3

WYJĄTKI

Doświadczenie - to nazwa, jaką nadajemy naszym błędom.
Oscar Wilde

Programiści nie są nieomylni. O tym wiedzą wszyscy, a najlepiej oni sami. W końcu to głównie do nich należy codzienna walka z większymi i mniejszymi błędami, wkradającymi się do kodu źródłowego. Dobrze, jeśli są to tylko usterki składniowe w rodzaju braku potrzebnego średnika albo domykającego nawiasu. Wtedy sam kompilator daje o nich znać.
Nieco gorzej jest, gdy mamy do czynienia z błędami objawiającymi się dopiero podczas działania programu. Może to spowodować nawet produkowanie nieprawidłowych wyników przez naszą aplikację (błędy logiczne).
Wszystkie tego rodzaju sytuację mają jędna cechę wspólną. Można bowiem (i należy) im zapobiegać: możliwe i pożądane jest takie poprawienie kodu, aby błędy tego typu nie pojawiały się. Aplikacja będzie wtedy działała poprawnie…

Ale czy na pewno? Czy twórca aplikacji może przewidzieć wszystkie sytuacje, w jakich znajdzie się jego program? Nawet jeśli jego kod jest całkowicie poprawny i wolny od błędów, to czy gwarantuje to jego poprawne działanie za każdym razem?…
Gdyby odpowiedź na chociaż jedno z tych pytań brzmiała „Tak”, to programiści pewnie rwaliby sobie z głów o połowę mniej włosów niż obecnie. Niestety, nikt o zdrowym rozsądku nie może obiecać, że jego kod będzie zawsze działać zgodnie z oczekiwaniami. Naturalnie, jeżeli jest on napisany dobrze, to w większości przypadków tak właśnie będzie. Od każdej reguły zawsze jednak mogą wystąpić **wyjątki**…

W tym rozdziale będziemy mówić właśnie o takich wyjątkach - albo raczej o **sytuacjach wyjątkowych**. Poznamy możliwości C++ w zakresie obsługi takich niecodziennych zdarzeń i ogólne metody radzenia sobie z nimi.

Mechanizm wyjątków w C++

Czym właściwie jest taka sytuacja wyjątkowa, która może narobić tyle zamieszania?…
Otóż:

Sytuacja wyjątkowa (ang. *exceptional state*) ma miejsce wtedy, gdy **warunki zewnętrzne** uniemożliwiają danemu fragmentowi kodu poprawne wykonanie. Ów fragment **nie jest winny** zaistnienia sytuacji wyjątkowej.

Ogólnie sytuacją wyjątkową można nazwać każdy błąd występujący podczas działania programu, który nie jest spowodowany przez błędy w jego kodzie. To coś w rodzaju przykrej niespodzianki: nieprawidołowych danych, nieprzewidzianego braku zasobów, i tak dalej. Takie przypadki mogą zdarzyć się w każdym programie, nawet napisanym pozornie bezbłędnie i działającym doskonale w **zwykłych warunkach**. Sytuacje wyjątkowe, jak sama ich nazwa wskazuje, zdarzają się bowiem tylko w **warunkach wyjątkowych**…

Tradycyjne metody obsługi błędów

Wystąpieniu sytuacji wyjątkowej zwykle **nie można zapobiec** - a przynajmniej nie może tego zrobić ten kawałek kodu, w którym ona faktycznie występuje. Jego rolą powinno być zatem poinformowanie o zainstniałym zdarzeniu kodu, który stoi „wyżej" w strukturze programu. Kod wyższego poziomu może wtedy podjąć jakieś sensowne akcje, a jeśli nie jest to możliwe - w ostateczności zakończyć działanie programu.

Działania wykonywane w reakcji na błędy są dość specyficzne dla każdego programu. Obejmowac mogą na przykład zapisanie informacji o zdarzeniu w specjalnym dzienniku, pokazanie komunikatu dla użytkownika czy też jeszcze inne czynności. Tym zagadnieniem nie bedziemy się więc zajmować.
Zobaczmy raczej, jakimi sposobami może odbywać się powiadamianie o błędach. Tutaj istnieje kilka potencjalnym rozwiązań - niektóre są lepsze, inne nieco gorsze… Oto te najczęściej wykorzystywane.

Dopuszczalne sposoby

Do całkiem dobrych metod informowania o niespodziewanych sytuacjach należy zwracanie jakiejś specjalnej wartości - indykatora. Wywołujący daną funkcję może wtedy sprawdzić, czy błąd wystąpił, kontrolując rezultaty zwrócone przez podprogram.

Zwracanie nietypowego wyniku

Najprostszą drogą poinformowania o błędzie jest zwrócenie pewnej specjalnej wartości, która w normalnych warunkach nie m prawa wystąpić. Aby to zilustrować, załóżmy przez chwilę, że mamy napisać funkcję obliczającą pierwiastek kwadratowy z podanej liczby. Wiedząc to, ochoczo zabieramy się do pracy, produkując np. taki oto kod:

```
float Pierwiastek(float x)
{
   // stała określająca dokładność
   static const float EPSILON = 0.0001f;


   /* liczymy pierwiastek kwadratowy metodą Newtona */

   // wybieramy punkt początkowy (połowę wartości)
   float fWynik = x / 2;

   // wykonujemy tyle iteracji, aby otrzymać rozsądne przybliżenie
   while (abs(x - fWynik * fWynik) > EPSILON)
         fWynik = (fWynik + x / fWynik) / 2;

   // zwracamy wynik
   return fWynik;
}
```

Funkcja ta wykorzystuje iteracyjną metodę Newtona do obliczania pierwiastka, ale to nie jest dla nas zbyt ważne, bowiem dotyczy zwykłej sytuacji. My natomiast mówimy o sytuacjach niezwykłych. Co nią będzie dla naszej funkcji?…

Na pewno będzie to podanie jej liczby ujemnej. Dopóki pozostajemy na gruncie prostej matematyki, jest to dla nas błędna wartość - nie można wyciągnąć pierwiastka kwadratowego z liczby mniejszej od zera.
Nie można jednak wykluczyć, że nasza funkcja otrzyma kiedyś liczbę ujemną. Będzie to błąd, sytuacja wyjątkowa - i trzeba będzie na nią zareagować. Ściśle mówiąc, trzeba będzie poinformować o niej wywołującego funkcję.

Specjalny rezultat

Jak można to zrobić?… Prostym sposobem jest **zwrócenie specjalnej wartości**. Niech będzie to wartość, która w normalnych warunkach nie ma prawa być zwrócona. W tym przypadku powinna to być taka liczba, której prawidłowe zwrócenie przez `Pierwiastek()` nie powinno mieć miejsca.
Jaka to liczba? Oczywiście - dowolna liczba ujemna. Powiedzmy, że np. `-1`:

```
if (x < 0)      return -1;
```

Po dodaniu tego sprawdzenia funkcja będzie już odporna na sytuacje z nieprawidłowym argumentem. Wywołujący ją będzie musiał natomiast sprawdzać, czy rezultat funkcji nie jest przypadkiem informacją o błędzie - np. w ten sposób:

```
float fLiczba;
float fPierwiastek;

if ((fPierwiastek = Pierwiastek(fLiczba)) < 0)
   std::cout << "Nieprawidlowa liczba";
else
   std::cout << "Pierwiastek z " << fLiczba << " to " << fPierwiastek;
```

Jak widać, przy wykorzystaniu wartości zwracanej operatora przypisania nie jest to szczególnie uciążliwe.

Wady tego rozwiązania

Takie rozwiązanie ma jednak kilka mankamentów. Pierwszą widać już tutaj: nie wygląda ono szczególnie estetycznie od strony wywołującego. Druga kwestia jest poważniejsza.

Jest nią problem doboru wartości specjalnej, sygnalizującej błąd. Zwracam uwagę, że **nie ma ona prawa** pojawienia się w jakiejkolwiek poprawnej sytuacji - musi ona jednoznacznie identyfikować błąd, a nie przydatny rezultat.
W przypadku funkcji `Pierwiastek()` było to proste, gdyż potencjalnych wartości jest mnóstwo: możemy przecież wykorzystać wszystkie liczby ujemne - poprawnym wynikiem funkcji jest bowiem tylko liczba dodatnia. Nie zawsze jednak musi tak być - czas na kolejny przykład matematyczny, tym razem z logarytmem o dowolnej podstawie:

```
float LogA(float a, float x)     { return log(x) / log(a); }
```

Tutaj także możliwe jest podanie nieprawidłowych argumentów: wystarczy, żeby choć jeden z nich był ujemny lub aby podstawa logarytmu (`a`) była równa jeden. Nie warto polegać na reakcji funkcji bibliotecznej `log()` w razie zaistnienia takiej sytuacji; lepiej samemu coś na to poradzić.
No właśnie - ale co? Możemy oczywiście skontrolować poprawność argumentów funkcji:

```
if (a < 0 || a == 1.0f || x < 0)
   /* błąd, ale jak o nim powiedzieć?... */
```

ale nie bardzo wiadomo, jaką specjalną wartość należałoby zwrócić. W zakresie typu `float` nie ma bowiem żadnej „wolnej” liczby, ponieważ poprawny wynik logarytmu może być każdą liczbą rzeczywistą.
Ostatecznie można zwrócić zero, który to wynik zachodzi normalnie tylko dla `x` równego `1`. Wówczas jednak sprawdzanie potencjalnego błędu byłoby bardzo niewygodne:

```
// sprawdzamy, czy rezultat jest równy zero, a argument różny od jeden;
// jeżeli tak, to błąd
if (((fWynik = LogA(fPodstawa, fLiczba)) == 0.0f) && fLiczba != 1.0f)
   std::cout << "Zly argument funkcji";
```

```
else
   std::cout << "Logarytm o podst. " << fPodstawa << " z " << fLiczba
             << " wynosi " << fWynik;
```

To chyba przesądza fakt, iż łączenie informacji o błędzie z właściwym wynikiem nie jest dobrym pomysłem.

Oddzielenie rezultatu od informacji o błędzie

Obie te dane trzeba od siebie odseparować. Funkcja powinna zatem zwracać dwie wartości: jedną „właściwą" oraz drugą, informującą o powodzeniu lub niepowodzeniu operacji.

Ma to rozliczne zalety - między innymi:
- pozwala przekazać więcej danych na temat charakteru błędu
- upraszcza kontrolę poprawności wykonania funkcji
- umożliwia swobodę zmian w kodzie i ewentualne rozszerzenie funkcjonalności

Wydaje się jednak, że jest dość poważny problem: jak funkcja miałaby zwracać dwie wartości?… Cóż, chyba brak ci pomysłowości - istnieje bowiem kilka dróg zrealizowania tego mechanizmu.

Wykorzystanie wskaźników

Nasza funkcja, oprócz normalnych argumentów, może przyjmować jeden wskaźnik. Za jego pośrednictwem przekazana zostanie dodatkowa wartość. Może to być informacja o błędzie, ale częściej (i wygodniej) umieszcza się tam właściwy rezultat funkcji.

Jak to wygląda? Oto przykład. Funkcja `StrToUInt()` dokonuje zamiany liczby naturalnej zapisanej jako ciąg znaków (np. `"21433"`) na typ `unsigned`:

```
#include <cmath>

bool StrToUInt(const std::string& strLiczba, unsigned* puWynik)
{
   // sprawdzamy, czy podany napis w ogóle zawiera znaki
   if (strLiczba.empty())  return false;


   /* dokonujemy konwersji */

   // zmienna na wynik
   unsigned uWynik = 0;

   // przelatujemy po kolejnych znakach, sprawdzając czy są to cyfry
   for (unsigned i = 0; i < strLiczba.length(); ++i)
         if (strLiczba[i] > '0' && strLiczba[i] < '9')
         {
               // OK - cyfra; mnożymy aktualny wynik przez 10
               // i dodajemy tę cyfrę
               uWynik *= 10;
               uWynik += strLiczba[i] - '0';
         }
         else
               // jeżeli znak nie jest cyfrą, to kończymy niepowodzeniem
               return false;

   // w przypadku sukcesu przepisujemy wynik i zwracamy true
   *puWynik = uWynik;
   return true;
}
```

Nie jest ona może najszybsza, jako że wykorzystuje najprostszy, „naturalny” algorytm konwersji. Nam jednak chodzi o coś innego: o sposób, w jaki funkcja zwraca rezultat i informację o ewentualnym błędzie.
Jak można zauważyć, typem zwracanym przez funkcję jest `bool`. Nie jest to więc zasadniczy wynik, lecz tylko znacznik powodzenia lub niepowodzenia działań. Zasadniczy rezultat to kwestia ostatniego parametru funkcji: należy tam przekazać wskaźnik na zmienną, która otrzyma wynikową liczbę.

Brzmi to może nieco skomplikowanie, ale w praktyce korzystanie z tak napisanej funkcji jest bardzo proste:

```
std::string strLiczba;
unsigned uLiczba;

if (StrToUInt(strLiczba, &uLiczba))
   std::cout << strLiczba << " razy dwa == " << uLiczba * 2;
else
   std::cout << strLiczba << " - nieprawidlowa liczba";
```

Możesz się spierać: „Ale przecież tutaj mamy wybitnego kandydata na połączenie rezultatu z informacją o błędzie! Wystarczy zmienić zwracany typ na `int` - wtedy wszystkie wartości ujemne mogłyby informować o błędzie!…”
Chyba jednak sam widzisz, jak to rozwiązanie byłoby naciągane. Nie dość, że użylibyśmy nieadekwatnego typu danych (który ma mniejszy zakres interesujących nas liczb dodatnich niż `unsigned`), to jeszcze ograniczylibyśmy możliwość przyszłej rozbudowy funkcji. Załóżmy na przykład, że na bazie `StrToUInt()` chcesz napisać funkcję `StrToInt()`:

```
bool StrToInt(const std::string& strLiczba, int* pnWynik);
```

Nie jest to trudne, jeżeli wykorzystujemy zaprezentowaną tu technikę informacji o błędach. Gdybyśmy jednak poprzestali na łączeniu rezultatu z informacją o błędzie, wówczas byłoby to problemem. Oto stracilibyśmy przecież całą „ujemną połówkę” typu `int`, bo ona teraz także musiałaby być przeznaczona na poprawne wartości.

Dla wprawy w ogólnym programowaniu możesz napisać funkcję `StrToInt()`. Jest to raczej proste: wystarczy dodać sprawdzanie znaku ‘minus’ na początku liczby i nieco zmodyfikować pętlę `for`.

Widać więc, że mimo pozornego zwiększenia poziomu komplikacji, ten sposób informowania o błedach jest lepszy. Nic dziwnego, że stosują go zarówno funkcje Windows API, jak i interfejsu DirectX.

Użycie struktury

Dla nieobytych ze wskaźnikami (mam nadzieję, że do nich nie należysz) sposób zaprezentowany wyżej może się wydawać dziwny. Istnieje też nieco inna metoda na odseparowanie właściwego rezultatu od informacji o błędzie.

Otóż parametry funkcji pozostawiamy bez zmian, natomiast inny będzie typ zwracany przez nią. W miejsce pojedynczej wartości (jak poprzednio: `unsigned`) użyjemy struktury:

```
struct RESULT
{
   unsigned uWynik;
   bool bBlad;
```

```
};
```

Zmodyfikowany prototyp będzie więc wyglądał tak:

```
RESULT StrToUInt(const std::string& strLiczba);
```

Myślę, że nietrudno zgadnąć, jakie zmiany zajdą w treści funkcji.

