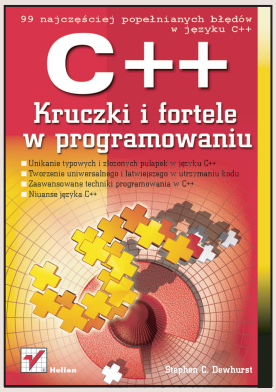
C++. Kruczki i fortele w programowaniu



Autor: Stephen C. Dewhurst
Tłumaczenie: Tomasz Żmijewski
ISBN: 83-7361-346-3
Tytuł oryginału: C++ Gotchas: Avoiding Common Problems in Coding and Design
Format: B5, stron: 266
Przykłady na ftp: 28 kB

„C++. Kruczki i fortele w programowaniu" to pomoc dla zawodowych programistów pozwalająca uniknąć lub poprawić dziewięćdziesiąt dziewięć najczęściej popełnianych i najbardziej szkodliwych błędów projektowych i programowych w C++. Jest to też książka, dzięki której można poznać niektóre niestandardowe cechy języka C++ i techniki programistyczne.

W książce omówiono typowe błędy występujące niemalże we wszystkich programach utworzonych w C++. Każdy z nich został starannie opisany, przedstawiono również konsekwencje wynikające z ich pojawienia się w kodzie programu i szczegółowy opis sposobów na ich uniknięcie. „C++. Kruczki i fortele w programowaniu" to książka o tym, jak uniknąć największych zagrożeń związanych z programowaniem w C++. Gotowa i praktyczna wiedza dla programistów, która pozwoli im uzyskać status ekspertów.

Omówione błędy dotyczą:

- Podstaw języka C++
- Składni języka
- Preprocesora
- Konwersji
- Inicjalizacji
- Zarządzania pamięcią i zasobami
- Polimorfizmu
- Projektowania klas
- Projektowania hierarchii

Stephen C. Dewhurst był jednym z pierwszych użytkowników C++ w Bell Labs. Ma ponad osiemnaście lat doświadczenia w stosowaniu języka C++ do takich zagadnień, jak projektowanie kompilatorów, zabezpieczenia, handel elektroniczny oraz elementy telekomunikacji wbudowane w większe systemy. Jest współautorem wydanej przez Prentice Hall w 1989 roku książki „Programming in C++", redaktorem „C/C++ Users Journal", poza tym prowadził kolumnę w „C++ Report". Jest autorem dwóch kompilatorów C++, napisał wiele artykułów o budowie kompilatorów i technikach programowania w C++.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Zarządzanie pamięcią i zasobami

C++ cechuje się ogromną elastycznością w zarządzaniu pamięcią ale niewielu programistów C++ w pełni rozumie dostępne mechanizmy. W tym obszarze języka przeciążanie, ukrywanie nazw, konstruktory, destruktory, wyjątki, funkcje statyczne wirtualne, funkcje operatory i funkcje zwykłe — wszystkie razem zapewniają niezwykłą elastyczność i możliwość sterowania zarządzaniem pamięcią. Niestety, wszystko się tu nieco komplikuje, co jest chyba zresztą nieuniknione.

W tym rozdziale przyjrzymy się różnym cechom języka C++ związanym z zarządzaniem pamięcią, pokażemy, jak czasem cechy te przedziwnie się splatają i jak można ich wzajemne interakcje uprościć.

Jako że pamięć jest jednym z wielu zasobów wykorzystywanych przez program, warto się dowiedzieć, w jaki sposób można ją wiązać z innymi zasobami i jak można zarządzać nią i innymi zasobami.

Zagadnienie 60: nieodróżnianie alokacji danej typu nietablicowego i tablicy

Czy obiekt `Widget` jest tym samym, co tablica obiektów `Widget`? Oczywiście, że nie. Dlatego zatem tak wielu programistów C++ ze zdziwieniem stwierdza, że inne operatory służą do alokacji i zwalniania tablic, a inne do typów nietablicowych?

Wiadomo, jak zaalokować i zwolnić pojedynczy obiekt `Widget` — należy użyć operatorów `new` i `delete`:

```
Widget *w = new Widget( arg );
// ...
delete w;
```

W przeciwieństwie do większości operatorów C++, zachowania operatora `new` nie można modyfikować przez przeciążanie. Operator `new` zawsze wywołuje funkcję `operator new`, aby uzyskać pamięć, potem ewentualnie ją zainicjalizować. W przypadku pokazanego powyżej obiektu `Widget` użycie operatora `new` spowoduje wywołanie funkcji `operator new` z jednym parametrem typu `size_t`, potem na niezainicjalizowanej pamięci zostanie wywołany konstruktor obiektu `Widget`.

Operator `delete` wywołuje destruktor obiektu `Widget` i następnie wywołuje funkcję `operator delete`, która zwolni wcześniej zaalokowaną na obiekt `Widget` pamięć.

Jeśli trzeba zmodyfikować sposób alokowania i zwalniania pamięci, korzysta się z przeciążania, podmiany lub ukrywania funkcji `operator new` i `operator delete`, a nie przez modyfikowanie operatorów `new` i `delete`.

Wiadomo też, jak alokować i zwalniać tablice obiektów `Widget`. Nie używa się do tego operatorów `new` ani `delete`:

```
w = new Widget[n];
// ...
delete [] w;
```

Zamiast tego używa się operatorów `new []` i `delete []` (czytanych jako „tablicowe `new`" i „tablicowe `delete`"). Tak jak `new` i `delete`, tak i ich tablicowe odpowiedniki nie mogą być modyfikowane. Tablicowy `new` najpierw wywołuje funkcję `operator new[]` alokującą pamięć, potem (w miarę potrzeb) przeprowadza inicjalizację każdego zaalokowanego elementu tablicy, od pierwszego do ostatniego. Tablicowy `delete` niszczy wszystkie elementy tablicy w kolejności odwrotnej niż były one inicjalizowane, następnie wywołuje funkcję `operator delete[]` w celu odzyskania pamięci.

Zauważmy na marginesie, że często lepiej zamiast tablic skorzystać obiektu `vector` z biblioteki standardowej. Obiekt `vector` działa niemalże równie szybko jak tablica a jego użycie jest bezpieczniejsze i jest on elastyczniejszy. Obiekt `vector` można zwykle traktować jako „inteligentną" tablicę. Jednak podczas niszczenia obiektu `vector` jego elementy są niszczone od pierwszego do ostatniego — odwrotnie niż w tablicy.

Funkcje zarządzające pamięcią muszą być odpowiednio sparowane. Jeśli za pomocą `new` alokujemy pamięć, musimy ją potem zwolnić za pomocą `delete`. Jeśli pamięć alokujemy za pomocą `malloc`, musimy ją zwolnić za pomocą `free`. Czasami użycie par `free` i `new` lub `malloc` i `delete` zadziała w pewnych warunkach, ale nie ma takiej gwarancji:

```
int *ip = new int(12);
// ...
free( ip ); // źle!
ip = static_cast<int *>(malloc( sizeof(int) ));
*ip = 12;
// ...
delete ip; // źle!
```

Taka sama zasada dotyczy także alokacji i usuwania tablic. Typowym błędem jest alokowanie pamięci na tablicę za pomocą tablicowego `new` i potem zwalnianie tej pamięci za pomocą nietablicowego `delete`. Tak jak w przypadku użycia pary `new/delete` może to zadziałać ale jest to błąd i wcześniej czy później się ujawni:

```
double *dp = new double[1];
// ...
delete dp; // źle!
```

Warto zauważyć, że kompilator nie może ostrzec o nieprawidłowym usuwania tablicy jako typu nietablicowego, gdyż nie jest w stanie odróżnić wskaźnika tablicy od wskaźnika pojedynczego elementu. Zwykle tablicowy operator `new` obok rezerwowanej pamięci umieści też informacje o wielkości bloku i liczbie elementów tablicy. Informacje te są potem sprawdzane i wykorzystywane przez tablicowy operator `delete` podczas usuwania tablicy.

