpp_8.1

Różnicie między wywołaniem obsługi błędów, a wywołaniem funkcji

- Obiekty zwracane
  - Funkcja może zwracać obiekty ściśle określonego typu i żadne inne
  - Instrukcja **`throw`** może wyrzucać obiekty dowolnego typu
- Różnica w przeniesieniu sterowania
  - Instrukcja **`return`** powoduje powrót do miejsca, skąd funkcja została wywołana
  - Instrukcja **`throw`** powoduje bezpowrotne opuszczenie wszystkich dalszych instrukcji (funkcji) i przenosi wykonanie do bloku **`catch`**

Kolejność bloków `catch`

- Kolejność bloków obsługi wyjątków ma istotne znaczenie
- Sytuacja bardzo podobna do instrukcji warunkowej **`if`**, **`else if`** i **`else`**
- Nie ma znaczenie czy np. w następnym bloku dopasowanie obiektu jest lepsze, zawsze wykonany zostanie ten blok, do którego jako pierwszego rzucany obiekt pasuje
  - Może to mieć szczególne znaczenie jeśli posługujemy się hierarchią klas

Bloki `try` i `catch` można zagnieżdżać

- Czasami może wydawać się lepsze zastosowanie zagnieżdżonej struktury bloków **`try`** i **`catch`**
  - Jeżeli rozróżnimy sytuacje wyjątkowe, z którymi możemy sobie poradzić lokalnie od sytuacji trudniejszych kiedy obsługa wyjątku musi odbyć się w dalszej części programu
- Jeśli instrukcja **`throw`** występuje w zagnieżdżonym bloku **`try`** to najpierw następuje próba obsługi wyjątku w blokach **`catch`** stojących bezpośrednio za nim. Dopiero jeżeli tam nie będzie możliwe obsłużenie wyjątku sprawdzane są bloki znajdujące się za zewnętrznym blokiem **`try`**
- Przykład cpp_8.2

Dopasowywanie typów w blokach `catch`

- Dana procedura obsługi nadaje się do pracy z danym typem jeżeli
  - Typ argumentu rzucanego jest taki sam jak typ argumentu oczekiwanego
  - Jeżeli typ argumentu oczekiwanego ma dodatkowo przydomek `const`
  - Gdy rzucamy dany typ, a oczekiwanym typem jest referencja do niego
  - Typ argumentu oczekiwanego jest publiczną klasą podstawową w stosunku do typu rzucanego
    - Bardzo nietypowe!!!
  - Typ argumentu rzucanego jest wskaźnikiem do jakiegoś typu, a oczekiwany typ jest wskaźnikiem do którego typ rzucany może być skonwertowany za pomocą konwersji standardowej
- Przykład cpp_8.3

Rzucanie obiektu klasy pochodnej, a odbieranie obiektu klasy bazowej

- Przy normalnym wywołaniu funkcji taka sytuacja nie może mieć miejsca (do przesyłanie obiektów używany jest stos)
- Dlaczego jest to możliwe
  - Ponieważ nie obowiązują zwykłe reguły związane ze stosem - sam stos nie bierze udziału w przekazywaniu argumentu wyjątku
  - Obiekt rzucany jest kopiowany do obiektu statycznego
- Odbierając obiekt klasy bazowej tracimy część informacji związanej z klasą pochodną
  - Operujemy na obiekcie klasy podstawowej, który nie da się przekształcić w obiekt klasy pochodnej nawet za pomocą rzutowania
- Ale informacja o obiekcie klasy pochodnej nie jest jeszcze bezpowrotnie stracona
  - Może zostać użyta dalej jeżeli wywołamy instrukcję `throw ;`
- Przykład cpp_8.4

Funkcyjny blok try-catch

- Istnieje możliwość ustanowienia bloku `try` wokół całej funkcji
  - Jest to wtedy cześć definicji funkcji
  - W szczególności interesujące jeśli dotyczy konstruktora z listą inicjalizacyjną, która wtedy też jest nim objęta
    - Wszystko co został skonstruowane zostaje w tej sytuacji zniszczone przed wejściem do bloku `catch`
    - W przypadku konstruktorów i destruktorów jeśli nie zostanie wyrzucony wyjątek nastąpi to automatycznie (`throw;`)
    - Dla wszystkich innych funkcji osiągnięcie końca bloku `catch` jest równoważne instrukcji `return;`
      - Czym to skutkuje?
  - Głównym celem tego bloku jest logowanie lub modyfikowanie czegoś a potem ponowne wyrzucenie kolejnego wyjątku
    - Bardzo rzadko używane w przypadku innych funkcji niż konstruktor
- Przykład 8.4a