Wywołanie tak spreparowanej funkcji nie odbiega od wywołania funkcji z „wymieszanym" rezultatem. Musi ono wyglądać co najmniej tak:

```
RESULT Wynik = StrToUInt(strLiczba);
if (Wynik.bBlad)
   /* błąd */
```

Można też użyć warunku:

```
if ((Wynik = StrToUInt(strLiczba)).bBlad)
```

który wygląda pewnie dziwnie, ale jest składniowo poprawny, bo przecież wynikiem przypisania jest zmienna typu `RESULT`.

Tak czy inaczej, nie jest to zbyt pociągająca droga. Jest jeszcze gorzej, jeśli uświadomimy sobie, że dla każdego możliwego typu rezultatu należałoby definiować odrębną strukturę. Poza tym prototyp funkcji staje się mniej czytelny, jako że typ jej właściwego rezultatu (`unsigned`) już w nim nie występuje. [121]

Dlatego też o wiele lepiej używać metody z dodatkowym parametrem wskaźnikowym.

Niezbyt dobre wyjścia

Oba zaprezentowane w poprzednim paragrafie sposoby obsługi błędów zakładały proste poinformowanie wywołującego funkcję o zainstniałym problemie. Mimo tej prostoty, sprawdzają się one bardzo dobrze.

Istnieją aczkolwiek także inne metody raportowania błędów, które nie mają już tak licznych zalet i nie są szeroko stosowane w praktyce. Oto te metody.

Wywołanie zwrotne

Idea **wywołania zwrotnego** (ang. *callback*) jest nieskomplikowana. Jeżeli w pisanej przez nas funkcji zachodzi sytuacja wyjątkowa, wywołujemy inną funkcję pomocniczną. Taka funkcja może pełnić rolę „ratunkową" i spróbować naprawić okoliczności, które doprowadziły do powstania problemu - jak np. błędne argumenty dla naszej funkcji. W ostateczności może to być tylko sposób na powiadomienie o nienaprawialnej sytuacji wyjątkowej.

Uwaga o wygodnictwie

Zaleta wywołania zwrotnego uwidacznia się w powyższym opisie. Przy jego pomocy nie jesteśmy skazani na bierne przyjęcie do wiadomości wystąpienia błędu; przy odrobinie dobrej woli można postarać się go naprawić.
Nie zawsze jest to jednak możliwe. Można wprawdzie poprawić nieprawidłowy parametr, przekazany do funkcji, ale już nic nie zaradzimy chociażby na brak pamięci.

[121] Wykorzystanie szablonów zlikwidowałoby obie te niedogodności, ale czy naprawdę są one tego warte…?

Poza tym, technika *callback* z góry czyni pesymistyczne założenie, że sytuacje wyjątkowe będą trafiały się na tyle często, że konieczny staje się mechanizm wywołań zwrotnych. Jego stosowanie nie zawsze jest współmierne do problemu, czasem jest to zwyczajne strzelanie z armaty do komara. Przykładowo, w funkcji `Pierwiastek()` spokojnie możemy sobie pozwolić na inne sposoby informowania o błędach - nawet w obliczu faktu, że naprawienie nieprawidłowego argumentu byłoby przecież możliwe. Funkcja ta nie jest bowiem na tyle kosztowna, aby opłacało się chronić ją przed niespodziewanym zakończeniem.
Dlaczego jednak wywołanie zwrotne jest taki „ciężkim" środkiem? Otóż wymaga ono specjalnych przygotowań. Od strony programisty-klienta obejmują one przede wszystkim napisania odpowiednich funkcji zwrotnych. Od strony piszącego kod biblioteczny wymagają natomiast gruntowego obmyślenia mechanizmu takich funkcji zwrotnych: tak, aby nie mnożyć ich ponad miarę, a jednocześnie zapewnić dla siebie pewną wygodę i uniwersalność.

Uwaga o logice

Funkcje *callback* są też bardzo kłopotliwe z punktu widzenia logiki programu i jego konstrukcji. Zakładają bowiem, by kod niższego poziomu - jak funkcje biblioteczne w rodzaju wspomnianej `Pierwiastek()` lub `StrToUInt()` - wywoływały kod wyższego poziomu, związany bezpośrednio z działaniem samej aplikacji. Łamie to naturalną hierarchię „warstw" kodu i burzy porządek jego wykonywania.

Uwaga o niedostatku mechanizmów

Wreszcie trzeba wspomnieć, że w C++ nie ma dobrych sposobów na realizację funkcji zwrotnych. Owszem, mamy wskaźniki na funkcje - jednak one pozwalają pokazywać jedynie na funkcje globalne lub statyczne metody klas. Nie posiadamy natomiast niezbędnego w programowaniu obiektowym mechanizmu **wskaźnika na niestatyczną metodę obiektu** (ang. *closure*), przez co trudno jest zrealizować *callback*.

W poprzednim rozdziale opisałem pewien sposób na obejście tego problemu, ale jak wszystkie połowiczne rozwiązania, nie jest on zbyt elegancki…

Zakończenie programu

Wyjątkowy błąd może spowodować jeszcze jedną możliwą akcję: natychmiastowe zakończenie działania programu.

Brzmi to bardzo drastycznie i takie jest w istocie. Naprawdę trudno wskazać sytuację, w której byłoby konieczne przerwanie wykonywania aplikacji - zwłaszcza niepoprzedzone żadnym ostrzeżeniem czy zapytaniem do użytkownika. Chyba tylko krytyczne braki pamięci lub niezbędnych plików mogą być tego częściowym usprawiedliwieniem.
Na pewno jednak fatalnym pomysłem jest stosowanie tego rozwiązania dla każdej sytuacji wyjątkowej. I chyba nawet nie muszę mówić, dlaczego…

Wyjątki

Takie są tradycyjne sposobu obsługi sytuacji wyjątkowych. Były one przydatne przez wiele lat i nadal nie straciły nic ze swojej użyteczności. Nie myśl więc, że mechanizm, który zaraz pokażę, może je całkowicie zastąpić.

Tym mechanizmem są **wyjątki** (ang. *exceptions*). Skojarzenie tej nazwy z sytuacjami wyjątkowymi jest jak najbardziej wskazane. Wyjątki służą właśnie do obsługi niecodzienych, niewystępujących w normalnym toku programu wypadków.
Spójrzmy więc, jak może się to odbywać w C++.

Rzucanie i łapanie wyjątków

Technikę obsługi wyjątków można streścić w trzech punktach, które od razu wskażą nam jej najważniejsze elementy. Tak więc, te trzy założenia wyjątków są następujące:

- jeżeli piszemy kod, w którym może zdarzyć się coś wyjątkowego i niecodziennego, czyli po prostu sytuacja wyjątkowa, oznaczamy go odpowiednio. Tym oznaczeniem jest ujęcie kodu w blok `try` ('spróbuj'). To całkiem obrazowa nazwa: kod wewnątrz tego bloku nie zawsze może być poprawnie wykonany, dlatego lepiej jest mówić o **próbie** jego wykonania: jeżeli się ona powiedzie, to bardzo dobrze; jeżeli nie, będziemy musieli coś z tym fantem zrobić…
- załóżmy, że wykonuje się nasz kod wewnątrz bloku `try` i stwierdzamy w nim, że zachodzi sytuacja wyjątkowa, którą należy zgłosić. Co robimy? Otóż używamy instrukcji `throw` ('rzuć'), podając jej jednocześnie tzw. **obiekt wyjątku** (ang. *exception object*). Ten obiekt, mogący być dowolnym typem danych, jest zwykle informacją o rodzaju i miejscu zainstniałego błędu
- rzucenie obiektu wyjątku powoduje przerwanie wykonywania bloku `try`, zaś nasz rzucony obiekt „leci" sobie przez chwilę - aż zostanie przez kogoś **złapany**. Tym zaś zajmuje się blok `catch` ('złap'), następujący bezpośrednio po bloku `try`. Jego zadaniem jest reakcja na sytuację wyjątkową, co zazwyczaj wiąże się z odczytaniem obiektu wyjątku (rzuconego przez `throw`) i podjęciem jakiejś sensownej akcji

A zatem mechanizmem wyjątków żądzą te trzy proste zasady:

Blok `try` obejmuje **kod**, w którym **może zajść sytuacja wyjątkowa**.

Instrukcja `throw` **wewnątrz bloku `try`** służy do **informowania** o takiej sytuacji przy pomocy **obiektu wyjątku**.

Blok `catch` **przechwytuje obiekty** wyrzucone przez `throw` i **reaguje** na zainstaniałe sytuacje wyjątkowe.

Tak to wygląda w teorii - teraz czas na obejrzenie kodu obsługi wyjątków w C++.

Blok `try`-`catch`

Obsługa sytuacji wyjątkowych zawiera się wewnątrz bloków `try` i `catch`. Wyglądają one na przykład tak:

```
try
{
   ryzykowne_instrukcje
}
catch (...)
{
   kod_obsługi_wyjątków
}
```

ryzykowne_instrukcje zawarte wewnątrz bloku `try` są kodem, który poddawany jest pewnej specjalnej ochronie na wypadek wystąpienia wyjątku. Na czym ta ochrona polega - będziemy mówić w następnym podrozdziale. Na razie zapamiętaj, że w bloku `try` umieszczamy kod, którego wykonanie może spowodować sytuację wyjątkową, np. wywołania funkcji bibliotecznych.
Jeżeli tak istotnie się stanie, to wówczas sterowanie przenosi się do bloku `catch`. Instrukcja `catch` „łapie" występujące wyjątki i pozwala przeprowadzić ustalone działania w reakcji na nie.

Instrukcja `throw`

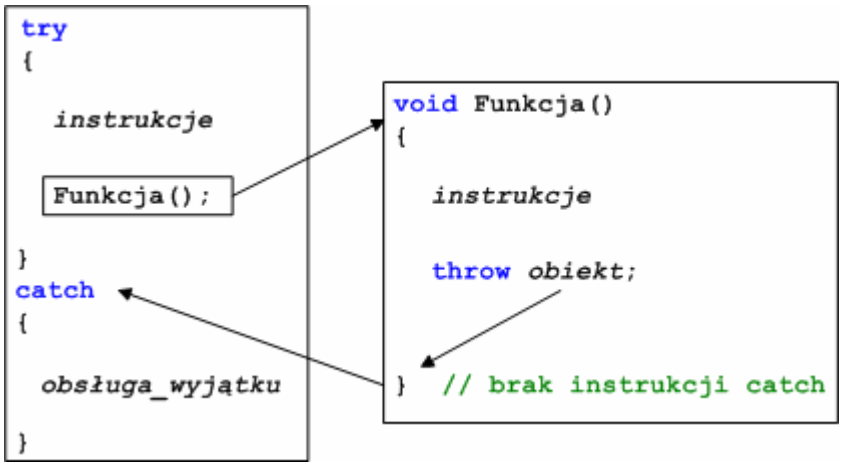
Kiedy wiadomo, że wystąpiła sytuacja wyjątkowa?… Otóż musi ona zostać zasygnalizowana przy pomocy instrukcji `throw`:

```
throw obiekt;
```

Wystąpienie tej instrukcji powoduje natychmiastowe przerwanie normalnego toku wykonywania programu. Sterowanie przenosi się wtedy do najbliższego pasującego bloku `catch`.
Rzucony *`obiekt`* pełni natomiast funkcję informującą. Może to być wartość d**owolnego typu** - również będąca obiektem zdefiniowanej przez nas klasy, co jest szczególnie przydatne. *`obiekt`* zostaje „wyrzucony” poza blok `try`; można to porównać do pilota katapultującego się z samolotu, który niechybnie ulegnie katastrofie. Wystąpienie `throw` jest bowiem sygnałem takiej katastrofy - sytuacji wyjątkowej.

Wędrówka wyjątku

Zaraz za blokiem `try` następuje najczęściej odpowiednia instrukcja `catch`, która złapie obiekt wyjątku. Wykona potem odpowiednie czynności, zawarte w swym bloku, a następnie program rozpocznie wykonywanie dalszych instrukcji, **zaraz za blokiem `catch`**.
Jeśli jednak wyjątek nie zostanie przechwycony, to może on opuścić swą macierzystą funkcję i dotrzeć do tej, którą ją wywołała. Jeśli i tam nie znajdzie odpowiadającego bloku `catch`, to wyjdzie jeszcze bardziej „na powierzchnię”. W przypadku gdy i tam nie będzie pasującej instrukcji `catch`, będzie wyskakiwał jeszcze wyżej, i tak dalej.
Proces ten nazywamy **odwijaniem stosu** (ang. *stack unwinding*) i trwa on dopóki jakaś instrukcja `catch` nie złapie lecącego wyjątku. W skrajnym (i nieprawidłowym) przypadku, odwijanie może zakończyć się przerwaniem działania programu - mówimy wtedy, że wystąpił **niezłapany wyjątek** (ang. *uncaught exception*).



Schemat 39. Wędrówka wyjątku rzuconego w funkcji

Zarówno o odwijaniu stosu, jak i o łapaniu i niezłapaniu wyjątków będziemy szerzej mówić w przyszłym podrozdziale.

`throw` a `return`

Instrukcja `throw` jest trochę podobna do instrukcji `return`, której używamy do zakończenia funkcji i zwrócenia jej rezultatu. Istnieją jednak ważne różnice:

- `return` powoduje zawsze przerwanie tylko jednej funkcji i powrót do miejsca, z którego ją wywołano. `throw` może natomiast wcale nie przerywać wykonywania

funkcji (jeżeli znajdzie w niej pasującą instrukcję `catch`), lecz równie dobrze może przerwać działanie wielu funkcji, a nawet całego programu
- w przypadku `return` możliwe jest „rzucenie" obiektu należącego tylko do jednego, ściśle określonego typu. Tym typem jest typ zwracany przez funkcję, określany w jej deklaracji. `throw` może natomiast wyrzucać obiekt **dowolnego typu**, zależnie od potrzeb
- `return` jest normalnym sposobem powrotu z funkcji, który stosujemy we wszystkich typowych sytuacjach. `throw` jest zaś używany w sytuacjach wyjątkowych; nie powinno się używać go jako zamiennika dla `return`, bo przeznaczenie obu tych instrukcji jest inne

Widać więc, że mimo pozornego podobieństwa instrukcje te są zupełnie różne. `return` jest typową instrukcją języka programowania, bez której tworzenie programów byłoby niemożliwe. `throw` jest z kolei częścią większej całości - mechanizmu obsługi wyjątków - będącym po prostu specjalnym mechanizmem radzenia sobie z sytuacjami kryzysowymi. Mimo jej przydatności, stosowanie tej techniki nie jest obowiązkowe.
Skoro jednak mamy wybierać między używaniem a nieużywaniem wyjątków (a takich wyborów będziesz dokonywał często), należy wiedzieć o wyjątkach coś więcej. Dlatego też kontynuujemy zajmowanie się tym tematem.

Właściwy chwyt

W poprzednich akapitach kilkakrotnie używałem sformułowania „pasujący blok `catch`" oraz „odpowiednia instrukcja `catch`". Cóż one znaczą?…

Jedną z zalet mechanizmu wyjątków jest to, że instrukcja `throw` może wyrzucać obiekty dowolnego typu. Poniższe wiersze są więc całkowicie poprawne:

```
throw 42u;
throw "Straszny blad!";
throw CException("Wystapil wyjatek", __FILE__, __LINE__);
throw 17.5;
```

Te cztery instrukcje `throw` rzucają (odpowiednio) obiekty typów `unsigned`, `const char[]`, zdefiniowanej przez użytkownika klasy `CException` oraz `double`. Wszystkie one są zapewne cennymi informacjami o błędach, które należałoby odczytać w bloku `catch`. Niewykluczone przecież, że nawet najmniejsza pomoc „z miejsca katastrofy" może być dla nas przydatna.