Format tych informacji jest prawdopodobnie inny niż informacje uzyskiwane w przypadku nietablicowego operatora `new`. Jeśli do zwolnienia pamięci zaalokowanej tablicowym `new` użyje się nietablicowego `delete`, informacje o wielkości i liczbie elementów, przeznaczone dla tablicowego `delete`, zostaną przez nietablicowe `delete` prawdopodobnie źle zinterpretowane, czego skutki są trudne do przewidzenia. Może też się zdarzyć, że pamięć na dane typów nietablicowych i na dane tablicowe pochodzi z różnych obszarów. Użycie nietablicowego operatora `delete` do zwolnienia pamięci zaalokowanej na tablicę w obszarze tablic może doprowadzić do katastrofy.

```
delete [] dp; // prawidłowo
```

Opisane mylenie alokacji pamięci na dane typów nietablicowych i tablice występuje także przy kodowaniu metod do zarządzania pamięcią:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   // ...
};
```

Autor klasy `Widget` postanowił napisać specjalną wersję funkcji do zarządzania pamięcią na obiekty `Widget` ale nie wziął pod uwagę faktu, że tablicowe operatory `new` i `delete` mają inne nazwy funkcji niż ich nietablicowe odpowiedniki, wobec czego nie są ukrywane przez zdefiniowane w pokazanej klasie metody:

```
Widget *w = new Widget( arg ); // OK
// ...
delete w; // OK
w = new Widget[n];            // ups!
// ...
delete [] w;                  // ups!
```

W klasie `Widget` nie zadeklarowano funkcji `operator new[]` ani `operator delete[]`, więc do zarządzania pamięcią na tablice obiektów `Widget` będą używane ogólne wersje tych funkcji. Jest to prawdopodobnie sytuacja błędna, autor klasy `Widget` powinien podać tablicowe funkcje zarządzania pamięcią.

Jeśli jednak jest to postępowanie celowe, autor klasy powinien jasno wskazać to przyszłym serwisantom kodu, gdyż inaczej wcześniej czy później ktoś „poprawi” kod dodając „brakujące” funkcje. Najlepszym sposobem udokumentowania takiej decyzji projektowej jest wstawienie nie komentarza, ale odpowiedniego kodu:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   void *operator new[]( size_t n )
      {return ::operator new[](n); }
   void operator delete[]( void *p, size_t )
      { ::operator delete[](p); }
   // ...
};
```

Jeśli powyższe funkcje zostaną zaimplementowane jako funkcje włączane (`inline`), koszt ich wykonania będzie zerowy a ich istnienie powinno odstręczyć wszystkich opiekunów od pisania ich „poprawionych" wersji uświadamiając im, że autor celowo stosuje globalne tablicowe funkcje `new` i `delete`.

Zagadnienie 61: sprawdzanie alokacji pamięci

Niektórych pytań po prostu nie należy zadawać. Jednym z nich jest pytanie, czy konkretny fragment pamięci został zaalokowany.

Warto się przekonać, jak naprawdę wygląda alokowanie pamięci w C++. Oto kod, w którym następuje staranne, każdorazowe sprawdzenie, czy alokowanie pamięci się powiodło:

```
bool error = false;
String **array = new String *[n];
if( array ) {
   for( String **p = array; p < array+n; ++p ) {
      String *tmp = new String;
      if( tmp )
         *p = tmp;
      else {
         error = true;
         break;
      }
   }
}
else
   error = true;
if( error )
   obslugaBledu();
```

Taki sposób kodowania powoduje mnóstwo kłopotów, ale i tak byłby tego wart, gdyby można było w ten sposób wykryć ewentualne problemy z alokacją pamięci. Niestety, nie można. Sam konstruktor klasy `String` może spowodować błąd alokacji pamięci a w takim przypadku nie ma prostej metody propagacji tego błędu poza konstruktor. Można — choć włos się jeży na głowie na samą myśl o tym — spowodować, aby konstruktor `String` tworzył obiekt w pewnym akceptowalnym, błędnym stanie i ustawiał

flagę, którą będzie mógł potem odczytać użytkownik. Nawet zakładając, że istnieje dostęp do klasy `String`, który pozwoli na zrealizowanie takiej implementacji, rozwiązanie to wymusza na autorze kodu i przyszłych serwisantach sprawdzanie jeszcze jednego dodatkowego warunku.

Można też nie przejmować się błędami. Kod sprawdzający błędy rzadko od razu bywa całkowicie poprawny a po pewnym czasie serwisowania zwykle jest już nieprawidłowy. Lepszym rozwiązaniem jest rezygnacja z jakichkolwiek kontroli:

```
String **array = new String *[n];
for( String **p = array; p < array+n; ++p )
   *p = new String;
```

Kod ten jest krótszy, bardziej zrozumiały i poprawny. Standardowe zachowanie `new` polega na wyrzucaniu w razie błędu alokacji pamięci wyjątku `bad_alloc`. Pozwala to zamknąć kod badania błędów w osobnym module, niezależnie od zasadniczej części programu. W ten sposób uzyskuje się czystszy, bardziej zrozumiały i szybciej działający program.

Tak czy inaczej próba sprawdzania wyniku standardowego wywołania `new` nigdy nie poinformuje użytkownika o błędzie, gdyż albo `new` zadziała prawidłowo, albo zostanie wyrzucony wyjątek:

```
int *ip = new int;
if( ip ) { // warunek zawsze jest prawdziwy
   // ...
}
else {
   // ten kod nigdy się nie wykona
}
```

Można też użyć funkcji `operator new` niewyrzucającej wyjątków, która w razie błędu zwraca wskaźnik pusty:

```
int *ip = new (nothrow) int;
if( ip ) { // warunek prawie zawsze prawdziwy
   // ...
}
else {
   // ten kod bardzo rzadko się wykona
}
```

Jednak w tym przypadku powraca się do problemów ze starą semantyką `new` a do tego programista jest zmuszony do korzystania z iście paskudnej składni. Lepiej unikać takiej zgodności ze starszymi wersjami i po prostu projektować programy i potem je kodować z użyciem operatora `new` wyrzucającego wyjątki.

Środowisko opisywanego programu także automatycznie będzie obsługiwało szczególnie paskudny problem związany z błędem alokacji pamięci. Przypomnijmy, że operator `new` używa wywołań dwóch funkcji: najpierw `operator new` alokującej pamięć a potem konstruktora inicjalizującego uzyskaną pamięć:

```
Cos *tp = new Cos( arg );
```

W razie przechwycenia wyjątku `bad_alloc` wiadomo, że wystąpił błąd alokacji pamięci — pytanie brzmi, gdzie on wystąpił. Błąd ten mógł pojawić się podczas alokacji pamięci na obiekt `Cos` lub w konstruktorze obiektu `Cos`. W pierwszym przypadku zwalnianie pamięci jest zbędne, gdyż wskaźnik `tp` nigdy nie wskazywał na nic. W drugim przypadku należałoby zwolnić ze sterty (niezainicjalizowaną) pamięć wskazywaną przez `tp`. Jednak sprawdzenie, z którym z przypadków ma się do czynienia, może być trudne lub niemożliwe.

Na szczęście środowisko opisywanego programu potrafi taką sytuację obsłużyć. Jeśli udało się zaalokować pamięć na obiekt `Cos`, natomiast zawiódł konstruktor wyrzucając wyjątek, zostanie wywołana odpowiednia funkcja `operator delete` zwalniająca pamięć (zobacz zagadnienie numer 62).

Zagadnienie 62: zastąpienie globalnych new i delete

Bardzo rzadko zdarza się, aby wskazane było zastępowanie standardowych, globalnych funkcji `operator new`, `operator delete` i ich tablicowych odpowiedników, mimo że standard na to pozwala. Standardowe funkcje są zwykle bardzo dobrze zoptymalizowane pod kątem różnorodnych zastosowań, zaś wersje podawane przez użytkownika rzadko będą lepsze (choć często rozsądne jest użycie operacji zarządzających pamięcią do udoskonalenia zarządzania pamięcią na potrzeby konkretnej klasy czy hierarchii).

Specjalne wersje funkcji `operator new` i `operator delete` implementujące inne zachowanie niż wersje standardowe zwykle zawierają mniej lub bardziej dokuczliwe błędy, gdyż niezawodność dużej części biblioteki standardowej i innych bibliotek zależy od domyślnej implementacji omawianych funkcji.

