BJARNE STROUSTRUP

Zaczerpnij wiedzę o C++ od samego twórcy tego języka!

Programowanie

Teoria i praktyka z wykorzystaniem C++

Wydanie III

Addison Wesley

Jak zacząć pracę w zintegrowanym środowisku programistycznym?
Jak profesjonalnie tworzyć programy użytkowe?
Jak korzystać z biblioteki graficznego interfejsu użytkownika?

Helion

Tytuł oryginału: Programming: Principles and Practice Using C++ (2nd Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-6312-0

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pcppt3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pcppt3.zip

Printed in Poland.

Spis treści

Rozdział 4. Wykonywanie obliczeń 103

Rozdział 5. Błędy 139

Rozdział 26. Testowanie 879

Rozdział 27. Język C 907

19
Wektory, szablony i wyjątki

„Sukces nigdy nie jest ostateczny."

— Winston Churchill

W tym rozdziale zostaną dokończone projekt i implementacja najczęściej używanego i najbardziej przydatnego kontenera STL — `vector`. Pokażemy, jak implementuje się kontenery o zmiennej liczbie elementów, jak tworzyć kontenery, w których typ elementów jest parametrem oraz co robić z błędami zakresu. Jak zwykle, opisywane techniki mają ogólne zastosowanie, a więc nie są ograniczone tylko do typu `vector`, a nawet tylko do implementacji kontenerów. Przedstawimy techniki bezpiecznego posługiwania się różnymi ilościami danych różnych typów. Dodatkowo dorzucimy pewną dozę realizmu i parę lekcji projektowania. Opisywane tu techniki wymagają znajomości szablonów i wyjątków. Dlatego napiszemy, jak definiuje się szablony, oraz opiszemy podstawowe techniki zarządzania zasobami, które mają kluczowe znaczenie dla właściwego posługiwania się wyjątkami.

19.1. Analiza problemów

Pod koniec rozdziału 18. doszliśmy z naszym wektorem do punktu, w którym możemy:

- Tworzyć wektory zmiennoprzecinkowych elementów o podwójnej precyzji (obiektów klasy `vector`) z dowolną liczbą elementów.
- Kopiować wektory za pomocą przypisania i inicjalizacji.
- Polegać na wektorach w sprawach poprawnego zwalniania pamięci, gdy wychodzą poza zakres dostępności.
- Uzyskiwać dostęp do elementów wektorów przy użyciu konwencjonalnej notacji indeksowej (zarówno po lewej, jak i prawej stronie przypisania).

Wszystkie te osiągnięcia są wartościowe, ale aby osiągnąć taki poziom zaawansowania, którego chcemy (pamiętając o typie `vector` z biblioteki standardowej), musimy poradzić sobie jeszcze z trzema problemami:

- Jak zmienić rozmiar wektora (zmienić liczbę elementów)?
- Jak wykrywać i raportować błędy dostępu do elementów spoza zakresu?
- Jak określić typ elementów wektora w postaci argumentu?

Jak na przykład zdefiniować nasz typ `vector`, aby było dozwolone pisanie poniższego kodu:

```
vector<double> vd;              // elementy typu double
for (double d; cin>>d; )        // Powiększa vd, aby zmieściły się wszystkie elementy

vector<char> vc(100);           // elementy typu char
int n;
cin>>n;
vc.resize(n);                   // Sprawia, że vc będzie zawierać n elementów
```

Oczywiście dobrze mieć do dyspozycji taki typ `vector`, który pozwala na to wszystko. Lecz dlaczego jest to ważne z programistycznego punktu widzenia? Co jest w tym ciekawego dla kogoś, kto zbiera wiedzę na temat wartościowych technik, które będzie mógł wykorzystać w przyszłości? Korzystamy z dwóch rodzajów elastyczności. Mamy jeden obiekt, `vector`, w którym możemy zmieniać dwie rzeczy:

- liczbę elementów,
- typ elementów.

Przydatność tego rodzaju zmienności ma fundamentalne znaczenie. Zawsze zbieramy jakieś dane. Kiedy rozejrzę się po swoim biurku, widzę stosy wyciągów bankowych, rachunki z karty kredytowej i rachunki za telefon. Każde z wymienionych jest w istocie listą wierszy informacji różnego typu — łańcuchy liter i wartości liczbowych. Przede mną leży telefon, w którym mam zapisane listy numerów telefonów i nazwisk. Na regałach po drugiej stronie pokoju stoją rzędy książek. Programy często wyglądają podobnie jak w tym opisie — zawierają kontenery różnego rodzaju elementów. Mamy różne rodzaje kontenerów (typ `vector` jest tylko jednym

z najczęściej używanych). Zapisane w nich są takie informacje jak numery telefoniczne, nazwiska, kwoty transakcji i dokumenty. W istocie wszystkie przykłady wzięte z mojego biurka i pokoju mają swój początek w jakimś programie komputerowym. Oczywistym wyjątkiem jest tu telefon — to **jest** komputer. Gdy patrzę na wyświetlone na jego wyświetlaczu cyfry, widzę dane wyjściowe programu, takiego samego jak te, które my piszemy. W istocie liczby te mogą być nawet przechowywane w wektorze `vector<Number>`.

Oczywiście kontenery nie mają takiej samej liczby elementów. Czy dałoby się żyć z wektorem, którego rozmiar został z góry ustalony na stałe? Innymi słowy, czy dałoby się żyć bez takich operacji jak `push_back()`, `resize()` itp.? Oczywiście że tak, ale stanowiłoby to niepotrzebne dodatkowe obciążenie dla programisty. Podstawową sztuczką, którą trzeba opanować przy pracy z kontenerami o stałym rozmiarze, jest przenoszenie elementów do nowego kontenera, gdy stary stanie się za mały. Moglibyśmy na przykład wczytywać dane do wektora, nigdy nie zmieniając jego rozmiarów w następujący sposób:

```
// Wczytuje elementy do wektora, nie używając funkcji push_back:
vector<double>* p = new vector<double>(10);
int n = 0; // liczba elementów
for (double d; cin>>d; ) {
    if (n==p->size()) {
        vector<double>* q = new vector<double>(p->size()*2);
        copy(p->begin(), p->end(), q->begin());
        delete p;
        p = q;
    }
    (*p)[n] = d;
    ++n;
}
```

Nie wygląda to zbyt elegancko. Masz pewność, że zrobiliśmy to dobrze? Skąd można mieć pewność? Zwróć uwagę, że nagle zaczęliśmy używać wskaźników i bezpośrednio zarządzać pamięcią. W tym przykładzie zaprezentowaliśmy styl programowania, który stosuje się w pracy „blisko" sprzętu — korzystanie tylko z podstawowych technik zarządzania pamięcią i używanie wyłącznie obiektów o stałym rozmiarze (tablice — podrozdział 18.6). Jednym z powodów do używania takich kontenerów jak `vector` jest uzyskanie lepszego efektu. Innymi słowy, chcemy, aby wektor wewnętrznie wykonywał operacje związane ze zmianą rozmiaru i zaoszczędził nam, użytkownikom, kłopotu i okazji do popełnienia błędu. Oznacza to, że interesują nas kontenery, które mogą się powiększać, aby pomieścić dokładnie tyle elementów, ile trzeba. Na przykład:

```
vector<double> vd;
for (double d;
cin>>d; ) vd.push_back(d);
```

Czy takie zmiany rozmiaru często się spotyka? Jeśli nie, to narzędzia je ułatwiające są tylko niewielkim udogodnieniem. Operacje tego typu są jednak bardzo częste. Jednym z oczywistych przykładów, gdzie są potrzebne, jest wczytywanie nieznanej liczby wartości. Inne przykłady to gromadzenie zbioru wyników wyszukiwania (nie wiadomo z góry, ile wyników zostanie zwróconych) i usuwanie jeden po drugim elementów z kolekcji. Dlatego nie pytamy, czy powinniśmy obsługiwać zmianę rozmiaru, lecz jak to zrobić.

Dlaczego w ogóle zaprzątamy sobie głowę zmienianiem rozmiaru? Czy nie można po prostu zaalokować wystarczająco dużo pamięci i już? Ten sposób działania wydaje się najprostszy i najbardziej wydajny. Miałoby to jednak sens tylko wówczas, gdybyśmy mogli mieć pewność, że rezerwujemy wystarczającą ilość pamięci i nie rezerwujemy jej o wiele za dużo — niestety nie możemy. Programiści, którzy próbują stosować to podejście, często muszą od nowa pisać duże partie kodu (jeśli skrupulatnie i systematycznie przeprowadzają testy pod kątem przepełnień) i liczyć się z katastrofalnymi efektami (jeśli niedbale podchodzili do kwestii testowania).

Oczywiście nie wszystkie wektory przechowują elementy tego samego typu. W zależności od programu potrzebne są wektory liczb typu `double`, do przechowywania odczytów temperatury, różnego rodzaju zapisów, łańcuchów, operacji, przycisków GUI, figur, dat, wskaźników na okna itp. Listę tę można ciągnąć w nieskończoność.

Istnieje wiele rodzajów kontenerów. Jest to bardzo ważny fakt i ze względu na jego implikacje nie można go przyjąć do wiadomości bez dokładnego przemyślenia. Dlaczego wszystkie kontenery nie mogą być po prostu wektorami? Gdybyśmy mogli poradzić sobie, mając tylko jeden rodzaj kontenera (np. `vector`), moglibyśmy zapomnieć o wszystkich troskach związanych z jego implementacją i uczynić go częścią języka. Gdybyśmy mogli poradzić sobie, mając tylko jeden rodzaj kontenera, nie musielibyśmy uczyć się o różnych rodzajach kontenerów — cały czas używalibyśmy wektorów.

Struktury danych odgrywają kluczową rolę w większości większych aplikacji. Napisano wiele grubych i bardzo przydatnych książek na temat sposobów organizacji danych. Większość zawartych w nich informacji odpowiada na pytanie: „Jak najlepiej przechować dane?”. Odpowiedź jest taka, że potrzebujemy wielu różnych rodzajów kontenerów, ale tematyka ta jest zbyt rozległa, aby się nią tutaj szczegółowo zajmować. Do tej pory wielokrotnie używaliśmy już wektorów (`vector`) i łańcuchów (`string` — typ ten jest kontenerem znaków). W dalszych rozdziałach opiszemy listy (`list`), słowniki (`map` — drzewa par wartości) i macierze. Ponieważ potrzebujemy wielu różnych kontenerów, narzędzia językowe i techniki programistyczne służące do ich budowy i używania będą nam bardzo pomocne. W istocie techniki przechowywania i dostępu do danych należą do fundamentalnych i najbardziej przydatnych technik wykorzystywanych w budowie wszystkich bardziej skomplikowanych form programowania.