Odwikłanie stosu

- Istnienie bloku **try** jest potrzebne gdyż w momencie rzucenia wyjątku następuje tzw. odwikłanie stosu
- Wykonywane jest sprzątanie obiektów automatycznych, które powstały w bloku **try**, aż do momentu wystąpienia sytuacji wyjątkowej
  - Wygląd stosu zostaje przywrócony do takiego jaki był przed wejściem do bloku **try**
  - Następuje to w łagodny sposób, tzn. wywoływane są chociażby destruktory
- Nie zostają zlikwidowane obiekty utworzone za pomocą operatora **new**!!!
  - Jednak na ogół tracimy dostęp do tych obiektów bo utracony zostanie wskaźnik do takiego obiektu, który jest za zwyczaj automatyczny
- Przykład cpp_8.5

Co zrobić z obiektami tworzonymi za pomocą `new`

- Po pierwsze możemy przed rzuceniem wyjątku skasować niepotrzebne już obiekty za pomocą operatora `delete`
- Postarać się o przekazanie adresu obiektu w taki sposób żeby „przeżył" odwikłanie stosu
  - Możemy mieć wskaźnik globalny
    - Nie jest to polecana metoda w szczególności jeżeli mam dużo takich wskaźników to wprowadzamy bałagan
  - Możemy wyposażyć obiekt, który będziemy wyrzucać w informację o pozostających obiektach stworzonych operatorem `new`

Co zrobić z obiektami tworzonymi za pomocą `new` . . .

- Wykorzystać inteligentny wskaźnik
  - Zaimplementować samemu zliczenie referencji
  - Wykorzystać gotowy z zewnętrznej biblioteki
  - Użyć istniejący wskaźnik z `std`
  - `template <typename T> class auto_ptr;`
    - Jest to szablon
      - `auto_ptr<MojaKlasa> ptr(new MojaKlasa);`
    - Zaimplementowany w postaci przenoszenia własności
    - Nie można go skopiować w normalnym tego słowa znaczeniu
    - Dwa takie wskaźniki nie mogą być w posiadaniu tego samego obiektu!
  - `template <typename T> class unique_ptr;`
    - *C++11*
- Przykład cpp_8.5a

`throw` i argumenty automatyczne

- Podczas odwikłania stosu wszystkie obiekty automatyczne zostają zniszczone
- Jeżeli argumentem instrukcji `throw` jest obiekt automatyczny to on też zostanie zniszczony
  - Ale informacja zostanie przekazana przez jego kopię, umieszczoną w obszarze zmiennych statycznych
- Dlaczego obiekt kopiowany jest do obszar statycznego?
  - Kompilator np. nie tworzy obiektu operatorem `new`, gdyż właśnie brak pamięci może być przyczyną wyrzucenia wyjątku
- Przykład cpp_8.6

`throw` i argumenty nie-automatyczne

- Oczywiście możemy za pomocą instrukcji `throw` rzucać obiekty nieautomatyczne
  - Obiekty globalne
  - Obiekty stworzone operatorem `new` (jeżeli powód rzucenia wyjątku nie jest brak pamięci)
- Jednak niezależnie jaki obiekt będziemy wyrzucać to `catch` zawsze odbiera kopię tego obiektu
- Przykład cpp_8.7

Wyjątki w destruktorach

- NIGDY nie należy rzucać wyjątków z destruktorów!!!
- Przyczyną dla której nie należy rzucać wyjątków z destruktorów jest ich sposób obsługi
  - W tym mechanizmie jest założenie, że nie wolno rzucać wyjątku dopóki poprzedni wyjątek nie został obsłużony przez kompilator
  - Rola kompilatora to przeniesienie sterowanie programu z punktu wyrzucenia wyjątku do odpowiedniego bloku `catch` (+odwikłanie stosu)
  - Nasza rola to reakcja na sytuacje wyjątkową w tym bloku
- Jeżeli jednak zostanie rzucony następny wyjątek przed obsłużeniem poprzedniego to program odmówi współpracy i zakończy brutalnie działanie
- Przykład cpp_8.8