Bezpieczniejszym rozwiązaniem jest przeciążenie globalnej funkcji `operator new`, a nie jej zastępowanie. Przykładowo, istnieje potrzeba wypełnienia zaalokowanej pamięci pewnym wzorcem (`pattern` poniżej):

```
void *operator new( size_t n, const string &pat ) {
   char *p = static_cast<char *>(::operator new( n ));
   const char *pattern = pat.c_str();
   if( !pattern[0] )
      pattern = "\0"; // uwaga: dwa znaki NUL
   const char *f = pattern;
   for( int i = 0; i < n; ++i ) {
      if( !*f )
         f = pattern;
      p[i] = *f++;
   }
   return p;
}
```

Pokazana wersja funkcji `operator new` ma parametr `string` kopiowany do nowo zaalokowanej pamięci. Kompilator odróżnia standardową funkcję od naszej dwuargumentowej funkcji będącej przeciążeniem wersji standardowej.

```
string fill( "<garbage>" );
string *string1 = new string( "Hello" ); // wersja standardowa
string *sstring2 =
   new (fill) string( "World!" );        // wersja przeciążona
```

Standard zawiera też definicję przeciążonej funkcji `operator new`, mającej oprócz wymaganego pierwszego parametru typu `size_t` drugi parametr typu `void *`. Implementacja po prostu zwraca drugi parametr (zapis `throw()` to opis wyjątków mówiących, że dana nie będzie przekazywała dalej żadnych wyjątków; można to spokojnie dalej pominąć).

```
void *operator new( size_t, void *p ) throw()
   { return p; }
```

Jest to wersja `new` tworząca obiekt we wskazanym miejscu (w przeciwieństwie jednak do standardowej, jednoparametrowej funkcji `operator new`, próba zastąpienia omawianej wersji funkcji jest niedopuszczalna). Pokazanej funkcji używa się przede wszystkim do wymuszenia na kompilatorze wywołania konstruktora. Przykładowo, w aplikacji zagnieżdżonej w innej można zechcieć zbudować obiekt „rejestru stanu" koniecznie pod wskazanym adresem:

```
class StatusRegister {
   // ...
};
void *regAddr = reinterpret_cast<void *>(0XFE0000);
// ...
// umieść obiekt rejestru pod adresem regAddr
StatusRegister *sr = new (regAddr) StatusRegister;
```

Oczywiście tworzone tą metodą obiekty muszą być kiedyś zniszczone. Jednak, skoro tak naprawdę żadna pamięć nie jest alokowana, konieczne jest zagwarantowanie, że żadna pamięć nie zostanie zwolniona. Przypomnijmy, że operator `delete` najpierw wywołuje destruktor usuwanego obiektu, dopiero potem wywołuje funkcję `operator delete` zwalniającą pamięć. W przypadku obiektu „zaalokowanego" operatorem `new` z adresem trzeba jawnie wywołać destruktor, aby uniknąć próby zwalniania pamięci:

```
sr->~StatusRegister(); // jawne wywołanie destruktora, brak operator delete
```

Operator `new` z adresem i jawne niszczenie są oczywiście przydatne, ale równie oczywiste jest, że w przypadku nieostrożnego użycia są niebezpieczne (w zagadnieniu numer 47 opisaliśmy przykład tego z biblioteki standardowej).

Zauważmy, że o ile możemy przeciążyć funkcję `operator delete`, przeciążone wersje nigdy nie zostaną wywołane przez standardowe wyrażenie `delete`:

```
void *operator new( size_t n, Buffer &buffer ); // przeciążony new
void operator delete( void *p,
   Buffer &buffer );                              // odpowiedni delete
// ...
Thing *thing1 = new Thing;                     // użycie standardowego operatora new
Buffer buf;
Thing *thing2 = new (buf) Thing;               // użycie przeciążonego operatora new
delete thing2;              // nieprawidłowo, powinno być użyte przeciążone delete
delete thing1;              // prawidłowo, użyto standardowego operatora delete
```

Tymczasem, jak w przypadku obiektu tworzonego przez `new` z adresem, programista jest zmuszony do jawnego wywołania destruktora obiektu, a potem do jawnego zwolnienia pamięci byłego obiektu przez wywołanie odpowiedniej funkcji `operator delete`:

```
thing2->~Thing();              // prawidłowo, niszczenie obiektu Thing
operator delete( thing2, buf ); // prawidłowo, użyto przeciążonego delete
```

W praktyce pamięć alokowana przez przeciążoną funkcję globalną `operator new` często jest błędnie zwalniana standardową funkcją globalną `operator delete`. Metodą uniknięcia tego błędu jest upewnienie się, że wszelka pamięć alokowana przeciążoną funkcją `operator new` otrzyma pamięć od standardowej funkcji globalnej `operator new`. Tak właśnie postąpiono powyżej, w przypadku pierwszej implementacji funkcji przeciążonej, dlatego wersja ta działa prawidłowo ze standardową funkcją globalną `operator delete`:

```
string fill( "<garbage>" );
string *string2 = new (fill) string( "World!" );
// ...
delete string2; // działa
```

Przeciążone wersje funkcji globalnej `operator new` powinny albo nie alokować żadnej pamięci, albo powinny być obudową standardowej funkcji.

Często najlepiej jest po prostu unikać robienia czegokolwiek związanego z globalnymi funkcjami zarządzającymi pamięcią, a za to dopracować zarządzanie pamięcią w poszczególnych klasach lub w całej ich hierarchii przez użycie metod `operator new`, `operator delete` i ich tablicowych odpowiedników.

Pod koniec zagadnienia numer 61 napomknęliśmy, że środowisko naszego programu wywoła „odpowiedni" operator `delete` w razie pojawienia się wyjątku w inicjalizacji wyrażenia `new`:

```
Thing *tp = new Thing( arg );
```

Jeśli alokacja pamięci na obiekt `Thing` powiedzie się ale konstruktor `Thing` wyrzuci wyjątek, środowisko naszego systemu wywoła odpowiednią funkcję `operator delete`, która zwolni niezainicjalizowaną pamięć wskazywaną przez `tp`. W pokazanym wyżej przypadku odpowiednią będzie albo globalna `operator delete(void *)`, albo metoda `operator delete` z taką samą sygnaturą. Jednak użycie innej funkcji `operator new` wymusiłoby użycie innej funkcji `operator delete`:

```
Thing *tp = new (buf) Thing( arg );
```

W tym przypadku odpowiednią funkcją jest dwuparametrowa wersja `operator delete` odpowiadająca przeciążonej `operator new` użytej do alokacji obiektu `Thing`, `operator delete( void *, Buffer &)`, i tak właśnie wersja zostanie wywołana.

C++ pozwala na dużą elastyczność przy definiowaniu zachowania funkcji zarządzających pamięcią, ale ceną za tę elastyczność jest złożoność. Standardowe, globalne wersje funkcji `operator new` i `operator delete` wystarczają w większości sytuacji. Bardziej skomplikowanych rozwiązań trzeba się chwytać tylko wtedy, gdy naprawdę nie można się bez nich obejść.

Zagadnienie 63: mylenie zakresu i wywołanie metod new i delete

Metody `operator new` i `operator delete` są wywoływane, kiedy tworzone i niszczone są obiekty zawierającej je klasy. Zakres, w którym jest wykonywana alokacja, nie ma znaczenia:

```
class String {
 public:
   void *operator new( size_t );        // metoda operator new
   void operator delete( void* );       // metoda operator delete
   void *operator new[]( size_t );      // metoda operator new[]
   void operator delete [] ( void * );  // metoda operator delete[]
   String( const char * = "" );
   // ...
};
void f() {
   String *sp = new String( "Sterta" ); // używany jest String::operator new
   int *ip = new int( 12 );             // używany jest ::operator new
   delete ip;                           // używany jest ::operator delete
   delete sp;                           // używany jest String::operator delete
}
```

Znowu, zakres alokacji nie ma znaczenia — to typ, na który alokowana jest pamięć, decyduje o tym, która funkcja zostanie użyta:

```
String::String( const char *s )
   : s_( strcpy( new char[strlen(s)+1], s ) )
   {}
```

Tablica znaków jest alokowana w zakresie klasy `String` ale alokacja wykorzystuje globalny operator tablicowy `new`, a nie takiż operator z klasy `String`: `char` to inny typ niż `String`. Przydatna może być jawna kwalifikacja:

```
String::String( const char *s )
   : s_( strcpy( reinterpret_cast<char *>
      (String::operator new[](strlen(s)+1 )),s ) )
   {}
```

Dobrze byłoby, gdyby można było w celu sięgnięcia do metody `operator new[]` klasy `String` napisać `String::new char[strlen(s)+1]` (zaleca się Czytelnikowi przeanalizowanie tego zagadnienia), ale taka składnia jest niepoprawna (choć możemy użyć zapisu `::new`, aby sięgnąć do globalnej funkcji `operator new` i `operator new[]` oraz `::delete`, aby sięgnąć do globalnej funkcji `operator delete` lub `operator delete[]`).