Dlatego też w mechanizmie wyjątków przewidziano sposób nie tylko na oddanie sterowania do bloku `catch`, ale też na przesłanie tam jednego obiektu. Jest to oczywiście ten obiekt, który podajemy instrukcji `throw`.
`catch` otrzymuje natomiast jego **lokalną kopię** - w podobny sposób, w jaki funkcje otrzymują kopie przekazanych im parametrów. Aby jednak tak się stało, blok `catch` musi **zadeklarować**, z **jakiego typu obiektami** chce pracować:

```
catch (typ obiekt)
{
   kod
}
```

W ten sposób bedzie miał dostęp do każdego złapanego *`obiektu`* wyjątku, który należy do podanego *`typu`*. Da mu to możliwość wykorzystania go - chociażby po to, aby wyświetlić użytkownikowi zawarte w nim informacje:

```
try
{
   srand (static_cast<unsigned>(time(NULL)))

   // losujemy rzucony wyjątek
   switch (rand() % 4)
   {
         case 0:     throw "Wyjatek tekstowy";
         case 1:     throw 1.5f;                    // wyjątek typu float
         case 2:     throw -12;                     // wyjątek typu int
         case 3:     throw (void*) NULL;            // pusty wskaźnik
   }
}
catch (int nZlapany)
{
   std::cout << "Zlapalem wyjatek liczbowy z wartoscia " << nZlapany;
}
```

Komunikaty o błędach powinny być w zasadzie kierowane do strumienia `cerr`, a nie `cout`. Tutaj jednak, dla zachowania prostoty, będę posługiwał się standardowym strumieniem wyjścia. O pozostałych dwóch rodzajach strumieni wyjściowych pomówimy w rozdziale o strumieniach STL.

W tym kawałku kodu blok `catch` złapie liczbę typu `int` - jeżeli takowa zostanie wyrzucona przez instrukcję `throw`. Przechwyci ją w postaci lokalnej zmiennej `nZlapany`, aby potem wyświetlić jej wartość w konsoli.

A co z pozostałymi wyjątkami? Nie mamy instrukcji `catch`, które by je łapały. Wobec tego zostaną one wyrzucone ze swej macierzystej funkcji i będą wędrowały tą ścieżką aż do natrafienia pasujących bloków `catch`. Jeżeli ich nie znajdą, spowodują zakończenie programu.
Powinniśmy zatem zapewnić obsługę także i tych wyjątków. Robimy w taki sposób, iż dopisujemy po prostu brakujące bloki `catch`:

```
catch (const char szNapis[])
{
   std::cout << szNapis;
}
catch (float fLiczba)
{
   std::cout << "Zlapano liczbe: " << fLiczba;
}
catch (void* pWskaznik)
{
   std::cout << "Wpadl wskaznik " << pWskaznik;
}
```

Bloków `catch`, nazywanych procedurami obsługi wyjątków (ang. *exception handlers*), może być dowolna ilość. Wszystko zależy od tego, **ile typów** wyjątków zamierzamy przechwytywać.

Kolejność bloków `catch`

Obecność kilku bloków `catch` po jednej instrukcji `try` to powszechna praktyka. Dzięki niej można bowiem zabezpieczyć się na okoliczność różnych rodzajów wyjątków. Warto więc o tym porozmawiać.

Dopasowywanie typu obiektu wyjątku

Załóżmy więc, że mamy taką oto sekwencję `try-catch`:

```
try
{
   // rzucamy wyjątek
   throw 90;
}
catch (float fLiczba)      { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (double fLiczba)     { /* ... */ }
```

W bloku `try` rzucamy jako wyjątek liczbę `90`. Ponieważ nie podajemy jej żadnych przyrostków, kompilator uznaje, iż jest to wartość typu `int`. Nasz obiekt wyjątku jest więc obiektem typu `int`, który leci na spotkanie swego losu.

Gdzie się zakończy jego droga?… Wszystko zależy od tego, który z trzech bloków catch przechwyci ten wyjątek. Wszystkie one są do tego zdolne: typ `int` pasuje bowiem zarówno do typu `float`, jak i `double` (no i oczywiście `int`).
Mówiąc „pasuje”, mam tu na myśli dokładnie taki sam mechanizm, jaki jest uruchamiany przy wywoływaniu funkcji z parametrami. Mając bowiem trzy funkcje:

```
void Funkcja1(float);
void Funkcja2(int);
void Funkcja3(double);
```

każdej z nich możemy przekazać wartość typu `int`. Naturalnie, jest on najbardziej zgodna z `Funkcja2()`, ale pozostałe też się do tego nadają. W ich przypadku zadziałają po prostu wbudowane, niejawne konwersje: kompilator zamieni liczbę na `int` na typ `float` lub `double`.

A jednak to tylko część prawdy. Zgodność typu wyjątku z typem zadeklarowanym w bloku `catch` to tylko jedno z kryterium wyboru - w dodatku wcale nie najważniejsze!
Otóż najpierw w grę wchodzi kolejność instrukcji `catch`. Kompilator przegląda je w takim samym porządku, w jakim występują w kodzie, i dla każdej z nich wykonuje test dopasowania argumentu. Jeśli stwierdzi **jakąkolwiek zgodność** (niekoniecznie najlepszą możliwą), **ignoruje wszystkie pozostałe** bloki `catch` i wybiera ten **pierwszy pasujący**.

Co to znaczy w praktyce? Spójrzmy na nasz przykład. Mamy obiekt typu `int`, który zostanie **kolejno** skonfrontowany z typami trzech bloków `catch`: `float`, `int` i `double`.
Wobec przedstawionych wyżej zasad, który z nich zostanie wybrany?…
Odpowiedź nie jest trudna. Już pierwsze dopasowanie `int` do `float` zakończy się sukcesem. Nie będzie ono wprawdzie najlepsze (wymagać będzie niejawnej konwersji), ale, jak podkresliłem, kompilator poprzestanie właśnie na nim. Porządek bloków `catch` weźmie po prostu górę nad ich zgodnością.
Pamiętaj więc zasadę dopasowywania typu obiektu rzuconego do wariantów `catch`:

Typy w blokach `catch` są sprawdzane wedle ich **kolejności w kodzie**, a wybierana jest **pierwsza pasująca** możliwość. Przy dopasowywania brane są pod uwagę **wszystkie niejawne konwersje**.

Szczególnie natomiast weź sobie do serca, iż:

Kolejność bloków `catch` często **ma znaczenie**.

Mimo że z pozoru przypominają one funkcje, funkcjami nie są. Obowiązują w nich więc inne zasady wyboru właściwego wariantu.

Szczegóły przodem

Jak w takim razie należy ustawiać procedury obsługi wyjątków, aby działały one zgodnie z naszymi życzeniami?… Popatrzmy wpierw na taki przykład:

```
try
{
   // ...
   throw 16u;         // unsigned
   // ...
   throw -87;         // int
   // ...
   throw 9.242f;      // float
   // ...
   throw 3.14157;     // double
}
catch (double fLiczba)     { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (float fLiczba)      { /* ... */ }
catch (unsigned uLiczba)   { /* ... */ }
```

Pytanie powinno tutaj brzmieć: co jest źle na tym obrazku? Domyślasz się, że chodzi o kolejność bloków `catch`. Sprawdźmy.
W bloku `try` rzucamy jeden z czterech wyjątków - typu `unsigned`, `int`, `float` oraz `double`. Co się z nimi dzieje? Oczywiście trafiają do odpowiednich blobków `catch`… czy aby na pewno?

Niezupełnie. Wszystkie te liczby zostaną bowiem od razu dopasowane do pierwszego wariantu z parametrem `double`. Typ `double` swobodnie potrafi pomieścić wszystkie cztery typy liczbowe, zatem wszystkie cztery wyjątkie trafią wyłącznie do pierwszego bloku `catch`! Pozostałe trzy są w zasadzie zbędne!
Kolejność procedur obsługi jest zatem nieprawidłowa. Poprawnie powinny być one ułożone w ten sposób:

```
catch (unsigned uLiczba)   { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (float fLiczba)      { /* ... */ }
catch (double fLiczba)     { /* ... */ }
```

To gwarantuje, że wszystkie wyjątki trafią do tych bloków `catch`, które im dokładnie odpowiadają. Korzystamy tu z faktu, że:

- typ `unsigned` w pierwszym bloku przyjmie tylko wyjątki typu `unsigned`
- typ `int` w drugim bloku mógłby przejąć zarówno liczby typu `unsigned`, jak i `int`. Te pierwszą są jednak przechwycane przez poprzedni blok, zatem tutaj trafiają wyłącznie wyjątki faktycznego typu `int`
- typ `float` może przyjąć typy `unsigned`, `int` i `float`. Pierwsze dwa są już jednak obsłużone, więc ten blok `catch` dostaje tylko „prawdziwe” liczby zmiennoprzecinkowe pojedynczej precyzji
- typ `double` pasuje do każdej liczby, ale tutaj blok `catch` z tym typem dostanie jedynie te wyjątki, które są faktycznie typu `double`. Pozostałe liczby zostaną przechwycone przez poprzednie warianty

Między typami `unsigned`, `int`, `float` i `double` zachodzi tu po prosta relacja polegająca na tym, że każdy z nich jest szczególnym przypadkiem następnego:

unsigned ⊂ int ⊂ float ⊂ double

„Najbardziej szczególny" jest typ `unsigned` i dlatego on występuje na początku. Dalej mamy już coraz bardziej ogólne typy liczbowe.
Taka zasada konstrurowania sekwencji bloków `catch` jest poprawna w każdym przypadku, nie tylko dla typów liczbowych,

Umieszczając kilka bloków `catch` jeden po drugim, zadbaj o to, aby występowały one w porządku **rosnącej ogólności**. Niech **najpierw** pojawią się bloki o **najbardziej wyspecjalizowanych typach**, a dopiero potem typy **coraz bardziej ogólne**.

Możesz kręcić nosem na takie nieścisłe sformulowania. Bo i co to znaczy, że dany typ jest ogólniejszy niż inny?… W grę wchodzą tu niejawne konwersje - jak wiemy, kompilator stosuje je przy dopasowywaniu w blokach `catch`. Można zatem powiedzieć, że:

Typ *A* jest **ogólniejszy** od typu *B*, jeżeli istnieje **niejawna konwersja z *B* do *A***, niepowodująca utraty danych.

W tym sensie double jest ogólniejszy od każdego z typów: `unsigned`, `int` i `float`, ponieważ w każdym przypadku istnieją niejawne konwersje standardowe, zamieniające te typy na `double`. To zresztą zgodne ze zdrowym rozsądkiem i wiedzą matematyczną, która mówi, nam że liczby naturalne i całkowite są także liczbami rzeczywistymi.
Innym rodzajem konwersji, który będzie nas interesował w tym rozdziale, jest zamiana odwołania do obiektu klasy pochodnej na odwołanie do obiektu klasy bazowej. Użyjemy jej do budowy hierarchii klas dla wyjątków.

Zagnieżdżone bloki `try`-`catch`

Wewnątrz bloku `try` może znaleźć się dowolny kod, jaki może być umieszczany we wszystkich blokach instrukcji C++. Przypisania, instrukcje warunkowe, pętle, wywołania funkcji - wszystko to jest dopuszczalne. Co więcej, w bloku `try` mogę się znaleźć… inne bloki `try`-`catch`. Nazywami je wtedy **zagnieżdżonymi**, zupełnie tak samo jak zagnieżdżone instrukcje `if` czy pętle.

Formalnie składnia takiego zagnieżdżenia może wyglądać tak:

```
try
{
   try
   {
         ryzykowne_instrukcje_wewnętrzne
   }
   catch (typ_wewnętrzny_1 obiekt_wewnętrzny_1)
   {
         wewnętrzne_instrukcje_obsługi_1
   }
   catch (typ_wewnętrzny_2 obiekt_wewnętrzny_2)
   {
         wewnętrzne_instrukcje_obsługi_2
   }
   // ...

   ryzykowne_instrukcje_zewnętrzne
}
catch (typ_zewnętrzny_1 obiekt_zewnętrzny_1)
{
   zewnętrzne_instrukcje_obsługi_1
```

```
}
catch (typ_zewnętrzny_1 obiekt_zewnętrzny_2)
{
   zewnętrzne_instrukcje_obsługi_2
}
// ...

dalsze_instrukcje
```

Mimo pozornego skomplikowania jej funkcjonowanie jest intuicyjne. Jeżeli podczas wykonywania *ryzykownych_instrukcji_wewnętrznych* rzucony zostanie wyjątek, to wpierw będzie on łapany przez wewnętrzne bloki `catch`. Dopiero gdy one przepuszczą wyjątek, do pracy wezmą się bloki zewnętrzne.
Jeżeli natomiast któryś z zestawów `catch` (wewnętrzny lub zewnętrzny) wykona swoje zadanie, to program będzie kontynuował od następnych linijek po tym zestawie. Tak więc w przypadku, gdy wyjątek złapie wewnętrzny zestaw, wykonywane będą *ryzykowne_instrukcje_zewnętrzne*; jeśli zewnętrzny - *dalsze_instrukcje*.
No a jeśli żaden wyjątek nie wystąpi? Wtedy wykonają się wszystkie instrukcje poza blokami `catch`, czyli: *ryzykowne_instrukcje_wewnętrzne*, *ryzykowne_instrukcje_zewnętrzne* i wreszcie *dalsze_instrukcje*.

Takie dosłowne zagnieżdżanie bloków `try-catch` jest w zasadzie rzadkie. Częściej wewnętrzny blok występuje w funkcji, której wywołanie mamy w zewnętrznym bloku. Oto przykład:

```
void FunkcjaBiblioteczna()
{
   try
   {
         // ...
   }
   catch (typ obiekt)
   {
         // ...
   }
   // ...
}

void ZwyklaFunkcja()
{
   try
   {
         FunkcjaBiblioteczna();
         // ...
   }
   catch (typ obiekt)
   {
         // ...
   }
}
```

Takie rozwiązanie ma prostą zaletę: `FunkcjaBiblioteczna()` może złapać i obsłużyć te wyjątki, z którymi sama sobie poradzi. Jeżeli nie potrzeba angażować w to wywołującego, jest to duża zaleta. Część wyjątków najprawdopodobniej jednak opuści funkcję - tylko tymi będzie musiał zająć się wywołujący. Wewnętrzne sprawy wywoływanej funkcji (także wyjątki) pozostaną jej wewnętrznymi sprawami.
Ogólnie można powiedzieć, że:

Wyjątki powinny być łapane w jak **najbliższym od ich rzucenia** miejscu, w którym **możliwe jest ich obsłużenie**.

O tej ważnej zasadzie powiemy sobie jeszcze przy okazji uwag o wykorzystaniu wyjątków.