Na najniższym poziomie pamięci wszystkie obiekty mają stały rozmiar i nie istnieją żadne typy. My wprowadzamy takie narzędzia językowe i techniki programistyczne umożliwiające nam tworzenie kontenerów obiektów różnych typów, które będą mogły przechowywać zmienną liczbę elementów. Dzięki temu zyskamy bardzo dużą elastyczność działania i wartościowe udogodnienia.

19.2. Zmienianie rozmiaru

Jakie narzędzia do zmieniania rozmiaru oferuje typ `vector` z biblioteki standardowej? Udostępnia trzy proste operacje. Jeśli dany jest wektor:

```
vector<double> v(n); // v.size()==n
```

można zmienić jego rozmiar na trzy sposoby:

```
v.resize(10);          // v zawiera teraz 10 elementów

v.push_back(7);        // Dodaje element o wartości 7 na końcu v
                       // v.size() zwiększa się o 1
v = v2;                // Przypisanie innego wektora. v jest teraz kopią v2
                       // v.size() teraz równa się v2.size()
```

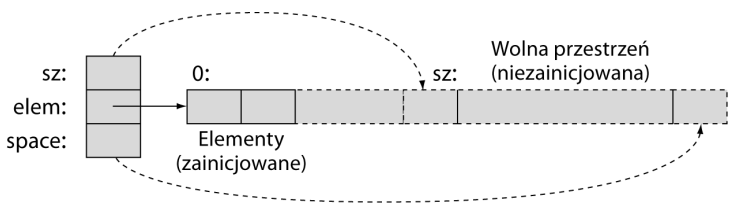
Znajdujący się w bibliotece standardowej wektor oferuje więcej operacji, które pozwalają zmienić rozmiar kontenera, są to np. `erase()` i `insert()` (dodatek B.4.7). Na razie zajmiemy się jednak implementacją w naszym wektorze tylko tych trzech podstawowych.

19.2.1. Reprezentacja

W podrozdziale 19.1 pokazaliśmy najprostszy sposób zmiany rozmiaru — zarezerwowanie pamięci dla nowej liczby elementów i skopiowanie starych elementów do tej nowej przestrzeni. Gdy jednak robi się to często, metoda ta staje się nieefektywna. W praktyce jeśli trzeba zmienić rozmiar raz, to zazwyczaj będzie trzeba to zrobić wiele razy. Rzadko na przykład zdarza się, aby funkcja `push_back()` była wywoływana tylko raz. Można zoptymalizować program, przewidując takie zmiany. W istocie wszystkie implementacje wektora pamiętają zarówno liczbę elementów, jak i ilość „wolnej przestrzeni" zarezerwowanej dla „przyszłych rozszerzeń". Na przykład:

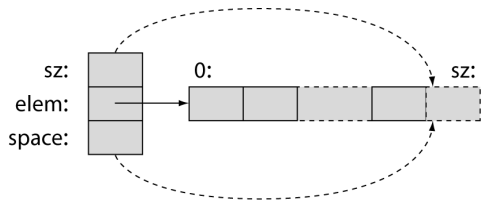
```
class vector {
    int sz;        // liczba elementów
    double* elem;  // adres pierwszego elementu
    int space;     // liczba elementów plus „wolna przestrzeń"
                   // dla nowych elementów („bieżący rozmiar alokacji")

public:
    // …
};
```

Można to przedstawić graficznie w następujący sposób:



Ponieważ liczenie elementów zaczyna się od 0, zmienną `sz` (liczba elementów) reprezentujemy jako odwołującą się do miejsca o jeden za ostatnim elementem, a `space` jako odwołującą się do miejsca o jeden za ostatnim alokowanym miejscem. Pokazane wskaźniki to w istocie `elem+sz` i `elem+space`.

Początkowo po utworzeniu wektora zmienna `space` ma wartość `sz`, czyli brak jest „wolnej przestrzeni":

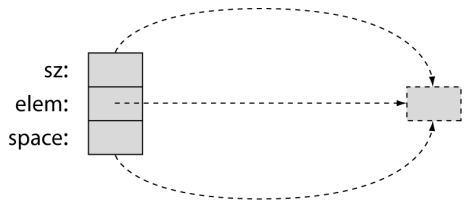


Rezerwacja dodatkowych miejsc zaczyna się dopiero po pierwszej próbie zmiany liczby elementów. Zwykle `space==sz`, a więc nie ma żadnego nadmiaru w wykorzystaniu pamięci, dopóki nie zostanie użyta funkcja `push_back()`.

Konstruktor domyślny (tworzący wektory bez elementów) ustawia składowe całkowitoliczbowe na `0`, a wskaźnik na `nullptr`:

```
vector::vector() :sz{0}, elem{nullptr}, space{0} { }
```

To daje:



Ten jeden element poza granicami jest tylko wytworem wyobraźni. Konstruktor domyślny nie rezerwuje pamięci wolnej i zajmuje minimalną ilość pamięci (ale zobacz 16. zadanie pracy domowej).

Należy zauważyć, że nasz wektor ilustruje techniki, które można zastosować do zaimplementowania standardowego wektora (i innych struktur danych), ale w implementacjach biblioteki standardowej zezwolono na dość dużą dowolność, dlatego `std::vector` w Twoim systemie może korzystać z innych technik.

19.2.2. Rezerwacja pamięci i pojemność kontenera

Fundamentalną operacją wykonywaną przy zmianie rozmiaru (czyli zmianie liczby elementów) jest `vector::reserve()`. Za jej pomocą dodajemy przestrzeń dla nowych elementów:

```
void vector::reserve(int newalloc)
{
    if (newalloc<=space) return;                // Nigdy nie zmniejszaj obszaru alokacji
    double* p = new double[newalloc];           // Alokuje nową przestrzeń
    for (int i=0; i<sz; ++i) p[i] = elem[i]; // Kopiuje stare elementy
    delete[] elem;                              // Dealokuje starą przestrzeń
    elem = p;
    space = newalloc;
}
```

Należy zauważyć, że elementy zarezerwowanej przestrzeni nie są inicjalizowane. Jest tylko rezerwowana pamięć, a jej wykorzystanie do przechowywania elementów jest zadaniem funkcji `push_back()` i `resize()`.

Oczywiście użytkownika może interesować ilość dostępnej wektorowi wolnej przestrzeni, dlatego (analogicznie do standardowej wersji) utworzymy funkcję składową służącą do sprawdzania tej informacji:

```
int vector::capacity() const { return space; }
```

To znaczy, że dla wektora `v` `v.capacity()-v.size()` oznacza liczbę elementów, które można wstawić (`push_back()`), nie powodując realokacji.

19.2.3. Zmienianie rozmiaru

Dzięki temu, że mamy funkcję `reserve()`, zaimplementowanie funkcji `resize()` dla naszego wektora będzie łatwe. Musimy pamiętać o kilku przypadkach:

- Nowy rozmiar jest większy od starego obszaru alokacji.
- Nowy rozmiar jest większy od starego rozmiaru, ale mniejszy lub równy staremu obszarowi alokacji.
- Nowy rozmiar jest równy staremu rozmiarowi.
- Nowy rozmiar jest mniejszy od starego rozmiaru.

Zobaczmy, co możemy zrobić:

```
void vector::resize(int newsize)
    // Zmienia rozmiar wektora na newsize
    // Każdy nowy element inicjalizuje domyślną wartością 0.0
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) elem[i] = 0; // Inicjalizuje nowe elementy
    sz = newsize;
}
```

Zadanie związane z zarządzaniem pamięcią oddelegowaliśmy do funkcji `reserve()`. Znajdująca się w niej pętla inicjalizuje nowe elementy, jeśli takie są.

Nie zajęliśmy się bezpośrednio żadnym przypadkiem, ale można sprawdzić, że wszystkie są obsługiwane poprawnie.

WYPRÓBUJ

Jakie przypadki należy rozważyć (i przetestować), aby przekonać się, czy funkcja `resize()` działa prawidłowo? Może `newsize==0` albo `newsize=-77`?

19.2.4. Funkcja push_back()

Na pierwszy rzut oka implementacja funkcji `push_back()` może wydawać się skomplikowana, ale jeśli ma się do dyspozycji funkcję `reserve()`, zadanie jest dużo łatwiejsze:

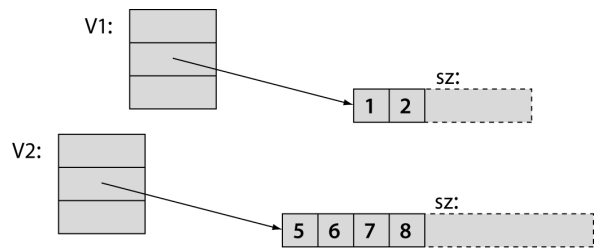
```
void vector::push_back(double d)
    // Zwiększa rozmiar wektora o jeden i inicjalizuje nowy element wartością d
{
```

```
    if (space==0) reserve(8); // Zaczyna od zarezerwowania przestrzeni dla 8 elementów
    else if (sz==space) reserve(2*space); // Rezerwuje więcej przestrzeni
    elem[sz] = d;                         // Dodaje d na końcu
    ++sz;                                 // Zwiększa rozmiar (sz określa liczbę elementów)
}
```

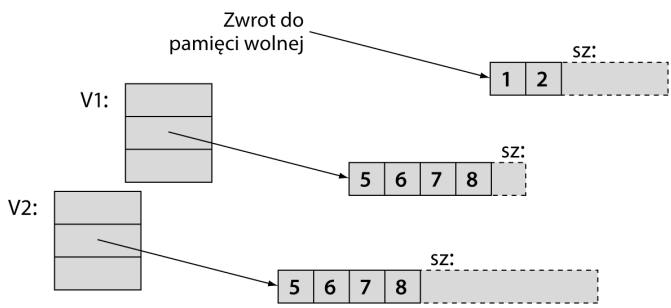
Innymi słowy, jeśli nie mamy zapasowej przestrzeni, podwajamy rozmiar obszaru alokacji. W praktyce jest to świetna strategia dla większości zastosowań wektorów. Zastosowano ją w większości standardowych implementacji tej struktury danych.