Zagadnienie 64: wyrzucanie literałów łańcuchowych

Wielu autorów piszących o programowaniu w C++ jako przykład użycia wyjątków prezentuje wyrzucanie literałów łańcuchowych:

```
throw "Niedopełnienie stosu!";
```

Autorzy ci wiedzą, że jest to niedobra praktyka, ale nadal ją stosują jako „przykład pedagogiczny". Niestety, często zapominają uprzedzić swoich czytelników, że stosowanie się do takiego wzorca może spowodować wiele szkód.

Nigdy nie należy jako obiektów wyjątków wyrzucać literałów łańcuchowych. Wynika to stąd, że takie obiekty wyjątków powinny być przechwytywane, zaś przechwytywanie odbywa się na podstawie ich typu, a nie wartości:

```
try {
   // ...
}
catch( const char *msg ) {
   string m( msg );
   if( m == "niedopełnienie stosu" ) // ...
   else if( m == "przekroczono czas połączenia" ) // ...
   else if( m == "naruszenie zasad bezpieczeństwa" ) // ...
   else thorw;
}
```

Praktycznym następstwem wyrzucania i przechwytywania literałów łańcuchowych jest to, że niemalże żadne informacje o wyjątku nie są zakodowane w typie obiektu wyjątku. Taki brak precyzji wymaga, aby fraza `catch` przechwytywała każdy wyjątek i sprawdzała, czy pasuje on do niej. Co gorsza, porównanie wartości jest bardzo podatne na pomyłki, często w trakcie serwisowania przestaje ono działać wskutek różnicy w wielkości liter czy zmiany formatowania „komunikatów". W powyższym przykładzie nigdy nie zauważymy, że wystąpiło niedopełnienie stosu.

Uwagi powyższe w takim samym stopniu dotyczą wyjątków innych predefiniowanych i standardowych typów. Wyrzucanie liczb całkowitych, zmiennoprzecinkowych, ciągów znaków `string` czy (przy wyjątkowo złym humorze) zbiorów wektorów liczb zmiennoprzecinkowych spowoduje podobne problemy. Krótko mówiąc, problem z wyrzucaniem jako wyjątków polega na tym, że kiedy taki wyjątek zostanie przechwycony, nie bardzo wiadomo, co on oznacza, więc nie można adekwatnie zareagować. Z miejsca wyrzucenia wyjątku otrzymuje się komunikat: „Stało się coś bardzo, bardzo złego. Zgadnij, co!". Nie ma żadnego innego wyboru jak przystąpić do zgadywania, co daje bardzo małe szanse na sukces.

Typ wyjątku to abstrakcyjny typ danych reprezentujący wyjątek. Wytyczne do jego projektowania nie różnią się od wytycznych projektowania innych abstrakcyjnych typów danych: należy zidentyfikować i nazwać pojęcie, zdecydować o dopuszczalnych operacjach danego pojęcia i zaimplementować je. Podczas implementacji trzeba wziąć

pod uwagę inicjalizację, kopiowanie i konwersje. Proste. Użycie literału łańcuchowego do reprezentowania wyjątku ma tyle sensu, co użycie literału jako liczby zespolonej. Teoretycznie może to zadziałać, ale praktycznie jest uciążliwe i podatne na błędy.

Jakie pojęcie abstrakcyjne chcemy opisać wyrzucając wyjątek odpowiadający niedopełnieniu stosu? No tak:

```
class StackUnderflow {};
```

Nierzadko zdarza się, że sam typ obiektu wyjątku zawiera wszystkie niezbędne informacje o tym wyjątku, często też typ wyjątku wystarcza do wyboru jawnie zadeklarowanych metod. Tym niemniej możliwość dodania tekstu objaśniającego jest wygodna. Rzadziej w obiekcie wyjątku można zapisać też dalsze informacje o zaistniałym wyjątku:

```
class StackUnderflow {
 public:
   StackUnderflow( const char *msg = "niedopełnienie stosu" );
   virtual ~StackUnderflow();
   virtual const char *what() const;
   // ...
};
```

Jeśli istnieje funkcja zwracająca komunikat tekstowy, powinna być ona wirtualną metodą o nazwie `what`, mającą pokazaną powyżej sygnaturę. W rzeczywistości często dobrym pomysłem jest wyprowadzenie typu wyjątku z jednego ze standardowych typów wyjątków:

```
class StackUnderflow : public std::runtime_error {
 public:
   explicit StackUnderflow( const char *msg = "niedopełnienie stosu" )
      : std::runtime_error( msg ) {}
};
```

Pozwala to przechwytywać wyjątki albo jako `StackUnderflow`, albo jako ogólniejszy typ `runtime_error`, albo jako bardzo ogólny typ `exception` (`exception` to publiczna klasa bazowa klasy `runtime_error`). Często też dobrze jest podać ogólniejszy ale niestandardowy typ wyjątku. Zwykle typ taki służy jako klasa bazowa wszystkich typów wyjątków, które mogą być wyrzucane z danego modułu lub biblioteki:

```
class ContainerFault {
 public:
   virtual ~ContainerFault();
   virtual const char *what() const = 0;
   // ...
};
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "niedopełnienie stosu" )
      : std::runtime_error( msg ) {}
   const char *what() const
      { return std::runtime_error::what(); }
};
```

W końcu trzeba też opisać prawidłowy sposób kopiowania i niszczenia typów wyjątków. W szczególności wyrzucenie wyjątku oznacza, że dopuszczalne musi być tworzenie obiektu przez kopiowanie, gdyż jest ono używane podczas pracy programu do wyrzucenia wyjątku (zobacz zagadnienie numer 65). Skopiowany wyjątek, kiedy będzie już obsłużony, musi zostać zniszczony. Często można pozwolić kompilatorowi napisać za programistę te operacje (zobacz zagadnienie numer 49).

```
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "niedopełnienie stosu" )
      : std::runtime_error( msg ) {}
   // StackUnderflow( const StackUnderflow & );
   // StackUnderflow &operator =( const StackUnderflow & );
   const char *what() const
      { return std::runtime_error::what(); }
};
```

Teraz już użytkownicy naszego stosu mogą wykryć niedopełnienie stosu jako `StackUnderflow` (wiedząc, że używają naszego typu, śledzą taki wyjątek), jako ogólniejszy `ContainerFault` (wiedząc, że używają naszej biblioteki kontenerów, są gotowi do obsłużenia wyjątków związanych z kontenerami), jako `runtime_error` (nie wiedząc nic o naszej bibliotece kontenerów chcą jednak obsłużyć wszelkie typowe błędy wykonania) lub jako `exception` (są gotowi obsłużyć wszelkie standardowe wyjątki).

Zagadnienie 65: nieprawidłowe korzystanie z mechanizmu wyjątków

Kwestie związane ogólnie z obsługą wyjątków i ich architekturą są nadal przedmiotem dyskusji. Jednak wytyczne zdefiniowane na niższym poziomie, dotyczące sposobu wyrzucania wyjątków i ich przechwytywania są równie dobrze rozumiane, co nagminnie naruszane.

Kiedy wykonywane jest wyrażenie wyrzucające wyjątek, mechanizm obsługi wyjątków kopiuje obiekt wyjątku w „bezpieczne", tymczasowe miejsce. Miejsce to w dużym stopniu zależy od używanego systemu ale zawsze gwarantuje się, że będzie ono dostępne aż do chwili obsługi wyjątku. Oznacza to, że tymczasowy obiekt będzie mógł być użyty aż do skończenia wykonywania ostatniej fraz `catch` korzystającej z tego obiektu, nawet jeśli na rzecz tego tymczasowego obiektu wyjątku wywołanych zostanie szereg innych fraz `catch`. Jest to ważne, gdyż — mówiąc najprościej — po wyrzuceniu wyjątku nagle wszystko staje się możliwe. Tymczasowy obiekt wyjątku jest cichym okiem cyklonu obsługi wyjątków.