Brak odpowiedniej obsługi wyjątku

- Program powinien obsługiwać wszystkie wyjątki
- Jeśli tak nie jest to program kończy działanie
  - Wywoływana jest funkcja `std::terminate()`, która to normalnie wywołuje funkcję `std::abort()`;
- Powody wywołanie `std::terminate()`
  - Nie złapany wyjątek
  - W mechanizmie obsługi wyjątków nastąpił wewnętrzny błąd
  - Jeżeli podczas odwikłania stosu zostanie rzucony następny wyjątek np. w destruktorze
  - Jeżeli między rzuceniem wyjątku, a złapaniem go w bloku `catch` wywołany zostanie konstruktor kopiujący, który rzuci wyjątek

Zmiana funkcji `std::terminate()`

- Istnieje możliwość wykonania innych czynności przez funkcje `std::terminate()` niż tylko wywołanie funkcji `std::abort()`
- Zmianę można dokonać za pomocą funkcji `std::set_terminate`
  - `void (*ptr)(); set_terminate(ptr);`
  - Czyli nasza funkcja wywoływana przez `terminate()` powinna być typu `void fun();`
  - Wg standardu nowa funkcja powinna na końcu wywołać funkcje `abort` lub `exit`
- W takim razie po co stosować inną funkcje?
  - Np. w trakcie testowania oprogramowanie, kiedy program przestaje działać z jakiś nieznanych nam przyczyn
- Przykład cpp_8.9

Funkcja deklaruje co może rzucać

- Jest to szczególnie istotne jeśli używamy bibliotekę i mamy dostępną tylko deklarację funkcji
  - Pomimo tego powinniśmy wiedzieć jakie wyjątki dana funkcja może wyrzucać, aby móc je obsłużyć
- Wprowadzono do deklaracji funkcji możliwość dodania informacji o typie rzucanych wyjątków
  - `void fun() throw(int, float, K); //` może rzucać typy `int, float, K` i żadne inne
  - `void fun(); //` funkcja może rzucić cokolwiek (kompatybilność ze starym zapisem)
  - `void fun() throw(); //` funkcja obiecuje, że nic nie będzie rzucać
- Zmiany w nowym standardzie (c++11)
  - `void fun() noexcept; //` funkcja obiecuje, że nic nie będzie rzucać
  - `void fun() noexcept(false); //` funkcja NIE obiecuje, że nic nie będzie rzucać