19.2.5. Przypisywanie

Przypisywanie wektorów można zdefiniować na kilka sposobów. Można na przykład postawić warunek, że oba zaangażowane w operację wektory muszą mieć taką samą liczbę elementów. Jednak w punkcie 18.3.2 stwierdziliśmy, że operacja przypisywania wektorów powinna mieć najogólniejsze i chyba najbardziej oczywiste działanie — w wyniku operacji `v1=v2` wektor `v1` powinien stać się kopią wektora `v2`. Rozważmy:



Oczywiście musimy skopiować wszystkie elementy, ale co zrobić z zapasową przestrzenią? Czy powinniśmy ją skopiować na końcu? Nie. Nowy wektor będzie zawierał kopie elementów starego wektora, ale ponieważ nie wiadomo, do czego będzie używany, nie zaprzątamy sobie głowy żadną dodatkową przestrzenią na końcu:



Oto przepis na najprostszą implementację tego:

- Zaalokuj pamięć dla kopii.
- Skopiuj elementy.
- Usuń stary obszar alokacji.
- Ustaw składowe `sz`, `elem` i `space` na nowe wartości.

Kod:

```
vector& vector::operator=(const vector& a)
    // Podobne do konstruktora kopiującego, ale musimy zajmować się starymi elementami
{
    double* p = new double[a.sz];                    // Alokuje nową przestrzeń
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i]; // Kopiuje elementy
    delete[] elem;                                   // Dealokuje starą przestrzeń
    space = sz = a.sz;                               // Ustawia nowy rozmiar
    elem = p;                                        // Ustawia nowe elementy
    return *this;                                    // Zwraca referencję do samego siebie
}
```

Tradycyjnie operator przypisania zwraca referencję do obiektu, do którego nastąpiło przypisanie. Służy do tego notacja `*this`, której opis znajduje się w podrozdziale 17.10.

Powyższa implementacja jest poprawna, ale jeśli się jej dokładnie przyjrzeć, można zauważyć, że wykonywanych jest za dużo operacji alokowania i dealokowania. Co jeśli wektor, do którego przypisujemy, ma więcej elementów niż przypisywany wektor? Co jeśli wektor, do którego przypisujemy, ma tyle samo elementów, co przypisywany wektor. Ten drugi przypadek występuje bardzo często. W obu przypadkach można skopiować elementy do już dostępnej w docelowym wektorze przestrzeni.

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this; // Samoprzypisanie — nic nie trzeba robić

    if (a.sz<=space) { // Wystarczająco dużo miejsca — nie trzeba przeprowadzać nowej alokacji
        for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i]; // kopiowanie elementów
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz];                    // alokacja nowej przestrzeni
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i]; // kopiowanie elementów
    delete[] elem;                                   // dealokacja starej przestrzeni
    space = sz = a.sz;                               // ustawianie nowego rozmiaru
    elem = p;                                        // ustawianie nowych elementów
    return *this;                                    // Zwraca referencję do samego siebie
}
```

Najpierw sprawdzamy, czy nie ma tu samoprzypisania (np. `v=v`). Jeśli jest, nic nie robimy. Z logicznego punktu widzenia test ten jest niepotrzebny, ale jednak stanowi w pewnych sytuacjach dobry środek optymalizacyjny. Poza tym przedstawia często spotykany sposób wykorzystania wskaźnika `this`, sprawdzając, czy argument `a` jest tym samym obiektem co ten, dla którego została wywołana funkcja składowa (tutaj `operator=()`). Przekonaj się, że powyższy kod będzie działał także po usunięciu wiersza z kodem `this==&a`. Także instrukcja `a.sz<=space` ma tylko funkcję optymalizacyjną. Sprawdź, że kod ten będzie działał nawet po jej usunięciu.

19.2.6. Podsumowanie dotychczasowej pracy nad typem vector

Mamy już prawie prawdziwy wektor liczb typu `double`:

```
// Prawie prawdziwy wektor liczb typu double:
class vector {
    /*
    inwariant:
    if 0<=n<sz, elem[n] jest elementem n
    sz<=space;
    Jeśli sz<space, jest miejsce dla (space–sz) liczb za elem[sz–1]
    */
    int sz;                                          // rozmiar
    double* elem;                                    // wskaźnik na elementy (lub 0)
    int space;                                       // liczba elementów plus liczba wolnych miejsc
public:
    vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit vector(int s) :sz{s}, elem{new double[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;          // inicjalizacja elementów
    }
    vector(const vector&);                           // konstruktor kopiujący
    vector& operator=(const vector&);                // przypisanie kopiujące

    vector(vector&&);                                // konstruktor przenoszący
    vector& operator=(vector&&);                     // przypisanie przenoszące

    ~vector() { delete[] elem; }                     // destruktor

    double& operator[ ](int n) { return elem[n]; } // dostęp do referencji zwrotnej
    const double& operator[](int n) const { return elem[n]; }

    int size() const { return sz; }

    int capacity() const { return space; }
    void resize(int newsize);                        // wzrost
    void push_back(double d);
    void reserve(int newalloc);
};
```

Należy zwrócić uwagę, że wektor ten posiada podstawowe operacje (podrozdział 18.4) — konstruktor, konstruktor domyślny, operacje kopiowania, destruktor. Ma operację dostępu do danych (operator indeksowania []), funkcje zwracające informacje o tych danych (`size()` i `capacity()`) oraz operacje umożliwiające kontrolę nad wzrostem struktury danych (`resize()`, `push_back()` oraz `reserve()`).

19.3. Szablony

Sęk w tym, że nie chcemy wektora, który pozwala przechowywać tylko wartości typu `double`. Chcemy mieć możliwość dowolnego określania typu danych, które będziemy przechowywać. Na przykład:

```
vector<double>
vector<int>
```

```
vector<Month>
vector<Window*>           // wektor wskaźników na okna
vector< vector<Record> > // wektor wektorów rekordów
vector<char>
```

Aby to zrobić, musimy wiedzieć, jak się definiuje szablony. Używamy ich od samego początku, ale do tej pory nie potrzebowaliśmy ich definiować. W bibliotece standardowej jest wszystko, czego do tej pory potrzebowaliśmy, ale nie możemy wierzyć w magię. Dlatego musimy dowiedzieć się, jak projektanci i implementatorzy biblioteki standardowej utworzyli takie rzeczy jak typ `vector` i funkcja `sort()` (podrozdział 21.1 i punkt B.5.4). Nie jest to tylko zagadnienie teoretyczne, ponieważ jak zwykle narzędzia i techniki wykorzystane w implementacji biblioteki standardowej będą nam bardzo przydatne w pisaniu własnego kodu. Na przykład w rozdziałach 21. i 22. opiszemy sposoby zastosowania szablonów w implementowaniu kontenerów i algorytmów biblioteki standardowej. W rozdziale 24. przedstawimy techniki projektowania macierzy do obliczeń naukowych.

Zasadniczo **szablon** (ang. *template*) to mechanizm pozwalający programiście używać typów jako parametrów klas i funkcji. Kompilator generuje odpowiednią klasę lub funkcję, gdy użytkownik poda jako argumenty konkretne typy.

19.3.1. Typy jako parametry szablonów

Chcemy zrobić tak, aby wektor przyjmował parametr, który będzie określał typ przechowywanych w tym wektorze argumentów. W związku z tym w naszym wektorze usuwamy słowo `double` i zamieniamy je na literę `T` będącą parametrem, któremu można nadawać „wartości" określające typy, np. `double`, `int`, `string`, `vector<Record>` czy `Window*`. W języku C++ notacja służąca do wprowadzania parametru typu `T` ma postać przyrostka `template<typename T>`, który oznacza „dla wszystkich typów `T`". Na przykład:

```
// Prawie prawdziwy wektor elementów typu T:
template<typename T> // Czytaj: „dla wszystkich typów T" (tak jak w matematyce)
    int sz;                                     // rozmiar
    T* elem;                                    // wskaźnik na elementy
    int space;                                  // rozmiar + wolna przestrzeń
public:
    vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit vector(int s) :sz{s}, elem{new T[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;   // elementy są zainicjalizowane
    }

    vector(const vector&);                      // konstruktor kopiujący
    vector& operator=(const vector&);           // przypisanie kopiujące

    vector(vector&&);                           // konstruktor przenoszący
    vector& operator=(vector&&);                // przypisanie przenoszące

    ~vector() { delete[] elem; }                // destruktor

    T& operator[](int n) { return elem[n]; } // dostęp: zwraca referencję
```

```
    const T& operator[](int n) const { return elem[n]; }

    int size() const { return sz; }          // bieżący rozmiar
    int capacity() const { return space; }

    void resize(int newsize);                // wzrost
    void push_back(const T& d);
    void reserve(int newalloc);
};
```

To jest nasz wektor liczb typu `double` z punktu 19.2.6 z zamienionymi wystąpieniami `double` na parametr szablonowy `T`. Oto przykładowe sposoby użycia tego szablonu klasowego:

```
vector<double> vd;          // T jest double
vector<int> vi;             // T jest int
vector<double*> vpd;        // T jest double*
vector< vector<int> > vvi; // T jest vector<int>, w którym T jest int
```

Jeśli chodzi o kompilator, można sobie wyobrazić, że gdy napotyka szablon, generuje klasę, wstawiając rzeczywisty typ (zwany argumentem szablonowym) w miejsce parametru szablonowego. Jeśli na przykład w kodzie znajdzie się `vector<char>`, kompilator wygeneruje coś takiego:

```
class vector_char {
    int sz;                                          // rozmiar
    char* elem;                                      // wskaźnik na elementy
    int space;                                       // rozmiar + wolna przestrzeń
public:
    vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit vector_char(int s) :sz{s}, elem{new char[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;          // elementy są zainicjalizowane
    }

    vector_char(const vector_char&);                 // konstruktor kopiujący
    vector_char& operator=(const vector_char &);     // przypisanie kopiujące

    vector_char(vector_char&&);                      // konstruktor przenoszący
    vector_char& operator=(vector_char&&);           // przypisanie przenoszące

    ~vector_char ();                                 // destruktor

    char& operator[] (int n) { return elem[n]; }     // dostęp: zwraca referencję
    const char& operator[] (int n) const { return elem[n]; }

    int size() const;                                // bieżący rozmiar
    int capacity() const;

    void resize(int newsize);                        // wzrost
    void push_back(const char& d);
    void reserve(int newalloc);
};
```

Dla `vector<double>` kompilator wygeneruje mniej więcej taki sam kod, jak w punkcie 19.2.6 (używając odpowiedniej wewnętrznej nazwy do oznaczenia `vector<double>`).

Czasami szablon klasy nazywa się **generatorem typów** (ang. *type generator*). Proces generowania typów (klas) z szablonu klasy przy użyciu argumentów szablonowych nazywa się **specjalizacją** (ang. *specialization*) lub **konkretyzacją szablonu** (ang. *template instantiation*). Na przykład `vector<char>` i `vector<Poly_line*>` są specjalizacjami szablonu klasy `vector`. W mało skomplikowanych przypadkach, jak nasz wektor, konkretyzacja jest bardzo prostą operacją. Natomiast w najbardziej ogólnych i zaawansowanych przypadkach konkretyzacja szablonu jest nieprawdopodobnie skomplikowana. Na szczęście dla użytkowników z tego rodzaju komplikacjami muszą radzić sobie osoby piszące kompilatory, a nie użytkownicy szablonów. Konkretyzacja szablonów (generowanie specjalizacji szablonów) odbywa się na etapie kompilacji lub konsolidacji — nie w czasie działania programu.

Oczywiście można używać funkcji składowych szablonów:

```
void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Witalis");
    // ...
}
```

Kompilator wygeneruje dla takiej funkcji odpowiedni kod. Jeśli na przykład zostanie zastosowane wywołanie `v.push_back("Witalis")`, kompilator wygeneruje następującą funkcję:

```
void vector<string>::push_back(const string& d) { /* ... */ }
```

z poniższej definicji szablonu:

```
template<typename T> void vector<T>::push_back(const T& d) { /* ... */ };
```

Dzięki temu da się znaleźć funkcję do wywołania dla instrukcji `v.push_back("Witalis")`. Innymi słowy, jeśli będzie potrzebna funkcja dla określonego typu obiektu i argumentu, kompilator napisze ją dla nas na podstawie szablonu.

Zamiast `template<typename T>` można napisać `template<class T>`. Obie te konstrukcje są równoważne, ale niektórzy wolą wersję z `typename`, „ponieważ jest bardziej przejrzysta” i „ponieważ słowo `typename` nikogo nie zmyli, że nie można jako argumentów szablonów używać typów wbudowanych, takich jak `int`”. Naszym zdaniem `class` oznacza typ, a więc nie ma różnicy. Poza tym słowo `class` jest krótsze.

19.3.2. Programowanie ogólne

Podstawą programowania ogólnego w języku C++ są szablony. W istocie programowanie ogólne w tym języku można najprościej zdefiniować jako „używanie szablonów”. Jednak definicja ta jest zbyt dużym uproszczeniem. Nie powinno się definiować fundamentalnych koncepcji programistycznych w kategoriach własności języka programowania. To języki programowania istnieją po to, aby wspierać techniki programistyczne, a nie na odwrót. Tak jak wszystkie popularne techniki, programowanie ogólne (ang. *generic programming*) ma wiele definicji. Poniżej przedstawiamy prostą i naszym zdaniem najlepszą:

> **Programowanie ogólne**: pisanie kodu, który może działać z różnymi typami przekazywanymi do niego jako argumenty, pod warunkiem że typy te spełniają określone wymagania syntaktyczne i semantyczne.

Na przykład elementy wektora muszą być typu, który można kopiować (poprzez konstruowanie kopiujące lub przypisanie kopiujące). W rozdziałach 20. i 21. przedstawimy szablony wymagające operacji arytmetycznych na swoich argumentach. Jeśli sparametryzuje się klasę, powstaje **szablon klasy** (ang. *class template*), który często nazywa się **typem parametryzowanym** (ang. *parametrized type*) lub **klasą parametryzowaną** (ang. *parametrized class*). Jeśli sparametryzuje się funkcję, powstaje **szablon funkcji** (ang. *function template*), który często nazywa się **funkcją parametryzowaną** (ang. *parametrized function*) lub czasami **algorytmem** (ang. *algorithm*). Dlatego programowanie ogólne czasami nazywa się „programowaniem algorytmicznym" — programista bardziej skupia się na algorytmach niż typach danych, które te algorytmy wykorzystują.

Ponieważ typy parametryzowane odgrywają w programowaniu bardzo ważną rolę, dokładniej zapoznamy się ze związaną z tą tematyką terminologią. Dzięki temu może unikniesz problemów ze zrozumieniem innych tekstów.

Rodzaj programowania ogólnego, w którym bezpośrednio wykorzystuje się parametry szablonowe, często nazywa się **polimorfizmem parametrycznym** (ang. *parametric polymorphism*). Dla kontrastu rodzaj polimorfizmu uzyskiwany za pomocą hierarchii klas i funkcji wirtualnych nazywa się **polimorfizmem ad hoc** i ten styl programowania określa się jako **programowanie obiektowe** (podrozdziały 14.3 i 14.4). Oba te style nazywa się **polimorfizmem**, ponieważ w każdym z nich programista dostarcza wielu wersji jednej koncepcji za pośrednictwem jednego interfejsu. Słowo **polimorfizm** pochodzi z języka greckiego i oznacza „wiele kształtów", co jest odniesieniem do możliwości wykorzystania wielu różnych typów poprzez jeden wspólny interfejs. W przykładach kodu przedstawionych w rozdziałach od 12. do 17. dosłownie posługiwaliśmy się różnymi kształtami (np. `Text`, `Circle` i `Polygon`) za pośrednictwem interfejsu zdefiniowanego przez `Shape`. Gdy używamy wektorów, wykorzystujemy ich wiele (np. `vector<int>`, `vector<double>` i `vector<Shape*>`) za pośrednictwem interfejsu zdefiniowanego przez szablon `vector`.

Styl programowania obiektowego (używanie hierarchii klas i funkcji wirtualnych) różni się od programowania ogólnego (używanie szablonów) pod kilkoma względami. Do najbardziej oczywistych należy to, że w programowaniu ogólnym wyboru funkcji do wywołania kompilator dokonuje w czasie kompilacji, podczas gdy w programowaniu obiektowym odbywa się to w czasie działania programu. Na przykład:

```
v.push_back(x); // Wstawia x do wektora v
s.draw();       // Rysuje figurę s
```

Dla wywołania `v.push_back(x);` kompilator sprawdzi typ elementu `x` i użyje odpowiedniej funkcji `push_back()`. Natomiast w przypadku wywołania `s.draw()` kompilator pośrednio wywoła jakąś funkcję `draw()` (używając tablicy `vtbl` — punkt 14.3.1). To sprawia, że programowanie obiektowe może zaoferować większą swobodę niż ogólne, chociaż to drugie jest bardziej regularne, łatwiejsze do zrozumienia i wydajniejsze (stąd nazwy „ad hoc" i „parametryczny").

Podsumowanie:

- *Programowanie ogólne* — realizowane za pomocą szablonów, wybory dokonywane są w czasie kompilacji.

- *Programowanie obiektowe* — realizowane za pomocą hierarchii klas i funkcji wirtualnych, wybory dokonywane są w czasie działania programu.

Można te dwa style łączyć, co bywa korzystne. Na przykład:

```
void draw_all(vector<Shape*>& v)
{
    for (int i=0; i<v.size(); ++i) v[i]->draw();
}
```

W tym kodzie wywoływana jest funkcja wirtualna (`draw()`) na rzecz klasy bazowej (`Shape`) przy użyciu funkcji wirtualnej — to jest zdecydowanie programowanie obiektowe. Jednak przechowujemy też elementy typu `Shape*` w wektorze, który jest typem parametryzowanym — a więc zastosowaliśmy też proste programowanie ogólne.

Zakładając, że nasyciłeś się już filozofią, spróbujemy odpowiedzieć na pytanie: po co w ogóle używa się szablonów. Ze względu na ich niedoścignioną elastyczność i świetną wydajność:

- Stosuj szablony wszędzie, gdzie kluczowe znaczenie ma wydajność (np. obliczenia liczbowe i programowanie systemów o ostrych ograniczeniach czasowych — rozdziały 24. i 25.).
- Stosuj szablony wszędzie, gdzie najważniejsze jest łączenie informacji z różnych typów (np. biblioteka standardowa C++ — rozdziały 20. i 21.).

19.3.3. Koncepcje

Szablony mają wiele wspaniałych cech, jak świetna elastyczność i prawie optymalna wydajność. Mimo to nie są niestety doskonałe. Jak zwykle z zaletami wiążą się pewne słabości. Największą wadą szablonów jest to, że ich elastyczność i wydajność są uzyskiwane kosztem słabego oddzielenia „wnętrza" szablonu (jego definicji) od interfejsu (deklaracji). Skutkiem tego jest niskiej jakości diagnostyka — często zgłaszane są wyjątkowo nietrafne komunikaty o błędach. Czasami komunikaty te pojawiają się na znacznie późniejszym etapie kompilacji, niż powinny.

Podczas kompilacji kodu wykorzystującego szablon kompilator „zagląda" do szablonu oraz do typów jego argumentów. Robi to po to, aby zdobyć informacje potrzebne do wygenerowania optymalnego kodu. Aby mieć wszystkie potrzebne informacje w zasięgu, aktualnie kompilatory wymagają podawania pełnej definicji szablonu wszędzie, gdzie został użyty. Dotyczy to definicji wszystkich funkcji składowych i wszystkich funkcji szablonowych, które są w nich wywoływane. Z tego powodu programiści piszący szablony często owoce swojej pracy umieszczają w plikach nagłówkowych. Nie jest to standardowy wymóg, ale dopóki nie upowszechnią się radykalnie ulepszone implementacje, zalecamy robienie tego także z własnymi szablonami — definicje wszystkich szablonów, które będą używane w więcej niż jednej jednostce translacyjnej, umieszczaj w plikach nagłówkowych.

Zacznij od pisania bardzo prostych szablonów i powoli zdobywaj doświadczenie. Jedną z przydatnych technik, którą możesz zastosować, jest ta, której użyliśmy do implementacji typu `vector` — najpierw napisz i przetestuj klasę przy użyciu jednego konkretnego typu. Gdy to się uda, zastąp określniki typów parametrami szablonu i wykonaj testy przy użyciu różnych argumentów szablonu. Korzystaj z ogólności, bezpieczeństwa i wydajności oferowanych przez biblioteki oparte na szablonach, np. bibliotekę standardową C++. Przykłady wykorzystania szablonów zostaną przedstawione w rozdziałach 20. i 21., w których opiszemy kontenery i algorytmy biblioteki standardowej.

W C++14 dodano mechanizm, który znacznie usprawnia sprawdzanie interfejsów szablonów. W C++11 na przykład piszemy tak:

```
template<typename T> // dla wszystkich typów T
class vector {
    // …
};
```

Nie możemy precyzyjnie określić naszych oczekiwań wobec typu argumentu `T`. W standardzie napisano, że te wymagania są, ale tylko po angielsku, a nie wyrażone w kodzie zrozumiałym dla kompilatora. Zestaw wymagań dotyczących argumentu szablonu nazywa się **koncepcją** (ang. *concept*). Argument szablonu musi spełniać wymagania — koncepcje — szablonu, do którego został zastosowany. Wektor na przykład wymaga, aby jego elementy można było kopiować lub przenosić, aby można było sprawdzać ich adresy oraz aby można było tworzyć elementy domyślne (w razie potrzeby). Innymi słowy: element musi spełniać wymagania, które moglibyśmy nazwać `Element`. W C++14 można taki zestaw wymagań zdefiniować za pomocą kodu:

```
template<typename T>        // dla wszystkich typów T
    requires Element<T>() // taki, że T jest Elementem
class vector {
    // …
};
```

Wynika z tego, że koncepcja to w istocie predykat typu, czyli ewaluowana w czasie kompilacji (`constexpr`) funkcja zwracająca prawdę, jeśli typ argumentu (tutaj `T`) ma właściwości wymagane przez koncepcję (tutaj `Element`), lub fałsz w przeciwnym przypadku.

Ten odrobinę przydługi zapis możemy skrócić do następującej postaci:

```
template<Element T> // dla wszystkich typów T, takich, że Element<T>() to prawda
class vector {
    // …
};
```

Jeśli nie mamy kompilatora obsługującego koncepcje C++14, wymagania możemy zdefiniować przy użyciu nazw i komentarzy:

```
template<typename Elem> // wymaga Element<Elem>()
class vector {
    // …
};
```

Kompilator nie rozumie nazw wymyślonych przez programistę i nie czyta komentarzy, ale bezpośrednie wyrażenie koncepcji pomaga nam w myśleniu, wspomaga projektowanie ogólnego kodu i ułatwia innym programistom jego zrozumienie. Później będziemy jeszcze używać niektórych popularnych i praktycznych koncepcji:

- `Element<E>()` — `E` może być elementem w kontenerze.
- `Container<C>()` — `C` może przechowywać elementy `Element` oraz można go przeglądać jako sekwencję `[begin():end())`.

- `Forward_iterator<For>()` — przy użyciu `For` można przeglądać sekwencję `[b:e)` (jak listę powiązaną, wektor lub tablicę).
- `Input_iterator<In>()` — przy użyciu `In` można odczytać sekwencję `[b:e)` tylko raz (jak strumień wejściowy).
- `Output_iterator<Out>()` — sekwencję można wysłać na wyjście za pomocą `Out`.
- `Random_access_iterator<Ran>()` — przy użyciu `Ran` można wielokrotnie odczytywać i zapisywać sekwencję `[b:e)`. Ponadto `Ran` obsługuje indeksowanie za pomocą operatora `[ ]`.
- `Allocator<A>()` — przy użyciu `A` można zajmować i zwalniać pamięć (jak pamięć wolna).
- `Equal_comparable<T>()` — dwa `T` można porównać za pomocą operatora `==`. Wynik jest wartością logiczną.
- `Equal_comparable<T,U>()` — `T` i `U` można porównać za pomocą operatora `==`. Wynik jest wartością logiczną.
- `Predicate<P,T>()` — możemy wywołać `P` z argumentem typu `T`, aby otrzymać wynik w postaci wartości logicznej.
- `Binary_predicate<P,T>()` — możemy wywołać `P` z dwoma argumentami typu `T`, aby otrzymać wynik w postaci wartości logicznej.
- `Binary_predicate<P,T,U>()` — możemy wywołać `P` z argumentami typów `T` i `U`, aby otrzymać wynik w postaci wartości logicznej.
- `Less_comparable<L,T>()` — możemy użyć `L` do porównania dwóch `T` przy użyciu operatora mniejszości `<`, aby otrzymać wynik w postaci wartości logicznej.
- `Less_comparable<L,T,U>()` — możemy użyć `L` do porównania `T` z `U` za pomocą operatora mniejszości `<`, aby otrzymać wynik w postaci wartości logicznej.
- `Binary_operation<B,T,U>()` — możemy użyć `B`, aby wykonać operację na dwóch `T`.
- `Binary_operation<B,T,U>()` — możemy użyć `B`, aby wykonać operację na `T` i `U`.
- `Number<N>()` — `N` zachowuje się jak liczba, tzn. obsługuje operacje `+`, `-`, `*` i `/`.

Dla kontenerów i algorytmów w bibliotece standardowej te koncepcje (i wiele innych) są opisane bardzo szczegółowo. W tej książce, szczególnie w rozdziałach 20. i 21., będziemy ich używać w celu nieformalnego dokumentowania naszych kontenerów i algorytmów.

Typ kontenera i iteratora, `T`, mają typ wartościowy (`Value_type<T>`), który jest typem elementu. Ten typ `Value_type<T>` często jest typem składowym `T::value_type` (zobacz wektor i listę w podrozdziale 20.5).

19.3.4. Kontenery a dziedziczenie

Istnieje jeden rodzaj kombinacji programowania obiektowego i ogólnego, którego każdy musi kiedyś spróbować, a który nigdy się nie udaje. Chodzi o używanie kontenerów obiektów klasy pochodnej jako kontenerów obiektów klasy bazowej. Na przykład:

```
vector<Shape> vs;
vector<Circle> vc;
vs = vc; // Błąd: wymagany vector<Shape>
void f(vector<Shape>&);
f(vc);    // Błąd: wymagany vector<Shape>
```

Ale czemu nie? Przecież można przekonwertować `Circle` na `Shape`! W istocie nie można. Można przekonwertować `Circle*` na `Shape*` i `Circle&` na `Shape&`, ale celowo uniemożliwiliśmy przypisywanie obiektów typu `Shape` do `Circle`, dzięki czemu nie trzeba się zastanawiać, co by się stało, gdyby umieszczono obiekt typu `Circle` (który ma promień) do zmiennej typu `Shape` (bez promienia) — punkt 14.2.4. Efektem tego — gdybyśmy na to pozwolili — byłoby „pocięcie" obiektu — klasowy odpowiednik obcinania liczb do formatu całkowitoliczbowego (punkt 3.9.2).

Spróbujemy zatem jeszcze raz przy użyciu wskaźników:

```
vector<Shape*> vps;
vector<Circle*> vpc;
vps = vpc; // Błąd: wymagany vector<Shape*>
void f(vector<Shape*>&);
f(vpc);     // Błąd: wymagany vector<Shape*>
```

Znowu spotykamy się z oporem ze strony systemu typów. Dlaczego? Pomyślmy, co mogłaby zrobić funkcja `f()`:

```
void f(vector<Shape*>& v)
{
    v. .push_back(new Rectangle{Point{0,0},Point{100,100}});
}
```

Oczywiście można wstawić element typu `Rectangle*` do wektora typu `vector<Shape*>`. Gdyby jednak wektor ten został gdzieś indziej potraktowany jako wektor typu `vector<Circle*>`, kogoś spotkałaby niemiła niespodzianka. Gdyby kompilator przepuścił powyższy przykład, co robiłby `Rectangle*` w `vpc`? Dziedziczenie to potężny i delikatny mechanizm, a szablony nie są jego wzbogaceniem. Istnieją sposoby na wyrażenie dziedziczenia za pomocą szablonów, ale nie jest to temat do uwzględnienia tej książce. Należy tylko zapamiętać: mimo iż „`D` jest `B`", nie oznacza to automatycznie, że „`C<D>` jest `C<B>`" dla dowolnego szablonu `C` — i należy to uważać za ochronę przed przypadkowym złamaniem zasad systemu typów (zobacz też punkt 25.4.4).

19.3.5. Liczby całkowite jako parametry szablonów

Oczywiście parametryzowanie klas typami jest bardzo przydatną techniką. A co z „innymi rzeczami", jak wartości całkowitoliczbowe i łańcuchy? W zasadzie każdy rodzaj argumentu może być przydatny, ale my zajmiemy się tylko parametrami w formie typów i liczb całkowitych. Inne rodzaje parametrów są rzadziej używane i ich obsługa w języku C++ wymaga od programisty szczegółowej wiedzy na temat tego języka.

Rozważymy jeden z najczęściej spotykanych przypadków użycia wartości całkowitoliczbowej jako argumentu szablonu — kontenera, którego liczba elementów jest znana w czasie kompilacji:

```
template<typename T, int N> struct array {
    T elem[N];                  // Przechowuje elementy w tablicy składowej

    // Polega na domyślnych konstruktorach, destruktorze i przypisaniu

    T& operator[] (int n);      // dostęp: zwraca referencję
    const T& operator[] (int n) const;

    T* data() { return elem; } // konwersja na T*
    const T* data() const { return elem; }

    int size() const { return N; }
};
```

Możemy powyższej tablicy użyć następująco (zobacz też podrozdział 20.7):

```
array<int,256> gb;                                          // 256 liczb całkowitych
array<double,6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 }; // Zwróć uwagę na inicjalizator!
const int max = 1024;

void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops; // Błąd: wartość n nie jest znana kompilatorowi
    // …
    array<char,max> loc2 = loc;                             // Tworzy kopię zapasową
    // …
    loc = loc2;                                             // przywrócenie
    // …
}
```

Bez wątpienia typ `array` jest bardzo prosty — znacznie prostszy i mniej funkcjonalny niż `vector`. Dlaczego więc w ogóle ktokolwiek miałby używać typu `array`? Jedną z odpowiedzi jest wydajność. Rozmiar tablicy jest znany w czasie kompilacji, dzięki czemu kompilator może alokować pamięć statyczną (dla obiektów globalnych, np. `gb`) i na stosie (dla obiektów lokalnych, np. `loc`). Operacje sprawdzania zakresu można przeprowadzać w odniesieniu do stałych (parametr `N` określający rozmiar).W większości programów taka niewielka poprawa wydajności nie ma znaczenia. Jeśli jednak pisze się jakiś ważny składnik systemu, np. sterownik sieci, to nawet najmniejsza poprawa może być ważna.

Co ważniejsze, niektóre programy po prostu nie mogą korzystać z pamięci wolnej. Jest to najczęściej oprogramowanie systemów wbudowanych i programy, w których najważniejsze jest bezpieczeństwo (rozdział 25.). W takich przypadkach tablice pozwalają korzystać z wielu zalet wektorów bez pogwałcenia najważniejszej zasady (nie używać pamięci wolnej).

Spróbujemy spojrzeć na to z innej strony. Zamiast pytać: „Czemu nie można użyć wektora?”, spytamy: „Czemu by nie użyć tablicy?”. Jak pamiętamy z podrozdziału 18.6, tablice często źle się zachowują — nie znają swojego rozmiaru, zamieniają się we wskaźniki przy każdej okazji, nie kopiują się tak, jak należy. Tablice, podobnie jak wektory, są tych wad pozbawione. Na przykład:

```
double* p = ad;          // Błąd: nie można stosować niejawnej konwersji na wskaźnik
double* q = ad.data(); // Dobrze: jawna konwersja

template<typename C> void printout(const C& c) // szablon funkcji
{
    for (int i = 0; i<c.size(); ++i) cout << c[i] <<'\n';
}
```

Funkcję `printout()` można wywołać zarówno przez tablicę, jak i wektor:

```
printout(ad); // wywołanie przez tablicę
vector<int> vi;
// …
printout(vi); // wywołanie przez wektor
```

Jest to prosty przykład programowania ogólnego zastosowanego w celu uzyskania dostępu do danych. Działa, ponieważ interfejsy tablic i wektorów (`size()` i indeksowanie) są takie same. Bardziej szczegółowy opis tego stylu programowania znajduje się w rozdziałach 20. i 21.

19.3.6. Dedukcja argumentów szablonu

Argumenty szablonu klasy podaje się przy tworzeniu obiektu konkretnej klasy. Na przykład:

```
array<char,1024> buf; // Dla buf T jest char, a N wynosi 1024
array<double,10> b2;  // Dla b2 T jest double, a N wynosi 10
```

Argumenty szablonu funkcji są natomiast zazwyczaj dedukowane przez kompilator na podstawie argumentów wywołania funkcji. Na przykład:

```
template<class T, int N> void fill(array<T,N>& b, const T& val)
{
    for (int i = 0; i<N; ++i) b[i] = val;
}
void f()
{
    fill(buf, 'x'); // Dla fill() T jest char, a N wynosi 1024,
                    // ponieważ to zawiera buf
    fill(b2,0.0);   // Dla fill() T jest double, a N wynosi 10,
                    // ponieważ to zawiera b2
}
```

Zapis `fill(buf, 'x')` jest technicznie skróconą wersją zapisu `fill<char, 1024>(buf, 'x')`, a `fill(b2,0)` odpowiada `fill<double,10>(b2,0)`, ale na szczęście rzadko trzeba podawać tyle szczegółów. Kompilator sam zdobędzie potrzebne mu informacje.

19.3.7. Uogólnianie wektora

Gdy uogólniliśmy nasz wektor i z klasy „`vector` liczb typu `double`" otrzymaliśmy szablon „`vector` typu `T`", nie zrewidowaliśmy definicji funkcji `push_back()`, `resize()` i `reserve()`. Musimy to teraz zrobić, ponieważ przy ich obecnej definicji w punktach 19.2.2 i 19.2.3 funkcje te przyjmują założenia odpowiadające typowi `double` niesprawdzające się dla wszystkich typów, które chcielibyśmy przechowywać w naszych wektorach:

- Co zrobimy z `vector<X>`, jeśli `X` nie będzie mieć domyślnej wartości?
- Jak zapewnić usuwanie elementów, gdy nie są już potrzebne?

Czy musimy rozwiązać te problemy? Moglibyśmy powiedzieć: „Nie próbuj tworzyć wektorów elementów typów, które nie mają domyślnej wartości" oraz „Nie używaj wektorów dla typów z destruktorami na takie sposoby, które mogą powodować problemy". Tego rodzaju ograniczenia w narzędziach przeznaczonych do użytku ogólnego denerwują użytkowników i wywołują wrażenie, że ich projektant nie przemyślał dokładnie problemu lub nie dba o swoich użytkowników. Często te podejrzenia się sprawdzają, ale projektanci biblioteki standardowej nie pozostawili takich kwiatków. Aby dorównać wektorowi z biblioteki standardowej, musimy te problemy rozwiązać.

Z typami bez wartości domyślnej możemy poradzić sobie poprzez umożliwienie użytkownikowi określenia tej wartości, gdy jest to potrzebne:

```
template<typename T> void vector<T>::resize(int newsize, T def = T());
```

To znaczy, że `T()` będzie wartością domyślną, jeśli użytkownik nie napisze inaczej. Na przykład:

```
vector<double> v1;
v1.resize(100);                 // Dodaje 100 kopii double(), tzn. 0.0
v1.resize(200, 0.0);            // Dodaje 200 kopii 0.0 — wpisanie 0.0 było zbędne
v1.resize(300, 1.0);            // Dodaje 300 kopii 1.0
struct No_default {
    No_default(int);            // Jedyny konstruktor struktury No_default
    // ...
};

vector<No_default> v2(10);      // Błąd: próba utworzenia 10 obiektów No_default()
vector<No_default> v3;
v3.resize(100, No_default(2)); // Dodaje 100 kopii No_default(2)
v3.resize(200);                 // Błąd: próba utworzenia 200 obiektów No_default()
```

Trudniejszy do rozwiązania jest problem destruktorów. W istocie musimy poradzić sobie z czymś bardzo niezręcznym — strukturą danych zawierającą część zainicjowanych i część niezainicjowanych danych. Do tej pory cały czas unikaliśmy niezainicjowanych danych i błędów, które wywołują. Teraz, jako implementatorzy typu `vector`, musimy stawić im czoła, abyśmy, jako użytkownicy, nie musieli tego robić, pisząc programy.

Najpierw musimy znaleźć sposób na pozyskiwanie i manipulowanie niezainicjowaną pamięcią. Na szczęście w bibliotece standardowej znajduje się klasa `allocator`, która dostarcza niezainicjowaną pamięć. Jej nieco uproszczona wersja wygląda następująco:

```
template<typename T> class allocator {
public:
    // ...
    T* allocate(int n);                 // Alokuje przestrzeń dla n obiektów typu T
    void deallocate(T* p, int n);       // Dealokuje n obiektów typu T, zaczynając od p

    void construct(T* p, const T& v); // Konstruuje T z wartością v w p
    void destroy(T* p);                 // Usuwa T z p
};
```

Jeśli chcesz zobaczyć wszystko, zajrzyj do książki *Język C++*[1], do punktu B.1.1 (`<memory>`) lub do biblioteki standardowej. Powyżej przedstawiliśmy cztery podstawowe operacje, które pozwalają:

- Alokować wystarczającą ilość pamięci, aby przechować obiekt typu `T` bez inicjalizacji.
- Utworzyć obiekt typu `T` w niezainicjowanej przestrzeni.
- Usunąć obiekt typu `T`, przywracając jego pamięć do niezainicjowanego stanu.
- Dealokować niezainicjowaną przestrzeń o rozmiarze odpowiadającym obiektowi typu `T`.

Oczywiście klasa `allocator` jest dokładnie tym, czego potrzebujemy do zaimplementowania funkcji `vector<T>::reserve()`. Najpierw dodamy wektorowi parametr w postaci obiektu tej klasy:

```
template<typename T, typename A = allocator<T> > class vector {
    A alloc; // Zarządza pamięcią dla elementów za pomocą funkcji allocate
    // ...
};
```

Poza dostarczeniem alokatora — i domyślnym użyciem standardowego zamiast używania operatora `new` — wszystko jest tak samo, jak wcześniej. Jako użytkownicy wektorów możemy zignorować alokatory, dopóki nie będziemy chcieli zmusić wektora, aby zarządzał pamięcią swoich elementów w jakiś nietypowy sposób. Jako implementatorzy wektora i studenci chcący pojąć fundamentalne problemy oraz nauczyć się podstawowych technik musimy dowiedzieć się, jak wektor może obchodzić się z niezainicjowaną pamięcią i prezentować swoim użytkownikom poprawnie skonstruowane obiekty. Zmiany są potrzebne tylko w składowych funkcjach wektora, które bezpośrednio operują na pamięci, np. `vector<T>::reserve()`:

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return; // Nigdy nie zmniejszaj obszaru alokacji
    T* p = alloc.allocate(newalloc);                    // Alokuje nową przestrzeń
    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]); // Kopiuje
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);        // Usuwa
    alloc.deallocate(elem,space);                       // Dealokuje starą przestrzeń
    elem = p;
    space = newalloc;
}
```

Przenosimy elementy w nowe miejsce, tworząc ich kopie w niezainicjowanej przestrzeni i usuwając oryginały. Nie możemy zastosować przypisania, ponieważ dla takich typów jak `string` oznacza ono, że obszar docelowy został już zainicjowany.

Mając funkcję `reserve()`, z łatwością możemy napisać funkcję `vector<T,A>::push_back()`:

```
template<typename T, typename A>
void vector<T,A>::push_back(const T& val)
{
    if (space==0) reserve(8);              // Zaczyna od przestrzeni dla 8 elementów
```

[1] Bjarne Stroustrup, *Język C++*, wydanie VI, Wydawnictwa Naukowo-Techniczne, Warszawa 2002.

```
    else if (sz==space) reserve(2*space); // Pobiera więcej przestrzeni
    alloc.construct(&elem[sz],val);       // Dodaje val na końcu
    ++sz;                                 // Zwiększa rozmiar
}
```

Podobnie problemów nie sprawi nam funkcja `vector<T,A>::resize()`:

```
template<typename T, typename A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) alloc.construct(&elem[i],val); // tworzenie
    for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]);     // usuwanie
    sz = newsize;
}
```

Należy zauważyć, że ze względu na to, iż niektóre typy nie mają konstruktora domyślnego, pozostawiliśmy możliwość podania wartości domyślnej dla nowych elementów.

Kolejna nowość w tym kodzie to usuwanie „nadmiarowych elementów" w przypadku zmiany rozmiaru wektora na mniejszy. Można myśleć, że destruktor ten zamienia obiekt o określonym typie na „surową pamięć".

Taka „zabawa z alokatorami" to zaawansowane i trudne programowanie. Nie podejmuj się tego, dopóki nie osiągniesz eksperckiego poziomu umiejętności.

19.4. Sprawdzanie zakresu i wyjątki

Patrzymy na dotychczasowy owoc naszej pracy i (z przerażeniem?) zauważamy, że nie sprawdzamy nigdzie zakresu. Implementacja operatora [] jest bardzo prosta:

```
template<typename T, typename A> T& vector<T,A>::operator[](int n)
{
    return elem[n];
}
```

Zatem rozważmy:

```
vector<int> v(100);
v[–200] = v[200]; // Ojej!
int i;
cin>>i;
v[i] = 999;       // Maltretowanie dowolnego miejsca w pamięci
```

Powyższy kod pomyślnie przejdzie kompilację i zadziała, uzyskując dostęp do pamięci, która nie należy do naszego wektora. To może oznaczać poważne kłopoty! W realnym programie niedopuszczalne jest pozostawienie takiego kodu. Spróbujemy poprawić naszą implementację wektora, aby pozbyć się tego problemu. Najłatwiej byłoby dodać kontrolowaną operację dostępu o nazwie `at()`:

```
struct out_of_range { /* … */ }; // Klasa do raportowania błędów zakresu

template<typename T, typename A = allocator<T> > class vector {
```

```
    // ...
    T& at(int n);                       // kontrola dostępu
    const T& at(int n) const;           // kontrola dostępu

    T& operator[](int n);               // brak kontroli dostępu
    const T& operator[](int n) const;   // brak kontroli dostępu
    // ...
};

template<typename T, typename A > T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<typename T, typename A > T& vector<T,A>::operator[](int n) // jak poprzednio
{
    return elem[n];
}
```

Teraz możemy napisać taki kod:

```
void print_some(vector<int>& v)
{
    int i = -1;
    while(cin>>i && i!= -1)
    try {
        cout << "v[" << i << "]==" << v.at(i) << "\n";
    }
    catch(out_of_range) {
        cout << "Zły indeks: " << i << "\n";
    }
}
```

Użyliśmy funkcji `at()` do uzyskania kontroli nad zakresem i przechwytujemy wyjątek `out_of_range` w przypadku niedozwolonej operacji dostępu.