Oto dlaczego nie należy wyrzucać wskaźników:

```
throw new StackUnderflow( "stos operatora" );
```

Adres obiektu `StackUnderflow` na stercie jest kopiowany w bezpieczne miejsce ale wskazywana pamięć, umieszczona na stercie, nie jest chroniona. Takie rozwiązanie dopuszcza też sytuację, że wskaźnik może odwoływać się do pamięci znajdującej się na stosie roboczym:

```
StackUnderflow e( "stos parametrów" );
throw &e;
```

W tym przypadku wskazywany obiekt wyjątku (należy pamiętać, że wyrzucany jest wskaźnik, a nie wskazywana przezeń wartość) odnosi się do pamięci, która może w chwili przechwycenia wyjątku być już niedostępna (a przy okazji: kiedy wyrzucamy literał łańcuchowy, cała tablica znaków jest kopiowana w tymczasowe miejsce, a nie adres pierwszego znaku; jest to jednak mało przydatna informacja, gdyż nigdy nie należy wyrzucać literałów łańcuchowych — zobacz zagadnienie numer 64). Co więcej, wskaźnik może być pusty (`null`). Komu takie komplikacje są potrzebne? Nie należy zatem wyrzucać wskaźników, lecz obiekty:

```
StackUnderflow e( "stos parametrów| );
throw e;
```

Obiekt wyjątku jest natychmiast kopiowany przez mechanizm obsługi wyjątków do tymczasowej lokalizacji, więc deklaracja `e` jest zbędna. Zwyczajowo wyrzuca się obiekty anonimowe:

```
throw StackUnderflow( "stos parametrów" );
```

Użycie obiektów tymczasowych jasno mówi, że obiekt `StackUnderflow` jest używany jedynie jako obiekt wyjątku, gdyż jego cykl życia jest ograniczony do wyrażenia wyrzucającego go. O ile jawnie zadeklarowana zmienna `e` zostanie także zniszczona podczas wykonywania instrukcji wyrzucającej wyjątek, to jej zakres sięga końca bloku zawierającego jej deklarację i w tym zakresie jest dostępna. Użycie anonimowych obiektów tymczasowych pomaga także ukrócić niektóre „tfurcze” próby obsługi wyjątków:

```
static StackUnderflow e( "stos parametrów" );
extern StackUnderflow *argstackerr;
argstackerr = &e;
throw e;
```

W tym przypadku nasz błyskotliwy programista postanowił zapamiętać adres obiektu wyjątku na później, prawdopodobnie w jakiejś frazie `catch`. Niestety, wskaźnik `argstackerr` nie wskazuje obiektu wyjątku (jest to tymczasowy obiekt w celowo nieujawnianym miejscu), ale wskazuje zniszczony już obiekt użyty do inicjalizacji tego wskaźnika. Kod obsługi wyjątków nie jest najlepszym miejscem na umieszczanie dobrze ukrytych błędów; powinien być to kod tak prosty, jak to możliwe.

Jak najlepiej przechwytywać wyjątki? Na pewno nie przez wartość:

```
try {
   // ...
}
catch( ContainerFault fault ) {
   // ...
}
```

Warto zastanowić się, co by się stało, gdyby taka fraza `catch` faktycznie przechwyciła wyrzucony gdzieś obiekt `StackUnderflow`. Odpowiedź brzmi: szatkowanie. Obiekt `StackUnderflow` jest przykładem `ContainerFault`, można by zainicjalizować `fault` wyrzuconym obiektem wyjątku, ale nastąpiłoby poszatkowanie wszystkich danych i metod klasy pochodnej (zobacz zagadnienie numer 30).

W tym konkretnym przypadku nie byłoby jednak problemów z szatkowaniem, gdyż `ContainerFault` — jako faktyczna klasa bazowa — jest klasą abstrakcyjną (zobacz zagadnienie numer 93). Wobec tego fraza `catch` jest błędna. Nie można przechwycić przez wartość obiektu wyjątku jako obiektu `ContainerFault`.

Przechwytywanie przez wartość naraża programistę na jeszcze inne ukryte problemy:

```
Catch( StackUnderflow fault ) {
   // częściowo uzdrawiamy sytuację...
   fault.modifyState(); // moja wina
   throw;               // ponownie wyrzucamy aktualny wyjątek
}
```

Nierzadko zdarza się, że jakaś fraza `catch` częściowo naprawia sytuację, zapisuje wynik tej naprawy w obiekcie wyjątku i ponownie wyrzuca tenże obiekt wyjątku, aby przetwarzanie było kontynuowane. Niestety, w pokazanym kodzie tak się nie dzieje. Fraza `catch` przechwytuje wyjątek, wykonuje częściową naprawę, zapisuje stan w lokalnej kopii obiektu wyjątku i ponownie wyrzuca (niezmieniony) obiekt wyjątku.

Dla uproszczenia i dla uniknięcia opisanych problemów zawsze wyrzuca się anonimowe obiekty tymczasowe a przechwytuje się je przez referencję.

Należy zachować ostrożność, aby nie doszło do ponownego przekazania kopii z opisem problemów do tej samej procedury obsługi. Zdarza się tak wtedy, gdy nowy wyjątek jest wyrzucany z procedury obsługi, podczas gdy ponownie powinien być wyrzucany wyjątek już istniejący:

```
catch( ContainerFault &fault ) {
   // robimy częściową naprawę...
   if( warunek )
      throw;                // ponowne wyrzucenie
   else {
      ContainerFault myFault( fault );
      myFault.modifyState(); // nadal moja wina
      throw myFault;         // nowy obiekt wyjątku
   }
}
```

Tym razem zapisane zmiany nie zostaną utracone, ale utracony zostanie pierwotny typ wyjątku. Załóżmy, że pierwotnym typem było `StackUnderflow`. Kiedy obiekt ten jest przechwytywany jako referencja `ContainerFault`, dynamicznym typem obiektu wyjątku nadal jest `StackUnderflow`, więc ponowne wyrzucenie tego obiektu pozwala znów przechwycić zarówno we frazie `catch` obiektu `StackUnderflow`, jak i `ContainerFault`. Jednak w chwili wyrzucania nowego obiektu wyjątku `myFault` jest on typu `ContainerFault` i nie może być przechwycony przez frazę `catch` obiektu `StackUnderflow`. Zwykle lepiej jest ponownie wyrzucać istniejący obiekt wyjątku, a nie obsługiwać pierwotny wyjątek. Zamiast tego lepiej jest wyrzucać nowy:

```
catch( ContainerFault &fault ) {
   // robimy częściową naprawę...
   if( !warunek )
      fault.modifyState();
   throw; // ponowne wyrzucenie
}
```

Na szczęście klasa bazowa `ContainerFault` jest abstrakcyjna, więc tutaj błąd nie może wystąpić. W ogóle klasy bazowe powinny być abstrakcyjne. Oczywiście, rada taka nie jest słuszna, jeśli konieczne jest wyrzucenie całkiem innego typu wyjątku:

```
catch( ContainerFault &fault ) {
   // robimy częściową naprawę...
   if( out_of_memory )
      throw bad_alloc(); // wyrzucamy nowy wyjątek
   fault.modifyState();
   throw;                // ponowne wyrzucenie
}
```

Inny typowy problem wiąże się z kolejnością fraz `catch`. Frazy te są sprawdzane kolejno (jak warunki `if-elseif`, a nie jak instrukcja `switch`), więc typy wyjątków należy porządkować od najbardziej szczegółowych do najogólniejszych. Jeśli chodzi o typy wyjątków niedających się tak uporządkować, trzeba po prostu logicznie zastanowić się, jaka kolejność będzie najlepsza:

```
catch( ContainerFault &fault ) {
   // robimy częściową naprawę...
   fault.modifyState(); // nie moja wina
   throw;
}
catch( StackUnderflow &fault ) {
   // ...
}
catch( exception & ) {
   // ...
}
```

Pokazana powyżej kolejność fraz `catch` nie pozwoli nigdy przechwycić wyjątku `StackUnderflow`, gdyż przed odpowiadającą mu frazą pojawia się fraza ogólniejszego wyjątku `ContainerFault`.