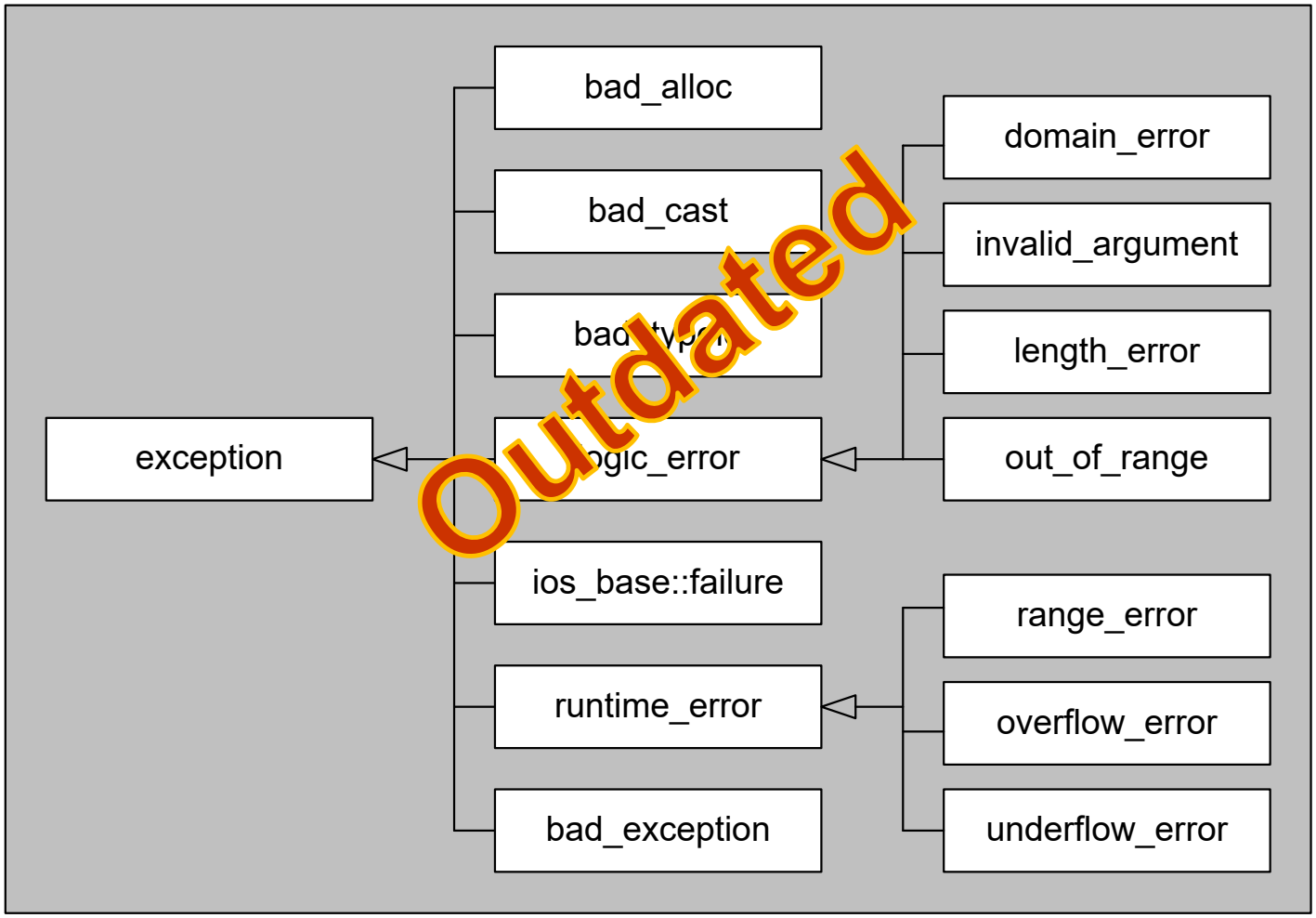
Niespodziewane wyjątki

- Z sytuacja nieoczekiwanych wyjątków mamy do czynienia kiedy funkcja obiecuję, że może rzucać jakieś wyjątki, a tak naprawdę może rzucić coś jeszcze innego
- Jeżeli funkcja wyrzuci taki nieoczekiwany wyjątek to zostanie wywołana wtedy specjalna metoda `std::unexpected();`
- Przez domniemanie funkcja ta wywołuje funkcję `std::terminate();`
- Mamy możliwość zmiany wywoływanej funkcji (tak jak poprzednio) za pomocą `std::set_unexpected`
  - `void (*ptr)(); set_unexpected(ptr);`
  - Niewątpliwie powinna tak czy inaczej zakończyć działanie programu lub ewentualnie rzuć nowy wyjątek np. `std::bad_exception`
- Przykład cpp_8.10

Klasy wyjątków

- W języku C++ zastosowano obiektowe podejście do wyjątków
  - Wyjątki są obiektami, w których umieszczane są informacje opisujące dany wyjątek
  - Dla różnych wyjątków mogą istnieć różne klasy
- Klasy wyjątków nie są wyjątkowymi klasami
  - Ich szczególne znaczenie odzwierciedla się tym, iż są używane przy instrukcjach **`throw`** i **`catch`**
- Klasy wyjątków powinny tworzyć hierarchię na szczycie, której znajduje się ogólna klasa wyjątków
  - Najczęściej jakaś standardowa klasa

Standardowe klasy wyjątków



Standardowe klasy wyjątków

- Sporo zmian od standardu c++11
- https://en.cppreference.com/w/cpp/error/exception