Ogólnie chodzi o to, że używamy operatora indeksowania `[]`, gdy wiemy, że mamy poprawny indeks, oraz funkcji `at()`, jeśli nie może wystąpić indeks spoza dozwolonego zakresu.

19.4.1. Dygresja — uwagi projektowe

Wszystko w porządku, ale czemu nie dodaliśmy sprawdzania zakresu do funkcji `operator[]()`? Wektor w bibliotece standardowej również udostępnia kontrolowaną funkcję `at()` i niekontrolowany operator `[]`. Spróbujemy wyjaśnić, czemu tak jest. Przemawiają za tym cztery podstawowe argumenty:

1. *Zgodność* — ludzie używali niekontrolowanych operatorów indeksowania długo przed dodaniem wyjątków do języka C++.
2. *Wydajność* — można zbudować kontrolowany operator dostępu na optymalnie szybkim niekontrolowanym operatorze dostępu, ale nie można zbudować optymalnie szybkiego operatora dostępu na bazie kontrolowanego operatora dostępu.

3. *Ograniczenia* — w niektórych środowiskach wyjątki nie są tolerowane.
4. *Opcjonalne sprawdzanie* — standard nie zabrania sprawdzania zakresu w typie `vector`, a więc jeśli tego potrzebujesz, użyj takiej implementacji, która to robi.

19.4.1.1. Zgodność

Ludzie naprawdę bardzo nie lubią, gdy ich stary kod przestaje działać. Jeśli jest program składający się z miliona wierszy kodu, jego przerobienie, aby poprawnie używał wyjątków, byłoby bardzo kosztowną zabawą. Możemy argumentować, że dzięki temu kod ten byłby lepszy, ale to nie my musimy za to płacić (czasem i pieniędzmi). Ponadto osoby odpowiedzialne za utrzymanie istniejącego kodu często argumentują, że niekontrolowany kod może w zasadzie być niebezpieczny, ale ich kod został przetestowany i jest używany od wielu lat, a więc wszystkie błędy zostały już znalezione. Możemy sceptycznie odnosić się do takich argumentów, ale nikt, kto nie musiał realnie podejmować takich decyzji, nie może wypowiadać się na ten temat zbyt protekcjonalnie. Oczywiście przed wprowadzeniem do biblioteki standardowej C++ typu `vector` nie było kodu, który by z niego korzystał. Były natomiast miliony wierszy kodu korzystających z bardzo podobnych wektorów, które (jako przed-standardowe) nie wykorzystywały wyjątków. Znaczne ilości tego kodu zostały później dostosowane do standardu.

19.4.1.2. Wydajność

W pewnych ekstremalnych przypadkach (np. przy programowaniu buforów dla interfejsów sieciowych lub macierzy do wysokowydajnych obliczeń naukowych) sprawdzanie zakresu może być ciężarem. Jednak w „zwykłym programowaniu", na którym spędzamy większość czasu, koszt sprawdzania zakresu rzadko ma znaczenie. Dlatego zalecamy używanie i sami używamy implementacji typu `vector` ze sprawdzaniem zakresu, gdy tylko możemy.

19.4.1.3. Ograniczenia

Ten argument także dotyczy tylko niektórych programistów i aplikacji. W istocie dotyczy mnóstwa programistów i nie należy go lekceważyć. Jeśli jednak zaczynasz pisać program w środowisku bez ostrych wymagań czasowych (zobacz punkt 25.2.1), staraj się stosować wyjątki do obsługi błędów i używać kontrolowanych wektorów.

19.4.1.4. Opcjonalne sprawdzanie

W standardzie ISO języka C++ jest napisane, że dostęp do wektora poza zakresem nie musi mieć żadnej określonej semantyki i należy go unikać. Jest w pełni zgodne ze standardem zgłaszanie wyjątku, gdy program próbuje uzyskać dostęp poza dozwolonym zakresem. Jeśli więc chcesz, aby klasa `vector` zgłaszała wyjątki, i nie obowiązują Cię poprzednie trzy powody, używaj kontrolowanej implementacji wektora. To robimy w tej książce.

Wniosek z tego taki, że realny projekt może być bardziej zagmatwany, niż byśmy chcieli, ale są też sposoby na poradzenie sobie z tym.

19.4.2. Wyznanie na temat makr

Podobnie jak nasz typ `vector`, większość implementacji wektora biblioteki standardowej nie gwarantuje sprawdzania zakresu przez operator indeksowania `[]`, ale udostępnia robiącą to funkcję `at()`. Skąd zatem biorą się te wyjątki `std::out_of_range` w naszych programach? W istocie wybraliśmy „opcję 4." z punktu 19.4.1 — implementacja wektora nie musi kontrolować zakresu dostępu operatora `[]`, ale nie jest to zabronione. Dlatego zadbaliśmy, aby to było robione. Mogłeś używać naszej wersji testowej o nazwie `Vector`, która kontroluje operator `[]`. To właśnie robimy, gdy piszemy kod. W ten sposób zmniejszamy liczbę błędów i skracamy czas potrzebny na debugowanie przy niewielkich kosztach wydajnościowych:

```
struct Range_error : out_of_range { // ulepszony wektor zgłaszający błędy zakresu
    int index;
    Range_error(int i) :out_of_range("Błąd zakresu"), index(i) { }
};

template<typename T> struct Vector : public std::vector<T> {
    using size_type = typename std::vector<T>::size_type;
    using vector<T>::vector;        // używa konstruktorów klasy vector<T> (podrozdział 20.5)

    T& operator[](size_type i)      // zamiast return at(i);
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
    const T& operator[](size_type i) const
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
};
```

Typu `Range_error` używamy, aby udostępnić nieprawidłowy indeks do debugowania. Oparcie typu `Vector` na `std::vector` pozwoliło nam korzystać z wszystkich funkcji składowych tej standardowej struktury. Pierwsza instrukcja `using` wprowadza synonim dla operatora `size_type` struktury `std::vector` (podrozdział 20.5). Natomiast druga pozwala korzystać z wszystkich konstruktorów standardowego wektora w naszym wektorze.

Ten typ `Vector` jest przydatny w debugowaniu niebanalnych programów. Alternatywą jest użycie systematycznie sprawdzanej implementacji kompletnego wektora z biblioteki standardowej — w istocie to **może** rzeczywiście być tym, czego używasz. Nie ma żadnego sposobu na dowiedzenie się, jaki dokładnie poziom sprawdzania oferują kompilator i biblioteka (poza tym, co gwarantuje standard).

W pliku *std_lib_facilities.h* zastosowaliśmy brzydką sztuczkę (zastąpienie za pomocą makra) polegającą na przedefiniowaniu nazwy `vector`, aby znaczyła tyle, co `Vector`:

```
// Obrzydliwa sztuczka przy użyciu makra, której celem jest uzyskanie kontrolowanego wektora:
#define vector Vector
```

Oznacza to, że zawsze gdy pisaliśmy `vector`, kompilator widział `Vector`. Ta sztuczka jest taka obrzydliwa, ponieważ programista patrzący na kod nie widzi tego samego, co kompilator. W realnych programach makra są poważnym źródłem wielu niejasnych błędów (podrozdział 27.8 i punkt A.17).

To samo zrobiliśmy, aby zapewnić kontrolowany dostęp do łańcuchów.

Nie ma niestety standardowego, przenośnego i jasnego sposobu na uzyskanie sprawdzania zakresu z implementacji operatora [] wektora. Można jednak zrobić to znacznie lepiej (także dla typu `string`), niż my zrobiliśmy. Oznacza to jednak zazwyczaj konieczność podmiany implementacji biblioteki standardowej, dostosowania opcji instalacji lub grzebania w kodzie źródłowym biblioteki standardowej. Żadna z tych czynności nie jest odpowiednia dla początkującego programisty — typu `string` używaliśmy już w rozdziale 2.

19.5. Zasoby i wyjątki

Zatem klasa `vector` może zgłaszać wyjątki i zalecamy, aby każda funkcja, która nie może wykonać swojego zadania, zgłaszała wyjątek informujący o problemie wywołującego (rozdział 5.). Nadszedł czas, aby pomyśleć, jak pisać kod, w którym musimy obsługiwać wyjątki zgłaszane przez operacje klasy `vector` i inne wywoływane przez nas funkcje. Naiwna odpowiedź typu: „przechwyć wyjątek za pomocą bloku `try`, napisz komunikat o błędzie i zamknij program" jest zbyt prosta dla większości niebanalnych systemów.

Jedna z fundamentalnych zasad programowania głosi, że każdy zajęty zasób musi zostać w jakiś sposób (pośredni lub bezpośredni) zwrócony do tej części systemu, która nim zarządza. Oto kilka przykładów zasobów:

- pamięć,
- blokada,
- uchwyt do pliku,
- uchwyt do wątku,
- gniazdo,
- okno.