Mechanizm obsługi wyjątków łatwo może prowadzić do dużej złożoności kodu ale niekoniecznie trzeba iść tą drogą. Wyrzucając i przechwytując wyjątki należy zachować prostotę.

Zagadnienie 66: nadużywanie adresów lokalnych

Nie należy zwracać wskaźników do zmiennych lokalnych. Większość kompilatorów ostrzega przed taką sytuacją i ostrzeżenie to należy traktować poważnie.

Znikanie ramek stosu

Jeśli zmienna jest zmienną automatyczną, używana przez nią pamięć znika w chwili powrotu z procedury:

```
char *newLabel1() {
   static int nrEt = 0;
   char bufor[16]; // zobacz zagadnienie numer 2
   sprintf( bufor, "etykieta%d", nrEt++ );
   return bufor;
}
```

Funkcja powyższa ma tę nieprzyjemną cechę, że czasem działa. Po wyjściu z tej funkcji ramka stosu jej odpowiadająca jest usuwana a związana z nią pamięć jest zwalniana (w tym także `bufor`), aby mogła być wykorzystana przez następną funkcję. Jeśli jednak otrzymana z `newLabel1` wartość zostanie przepisana przed wywołaniem następnej funkcji, uzyskany wskaźnik, choć już niepoprawny, może wskazywać jednak pożądaną wartość:

```
char *uniqueLab = newLabel1();
char mybuf[16], *pmybuf = mybuf;
while( *pmybuf++ = *uniqueLab++ );
```

Nie jest to kod, który da się zbyt długo serwisować. Serwisant może zdecydować się na alokację bufora na stercie:

```
char *pmybuf = new char[16];
```

Serwisant może też zrezygnować z ręcznego kopiowania bufora:

```
strcpy( pmybuf, uniqueLab );
```

Serwisant może postanowić o użyciu bardziej abstrakcyjnego typu danych niż bufor znakowy:

```
std::string mybuf( uniqueLab );
```

Każda z powyższych modyfikacji spowoduje, że zmodyfikowana będzie pamięć lokalna wskazywana przez `uniqueLab`.

Interferencje danych statycznych

Jeśli zmienna jest statyczna, późniejsze wywołanie tej samej funkcji wpłynie na wynik wywołania wcześniejszego:

```
char *newLabel2() {
   static int nrEt = 0;
   static char bufor[16];
   sprintf( bufor, "etykieta%d", nrEt++ );
   return bufor;
}
```

Pamięć na bufor jest dostępna także po wyjściu z funkcji ale kolejne użycie tej samej funkcji może zmienić zawartość tej pamięci:

```
// przypadek 1
cout << "pierwsza: " << newLabel2() << ' ';
cout << "druga: " << newLabel2() << endl;

// przypadek 2
cout << "pierwsza: " << newLabel2() << ' '
     << "druga: " << newLabel2() << endl;
```

W pierwszym przypadku zostaną pokazane dwie różne etykiety. W drugim — prawdopodobnie (choć niekoniecznie) ta sama etykieta zostanie pokazana dwa razy. Prawdopodobnie ktoś, kto jest świadom niezwykłej implementacji funkcji `newLabel2`, napisze tak jak w przypadku 1 — rozdzielając wywołania etykiet na odrębne instrukcje. Ktoś, kto będzie używał tej funkcji w przyszłości nie znając błędnej implementacji `newLabel2` może wywołania połączyć powodując przez to błąd. Co gorsze, połączone wywołanie może też działać prawidłowo, ale zachowywać się nieprzewidywalnie w przyszłości (zobacz zagadnienie numer 14).

Problemy z idiomami

Trzeba pamiętać o jeszcze jednym niebezpieczeństwie. Wiadomo, że użytkownicy funkcji zwykle nie mają dostępu do jej implementacji i muszą opierać się jedynie na jej deklaracji. Pomocą mogą być tu komentarze (zobacz zagadnienie numer 1), ale ważne jest takie projektowanie funkcji, aby ułatwiać prawidłowe jej wykorzystywanie.

Należy unikać zwracania referencji odwołującej się do pamięci zaalokowanej w danej funkcji. Użytkownicy będą notorycznie zapominać o zwolnieniu tej pamięci powodując przez to wycieki pamięci:

```
int &f()
   { return *new int( 5 ); }
// ...
int i = f(); // wyciek pamięci!
```

Prawidłowy kod musi przekształcać referencję na adres lub kopiować wynik i zwalniać pamięć. Byle nie na mojej zmianie, kolego:

```
int *ip = &f(); // jedna koszmarna metoda
int &tmp = f(); // kolejna
int i = tmp;
delete &tmp;
```

Wyjątkowo źle sprawdza się to w przypadku przeciążonych funkcji operatorów:

```
Zespolona &operator +( const Zespolona &a, const Zespolona &b )
   { return *new Zespolona( a.re+b.re, a.im+b.im ); }
// ...
Zespolona a, b, c;
a = b + c + a + b; // mnóstwo wycieków!
```

Należy zwracać wskaźnik do odpowiedniego miejsca w pamięci albo nie alokować pamięci i zwracać wynik przez wartość:

```
int *f() { return new int(5); }
Zespolona operator +( Zespolona a, Zespolona b )
   { return Zespolona( a.re+b.re, a.im+b.im ); }
```

Użytkownicy funkcji zwracającej wskaźnik spodziewają się, że mogą być odpowiedzialni za ewentualne zwalniania pamięci wskazywanej takim wskaźnikiem i zwykle starają się sprawdzić, czy faktycznie są za to odpowiedzialni (zwykle oznacza to przeczytanie komentarzy). Użytkownicy funkcji zwracającej referencję rzadko czują się odpowiedzialni za zwolnienie pamięci.

Problemy z zasięgiem lokalnym

Problemy, jakie napotykamy w związku z czasem życia zmiennych lokalnych mogą pojawiać się nie tylko na granicy między funkcjami, ale także w przypadku zagnieżdżonych zakresów w pojedynczych funkcjach:

```
void localScope( int x ) {
   char *cp = 0;
   if( x ) {
      char buf1[] = "asdf";
      cp = buf1;     // kiepski pomysł!
      char buf2[] = "qwerty";
      char *cp1 = buf2;
      // ...
   }
   if( x-1 ) {
      char *cp2 = 0; // zachodzi na buf1?
      // ...
   }
   if( cp )
      printf( cp );  // być może błąd...
}
```

Kompilatory są bardzo elastyczne, jeśli chodzi o sposób układania pamięci na zmienne lokalne. W zależności od używanego systemu i opcji kompilatora pamięć na `buf1` i `cp2` może się pokryć lub nie. Jest to dopuszczalne, gdyż `buf1` i `cp2` mają rozłączne zakresy i rozłączne okresy życia. W przypadku pokrycia wartość `buf1` zostanie uszkodzona i zachowanie `printf` może się zmienić (prawdopodobnie nic nie zostanie pokazane). Aby uzyskać przenośność, nie należy opierać się na konkretnym sposobie działania ramek stosu.

Korekta przez static

Czasami programista staje wobec poważnego błędu i okazuje się, że zastosowanie specyfikatora `static` powoduje, że wszystko nagle zaczyna działać poprawnie:

```
// ...
char buf[MAX];
long count = 0;
// ...
int i = 0;
while( i++ <= MAX )
   if( buf[i] == '\0' ) {
      buf[i] = '*';
      ++count;
```

```
    }
assert( count <= i );
// ...
```

Kod ten zawiera kiepsko napisaną pętlę, która czasami pisze za końcem tablicy `buf` powodując, że asercja zawodzi. Szukając na ślepo przyczyn błędu programista może zadeklarować `count` jako lokalną zmienną statyczną i kod nagle zacznie działać:

```
char buf[MAX];
static long count = 0;
// ...
count = 0;
int i = 0;
while( i++ <= MAX )
   if( buf[i] == '\0' ) {
      buf[i] = '*';
      ++count;
   }
assert( count <= i );
```

Wielu programistów nie będzie chciało narażać na szwank swojego szczęścia i zostawi swój kod w takiej postaci. Niestety, problem nie znikł: po prostu został przeniesiony. Siedzi sobie spokojnie i czeka, gotów do uderzenia w odpowiedniej chwili.