Złapanie i odrzucenie

Przy zagnieżdżaniu bloków `try` (nieważne, czy z pośrednictwem funkcji, czy nie) może wystąpić częsta w praktyce sytuacja. Możliwe jest mianowicie, że po złapaniu wyjątku przez bardziej wewnętrzny `catch` **nie potrafimy podjąć wszystkich akcji**, jakie byłyby dla niego konieczne. Przykładowo, możemy tutaj jedynie zarejestrować go w dzienniku błędów; bardziej użyteczną reakcję powinien zająć się „ktoś wyżej”.
Moglibyśmy pominąć wtedy ten wewnętrzny `catch`, ale jednocześnie pozbawilibyśmy się możliwości wczesnego zarejestrowania błędu. Lepiej więc pozostawić go na miejscu, a po zakończeniu zapisywania informacji o wyjątku **wyrzucić go ponownie**. Robimy to instrukcją `throw` bez żadnych parametrów:

```
throw;
```

Ta instrukcja powoduje ponowne rzucenie tego samego obiektu wyjątku. Teraz jednak będą mogły zająć się nim bardziej zewnętrzne bloki `catch`. Będą one pewnie bardziej kompetentne niż nasze siły szybkiego reagowania.

Blok `catch(...)`, czyli chwytanie wszystkiego

W połączeniu z zagnieżdżonymi blokami `try` i instrukcją `throw;` czesto występuje specjalny rodzaj bloku `catch`. Nazywany jest on **uniwersalnym**, a powstaje poprzez wpisanie po `catch` wielokropka (trzech kropek) w nawiasie:

```
try
{
   // instrukcje
}
catch (...)
{
   // obsługa wyjątków
}
```

Uniwersalność tego specjalnego rodzaju catch polega na tym, iż pasują do niego **wszystkie obiekty wyjątków**. Jeżeli kompilator, transportując wyjątek, natrafi na `catch(...)`, to **bezwarunkowo wybierze** właśnie ten wariant, nie oglądając się na żadne inne. `catch(...)` jest więc „wszystkożerny”: pochłania dowolne typy wyjątków.

‘Pochłania’ to zresztą dobre słowo. Wewnątrz bloku `catch(...)` nie mamy mianowicie żadnych informacji o obiekcie wyjątku. Nie tylko o jego wartości, ani nawet o jego typie. Wiemy jedynie, że **jakiś wyjątek wystąpił** - i skromną tą wiedzą musimy się zadowolić. Po co nam wobec tego taki dziwny blokcatch?… Jest on przydatny tam, gdzie możemy jakoś wykorzystać samo powiadomienie o wyjątku, nie znając jednak jego typu ani wartości. Wewnątrz `catch(...)` możemy jedynie podjać pewne domyślne działania. Możemy na przykład dokonać małego zrzutu pamięci (ang. *memory dump*), zapisując w bezpiecznym miejscu wartości zmiennych na wypadek zakończenia programu. Możemy też w jakiś sposób przygotować się do właściwej obsługi błędów.
Cokolwiek zrobimy, na koniec powinniśmy przekazać wyjątek dalej, czyli użyć konstrukcji:

```
throw;
```

Jeżeli tego nie zrobimy, to `catch(...)` zdusi w zarodku wszelkie wyjątki, nie pozwalając na to, by dotarły one dalej.

***

Na tym kończą się podstawowe informacje o mechanizmie wyjątków. To jednak nie wszystkie aspekty tej techniki. Musimy sobie jeszcze porozmawiać o tym, co dzieje się między rzuceniem wyjątku poprzez `throw` i jego złapaniem przy pomocy `catch`. Porozmawiamy zatem o odwijaniu stosu.

Odwijanie stosu

Odwijanie stosu (ang. *stack unwinding*) jest procesem ściśle związanym z wyjątkami. Jakkolwiek sama jego istota jest raczej prosta, musimy wiedzeć, jakie ma on konsekwencje w pisanym przez nas kodzie.

Między rzuceniem a złapaniem

Odwijanie stosu rozpoczyna się wraz z rzuceniem jakiegokolwiek wyjątku przy pomocy instrukcji `throw` i postępuje aż do momentu natrafienia na pasujący do niego blok `catch`. W skrajnym przypadku odwijanie może doprowadzić do zakończenia działania programu - jest tak jeśli odpowiednia procedura obsługi wyjątku nie zostanie znaleziona.

Wychodzenie na wierzch

Na czym jednak polega samo odwijanie?… Otóż można opisać je w skrócie jako **wychodzenie punktu wykonania ze wszystkich bloków kodu**. Co to znaczy, najlepiej wyjaśnić na przykładzie.

Załóżmy, że mamy taką oto sytuację:

```
try
{
    for (/* ... */)
    {
          switch (/* ... */)
          {
                case 1:
                      if (/* ... */)
                      {
                            // ...
                            throw obiekt;
                      }
          }
    }
}
catch
{
    // ...
}
```

Instrukcja throw występuje to wewnątrz 4 zagnieżdżonych w sobie bloków: `try`, `for`, `switch` i `if`. My oczywiście wiemy, że najważniejszy jest ten pierwszy, bo zaraz za nim występuje procedura obsługi wyjątku - `catch`.
Co się dzieje z wykonywaniem programu, gdy następuje sytuacja wyjątkowa? Otóż **nie skacze on od razu** do odpowiedniej instrukcji `catch`. Byłoby to może najszybsze z

punktu widzenia wydajności, ale jednocześnie całkowicie niedopuszczalne. Dlaczego tak jest - o tym powiemy sobie w następnym paragrafie.

Jak więc postępuje kompilator? Rozpoczyna to sławetne odwijanie stosu, któremu poświęcony jest cały ten podrozdział. Działa to mniej więcej tak, jakby dla każdego bloku, w którym się aktualnie znajdujemy, zadziałała instrukcja `break`. Powoduje to wyjście z danego bloku.
Po każdej takiej operacji jest poza tym sprawdzana obecność następującego dalej bloku `catch`. Jeżeli takowy jest obecny, i pasuje on do typu obiektu wyjątku, to wykonywana jest procedura obsługi wyjątku w nim zawarta. Proste i skuteczne :)

Zobaczmy to na naszym przykładzie. Instrukcja `throw` znajduje się tu przede wszystkim wewnątrz bloku `if` - i to on będzie w pierwszej kolejności odwinięty. Potem nie zostanie znaleziony blok `catch`, zatem opuszczone zostaną także bloki `switch`, `for` i wreszcie `try`. Dopiero w tym ostatnim przypadku natrafimy na szukaną procedurę obsługi, która zostanie wykonana.
Warto pamiętać, że - choć nie widać tego na przykładzie - odwijanie może też dotyczyć funkcji. Jeżeli zajdzie konieczność odwinięcia jej bloku, to sterowanie wraca do wywołującego funkcję.

Porównanie `throw` z `break` i `return`

Nieprzypadkowo porównałem instrukcję `throw` do `break`, a wcześniej do `return`. Czas jednak zebrać sobie cechy wyróżniające i odróżniające te trzy instrukcje. Oto stosowna tabela:

instrukcja → cecha ↓	***`throw`***	***`break`***	***`return`***
przekazywanie sterowania	do najbliższego pasującego bloku `catch`	jeden blok wyżej (wyjście z pętli lub bloku `switch`)	zakończenie działania funkcji i powrót do kodu, który ją wywołał
wartość	obiekt wyjątku dowolnego typu	nie jest związany z żadną wartością	wartość tego samego typu, jaki został określony w deklaracji funkcji
zastosowanie	obsługa sytuacji wyjątkowych	ogólne programowanie	

Tabela 20. Porównanie `throw` z `break` i `return`

Wszystkie te trzy własności trzech instrukcji są bardzo ważne i koniecznie musisz o nich pamiętać. Nie będzie to chyba dla ciebie problemem, skoro dwie z omawianych instrukcji znasz doskonale, a o wszystkich aspektach trzeciej porozmawiamy sobie jeszcze całkiem obszernie.

Wyjątek opuszcza funkcję

Rzucenie oraz złapanie i obsługa wyjątku może odbywać się w ramach tej samej funkcji. Często jednak mamy sytuację, w której to jedna funkcja sygnalizuje sytuację wyjątkową, a dopiero inna (wywołująca ją) zajmuje się reakcją na zainstniały problem. Jest to zupełnie dopuszczalne, co zresztą parokrotnie podkreślałem.

W procesie odwijania stosu obiekt wyjątku może więc opuścić swoją macierzystą funkcję. Nie jest to żaden błąd, lecz normalna praktyka. Nie zwalnia ona jednak z obowiązku złapania wyjątku: nadal ktoś musi to zrobić. Ktoś - czyli wywołujący funkcję.

Specyfikacja wyjątków

Aby jednak można było to uczynić, należy wiedzieć, **jakiego typu** wyjątki funkcja może wyrzucać na zewnątrz. Dzięki temu możemy opakować jej przywołanie w blok `try` i dodać za nim odpowiednie instrukcje `catch`, chwytające właściwe obiekty.

Skąd mamy uzyskać tę tak potrzebną wiedzę? Wydawałoby się, że nic prostszego. Wystarczy przejrzeć kod funkcji, znaleźć wszystkie instrukcje `throw` i określić typ obiektów, jakie one rzucają. Następnie należy odrzucić te, które są obsługiwane w samej funkcji i zająć się tylko wyjątkami, które z niej „uciekają".
Ale to tylko teoria i ma ona jedną poważną słabostkę. Wymaga przecież dostępu do kodu źródłowego funkcji, a ten nie musi być wcale osiągalny. Wiele bibliotek jest dostarczanych w formie skompilowanej, zatem nie ma szans na ujrzenie ich wnętrza. Mimo to ich funkcjom nikt całkowicie nie zabroni rzucania wyjątków.

Dlatego należało jakoś rozwiązać ten problem. Uzupełniono więc deklaracje funkcji o dodatkową informację - **specyfikację wyjątków**.

Specyfikacja albo **wyszczególnienie wyjątków** (ang. *exceptions' specification*) mówi nam, czy dana funkcja wyrzuca z siebie jakieś **nieobsłużone obiekty wyjątków**, a jeśli tak, to informuje także o ich **typach**.

Takie wyszczególnienie jest częścią deklaracji funkcji - umieszczamy je na jej końcu, np.:

```
void Znajdz(int* aTablica, int nLiczba) throw(void*);
```

Po liście parametrów (oraz ewentualnych dopiskach typu `const` w przypadku metod klasy) piszemy po prostu słowo `throw`. Dalej umieszczamy w nawiasie listę typów wyjątków, które będą opuszczały funkcję i których złapanie będzie należało do obowiązków wywołującego. Oddzielamy je przecinkami.
Ta lista typów jest nieobowiązkowa, podobnie zresztą jak cała fraza `throw()`. Są to jednak dwa szczególne przypadki - wyglądają one tak:

```
void Stepuj();
void Spiewaj() throw();
```

Brak specyfikacji oznacza tyle, iż dana funkcja **może rzucać na zewnątrz wyjątki dowolnego typu**. Natomiast podanie `throw` bez określenia typów wyjątków informuje, że funkcja **w ogóle nie wyrzuca wyjątków na zewnątrz**. Widząc tak zadeklarowaną funkcję możemy więc mieć pewność, że jej wywołania nie trzeba umieszczać w bloku `try` i martwić się o obsługę wyjątków przez `catch`.

Specyfikacja wyjątków jest **częścią deklaracji funkcji**, zatem będzie ona występować np. w pliku nagłówkowym zewnętrznej biblioteki. Jest to bowiem niezbędna informacja, potrzebna do korzystania z funkcji - podobnie jak jej nazwa czy parametry. Kiedy jednak tamte wiadomości podpowiadają, w jaki sposób wywoływać funkcję, wyszczególnienie `throw()` mówi nam, jakie wyjątki musimy przy okazji tego wywołania obsługiwać.
Warto też podkreślić, że mimo swej obecności w deklaracji funkcji, specyfikacja wyjątków **nie należy do typu funkcji**. Do niego nadal zaliczamy wyłącznie listę parametrów oraz typ wartości zwracanej. Na pokazane wyżej funkcje `Stepuj()` i `Spiewaj()` można więc pokazywać tym samym wskaźnikiem.

Kłamstwo nie popłaca

Specyfikacja wyjątków jest przyczeniem złożonym przez twórcę funkcji jej użytkownikowi. W ten sposób autor procedury zaświadcza, że jego dzieło będzie wyrzucało do wywołującego **wyjątki wyłącznie podanych typów**.

Niestety, życie i programowanie uczy nas, że niektóre obietnice mogą być tylko obiecankami. Załóżmy na przykład, że w nowej wersji biblioteki, z której pochodzi funkcja, dokonano pewnych zmian. Teraz rzucany jest jeszcze jeden, nowy typ wyjątków, którego obsługa spada na wywołującego.
Zapomniano jednak zmienić deklarację funkcji - wygląda ona nadal np. tak:

```
bool RobCos() throw(std::string);
```

Obiecywanym typem wyjątków jest tu **tylko i wyłącznie** `std::string`. Przypuśćmy jednak, że w wyniku poczynionych zmian funkcja może teraz rzucać także liczby typu `int` - typu, którego nazwa **nie występuje** w specyfikacji wyjątków.

Co się wtedy stanie? Czy wystąpi błąd?… Powiedzmy. Jednak to nie kompilator nam o nim powie. Nie zrobi tego nawet linker. Otóż:

O **rzuceniu przez funkcję niezadeklarowanego wyjątku** dowiemy się dopiero **w czasie działania programu**.

Wygląda to tak, iż program wywoła wtedy specjalną funkcję `unexpected()` (‘niespodziewany’). Jest to funkcja biblioteczna, uruchamiana w reakcji na niedozwolony wyjątek.
Co robi ta funkcja? Otóż… wywołuje ona drugą funkcję, `terminate()` (‘przerwij’). O niej będziemy jeszcze rozmawiać przy okazji niezłapanych wyjątków. Na razie zapamiętaj, że funkcja ta po prostu kończy działanie programu w mało porządny sposób.

Wyrzucenie przez funkcję **niezadeklarowanego wyjątku** kończy się **awaryjnym przerwaniem działania programu**.

Spytasz pewnie: „Dlaczego tak drastycznie?” Taka reakcja jest jednak uzasadniona, gdyż do czynienia ze zwyczajnym **oszustwem**.
Oto ktoś (twórca funkcji) deklaruje, że będzie ona wystrzeliwać z siebie wyłącznie określone typy wyjątków. My posłusznie podporządkowujemy się tej **obietnicy**: ujmujemy wywołanie funkcji w blok `try` i piszemy odpowiednie bloki `catch`. Wszystko robimy zgodnie ze specyfikacją `throw()`.
Tymczasem **zostajemy oszukani**. Obietnica została złamana: funkcja rzuca nam wyjątek, którego się zupełnie nie spodziewaliśmy. Nie mamy więc kodu jego obsługi - albo nawet gorzej: mamy go, ale nie tam gdzie trzeba. W każdym przypadku jest to sytuacja nie do przyjęcia i stanowi wystarczającą podstawę do zakończenia działania programu.

To domyślne możemy aczkolwiek zmienić. Nie zaleca się wprawdzie, aby mimo niespodziewanego wyjątku praca programu była kontynuowana. Jeżeli jednak napiszemy własną wersję funkcji `unexpected()`, będziemy mogli **odróżnić** dwie sytuacje:

- niezłapany wyjątek - czyli taki wyjątek, którego nie schwycił żaden blok `catch`
- nieprawidłowy wyjątek - taki, który nie powinien się wydostać z funkcji

Różnica jest bardzo ważna, bowiem w tym drugim przypadku nie jesteśmy winni zaistniałemu problemu. Dokładniej mówiąc, nie jest winny kod wywołujący funkcję - przyczyna tkwi w samej funkcji, a zawinił jej twórca. Jego obietnice dotyczące wyjątków okazały się obietnicami bez pokrycia.
Rozdzielenie tych dwóch sytuacji pozwoli nam uchronić się przed poprawianiem kodu, który być może wcale tego nie wymaga. Z powodu niezadeklarowanego wyjątku nie ma bowiem potrzeby dokonywania zmian w kodzie wywołującym funkcję. Później będą one oczywiście konieczne; później - to znaczy wtedy, gdy powiadomimy twórcę funkcję o jego niekompetencji, a ten z pokorą naprawi swój błąd.