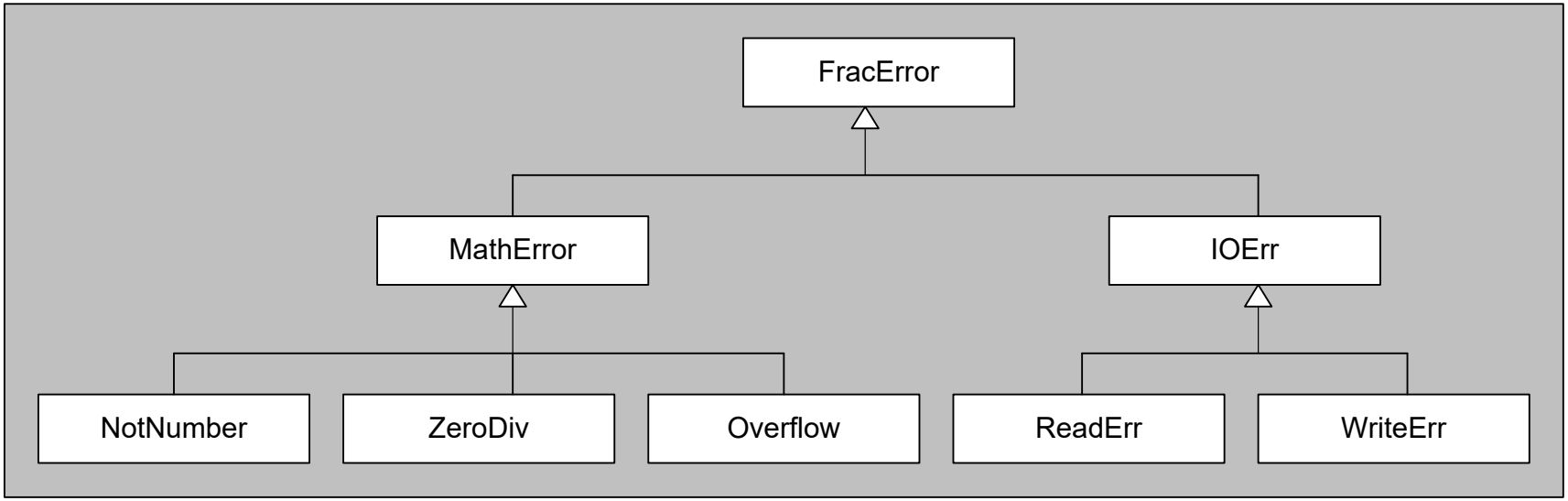
All exceptions generated by the standard library inherit from **std::exception**

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - tx_exception(TM TS)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- bad_variant_access(C++17)

Standardowe klasy wyjątków...

- Zdefiniowane są w pliku nagłówkowym **`exception`** (w większości)
- Definiowane nasze klasy wyjątków powinny być pochodne względem **`std::exception`**
  - Pozwala to na obsługę wyjątku w jednym bloku **`catch`**
- W klasie **`std::exception`** zdefiniowana jest wirtualna metoda **`what()`**, która zwraca komunikat specyficzny dla danej implementacji klasy
  - `const char* what() const throw();`
- Przykład cpp_8.11

Przykładowa hierarchia wyjątków



- Przykład cpp_8.12

Zwracanie kodu błędu, a rzucanie wyjątków

- Kiedy musimy (powinniśmy) rzucać wyjątek
  - W konstruktorze obiektu
  - Przy przeładowaniu operatorów
  - Przy oddzielaniu normalnych operacji od obsługi błędów
  - Przy przeniesieniu sterowania na dużą odległość
  - Kiedy funkcja powinna informować o różnych typach niepowodzeń
  - Jeżeli chcemy zobowiązać programistę do staranności
  - Przy szablonach klas
- Kiedy zwracać status błędu
  - W przypadku kiedy korzystamy z funkcji bibliotecznych, które wykorzystują ten mechanizm
    - Lepiej jest trzymać się jednej konwencji
  - POSIX
    - https://en.cppreference.com/w/cpp/error/errno_macros

Asercje, statyczne i dynamiczne

- Statyczne asercje pozwalają sprawdzać wrażenia stałe (`constexpr`) w czasie kompilacji programu
  - `static_assert (bool_constexpr, message)`
  - Przykład cpp_8.11a
- Dynamiczna asercja
  - Zdefiniowany w `<cassert>`
  - ```
    #ifdef NDEBUG
    #define assert(condition) ((void)0)
    #else
    #define assert(condition) /*implementation
    defined*/
    #endif
    ```
  - Przykład cpp_8.11b
    ```