Zmiana lokalnej zmiennej `count` na statyczną spowodowało przeniesienie jej poza ramkę stosu funkcji do całkiem innego obszaru pamięci, gdzie umieszczane są obiekty lokalne. Z powodu tego przeniesienia zmienna ta już nie jest nadpisywana. Jednak nie tylko zmienna `count` jest przedmiotem problemów, które określiliśmy wcześniej jako „interferencja zmiennych statycznych". Dotyczy to każdej innej zmiennej lokalnej, ewentualnie zmiennych, które powstaną — wszystkie one mogą być nadpisywane. Prawidłowym rozwiązaniem jest oczywiście nie ukrywanie błędu, lecz jego usunięcie:

```
char buf[MAX];
long count = 0;
// ...
int i = 0;
for( int i = 1; i < MAX; ++i )
   if( buf[i] == '\0' ) {
      buf[i] = '*';
      ++count;
   }
// ...
```

Zagadnienie 67: błąd pozyskania zasobu bierze się z inicjalizacji

Wstyd, jak wielu programistów C++ nie docenia cudownej symetrii konstruktorów i destruktorów. W większości przypadków chodzi tu o programistów, którzy korzystali z języków, które trzymały ich z daleka od kaprysów wskaźników i problemów

zarządzania pamięcią. Bezpieczeństwo i nieświadomość. Błoga nieświadomość. Programowanie zgodnie z metodą wytyczoną przez projektanta języka wymaga programowania w jeden, z góry określony sposób. Właściwy sposób.

Na szczęście C++ ma więcej względów dla praktyków i jest znacznie elastyczniejszy pod względem możliwych sposobów stosowania języka. Nie chodzi o to, iżby nie było w C++ ogólnych zasad i idiomów (zobacz zagadnienie numer 10). Jednym z najważniejszych idiomów jest to, że „pozyskiwanie zasobów bierze się z inicjalizacji". Jest to dość enigmatyczne stwierdzenie ale jego konsekwencje pozwalają prosto i elastycznie wiązać zasoby z pamięcią i zarządzać jednymi i drugimi w przewidywalny sposób.

Operacje tworzenia i niszczenia stanowią swoje wzajemne odbicia lustrzane. Podczas tworzenia obiektu klasy kolejność inicjalizacji zawsze jest taka sama: najpierw podobiekty wirtualnej klasy bazowej (zgodnie ze standardem, „w takiej kolejności, w jakiej pojawiają się przy analizie w głąb od lewej do prawej skierowanego grafu acyklicznego klas bazowych"), potem bezpośrednie klasy bazowe w kolejności ich występowania na liście klas bazowych w definicji omawianej klasy, potem niestatyczne pola danych w kolejności ich deklaracji, potem treść konstruktora. Niszczenie odbywa się w kolejności odwrotnej: treść destruktora, pola w odwrotnej kolejności deklaracji, bezpośrednie klasy bazowe w odwrotnej kolejności deklaracji, wirtualne klasy bazowe. Dobrze jest wyobrazić sobie budowanie jako proces wkładania pewnego ciągu na stos, a niszczenia — jako zdejmowanie ze stosu, oczywiście w odwrotnej kolejności. Symetria budowania i niszczenia jest uważana za tak ważną, że wszystkie konstruktory klas robią swoje inicjalizacje w takiej samej kolejności, nawet jeśli lista inicjalizacyjna pól jest zapisana w innej kolejności (zobacz zagadnienie numer 52).

Efektem ubocznym lub wynikiem inicjalizacji jest to, że konstruktor zbiera zasoby w miarę tworzenia obiektu. Często kolejność pobierania tych zasobów jest istotna (przykładowo, przed zapisem bazy danych trzeba ją zablokować. Trzeba pobrać uchwyt pliku przed pisaniem do tego pliku) i zwykle destruktor zwalnia zasoby w kolejności odwrotnej do kolejności ich pobierania. Może istnieć wiele konstruktorów, ale istnienie zawsze dokładnie jednego destruktora oznacza, że wszystkie konstruktory muszą wykonywać inicjalizacje swoich składników w takiej samej kolejności.

Tak na marginesie warto wspomnieć, że nie zawsze tak było. Krótko po powstaniu języka C++ kolejność inicjalizacji w konstruktorach nie była ustalona, co powodowało wiele trudności w przypadku złożonych projektów. Tak jak większość reguł języka C++, tak i ta jest wynikiem starannego projektu i praktycznego doświadczenia.

Symetria tworzenia i niszczenia zachodzi nawet wtedy, gdy przechodzimy od struktury obiektu do zastosowań wielu obiektów. Warto rozważyć prostą klasę śladu:

►►*zagadnienie67/trace.h*

```
class Trace {
  public:
    Trace( const char *msg )
        : m_( msg ) { std::cout << "Wejście do " << m_ << endl; }
    ~Trace()
        { std::cout << "Wyjście z " << m_ << endl; }
```

```
  private:
    const char *m_;
};
```

Taka klasa śladu jest być może nieco zbyt prosta, gdyż zakładamy, że jej inicjalizator jest poprawny i będzie istniał przynajmniej tak długo, jak długo istniał będzie obiekt `Trace`, ale dla naszych celów to wystarcza. Obiekt `Trace` pokazuje komunikat przy każdym jego tworzeniu i ponownie przy niszczeniu, więc za jego pomocą można śledzić przebieg wykonania programu:

►►*zagadnienie67/trace.cpp*

```
Trace a( "obszaru globalnego" );

void loopy( int cond1, int cond2 ) {
    Trace b( "treści funkcji" );
it: Trace c( "dalszej części treści" );
    if( cond1 == cond2 )
        return;
    if( cond1-1 ) {
        Trace d( "if" );
        static Trace stat( "lokalnych statycznych" );
        while( --cond1 ) {
            Trace e( "pętli" );
            if( cond1 == cond2 )
                goto it;
        }
        Trace f( "po pętli" );
    }
    Trace g( "po if" );
}
```

Wywołanie funkcji `loopy` z parametrami `4` i `2` da następujące wyniki:

```
Wchodzę do obszaru globalnego
Wchodzę do treści funkcji
Wchodzę do dalszej części treści
Wchodzę do if
Wchodzę do lokalnych statycznych
Wchodzę do pętli
Wychodzę z pętli
Wchodzę do pętli
Wychodzę z pętli
Wychodzę z if
Wychodzę z dalszej części treści
Wchodzę do dalszej części treści
Wychodzę z dalszej części treści
Wychodzę z treści funkcji
Wychodzę z lokalnych statycznych
Wychodzę z obszaru globalnego
```

Z komunikatów jasno wynika, jak czas życie obiektu `Trace` jest związany z aktualnym zakresem wykonania. W szczególności warto zwrócić uwagę na wpływ instrukcji `goto` i `return` na czas życia aktywnych obiektów `Trace`. Każda z tych gałęzi stanowi przykład praktyki kodowania, ale są to konstrukcje podobne do tych, jakie pojawiać się będą w większości praktycznie spotykanego oprogramowania.

```
void zrobDB() {
   blokujDB();
   // zrób z bazą danych co trzeba...
   odblokujDB();
}
```

W powyższym kodzie przed skorzystaniem z bazy danych najpierw ją zablokowano, a po wykonaniu zaplanowanych czynności odblokowano ją. Niestety, tego typu ostrożność załamuje się podczas serwisowania kodu, szczególnie, jeśli fragment kodu między blokowaniem a odblokowaniem jest długi:

```
void zrobDB() {
   blokujDB();
   // ...
   if( tak_mi_sie_podoba )
      return;
   // ...
   odblokujDB();
}
```

Teraz błąd powstaje za każdym razem, kiedy tak się podoba funkcji `zrobDB`: baza danych pozostanie zablokowana a to bez wątpienia przyczyni się do wielu kłopotów w innych miejscach systemu. Tak naprawdę nawet oryginalny kod nie był napisany poprawnie, gdyż wyrzucenie wyjątku po zablokowaniu bazy, ale przed jej zablokowaniem, da podobny efekt: baza będzie zablokowana.