Jak zatem możemy zmienić domyślną funkcję `unexpected()`? Czynimy to… wywołując inną funkcję - `set_unexpected()`:

```
unexpected_handler set_unexpected(unexpected_handler pfnFunction);
```

Tym, który ta funkcja przyjmuje i zwraca, to `unexpected_handler`. Jest to alias ta **wskaźnik do funkcji**: takiej, która nie bierze żadnych parametrów i nie zwraca żadnej wartości.
Poprawną wersją funkcji `unexpected()` może więc być np. taka funkcja:

```
void MyUnexpected()
{
   std::cout << "--- UWAGA: niespodziewany wyjątek ---" << std::endl;
   exit (1);
}
```

Po przekazaniu jej do `set_unexpected()`:

```
set_unexpected (MyUnexpected);
```

będziemy otrzymywali stosowną informację w przypadku wyrzucenia niedozwolonego wyjątku przez jakąkolwiek funkcję programu.

Niezłapany wyjątek

Przekonaliśmy się, że proces odwijania stosu może doprowadzić do przerwania działania funkcji i poznaliśmy tego konsekwencje. Nieprawidłowe sygnalizowanie lub obsługa wyjątków mogą nam jednak sprawić jeszcze jedną niespodziankę.
Odwijanie może się mianowicie zakończyć niepowodzeniem, jeśli żaden pasujący blok catch nie zostanie znaleziony. Mówimy wtedy, że wystąpił **nieobsłużony wyjątek**.

Co następuje w takim wypadku? Otóż program wywołuje wtedy funkcję `terminate()`. Jej nazwa wskazuje, że powoduje ona przerwanie programu. Faktycznie funkcja ta wywołuje inną funkcję - `abort()` ('przestań'). Ona zaś powoduje brutalne i nieznoszące żadnych kompromisów przerwanie działania programu. Po jej wywołaniu możemy w oknie konsoli ujrzeć komunikat:

```
Abnormal program termination
```

Taki też napis będzie pożegnaniem z programem, w którym wystąpi niezłapany wyjątek. Możemy to jednak zmienić, pisząc własną wersję funkcji `terminate()`.

Do ustawienia nowej wersji służy funkcja `set_terminate()`. Jest ona bardzo podobna do analogicznej funkcji `set_unexpected()`:

```
terminate_handler set_terminate(terminate_handler pfnFunction);
```

Występujący tu alias `terminate_handler` jest także wskaźnikiem na funkcję, która nic nie bierze i nie zwraca. W parametrze `set_terminate()` podajemy więc wskaźnik do nowej funkcji `terminate()`, a w zamian otrzymujemy wskaźnik do starej - zupełnie jak w `set_unexpected()`.
Oto przykładowa funkcja zastępcza:

```
void MyTerminate()
{
   std::cout << "--- UWAGA: blad mechanizmu wyjatkow ---" << std::endl;
```

```
    exit (1);
}
```

Wypisywany przez nas komunikat jest tak ogólny (nie brzmi np. `"niezlapany wyjatek"`), ponieważ `terminate()` jest wywoływana także w nieco innych sytuacjach, niż niezłapany wyjątek. Powiemy sobie o nich we właściwym czasie.

Zastosowana tutaj, jak w i `MyUnexpected()` funkcja `exit()` służy do normalnego (a nie awaryjnego) zamknięcie programu. Podajemy jej tzw. **kod wyjścia** (ang. *exit code*) - zwyczajowo zero oznacza wykonanie bez błędów, inna wartość to nieprawidłowe działanie aplikacji (tak jak u nas).

Porządki

Odwijanie stosu jest w praktyce bardziej złożonym procesem niż to się wydaje. Oprócz przetransportowania obiektu wyjątku do stosownego bloku catch kompilator musi bowiem zadbać o to, aby reszta programu nie doznała przy okazji jakichś obrażeń.
O co chodzi? O tym porozmawiamy sobie w tym paragrafie.

Niszczenie obiektów lokalnych

Wspominając o opuszczaniu kolejno zagnieżdżonych bloków czy nawet funkcji, posłużyłem się porównaniu z `break` i `return`. `throw` ma z nimi jeszcze jedną cechę wspólną - nie licząc tych odróżniających.

Wychodzenie z bloków przebiega mianowicie w sposób całkiem „czysty” - tak jak w normalnym kodzie. Oznacza, to że wszystkie stworzone **obiekty lokalne są niszczone**, a ich pamięć zwalniania.
W przypadku typów podstawowych oznacza to po prostu usunięcie zmiennych z pamięci. Dla klas mamy jeszcze wywoływanie destruktorów i wszystkie tego konsekwencje.

Można zatem powiedzieć, że:

Opuszczanie bloków kodu dokonywane podczas odwijania stosu przebiega tak samo, jak to się dzieje podczas normalnego wykonywania programu. Obiekty lokalne są więc **niszczone poprawnie**.

Sama nazwa ‘odwijanie stosu’ pochodzi zresztą od tego sprzątania, dokonywanego przy okazji „wychodzenia na wierzch” programu. Obiekty lokalne (zwane też automatycznymi) są bowiem tworzone na stosie, a jego odwinięcie to właśnie usunięcie tych obiektów oraz powrót z wywoływanych funkcji.

Wypadki przy transporcie

To niszczenie obiektów lokalnych może się wydawać tak oczywiste, że nie warto poświęcać temu aż osobnego paragrafu. Jest jednak coś na rzeczy: czynność ta może być bowiem powodem pewnych problemów, jeżeli nie będziemy jej świadomi. Jakich problemów?…

Niedozwolone rzucenie wyjątku

Musimy powiedzieć sobie o jednej bardzo ważnej zasadzie związanej z mechanizmem wyjątków w C++. Brzmi ona:

Nie należy rzucać następnego wyjątku w czasie, gdy **kompilator zajmuje się obsługą** poprzedniego.

Co to znaczy? Czy nie możemy używać instrukcji `throw` w blokach `catch`?…

Otóż nie - jest to dozwolone, ale w sumie nie o tym chcemy mówić :) Musimy sobie powiedzieć, co rozumiemy poprzez „obsługę wyjątku dokonywaną przez kompilator”.

Dla nas obsługą wyjątku jest kod w bloku `catch`. Aby jednak mógł on być wykonany, obiekt wyjątku oraz punkt sterowania programu muszą tam trafić. Tym zajmuje się kompilator - **to jest właśnie jego obsługa wyjątku**: dostarczenie go do bloku `catch`. Dalej nic go już nie obchodzi: kod z bloku `catch` jest traktowany jako normalne instrukcje, bowiem sam kompilator uznaje już, że z chwilą rozpoczęcia ich wykonywania jego praca została wykonana. Wyjątek został przyniesiony i to się liczy.
Tak więc:

Obsługa wyjątku dokonywana przez kompilator polega na jego **dostarczeniu go do odpowiedniego bloku `catch`** przy jednoczesnym **odwinięciu stosu**.

Teraz już wiemy, na czym polega zastrzeżenie podane na początku. Nie możemy rzucić następnego wyjątku w chwili, gdy kompilator zajmuje się jeszcze transportem poprzedniego. Inaczej mówiąc, między wykonaniem instrukcji `throw` a obsługą wyjątku w bloku `catch` **nie może wystapić następna instrukcja `throw`**.

Strefy bezwyjątkowe

„No dobrze, ale właściwie co z tego? Przecież po rzuceniu jednego wyjątku wszystkim zajmuje się już kompilator. Jak więc moglibyśmy rzucić kolejny wyjątek, zanim ten pierwszy dotrze do bloku `catch`?…”
Faktycznie, tak mogłoby się wydawać. W rzeczywistości istnieją aż dwa miejsca, z których można rzucić drugi wyjątek.

Jeśli chodzi o pierwsze, to pewnie się go domyślasz, jeżeli uważnie czytałeś opis procesu odwijania stosu i związanego z nim niszczenia obiektów lokalnych. Powiedziałem tam, że przebiega ono w identyczny sposób, jak normalnie. Pamięć jest zawsze zwalniania, a w przypadku obiektów klas **wywoływane są destruktory**.
Bingo! Destruktory są właśnie tymi procedurami, które są wywoływane podczas obsługi wyjątku dokonywanej przez kompilator. A zatem nie możemy wyrzucać z nich żadnych wyjątków, ponieważ może zdarzyć, że dany destruktor jest wywoływany podczas odwijania stosu.

Nie rzucaj wyjątków z **destruktorów**.

Druga sytuacja jest bardziej specyficzna. Wiemy, że mechanizm wyjątków pozwala na rzucanie obiektów dowolnego typu. Należą do nich także obiekty klas, które sami sobie zdefiniujemy. Definiowanie takich specjalnych klas wyjątków to zresztą bardzo pożądana i rozsądna praktyka. Pomówimy sobie jeszcze o niej.
Jednak niezależnie od tego, jakiego rodzaju obiekty rzucamy, kompilator z każdym postępuje tak samo. Podczas transportu wyjątku do `catch` czyni on przynajmniej jedną kopię obiektu rzucanego. W przypadku typów podstawowych nie jest to żaden problem, ale dla klas wykorzystywane są normalne sposoby ich kopiowania. Znaczy to, że **może zostać użyty konstruktor kopiujący** - nasz własny.
Mamy więc drugie (i na szczęście ostatnie) potencjalne miejsce, skąd można rzucić nowy wyjątek w trakcie obsługi starego. Pamiętajmy więc o ostrzeżeniu:

Nie rzucajmy nowych wyjątków z **konstruktorów kopiujących klas, których obiekty rzucamy jako wyjątki**.

Z tych dwóch miejsc (wszystkie destruktory i konstruktory kopiujące obiektów rzucanych) nie powinniśmy rzucać żadnych wyjątków. W przeciwnym wypadku kompilator uzna to za bardzo poważny błąd. Zaraz się przekonamy, jak poważny…

Biblioteka Standardowa udostępnia prostą funkcję `uncaught_exception()`. Zwraca ona `true`, jeżeli kompilator jest w trakcie obsługi wyjątku. Można jej użyć, jeśli koniecznie musimy rzucić wyjątek w destruktorze; oczywiście powinniśmy to zrobić tylko wtedy, gdy funkcja zwróci `false`.
Prototyp tej funkcji znajduje się w pliku nagłówkowym *exception* w przestrzeni nazw `std`.

Skutki wypadku

Co się stanie, jeżeli zignorujemy któryś z zakazów podanych wyżej i rzucimy nowy wyjątek w trakcie obsługi innego?…

Będzie to wtedy bardzo poważna sytuacja. Oznaczać ona będzie, że kompilator nie jest w stanie poprawnie przeprowadzić obsługi wyjątku. Inaczej mówiąc, **mechanizm wyjątków zawiedzie** - tyle że będzie to rzecz jasna nasza wina.
Co może wówczas zrobić kompilator? Niewiele. Jedyne, co wtedy czyni, to wywołanie funkcji `terminate()`. Skutkiem jest więc nieprzewidziane zakończenie programu.

Naturalnie, zmiana funkcji `terminate()` (poprzez `set_terminate()`) sprawi, że zamiast domyślnej będzie wywoływana nasza procedura. Pisząc ją powinniśmy pamiętać, że funkcja `terminate()` jest wywoływana w dwóch sytuacjach:

- gdy wyjątek nie został złapany przez żaden blok `catch`
- gdy został rzucony nowy wyjątek w trakcie obsługi poprzedniego

Obie są sytuacjami krytycznymi. Zatem niezależnie od tego, jakie dodatkowe akcje będziemy podejmować w naszej funkcji, zawsze musimy na koniec zamknąć nasz program. W aplikacjach konsolowych można uczynić to poprzez `exit()`.

Zarządzanie zasobami w obliczu wyjątków

Napisałem wcześniej, że transport rzuconego wyjątku do bloku `catch` powoduje zniszczenie wszystkich obiektów lokalnych znajdujących się „po drodze”. Nie musimy się o to martwić; zresztą, nie troszczyliśmy się o nie także i wtedy, gdy nie korzystaliśmy z wyjątków.

Obiekty lokalne nie są jednak jedynymi z jakich korzystamy w C++. Wiemy też, że możliwe jest dynamiczne tworzenie obiektów na stercie, czyli w rezerwuarze pamięci. Dokonujemy tego poprzez `new`.
Pamięć jest z kolei jednym z tak zwanych **zasobów** (ang. *resources*), czyli zewnętrznych „bogactw naturalnych” komputera. Możemy do nich zaliczyć nie tylko pamięć operacyjną, ale np. otwarte pliki dyskowe, wyłączność na wykorzystanie pewnych urządzeń lub aktywne połączenia internetowe. Właściwe korzystanie z takich zasobów jest jednym z zadań każdego poważnego programu.

Zazwyczaj odbywa się ono według prostego schematu:

- najpierw pozyskujemy żądany zasób w jakiś sposób (np. alokujemy pamięć poprzez `new`)
- potem możemy do woli korzystać z tego zasobu (np. zapisywac dane do pamięci)
- na koniec zwalniamy zasób, jeżeli nie jest już nam potrzebny (czyli korzystamy z `delete` w przypadku pamięci)

Najbardziej znany nam zasob, czyli pamięć opercyjna, jest przez nas wykorzystywany choćby tak:

```
CFoo* pFoo = new CFoo;      // alokacja (utworzenie) obiektu-zasobu
// (robimy coś...)
```

```
delete pFoo;                // zwolnienie obiektu-zasobu
```

Między stworzeniem a zniszczeniem obiektu może jednak zajść sporo zdarzeń. W szczególności: możliwe jest rzucenie wyjątku.
Co się wtedy stanie?… Wydawać by się mogło, że obiekt zostanie zniszczony, bo przecież tak było zawsze… Błąd! Obiekt, na który wskazuje `pFoo` **nie zostanie zwolniony** z prostego powodu: nie jest on obiektem lokalnym, rezydującym na stosie, lecz tworzonym dynamicznie na stercie. Sami wydajemy polecenie jego utworzenia (`new`), więc również sami musimy go potem usunąć (poprzez `delete`). Zostanie natomiast zniszczony wskaźnik na niego (zmienna `pFoo`), bo jest to zmienna lokalna - co aczkolwiek nie jest dla nas żadną korzyścią.

Możesz zapytać: „A w czym problem? Skoro pamięć należy zwolnić, to zróbmy to przed rzuceniem wyjątku - o tak:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...
   if (warunek_rzucenia_wyjątku)
   {
         delete pFoo;
         throw wyjątek;
   }
   // ...

   delete pFoo;
}
catch (typ obiekt)
{
   // ...
}
```

To powinno rozwiązać problem.”
Taki sposób to jednak oznaka skrajnego i niestety nieuzasadnionego optymizmu. Bo kto nam zagwarantuje, że wyjątki, które mogą nam przeszkadzać, będą rzucane **wyłącznie przez nas**?… Możemy przecież wywołać jakąś zewnętrzną funkcję, która sama będzie wyrzucała wyjątki - nie pytając nas o zgodę i nie bacząc na naszą zaalokowaną pamięć, o której przecież **nic nie wie**!
„To też nie katastrofa”, odpowiesz, „Możemy przecież wykryć rzucenie wyjątku i w odpowiedzi zwolnić pamięć:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...
   try
   {
         // wywołanie funkcji potencjalnie rzucającej wyjątki
         FunkcjaKtoraMozeWyrzucicWyjatek();
   }
   catch (...)
   {
         // niszczymy obiekt
         delete pFoo;

         // rzucamy dalej otrzymany wyjątek
```

```
            throw;
        }
        // ...

        delete pFoo;
    }
    catch (typ obiekt)
    {
        // ...
    }
```

Blok `catch(...)` złapie nam wszystkie wyjątki, a my w jego wnętrzu zwolnimy pamięć i rzucimy je dalej poprzez `throw;`. Wszystko proste, czyż nie?"
Brawo, twoja pomysłowość jest całkiem duża. Już widzę te dziesiątki wywołań funkcji bibliotecznych, zamkniętych w ich własne bloki `try-catch(...)`, które dbają o zwalnianie pamięci… Jak sądzisz, na ile eleganckie, efektywne (zarówno pod względem czasu wykonania jak i zakodowania) i łatwe w konserwacji jest takie rozwiązanie?…

Jeżeli zastanowisz się nad tym choć trochę dłuższą chwilę, to zauważysz, że to bardzo złe wyjście. Jego stosowanie (podobnie zresztą jak `delete` przed `throw`) jest świadectwem koszmarnego stylu programowania. Pomyślmy tylko, że wymaga to wielokrotnego napisania instrukcji `delete` - powoduje to, że kod staje się bardzo nieczytelny: na pierwszy rzut oka można pomyśleć, że kilka(naście) razy usuwany jest obiekt, który tworzymy tylko raz. Poza tym obecność tego samego kodu w wielu miejscach znakomicie utrudnia jego zmianę.
Być może teraz pomyślałeś o preprocesorze i jego makrach… Jeśli naprawdę chciałbyś go zastosować, to bardzo proszę. Potem jednak nie narzekaj, że wyprodukowałeś kod, który stanowi zagadkę dla jasnowidza.

Teraz możesz się oburzyć: „No to co należy zrobić?! Przecież nie możemy dopuścić do powstawania wycieków pamięci czy niezamykania plików! Może należy po prostu zrezygnować z tak nieprzyjaznego narzędzia, jak wyjątki?" Cóż, możemy nie lubić wyjątków (szczególnie w tej chwili), ale nigdy od nich nie uciekniemy. Jeżeli sami nie będziemy ich stosować, to użyje ich ktoś inny, którego kodu my będziemy potrzebowali. Na wyjątki nie powinniśmy się więc obrażać, lecz spróbować je zrozumieć. Rozwiązanie problemu zasobów, które zaproponowaliśmy wyżej, jest złe, ponieważ próbuje wtrącić się w automatyczny proces odwijania stosu ze swoim ręcznym zwalnianiem zasobów (tutaj pamięci). Nie tędy droga; należy raczej zastosować taką metodę, która pozwoli nam czerpać korzyści z automatyki wyjątków.
Teraz poznamy właściwy sposób dokonania tego.

Problem z niezwolnionymi zasobami występuje we wszystkich językach, w których funkcjonują wyjątki. Trzeba jednak przyznać, że w większości z nich poradzono sobie z nim znacznie lepiej niż w C++. Przykładowo, Java i Object Pascal posiadają możliwość zdefiniowania dodatkowego (obok `catch`) bloku `finally` ('nareszcie'). W nim zostaje umieszczany kod wykonywany zawsze - niezależnie od tego, czy wyjątek w `try` wystąpił, czy też nie. Jest to więc idealne miejsce na instrukcje zwalniające zasoby, pozyskane w bloku `try`. Mamy bowiem gwarancję, iż zostaną one poprawnie oddane niezależnie od okoliczności.

Opakowywanie

Pomysł jest dość prosty. Jak wiemy, podczas odwijania stosu niszczone są wszystkie obiekty lokalne. W przypadku, gdy są to obiekty naszych własnych klas, do pracy ruszają wtedy destruktory tych klas. Właśnie we wnętrzu tych destruktorów możemy umieścić kod zwalniający przydzieloną pamięć czy jakikolwiek inny zasób.

Wydaje się to podobne do ręcznego zwalniania zasobów przed rzuceniem wyjątku lub w blokach `catch(...)`. Jest jednak jedna bardzo ważna różnica: **nie musimy tutaj wiedzieć**, w którym dokładnie miejscu wystąpi wyjątek. Kompilator bowiem i tak wywoła destruktor obiektu - nieważne, gdzie i jaki wyjątek został rzucony.

Skoro jednak mamy używać destruktorów, to trzeba rzecz jasna zdefiniować jakieś klasy. Potem zaś należy w bloku `try` tworzyć obiekty tychże klas, by ich destruktory zostały wywołane w przypadku wyrzucenia jakiegoś wyjątku.
Jak to należy uczynić? Kwestia nie jest trudna. Najlepiej jest zrobić tak, aby dla każdego pojedynczego zasobu (jak zaalokawany blok pamięci, otwarty plik, itp.) istniał jeden obiekt. W momencie zniszczenia tego obiektu (z powodu rzucenia wyjątku) zostanie wywołany destruktor jego klasy, który zwolni zasób (czyli np. usunie pamięć albo zamknie plik).

Destruktor wskaźnika?…

To bardzo proste, prawda? ;) Ale żeby było jeszcze łatwiejsze, spójrzmy na prosty przykład. Zajmiemy się zasobem, który najbardziej znamy, czyli pamięcią operacyjną; oto przykład kodu, który może spowodować jej wyciek:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...

         throw "Cos sie stalo";
         // obiekt niezwolniony, mamy wyciek!
}
// (tutaj catch)
```

Przyczyna jest oczywiście taka, iż odwijanie stosu nie usunie obiektu zaalokowanego dynamicznie na stercie. Usunięty zostanie rzecz jasna **sam wskaźnik** (czyli zmienna `pFoo`), ale na tym się skończy. Kompilator nie zajmie się obiektem, na który ów wskaźnik pokazuje.

Zapytasz: „A czemu nie? Przecież mógłby to zrobić”. Pomyśl jednak, że nie musi to być wcale jedyny wskaźnik pokazujący na dynamiczny obiekt. W przypadku usunięcia obiektu wszystkie pozostałe stałyby się nieważne. Oprócz tego byłoby to złamanie zasady, iż obiekty stworzone jawnie (poprzez `new`) muszą być także jawnie zniszczone (przez `delete`).

My jednak chcielibyśmy, aby wraz z końcem życia wskaźnika skończył się także żywot pamięci, na którą on pokazuje. Jak można to osiągnąć?
Cóż, gdyby nasz wskaźnik był obiektem jakiejś klasy, wtedy moglibyśmy napisać instrukcję `delete` w jej destruktorze. Tak jest jednak nie jest: wskaźnik to typ wbudowany[122], więc nie możemy napisać dlań destruktora - podobnie jak nie możemy tego zrobić dla typu `int` czy `float`.

Sprytny wskaźnik

Wskaźnik musiałby więc być klasą… Dlaczego nie? Podkreślałem w zeszłym rozdziale, że klasy w C++ są tak pomyślane, aby mogły one naśladować typy podstawowe. Czemu zatem nie możnaby stworzyć sobie takiej klasy, która działaby jak wskaźnik - typ

[122] Wskaźnik może wprawdzie pokazywać na typ zdefiniowany przez użytkownika, ale sam zawsze będzie typem wbudowanym. Jest to przecież zwykła liczba - adres w pamięci.

wbudowany? Wtedy mielibyśmy pełną swobodę w określeniu jej destruktora, a także innych metod.

Oczywiście, nie my pierwsi wpadliśmy na ten pomysł. To rozwiązanie jest szeroko znane i nosi nazwę **sprytnych wskaźników** (ang. *smart pointers*). Takie wskaźniki są podobne do zwykłych, jednak przy okazji oddają jeszcze pewne dodatkowe przysługi. W naszym przypadku chodzi o dbałość o zwolnienie pamięci w przypadku wystąpienia wyjątku. Sprytny wskaźnik jest klasą. Ma ona jednak odpowiednio przeciążone operatory - tak, że korzystanie z jej obiektów niczym nie różni się od korzystania z normalnych wskaźników. Popatrzmy na znany z zeszłego rozdziału przykład:

```
class CFooSmartPtr
{
   private:
         // opakowywany, właściwy wskaźnik
         CFoo* m_pWskaznik;

   public:
         // konstruktor i destruktor
         CFooSmartPtr(CFoo* pFoo) : m_pWskaznik(pFoo)    { }
         ~CFooSmartPtr()   { if (m_pWskaznik) delete m_pWskaznik; }

         //-------------------------------------------------------------

         // operator dereferencji
         CFoo& operator*()       { return *m_pWskaznik; }

         // operator wyłuskania
         CFoo* operator->()      { return m_pWskaznik; }
};
```

Jest to inteligentny wskaźnik na obiekty klasy `CFoo`; docelowy typ jest jednak nieistotny, bo równie dobrze możnaby pokazywać na liczby typu `int` czy też inne obiekty. Ważna jest zasada działania - zupełnie nieskomplikowana.
Klasy `CFooSmartPtr` używamy po prostu zamiast typu `CFoo*`:

```
try
{
   CFooSmartPtr pFoo = new CFoo;

   // ...

         throw "Cos sie stalo";
         // niszczony obiekt pFoo i wywoływany destruktor CFooSmartPtr
}
// (tutaj catch)
```

Dzięki przeciążeniu operatorów korzystamy ze sprytnego wskaźnika dokładnie w ten sam sposób, jak ze zwykłego. Poza tym rozwiązujemy problem ze zwolnieniem pamięci: zajmuje się tym destruktor klasy `CFooSmartPtr`. Stosuje on operator `delete` wobec właściwego, wewnętrznego wskaźnika (typu „normalnego”, czyli `CFoo*`), usuwając stworzony dynamicznie obiekt. Robi niezależnie od tego, gdzie i kiedy (i czy) wystąpił jakikolwiek wyjątek. Wystarczy, że zostanie zlikwidowany obiekt `pFoo`, a to pociągnie za sobą zwolnienie pamięci.

I o to nam właśnie chodziło. Wykorzystaliśmy mechanizm odwijania stosu do zwolnienia zasobów, które normalnie byłyby pozostawione same sobie. Nasz problem został rozwiązany.

Nieco uwag

Aby jednak nie było aż tak bardzo pięknie, na koniec paragrafu muszę jeszcze trochę poględzić :) Chodzi mianowicie o dwie ważne sprawy związane ze sprytnymi wskaźnikami, których używamy w połączeniu z mechanizmem wyjątków.

Różne typy wskaźników

Zaprezentowana wyżej klasa `CFooSmartPtr` jest typem inteligentnego wskaźnika, który może pokazywać na obiekty jakiejś zdefiniowanej wcześniej klasy `CFoo`. Przy jego pomocy nie możemy odnosić się do obiektów innych klas czy typów podstawowych.

Jeśli jednak będzie to konieczne, wówczas musimy niestety napisać nową klasę wskaźnika. Nie jest to trudne: wystarczy w definicji `CFooSmartPtr` zmienić wystąpienia `CFoo` np. na `int`. W następnym rozdziale poznamy zresztą o wiele bardziej efektywną technikę (mianowicie szablony), która uwolni nas od tej żmudnej pracy.
Za chwilę też przyjrzymy się rozwiązaniu, jakie przygotowali dla nas sami twórcy C++ w Bibliotece Standardowej.

Używajmy tylko tam, gdzie to konieczne

Muszę też powtórzyć to, o czym już wspomniałem przy pierwszym spotkaniu ze sprytnymi wskaźnikami. Otóż trzeba pamiętać, że nie są one uniwersalnym lekiem na wszystkie bolączki programisty. Nie należy ich stosować wszędzie, ponieważ każdy rodzaj inteligentnego wskaźnika (my na razie poznaliśmy jeden) ma ściśle określone zastosowania.
W sytuacjach, w których z powodzeniem sprawdzają się zwykłe wskaźniki, powinniśmy nadal z nich korzystać. Dopiero w takich przypadkach, gdy są one niewystarczające, musimy sięgnąć go bardziej wyrafinowane rozwiązania. Takim przypadkiem jest właśnie rzucanie wyjątków.

Co już zrobiono za nas

Metoda opakowywania zasobów może się wydawać nazbyt praco- i czasochłonna, a przede wszystkim wtórna. Stosując ją pewnie szybko zauważyłbyś, że napisane przez ciebie klasy powinny być obecne w niemal każdym programie korzystającym z wyjątków.

Naturalnie, mogą być one dobrym punktem wyjścia dla twojej własnej biblioteki z przydatnymi kodami, używanymi w wielu aplikacjach. Niewykluczone, że kiedyś będziesz musiał napisać przynajmniej kilka takich klas-opakowań, jeżeli zechcesz skorzystać z zasobów innych niż pamięć operacyjna czy pliki dyskowe.
Na razie jednak lepiej chyba sprawdzą się narzędzia, które otrzymujesz wraz z językiem C++ i jego Biblioteką Standardową. Zobaczmy pokrótce, jak one działają; ich dokładny opis znajdziesz w kolejnych rozdziałach, poświęconych samej tylko Bibliotece Standardowej.

Klasa `std::auto_ptr`

Sprytne wskaźniki chroniące przed wyciekami pamięci, powstającymi przy rzucaniu wyjątków, są dość często używane w praktyce. Samodzielne ich definiowanie byłoby więc uciążliwe. W C++ mamy więc już stworzoną do tego klasę `std::auto_ptr`.
Ściślej mówiąc, `auto_ptr` jest szablonem klasy. Co to dokładnie znaczy, dowiesz się w następnym rozdziale. Póki co będziesz wiedział, iż pozwala to na używanie `auto_ptr` w charakterze wskaźnika do **dowolnego typu** danych. Nie musimy już zatem definiować żadnych klas.

Aby skorzystać z `auto_ptr`, trzeba jedynie dołączyć standardowy plik nagłówkowy *memory*:

```
#include <memory>
```

Teraz możemy już korzystać z tego narzędzia. Z powodzeniem może ono zastąpić naszą pieczołowicie wypracowaną klasę `CFooSmartPtr`:

```
try
{
   std::auto_ptr<CFoo> pFoo(new CFoo);

   // ...

         throw "Cos sie stalo";
         // przy niszczeniu wskaźnika auto_ptr zwalniana jest pamięć
}
// (tutaj catch)
```

Konstrukcja `std::auto_ptr<CFoo>` pewnie wygląda nieco dziwnie, ale łatwo się do niej przyzwyczaisz, gdy już poznasz szablony. Można z niej także wydedukować, że w nawiasach kątowych `<>` podajemy typ danych, na który chcemy pokazywać poprzez `auto_ptr` - tutaj jest to `CFoo`. Łatwo domyślić się, że chcąc mieć wskaźnik na typ `int`, piszemy `std::auto_ptr<int>`, itp.
Zwróćmy jeszcze uwagę, w jaki sposob umieszcza się instrukcję `new` w deklaracji wskaźnika. Z pewnych powodów, o których nie warto tu mówić, konstruktor klasy `auto_ptr` jest opatrzony słówkiem `explicit`. Dlatego też nie można użyć znaku `=`, lecz trzeba jawnie przekazać parametr, będący normalnym wskaźnikiem do zaalokowanego poprzez `new` obszaru pamięci.
W sumie więc składnia deklaracji wskaźnika `auto_ptr` wygląda tak:

```
std::auto_ptr<typ> wskaźnik(new typ[(parametry_konstruktora_typu)]);
```

O zwolnienie pamięci nie musimy się martwić. Destruktor `auto_ptr` usunie ją zawsze; niezależnie od tego, czy wyjątek faktycznie wystąpi.

Pliki w Bibliotece Standardowej

Oprócz pamięci drugim ważnym rodzajem zasobów są pliki dyskowe. O dokładnym sposobie ich obsługi powiemy sobie aczkolwiek dopiero wtedy, gdy zajmiemy się strumieniami Biblioteki Standardowej.

Tutaj chcę tylko wspomnieć, że metody dostępu do plików, jakie są tam oferowane, całkowicie poprawnie współpracują z wyjątkami. Oto przykład:

```
#include <fstream>

try
{
   // stworzenie strumienia i otwarcie pliku do zapisu
   std::ofstream Plik("plik.txt", ios::out);

   // zapisanie czegoś do pliku
   Plik << "Coś";

   // ...

         throw "Cos sie stalo";
         // strumień jest niszczony, a plik zamykany
}
// (tutaj catch)
```

Plik reprezentowany przez strumień `Plik` zostanie zawsze zamknięty. W każdym przypadku - wystąpienia wyjątku lub nie - wywołany bowiem będzie destruktor klasy `ofstream`, a on tym się właśnie zajmie. Nie trzeba więc martwić się o to.

***

Tak zakończymy omawianie procesu odwijania stosu i jego konsekwencji. Teraz zobaczysz, jak w praktyce powinno się korzystać z mechanizmu wyjątków w C++.

Wykorzystanie wyjątków

Dwa poprzednie podrozdziały mówiły o tym, czym są wyjątki i jak działa ten mechanizm w C++. W zasadzie na tym możnaby poprzestać, ale taki opis na pewno nie będzie wystarczający. Jak każdy element języka, także i wyjątki należy używać we właściwy sposób; korzystaniu z wyjątków w praktyce zostanie więc poświęcony ten podrozdział.

Wyjątki w praktyce

Zanim z pieśnią na ustach zabierzemy się do wykorzystywania wyjątków, musimy sobie odpowiedzieć na jedno fundamentalne pytanie: czy tego potrzebujemy? Takie postawienie sprawy jest pewnie zaskakujące - dotąd wszystkie poznawane przez nas elementy C++ były właściwie niezbędne do efektywnego stosowania tego języka. Czy z wyjątkami jest inaczej? Przyjrzyjmy się sprawie bliżej…

Może powiedzmy sobie o dwóch podstawowych sytuacjach, kiedy wyjątków **nie powinniśmy** stosować. W zasadzie można je zamknąć w jedno stwierdzenie:

Nie powinno się wykorzystywać wyjątków tam, gdzie z powodzeniem wystarczają inne techniki sygnalizowania i obsługi błędów.

Oznacza to, że:

- nie powinniśmy „na siłę” dodawać wyjątków do istniejącego programu. Jeżeli po przetestowaniu działa on dobrze i efektywnie bez wyjątków, nie ma żadnego powodu, aby wprowadzać do kodu ten mechanizm
- dla tworzonych od nowa, lecz krótkich programów wyjątki mogą być zbyt potężnym narzędziem. Wysiłek włożony w jego zaprogramowanie (jak się zaraz przekonamy - wcale niemały) nie musi się opłacać. Co oznacza pojęcie ‘krótki program’, to już każdy musi sobie odpowiedzieć sam; zwykle uważa się, że krótkie są te aplikacje, które nie przekraczają rozmiarami 1000-2000 linijek kodu

Widać więc, że nie każdy program musi koniecznie stosować ten mechanizm. Są oczywiście sytuacje, gdy obyć się bez niego jest bardzo trudno, jednak nadużywanie wyjątków jest zazwyczaj gorsze niż ich niedostatek. O obu sprawach (korzyściach płynących z wyjątków i ich przesadnemu stosowaniu) powiemy sobie jeszcze później.

Załóżmy jednak, że zdecydowaliśmy się wykorzystywać wyjątki. Jak poprawnie zrealizować te intencje? Jak większość rzeczy w programowaniu, nie jest to trudne :) Musimy mianowicie:

- pomyśleć, jakie sytuacje wyjątkowe mogą wystąpić w naszej aplikacji i wyróżnić wśród nich poszczególne rodzaje, a nawet pewną hierarchię. To pozwoli na stworzenie odpowiednich klas dla obiektów wyjątków, czym zajmiemy się w pierwszym paragrafie

- we właściwy sposób zorganizować obsługę wyjątków - chodzi głównie o rozmieszczenie bloków `try` i `catch`. Ta kwestia będzie przedmiotem drugiego paragrafu

Potem możemy już tylko mieć nadzieję, że nasza ciężko wykonana praca… nigdy nie będzie potrzebna. Najlepiej przecież byłoby, aby sytuacje wyjątkowe nie zdarzały się, a nasze programy działały zawsze zgodnie z zamierzeniami… Cóż, praca programisty nie jest usłana różami, więc tak nigdy nie będzie. Nauczmy się więc poprawnie reagować na wszelkiego typu nieprzewidziane zdarzenia, jakie mogą się przytrafić naszym aplikacjom.

Projektowanie klas wyjątków

C++ umożliwia rzucenie w charakterze wyjątków obiektów dowolnych typów, także tych wbudowanych. Taka możliwość jest jednak mało pociągająca, jako że pojedyncza liczba czy napis nie niosą zwykle wystarczającej wiedzy o powstałej sytuacji.
Dlatego też powszechną praktyką jest tworzenie własnych typów (klas) dla obiektów wyjątków. Takie klasy zawierają w sobie więcej informacji zebranych „z miejsca katastrofy", które mogą być przydatne w rozpoznaniu i rozwiązaniu problemu.

Definiujemy klasę

Co więc powinien zawierać taki obiekt? Najważniejsze jest ustalenie rodzaju błędu oraz miejsca jego wystąpienia w kodzie. Typowym zestawem danych dla wyjątku może być zatem:

- nazwa pliku z kodem i numer wiersza, w którym rzucono wyjątek. Do tego można dodać jeszcze datę kompilacji programu, aby rozróżnić jego poszczególne wersje
- dane identyfikacyjne błędu - w najprostszej wersji tekstowy komunikat

Nasza klasa wyjątku mogłaby więc wyglądać tak:

```
#include <string>

class CException
{
   private:
         // dane wyjątku
         std::string m_strNazwaPliku;
         unsigned    m_uLinijka;
         std::string m_strKomunikat;

   public:
         // konstruktor
         CException(const std::string& strNazwaPliku,
                    unsigned uLinijka,
                    const std::string& strKomunikat)
               : m_strNazwaPliku(strNazwaPliku),
                 m_uLinijka(uLinijka),
                 m_strKomunikat(strKomunikat)             { }

         //-------------------------------------------------------------

         // metody dostępowe
         std::string NazwaPliku() const      { return m_strNazwaPliku; }
         unsigned    Linijka() const         { return m_uLinijka; }
         std::string Komunikat() const       { return m_strKomunikat; }
};
```

Dość obszerny konstruktor pozwala na podanie wszystkich danych za jednym zamachem, w instrukcji `throw`:

```
throw CException(__FILE__, __LINE__, "Cos sie stalo");
```

Dla wygody można sobie nawet zdefiniować odpowiednie makro, jako że `__FILE__` i `__LINE__` pojawią się w każdej instrukcji rzucenia wyjątku. Jest to szczególnie przydatne, jeżeli do wyjątku dołączymy jeszcze inne informacje pochodzące predefiniowanych symboli preprocesora.

Także konstruktor klasy może dokonywać zbierania jakichś informacji od programu. Mogą to być np. zrzuty pamięci (ang. *memory dumps*), czyli obrazy zawartości kluczowych miejsc pamięci operacyjnej. Takie zaawansowane techniki są aczkolwiek przydatne tylko w naprawdę dużych programach.

Po złapaniu takiego obiektu możemy pokazać związane z nim dane - na przykład tak:

```
catch (CException& Wyjatek)
{
   std::cout << "     Wystapil wyjatek      " << std::endl;
   std::cout << "---------------------------" << std::endl;

   std::cout << "Komunikat:\t"   << Wyjatek.Komunikat() << std::endl;
   std::cout << "Plik:\t"        << Wyjatek.NazwaPliku() << std::endl;
   std::cout << "Wiersz kodu:\t" << Wyjatek.Linijka() << std::endl;
}
```

Jest to już całkiem zadowalająca informacja o błędzie.

Hierarchia wyjątków

Pojedyncza klasa wyjątku rzadko jest jednak wystarczająca. Wadą takiego skromnego rozwiązania jest to, że ze względu na charakter danych o sytuacji wyjątkowej, jakie zawiera obiekt, ograniczamy sobie możliwość obsługi wyjątku. W naszym przypadku trudno jest podjąć jakiekolwiek działania poza wyświetleniem komunikatu i zamknięciem programu.

Dla zwiększenia pola manewru możnaby dodać do klasy jakieś pola typu wyliczeniowego, określające bliżej rodzaj błędu; wówczas w bloku `catch` pojawiłaby się pewnie jakaś instrukcja `switch`.
Jest aczkolwiek praktyczniejsze i bardziej elastyczne wyjście: możemy użyć dziedziczenia.

Okazuje się, że rozsądne jest stworzenie hierarchii sytuacji wyjątków i odpowiadającej jej hierarchii klas wyjątków. Opiera się to na spostrzeżeniu, że możliwe błędy możemy najczęściej w pewien sposób sklasyfikować. Przykładowo, możnaby wyróżnić wyjątki związane z pamięcią, z plikami dyskowymi i obliczeniami matematycznymi: wśród tych pierwszych mielibyśmy np. brak pamięci (ang. *out of memory*) i błąd ochrony (ang. *access violation*); dostęp do pliku może być niemożliwy chociażby z powodu jego braku albo nieobecności dysku w napędzie; działania na liczbach mogą wreszcie doprowadzić do dzielenia przez zero lub wyciągania pierwiastka z liczby ujemnej.
Taki układ, oprócz możliwości rozróżnienia poszczególnych typów wyjątków, ma jeszcze jedną zaletę. Można bowiem dla każdego typu zakodować specyficzny dla niego sposób obsługi, stosując do tego metody wirtualne - np. w ten sposób:

```
// klasa bazowa
class IException
{
   public:
         // wyświetl informacje o wyjątku
```

```
        virtual void Wyswietl();
};

//---------------------------------------------------------------------

// wyjątek związany z pamięcią
class CMemoryException : public IException
{
   public:
        // działania specyficzne dla tego rodzaju wyjątku
        virtual void Wyswietl();
};

// wyjątek związany z plikami
class CFilesException : public IException
{
   public:
        // działania specyficzne dla tego rodzaju wyjątku
        virtual void Wyswietl();
};
```

Pamiętajmy jednak, że nadmierne rozbudowywanie hierarchii też nie ma zbytniego sensu. Nie wydaje się na przykład słuszne wyróżnianie osobnych klas dla wyjątków dzielenia przez zero, pierwiastka kwadratowego z liczy ujemnej oraz podniesienia zera do potęgi zerowej. Jest bowiem wielce prawdopodobne, że jedyna różnica między tymi sytuacjami będzie polegała na treści wyświetlanego komunikatu. W takich przypadkach zdecydowanie wystarczy pojedyncza klasa.

Organizacja obsługi wyjątków

Zdefiniowana uprzednio klasę lub jej hierarchię będziemy pewnie mieli okazję nieraz wykorzystać. Ponieważ nie jest to takie oczywiste, warto poświęcić temu zagadnieniu osobny paragraf.

Umiejscowienie bloków `try` i `catch`

Wydawałoby się, że obsługa wyjątków to bardzo prosta czynność - szczególnie, jeśli mamy już zdefiniowany dla nich odpowiednie klasy. Niestety, polega to na czymś więcej niż tylko napisaniu „niepewnego” kodu w bloku `try` i instrukcji obsługi błędów `catch`.

Kod warstwowy

Jednym z podstawowych powodów, dla których wprowadzono wyjątki w C++, była konieczność zapewnienia jakiegoś sensownego sposobu reakcji na błędy w programach o skomplikowanym kodzie. Każdy większy (i dobrze napisany) program ma bowiem skłonność do „rozwarstwiania” kodu.
Nie jest to bynajmniej niepożądane zjawisko, wręcz przeciwnie. Polega ono na tym, że w aplikacji możemy wyróżnić fragmenty wyższego i niższego poziomu. Te pierwsze odpowiadają za całą logikę aplikacji, w tym za jej komunikację z użytkownikiem; te drugie wykonują bardziej wewnętrzne czynności, takie jak na przykład zarządzanie pamięcią operacyjną czy dostęp do plików na dysku.

Taki podział jest korzystny, ponieważ ułatwia konserwację programu, a także wykorzystywanie pewnych fragmentów kodu (zwłaszcza tych niskopoziomowych) w kolejnych projektach. Funkcje odpowiedzialne za pewne proste czynności, jak wspomniany dostęp do plików nie muszą nic wiedzieć o tym, kto je wywołuje - właściwie to nawet **nie powinny**. Innymi słowy:

Kod niższego poziomu powinien być zazwyczaj **niezależny** od kodu wyższego poziomu.

Tylko wtedy zachowujemy wymienione wyżej zalety „warstwowości” programu.

Podawanie błędów wyżej

Podział warstwowy wymusza poza tym dość ściśle ustalony przepływ danych w aplikacji. Odbywa się on zawsze tak, że kod wyższego poziomu przekazuje do niższych warstw konieczne informacje (np. nazwę pliku, który ma być otwarty) i odbiera rezultaty wykonanych operacji (czyli zawartość pliku). Potem wykorzystuje je do swych własnych zadań (np. do wyświetlenia pliku na ekranie).
Ten naturalny układ działa dobrze… dopóki się nie zepsuje :) Przyczyną mogą być sytuacje wyjątkowe występujące w kodzie niższego poziomu. Typowym przykładem może być brak żądanego pliku, wobec czego jego otwarcie nie jest możliwe. Funkcja, która miała tego dokonać, nie będzie potrafiła poradzić sobie z tym błędem, ponieważ nazwa pliku do otwarcie pochodziła z zewnątrz - „z góry”. Może jedynie poinformować wywołującego o zainstniałej sytuacji.

I tutaj wkraczają na scenę opisane na samym początku rozdziału mechanizmy obsługi błędów. Jednym z nich są właśnie wyjątki.

Dobre wypośrodkowanie

Ich stosowanie jest szczególnie wskazane właśnie wtedy, gdy nasz kod ma kilka logicznych warstw, co zresztą powinno zdarzać się jak najczęściej. Wówczas odnosimy jedną zasadniczą korzyść: nie musimy martwić się o sposób, w jaki informacja o błędzie dotrze z „pokładów głębinowych” programu, gdzie wystąpiła, na „górne piętra”, gdzie mogłaby zostać właściwie obsłużona.

Naszym problemem jest jednak co innego. O ile zazwyczaj dokładnie wiadomo, gdzie wyjątek należy rzucić (wiadomo - tam gdzie coś się nie powiodło), o tyle trudność może sprawić wybranie właściwego miejsca na jego złapanie:

- jeżeli będzie on „za nisko”, wtedy najprawdopodobniej nie będzie możliwe podjęcie żadnych rozsądnych działań w reakcji na wyjątek. Przykładowo, wymieniona funkcja otwierająca plik nie powinna sama łapać wyjątku, który rzuci, bo będzie wobec niego bezradna. Skoro przecież rzuciła ten wyjątek, jest to właśnie znak, iż nie radzi sobie z powstała sytuacją i oddaje inicjatywę komuś bardziej kompetentnemu
- z drugiej strony, umieszczenie bloków `catch` „za wysoko” powoduje zbyt duże zamieszanie w funkcjonowaniu programu. Powoduje to, że punkt wykonania przeskakuje o całe kilometry, niespodziewanie przerywając wszystko znajdujące się po drodze zdania. Nie należy bowiem zapominać, że po rzuceniu wyjątku **nie ma już powrotu** - dalsze wykonywanie zostanie co najwyżej podjęte po wykonaniu bloku `catch`, który ten wyjątek. Całkowitym absurdem jest więc np. ujęcie całej zawartości funkcji `main()` w blok `try` i obsługa wszystkich wyjątków w następującym dalej bloku `catch`. Nietrudno przecież domyślić się, że takie rozwiązanie spowoduje zakończenie programu po każdym wystąpieniu wyjątku

Pytanie brzmi więc: jak osiągnąć rozsądny kompromis? Trzeba pogodzić ze sobą dwie racje:

- konieczność sensownej obsługi wyjątku
- konieczność przywrócenia programu do normalnego stanu

Należy więc łapać wyjątek w takim miejscu, w którym **już możliwe** jest jego **obsłużenie**, ale jednocześnie po jego zakończeniu program powinien **nadal móc** podjąć podjąć w miarę **normalną pracę**.

Przykład?… Jeżeli użytkownik wybierze opcję otwarcia pliku, ale potem poda nieistniejącą nazwę, program powinien po prostu poinformować o tym i ponownie zapytać o nazwę

pliku. Nie może natomiast zmuszać użytkownika do ponownego wybrania opcji otwarcia pliku. A już na pewno nie może niespodziewanie kończyć swojej pracy - to byłoby wręcz skandaliczne.

Chwytanie wyjątków w blokach `catch`

Poprawne chwytanie wyjątków w blokach `catch` to kolejne (ostatnie już na szczęście) zagadnienie, o którym musimy pamiętać. Wiesz na ten temat już całkiem sporo, ale nigdy nie zaszkodzi powtórzyć sobie przyswojone wiadomości i przyswoić nowe.

Szczegóły przodem - druga odsłona

Swego czasu zwróciłem ci uwagę na ważną sprawę kolejności bloków `catch`. Uświadomiłem, że ich działanie tylko z pozoru przypomina przeciążone funkcje, jako że porządek dopasowywania obiektu wyjątku ściśle pokrywa się z porządkiem samych bloków `catch`, a same dopasowywanie kończy przy pierwszym sukcesie.

W związku należy tak ustawiać bloki `catch`, aby na początek szły te, które precyzyjniej opisują typ wyjątku. Gdy zdefiniujemy sobie hierarchię klas wyjątków, ta zasada zyskuje jeszcze pewniejszą podstawę. W przypadku typów podstawowych (`int`, `double`…) może być dość trudne wyobrażenie się relacji „typ ogólny - typ szczegółowy”. Natomiast dla klas jest to oczywiste: wchodzi tu bowiem w grę jednoznaczny związek **dziedziczenia**. Jakie są więc konkretne wnioski? Ano takie, że:

Gdy stosujemy **hierarchię klas wyjątków**, powinniśmy **najpierw** próbować **łapać obiekty klas pochodnych**, a dopiero **potem obiekty klas bazowych**.

Mam nadzieję, iż wiesz doskonale, z jakiej fundamentalnej reguły programowania obiektowego wynika powyższa zasada[123].
Jeżeli zastosujemy klasy wyjątków z poprzedniego paragrafu, to ilustracją może być taki kawałek kodu:

```
try
{
   // ...
}
catch (CMemoryException& Wyjatek)
{
   // ...
}
catch (CFilesException& Wyjatek)
{
   // ...
}
catch (IException& Wyjatek)
{
   // ...
}
```

Instrukcje chwytające bardziej wyspecjalizowane wyjątki - `CMemoryException` i `CFilesException` - umieszczamy na samej górze. Dopiero niżej zajmujemy się pozostałymi wyjątkami, chwytając obiekty typu bazowego `IException`. Gdybyśmy czynili to na początku, złapalibyśmy absolutnie wszystkie swoje wyjątki - nie dając sobie szansy na rozróżnienie błędów pamięci od wyjątków plikowych lub innych.

[123] Oczywiście wynika ona stąd, że obiekt klasy pochodnej jest jednocześnie obiektem klasy bazowej. Albo też stąd, że zawsze istnieje niejawna konwersja z klasy pochodnej na klasy bazowej - jakkolwiek to wyrazimy, będzie poprawnie.

Widać więc po raz kolejny, że właściwe uporządkowanie bloków `catch` ma niebagatelne znaczenie.

Lepiej referencją

We wszystkich przytoczonych ostatnio kodach łapałem wyjatki poprzez referencje do nich, a nie poprzez same obiekty. Zbywaliśmy to dotąd milczeniem, ale czas ten fakt wyjaśnić.

Przyczyna jest właściwie całkiem prosta. Referencje są, jak pamiętamy, zakamuflowanymi wskaźnikami: faktycznie różnią się od wskaźników tylko drobnymi szczegółami, jak choćby składnią. Zachowują jednak ich jedną cenną właściwość obiektową: pozwalają na stosowanie polimorfizmu metod wirtualnych.
To doskonalne znane nam zjawisko jest więc możliwe do wykorzystania także przy obsłudze wyjątków. Oto przykład:

```
try
{
   // ...
}
catch (IException& Wyjatek)
{
   // wywołanie metody wirtualnej, późno wiązanej
   Wyjatek.Wyswietl();
}
```

Metoda wirtualna `Wyswietl()` jest tu późno wiązana, zatem to, który jej wariant - z klasy podstawowej czy pochodnej - zostanie wywołany, decyduje się podczas działania programu. Jest to więc inny sposób na swoiste rozróżnienie typu wyjątku i podjęcie działań celem jego obsługi.

Uwagi ogólne

Na sam koniec podzielę się jeszcze garścią uwag ogólnych dotyczących wyjątków. Przede wszystkim zastanowimy się nad korzyściami z używania tego mechanizmu oraz sytuacjami, gdzie często jest on nadużywany.

Korzyści ze stosowania wyjątków

Podstawowe zalety wyjątków przedstawiłem na początku rozdziału, gdy porównywałem je z innymi sposobami obsługi błędów. Teraz jednak masz już za sobą dogłębne poznanie tej techniki, więc pewnie zwątpiłeś w te przymioty ;) Nawet jeśli nie, to pokazane niżej argumenty przemawiające na korzyść wyjątków mogą pomóc ci w decyzji co do ich wykorzystania w konkretnej sytuacji.

Informacja o błędzie w każdej sytuacji

Pierwszą przewagą, jaką wyjątki mają nad innymi sposobami sygnalizowania błędów, jest uniwersalność: możemy je bowiem stosować w każdej sytuacji i w każdej funkcji.

No ale czy to coś nadzwyczajnego? Przecież wydawałoby się, że zarówno technika zwracania kodu błędu jak i wywołanie zwrotne, może być zastosowane wszędzie. To jednak nieprawda; oba te sposoby wymagają odpowiedniej deklaracji funkcji, uwzględniającej ich wykorzystanie. A nagłówek funkcji może być często ograniczony przez sam język albo inne czynniki - jest tak na przykład w:

- konstruktorach
- większości przeciążonych operatorów
- funkcjach zwrotnych dla zewnętrznych bibliotek

Do tej grupy możnaby zaliczyć też destruktory, ale jak przecież, z destruktorów **nie można** rzucać wyjątków.

Dzięki temu, że wyjątki nie opierają się na normalnym sposobie wywoływania i powrotu z funkcji, mogą być używane także i w tych specjalnych funkcjach.

Uproszczenie kodu

Jakkolwiek dziwnie to zabrzmi, wyjątki umożliwiają też znaczne uproszczenie kodu i uczynienie go przejrzystszym. Jest tak, gdyż pozwalają one przenieść sekwencje odpowiedzialne za obsługę błędów do osobnych bloków, z dala od właściwych instrukcji.

W normalnym kodzie procedury wyglądają mniej więcej tak:
- zrób coś
- *sprawdź, czy się udało*
- zrób coś innego
- *sprawdź, czy się udało*
- zrób jeszcze coś
- *sprawdź, czy nie było błędów*
- itd.

Wyróżnione tu sprawdzenia błędów są realizowane zwykle przy pomocy instrukcji `if` lub `switch`. Przy ich użyciu kod staje się więc plątaniną instrukcji warunkowych, raczej trudnych do czytania.
Gdy zaś używamy wyjątków, to obsługa błędów przenosi się na koniec algorytmu:
- zrób coś
- zrób coś innego
- zrób jeszcze coś
- itd.
- *obsłuż ewentualne niepowodzenia*

Oczywiście dla tych, którzy nie dbają o porządek w kodzie, jest to żaden argument, ale ty się chyba do nich nie zaliczasz?

Wzrost niezawodności kodu

Wreszcie można wytoczyć najcięższe działa. Wyjątki nie pozwalają na obojętność - na ignorowanie błędów.

Poprzedni akapit uświadamia, że tradycyjne metody w rodzaju zwracania rezultatu muszą być aktywnie wspomagane przez programistę, który używa wykorzystujących je funkcji. Nie musi jednak tego robić; kod skompiluje się tak samo poprawnie, jeżeli wartości zwracane zostaną całkowicie pominięte. Co więcej, może to prowadzić do pominięcia krytycznych błędów, które wprawdzie nie dają natychmiast katastrofalnych rezultatów, ale potrafią „przyczaić się” w zakamarkach aplikacji, by ujawnić się w najmniej spodziewanym momencie.
Mechanizm wyjątków jest skonstruowany zupełnie przeciwnie. Tutaj nie trzeba się wysilać, aby błąd dał znać o sobie, bowiem wyjątek zawsze wywoła jakąś reakcję - choćby nawet awaryjne zakończenie programu. Natomiast świadome zignorowanie wyjątku wymaga z kolei pewnego wysiłku.

Tak więc tutaj mamy do czynienia z sytuacją, w której to nie programista szuka błędu, lecz błąd szuka programisty. Jest to naturalnie znacznie lepsza sytuacja z punktu widzenia niezawodności programu, bo pozwala na łatwiejsze odszukanie występujących weń błędów.

Nadużywanie wyjątków

Czytając o zaletach wyjątków, nie można wpaść w bezkrytyczny zachwyt nad nimi. One nie są ani obowiązkową techniką programistyczną, ani też nie są lekarstwem na błędy w programach, ani nawet nie są pasującym absolutnie wszędzie rozwiązaniem. Wyjątków łatwo można nadużyć i dlatego chcę się przed tym przestrzec.

Nie używajmy ich tam, gdzie wystarczą inne konstrukcje

Początkujący programiści mają czasem skłonność do uważania, iż **każde niepowodzenie** wykonania jakiegoś zadania zasługuje na rzucenie wyjątku. Oto (zły) przykład:

```
// funkcja wyszukuje liczbę w tablicy
unsigned Szukaj(const CIntArray& aTablica, int nLiczba)
{
   // pętla porównuje kolejne elementy tablicy z szukaną liczbą
   for (unsigned i = 0; i < aTablica.Rozmiar{]; ++i)
         if (aTablica[i] == nLiczba)
               return i;

   // w razie niepowodzenia - wyjątek?...
   throw CError(__FILE__, __LINE__, "Nie znaleziono liczby");
}
```

Rzucanie wyjątku w razie nieznalezienia elementu tablicy to gruba przesada. Pomyślmy tylko, że kod wykorzystujący tę funkcję musiałby wyglądać mniej więcej tak:

```
// szukamy liczby nZmienna w tablicy aTablicaLiczb

try
{
   unsigned uIndeks = Szukaj(aTablicaLiczb, nZmienna);

   // zrób coś ze znalezioną liczbą...
}
catch (CError& Wyjatek)
{
   std::cout << Wyjatek.Komunikat() << std::endl;
}
```

Może i ma on swój urok, ale chyba lepiej skorzystać z mniej urokliwej, ale na pewno prostszej instrukcji `if`, porównującej po prostu rezultat funkcji `Szukaj()` z jakąś ustaloną stałą (np. `-1`), oznaczającą niepowodzenie szukania. Pozwoli to na wydorębnienie sytuacji faktycznie wyjątkowych od tych, które zdarzają się w normalnym toku działania programu. Nieobecność liczby w tablicy należy zwykle do tej drugiej grupy i nie jest wcale krytyczna dla funkcjonowania aplikacji - *ergo*: nie wymaga zastosowania wyjątków.

Nie używajmy wyjątków na siłę

Nareszcie, muszę powstrzymać wszystkich tych, którzy z zapałem rzucili się do implementacji wyjątków w swych gotowych i działających programach. Niesłusznie! Prawdopodobnie będzie to kawał ciężkiej, nikomu niepotrzebnej roboty. Nie ma sensu jej wykonywać, ponieważ zysk zwykle będzie nieadekwatny do włożonego wysiłku.

Co najwyżej można pokusić się o zastosowanie wyjątków w przypadku, gdy nowa wersja danego programu wymaga napisania jego kodu od nowa. Decyzja o tym, czy tak ma się stać w istocie, powinna być podjęta jak najwcześniej.

***

Praktyczne wykorzystanie wyjątków to sztuka, jak zresztą całe programowanie. Najlepszym nauczycielem będzie tu doświadczenie, ale jeśli zawartość tego podrozdziału pomoże ci choć trochę, to jego cel będę mógł uważać za osiągnięty.

Podsumowanie

Ten rozdział omawiał mechanizm wyjątków w języku C++. Rozpoczął się od przedstawienia kilku popularnych sposobów radzenia sobie z błędami, jakie moga wystapić w trakcie działania programu. Później poznałeś same wyjątki oraz podstawowe informacje o nich. Dalej zajęliśmy się zagadnieniem odwijania stosu i jego konsekwencji, by wreszcie nauczyć się wykorzystywać wyjątki w praktyce.

Pytania i zadania

Rozdział kończymy tradycyjną porcją pytań i ćwiczeń.

Pytania

1. Kiedy możemy mówić, iż mamy do czynienia z sytuacją wyjątkową?
2. Dlaczego specjalny rezultat funkcji nie zawsze jest dobrą metodą informowania o błędzie?
3. Czy różni się `throw` od `return`?
4. Dlaczego kolejność bloków `catch` jest ważna?
5. Jaka jest rola bloku `catch(...)`?
6. Czym jest specyfikacja wyjątków? Co dzieje się, jeżeli zostanie ona naruszona?
7. Które obiekty są niszczone podczas odwijania stosu?
8. W jakich funkcjach nie należy rzucać wyjątków?
9. W jaki sposób możemy zapewnić zwolnienie zasobów w przypadku wystąpienia wyjątku?
10. Dlaczego warto definiować własne klasy dla obiektów wyjątków?

Ćwiczenia

1. Zastanów się, jakie informacje powinien zawierać dobry obiekt wyjątku. Które z tych danych dostarcza nam sam kompilator, a które trzeba zapewnić sobie samemu?
2. (**Trudne**) Mechanizm wyjątków został pomyślany do obsługi błędów w trakcie działania programu. To jednak nie są jego jedyne możliwe zastosowanie; pomyśl, do czego potencjalnie przydatne mogą być jeszcze wyjątki - a szczególnie towarzyszący im proces odwijania stosu…