Można by próbować zaradzić temu jawnie uwzględniając wyjątki i utrudniając lekturę serwisantom:

```
void zrobDB() {
   zablokujDB();
   try {
      // zrób z bazą danych co trzeba...
   }
   catch( ... ) {
      odblokujDB();
      throw;
   }
   odblokujDB();
}
```

Takie rozwiązanie jest przegadane, mało eleganckie, trudne do serwisowania i spowoduje, że jego twórca będzie postrzegany jako przedstawiciel Wydziału Wydziału Nadmiarowości. Prawidłowo napisany, bezpieczny z uwagi na wyjątki kod zwykle zawiera kilka bloków `try`. Zamiast tego można skorzystać z zasady pozyskiwania zasobów w ramach inicjalizacji:

```
class DBBlokuj {
 public:
   DBBlokuj() { blokujDB(); }
   ~DBBlokuj() { odblokujDB(); }
};

void zrobDB() {
   DBBlokuj lock;
   // tutaj robimy z bazą danych co trzeba...
}
```

Tworzenie obiektu `DBBlokada` powoduje, że jest pozyskiwana blokada bazy danych. Kiedy obiekt `DBBlokada` wychodzi poza zakres z jakiegokolwiek powodu, jego destruktor odzyska zasób i odblokuje bazę danych. Idiom taki jest tak często używany w C++, że często w ogóle się go nie zauważa. Jednak za każdym razem, kiedy używa się standardowego `string`, `vector`, `list` lub innego typu, korzysta się z tego właśnie idiomu.

Na marginesie warto ostrzec przed dwoma typowymi problemami związanymi z użyciem klas uchwytów zasobów takich, jak `DBBlokada`:

```
void zrobDB() {
   DBBlokada blokada1; // poprawnie
   DBBlokada blokada2(); // ups!
   DBBlokada(); // ups!
   // tutaj robimy z bazą danych co trzeba...
}
```

Deklaracja zmiennej `blokada1` jest prawidłowa: mamy obiekt `DBBlokada`, który pojawia się w zakresie tuż przed średnikiem w deklaracji i wychodzi poza zakres pod koniec bloku zawierającego jego deklarację (tutaj jest to koniec funkcji). Deklaracja `blokada2` jest deklaracją funkcji bezparametrowej, zwracającej `DBBlokada` (zobacz zagadnienie numer 19). Nie jest to błąd, ale prawdopodobnie nie o to chodziło programiście, gdyż nie zachodzi ani blokowanie, ani odblokowywanie.

Następny wiersz to deklaracja tworząca anonimowy obiekt tymczasowy `DBBlokada`. To faktycznie zablokuje bazę danych, ale z uwagi na to, że anonimowy obiekt tymczasowy wychodzi poza zakres zaraz za zawierającym go wyrażeniem (tuż przed średnikiem), baza danych natychmiast zostanie odblokowana. Prawdopodobnie nie o to chodziło.

Standardowy szablon `auto_ptr` jest przydatnym uchwytem zasobu ogólnego przeznaczenia obiektu alokowanego na stercie. Zobacz zagadnienia numer 10 i 68.

Zagadnienie 68: nieprawidłowe użycie auto_ptr

Standardowy szablon `auto_ptr` jest prostym i wygodnym uchwytem zasobu, mającym niezwykłą semantykę kopiowania (zobacz zagadnienie numer 10). Większość przypadków użycia `auto_ptr` nie wymaga komentarza:

```
template <typename T>
void print( Container<T> &c ) {
   auto_ptr< Iter<T> > i(c.genIter() );
   for( i->reset(); !i->done(); i->next() ) {
      cout << i->get() << endl;
      examine( c );
   }
   // niejawne czyszczenie...
}
```

Szablonu auto_ptr często używa się, aby zapewnić zwalnianie pamięci i zasobów w formie obiektów alokowanych na stercie, które są zwalniane, kiedy wskazujący je wskaźnik wychodzi poza zakres (bardziej skomplikowaną analizę hierarchii można zobaczyć w zagadnieniu 90). Powyżej zakładamy, że pamięć na Iter<T> zwracana przez genIter jest alokowana ze sterty. Wobec tego auto_ptr< Iter<T> > wywoła operator delete, aby odzyskać pamięć obiektu w przypadku, kiedy auto_ptr wyjdzie poza zakres.

Jednak z użyciem auto_ptr wiążą się dwa typowe błędy. Pierwszy to założenie, że auto_ptr może odnosić się do tablicy.

```
void calc( double src[], int len ) {
   double *tmp = new double[len];
   // ...
   delete [] tmp;
}
```

Funkcja calc jest podatna na błędy, gdyż zaalokowana tablica tmp nie zostanie odzyskana, jeśli podczas wykonania funkcji pojawi się wyjątek lub jeśli niedbały serwisant spowoduje wcześniejsze wyjście z funkcji. Potrzebny jest uchwyt zasobu, jest nim standardowo auto_ptr:

```
void calc( double src[], int len ) {
   auto_ptr<double> tmp( new double[len] );
   // ...
}
```

Jednak auto_ptr to standardowy uchwyt zasobu pojedynczego obiektu, a nie tablicy obiektów. Kiedy tmp wychodzi poza zakres i aktywowany jest jego destruktor, do tablicy wartości double zaalokowanej tablicowym operatorem new (zobacz zagadnienie numer 60) stosowane jest nietablicowe zwalnianie pamięci. Wynika to stąd, że, niestety, kompilator nie potrafi odróżnić wskaźnika tablicy od wskaźnika pojedynczego obiektu. Szczęśliwym zbiegiem okoliczności kod ten może czasem działać w niektórych systemach, zaś problem może zostać wykryty jedynie przy przenoszeniu oprogramowania na inny system lub podczas aktualizacji nowej wersji systemu istniejącego.

Lepszym rozwiązaniem jest użycie do standardowej struktury vector zamiast array. Standardowa struktura vector to w zasadzie uchwyt zasobu tablicy, rodzaj „autotablicy" wzbogacony o szereg dodatkowych możliwości. Jednocześnie zapewne dobrym pomysłem jest pozbycie się prymitywnego i niebezpiecznego używania jako parametrów formalnych wskaźników przysłaniających tablicę:

```
void calc( vector<double> &src ) {
   vector<double> tmp<srcsize();
   // ...
}
```

Innym typowym błędem jest użycie auto_ptr jako typu elementu z kontenera STL. Kontenery STL nie narzucają zbyt dużych wymagań na swoje elementy, ale wymagają klasycznej semantyki kopiowania.

W standardzie auto_ptr zdefiniowano w taki sposób, że niepoprawne jest ukonkretnienie kontenera STL typem elementu auto_ptr. Próba spowoduje wyświetlenie błędu kompilacji (prawdopodobnie dość mocno ukrytego). Jednak wiele istniejących implementacji nadal nie nadąża za standardem.

W powszechnych, przestarzałych implementacjach `auto_ptr` jego semantyka kopii doskonale odpowiada użyciu go jako typu elementu kontenera. Oznacza to, że do chwili otrzymania innej lub nowszej wersji biblioteki standardowej, taki kod nie będzie się kompilował. Bardzo to niedogodne, ale łatwo to naprawić.

Gorsza sytuacja zachodzi, kiedy implementacja `auto_ptr` nie w pełni jest zgodna ze standardem, więc można użyć jej do ukonkretnienia kontenera STL, ale semantyka kopiowania nie jest tym, czego wymaga STL. Jak opisano w zagadnieniu numer 10, kopiowanie `auto_ptr` przenosi kontrolę wskazywanego obiektu i ustawia źródło kopiowania na `null`:

```
auto_ptr<Pracownik> e1( new Godzinowy );
auto_ptr<Pracownik> e2( e1 ); // e1 jest null
e1 = e2;                      // e2 jest null
```

Opisana właściwość jest w pewnych sytuacjach całkiem przydatna, ale nie o nią chodzi w przypadku elementów będących kontenerami STL:

```
vector< auto_ptr<Pracownik> > listaplac;
// ...
list< auto_ptr<Pracownik> > temp;
copy( listaplac.begin(), listaplac(end), wstawianie_wstecz(temp) );
```

W niektórych systemach taki kod da się skompilować i uruchomić, ale zwykle nie warto się tym zajmować. Struktura `vector` obiektów `Pracownik` zostanie skopiowana na listę, ale po zakończeniu kopiowania `vector` będzie zawierała wszystkie puste wskaźniki (`null`)!

Należy unikać używania `auto_ptr` jako kontenera STL, nawet jeśli używany aktualnie system na to pozwala.