Podstawowa definicja zasobu mówi, że jest to coś, co można zająć i trzeba oddać (zwolnić) lub musi zostać odzyskane przez jakiegoś „zarządcę zasobów". Najprostszym przykładem jest pamięć wolna, którą możemy zajmować za pomocą operatora `new` i zwracać za pomocą operatora `delete`. Na przykład:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // zajmowanie pamięci
    // ...
    delete[] p;          // zwalnianie pamięci
}
```

W punkcie 17.4.6 zaznaczyliśmy, że trzeba pamiętać o zwalnianiu pamięci, co nie zawsze jest łatwe. Gdy doda się do tego wszystkiego wyjątki, mogą wystąpić wycieki pamięci. Powodem tego najczęściej są ignorancja i nieuwaga. Zwłaszcza taki kod jak funkcji `suspicious()`, która

bezpośrednio używa operatora `new` i uzyskany wskaźnik przypisuje do zmiennej lokalnej, należy traktować bardzo podejrzliwie.

Obiekt odpowiedzialny za zwalnianie zasobów, taki jak `vector`, nazywamy **właścicielem** lub **uchwytem** do zasobu.

19.5.1. Potencjalne problemy z zarządzaniem zasobami

Jednym z powodów do podejrzliwego traktowania pozornie niegroźnych przypisań wskaźników jak poniższe:

```
int* p = new int[s]; // zajmowanie pamięci
```

jest to, że trudno sprawdzić, czy ten operator `new` ma odpowiednik w postaci operatora `delete`. Funkcja `suspicious()` ma przynajmniej instrukcję `delete[] p;` mogącą zwolnić pamięć, ale wyobrazimy sobie kilka rzeczy, które mogłyby to uniemożliwić. Co moglibyśmy wstawić w miejsce trzech kropek, aby spowodować wyciek pamięci? Powinniśmy przedstawić takie przykłady, które dadzą do myślenia i nauczą podejrzliwie traktować taki kod. Ponadto powinny nakłaniać do docenienia prostych i wartościowych alternatywnych rozwiązań.

Może zanim dojdziemy do operatora `delete p`, nie będzie już wskazywany ten sam obiekt:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // zajmowanie pamięci
    // ...
    if (x) p = q;        // przestawienie p na inny obiekt
    // ...
    delete[] p;          // zwalnianie pamięci
}
```

Instrukcję `if (x)` zastosowaliśmy w tym kodzie po to, aby uniemożliwić zgadnięcie, czy wartość `p` została zmieniona czy nie. Może nigdy nie dojdziemy do `delete`:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // zajmowanie pamięci
    // ...
    if (x) return;
    // ...
    delete[] p;          // zwalnianie pamięci
}
```

Może nigdy nie dojdziemy do `delete`, ponieważ zgłosimy wyjątek:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // zajmowanie pamięci
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);
    // ...
    delete[] p;          // zwalnianie pamięci
}
```

Ta ostatnia możliwość interesuje nas teraz najbardziej. Wiele osób, które spotykają się z tym problemem po raz pierwszy, uznaje, że dotyczy on wyjątków, a nie zarządzania pamięcią. Gdy niepoprawnie określą główną przyczynę, opracowują rozwiązanie polegające na przechwytywaniu wyjątków:

```
void suspicious(int s, int x) // zagmatwany kod
{
    int* p = new int[s];      // zajmowanie pamięci
    vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
        // ...
    } catch (...) {           // przechwytywanie wszystkich wyjątków
        delete[] p;           // zwalnianie pamięci
        throw;                // ponowne zgłoszenie wyjątku
    }
    // ...
    delete[] p;               // zwalnianie pamięci
}
```

To rozwiązuje problem kosztem dodania pewnej ilości kodu i zduplikowania procedury zwalniającej zasoby (`delete[] p;`). Innymi słowy rozwiązanie to jest brzydkie. Co gorsza, nie stanowi dobrego uogólnienia. Wyobraź sobie zajęcie większej ilości zasobów:

```
void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int>v1;
    // ...
    int* q = new int[s];
    vector<double> v2;
    // ...
    delete[] p;
    delete[] q;
}
```

Należy zauważyć, że jeśli operator `new` nie znajdzie wystarczająco pamięci wolnej do alokowania, zgłosi wyjątek `bad_alloc` z biblioteki standardowej. Ten problem także można rozwiązać za pomocą techniki `try-catch`, ale będzie to wymagało zastosowania kilku bloków `try`, a powstały kod będzie powtarzalny i brzydki. Nie lubimy powtarzalnego i brzydkiego kodu, ponieważ „powtarzalny” oznacza problemy z utrzymaniem, a „brzydki”, że trudno go dobrze napisać, poprawnie odczytać i utrzymać.

WYPRÓBUJ

Dodaj do powyższego kodu bloki `try`, które zapewnią poprawne zwolnienie wszystkich zasobów we wszystkich przypadkach, w których mógłby zostać zgłoszony wyjątek.

19.5.2. Zajmowanie zasobów jest inicjalizacją

Na szczęście nie musimy zapychać naszego kodu skomplikowanymi strukturami `try-catch`, aby poradzić sobie z potencjalnymi wyciekami pamięci. Rozważmy:

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // …
}
```

To jest lepsze. Co ważniejsze, jest **oczywiste**, że to jest lepsze. Zasób (tutaj pamięć wolna) jest zajmowany przez konstruktor i zwalniany przez odpowiadający mu destruktor.

Ten „problem z wyjątkami" rozwiązaliśmy już przy okazji rozwiązywania problemów z wyciekami w implementacji wektora. Rozwiązanie to jest bardzo ogólne, ponieważ ma zastosowanie do wszystkich rodzajów zasobów — zajmuje się zasób w konstruktorze i zwalnia go w odpowiednim destruktorze. Do przykładów zasobów, które najlepiej w ten sposób obsługiwać, należą blokady baz danych, gniazda oraz bufory wejścia i wyjścia (typ `iostream` robi to za nas). Technikę tę często określa się niezgrabną frazą Resource Acquisition is Initialization w skrócie RAII (zajmowanie zasobów jest inicjalizacją).

Wróćmy do powyższego kodu. W jakikolwiek sposób nie opuścilibyśmy funkcji `f()`, destruktory obiektów `p` i `q` zostaną prawidłowo wywołane — ponieważ `p` i `q` nie są wskaźnikami, nie możemy do nich przypisywać, instrukcja `return` nie uniemożliwi wywołania destruktorów i żaden nie zgłosi wyjątku. Można sformułować następującą ogólną zasadę: jeśli wątek wykonawczy wychodzi z zakresu, wywoływane są destruktory wszystkich w pełni uformowanych obiektów i podobiektów. Obiekt uznaje się za w pełni utworzony, gdy jego konstruktor zakończy działanie. Zagłębianie się w szczegóły implikacji, które niosą te dwa stwierdzenia, mogłoby wywołać bóle głowy. Można po prostu powiedzieć, że konstruktory i destruktory są wywoływane, gdy są potrzebne.

W szczególności należy używać typu `vector` zamiast bezpośrednio używać operatorów `new` i `delete`, gdy potrzebna jest niestała ilość pamięci w obrębie określonego zakresu.

19.5.3. Gwarancje

Co możemy zrobić, jeśli nie da się utrzymać wektora w jednym zakresie (i jego podzakresach)? Na przykład:

```
vector<int>* make_vec()                    // Tworzy pełny wektor
{
    vector<int>* p = new vector<int>; // Alokacja w pamięci wolnej
    // … Napełnia wektor danymi. Tu może zostać zgłoszony wyjątek
    return p;
}
```

Jest to przykład często spotykanego rodzaju kodu — wywołanie funkcji do utworzenia skomplikowanej struktury danych i zwrócenie tej struktury w wyniku. Sęk w tym, że jeśli w czasie napełniania wektora wystąpi wyjątek, funkcja `make_vec()` straci nad tym wektorem kontrolę. Inny problem polega na tym, że jeśli funkcja ta z powodzeniem zakończy działanie, ktoś będzie musiał usunąć zwrócony przez nią obiekt (punkt 17.4.6).

Możemy poradzić sobie z tą możliwością zgłoszenia wyjątku, dodając blok `try`:

```
vector<int>* make_vec()                    // Tworzy napełniony wektor
{
    vector<int>* p = new vector<int>; // Alokacja w pamięci wolnej
    try {
        // Napełnia wektor danymi. Tu może zostać zgłoszony wyjątek
        return p;
    }
    catch (...) {
        delete p; // Lokalne czyszczenie
        throw;    // Ponowne zgłoszenie wyjątku, aby umożliwić wywołującemu poradzenie sobie
                  // z tym, że funkcja make_vec() nie była w stanie wykonać swojego zadania
    }
}
```

Powyższa funkcja `make_vec()` ilustruje powszechnie stosowaną technikę obsługi błędów — próbuje wykonać swoje zadanie i jeśli się jej nie uda, czyści wszelkie lokalne zasoby (tu usuwa wektor z pamięci wolnej) oraz zgłasza problem w postaci wyjątku. W tym przypadku wyjątek został zgłoszony przez jakąś inną funkcję (np. `vector::at()`). Funkcja `make_vec()` tylko zgłasza go ponownie za pomocą instrukcji `throw`. Jest to prosta i efektywna metoda obsługi błędów, którą można stosować systematycznie.

- *Podstawowa gwarancja* — zadaniem struktury `try-catch` jest zagwarantowanie, że funkcja `make_vec()` zakończy działanie z powodzeniem lub zgłosi wyjątek, nie przepuszczając żadnego wycieku zasobów. Często nazywa się to **podstawową gwarancją** (ang. *basic guarantee*). Wszelki kod wchodzący w skład programu, który może odzyskać sprawność po wyjątku, powinien spełniać warunki podstawowej gwarancji. Zapewnia to cały kod biblioteki standardowej.

- *Pełna gwarancja* — jeśli poza podstawową gwarancją funkcja zapewnia też, że wszystkie obserwowalne wartości (takie, które nie są w niej lokalne) zostaną przywrócone do pierwotnego stanu sprzed wywołania, mówi się, że funkcja taka zapewnia **pełną gwarancję** (ang. *strong guarantee*). Ten rodzaj gwarancji jest najlepszy, gdy pisze się funkcję, która albo zrobi wszystko, co powinna, albo nic poza tym, że został zgłoszony wyjątek oznaczający błąd.

- *Gwarancja niezgłaszania wyjątku* — gdybyśmy nie mogli wykonać prostych operacji bez ryzyka, że się nie powiodą i nie spowodują zgłoszenia wyjątku, nie moglibyśmy pisać kodu oferującego podstawową i pełną gwarancję. Na szczęście wszystkie wbudowane narzędzia języka C++ oferują gwarancję niezgłaszania wyjątku — innymi słowy nie mogą one zgłaszać wyjątków. Aby uniknąć zgłaszania wyjątków, należy unikać instrukcji `throw` oraz operatorów `new` i `dynamic_cast` (punkt A.5.7).

Podstawowa i pełna gwarancja są bardzo przydatne w rozpatrywaniu poprawności programów. Technika RAII jest podstawą prostej i wydajnej implementacji kodu napisanego zgodnie z tymi zasadami.

Oczywiście zawsze należy unikać niezdefiniowanych (i zazwyczaj tragicznych) operacji, takich jak dereferencja `0`, dzielenie przez `0` czy dostęp do nieistniejących elementów tablicy. Przechwytywanie wyjątków nie stanowi ochrony przed łamaniem podstawowych zasad języka programowania.

19.5.4. Obiekt unique_ptr

Funkcja `make_vec()` jest przydatnym rodzajem funkcji, która przestrzega podstawowych zasad dobrego zarządzania zasobami w obecności wyjątków. Oferuje podstawową gwarancję — powinny to robić wszystkie dobre funkcje, jeśli chcemy mieć możliwość odzyskiwania sprawności po wystąpieniu wyjątku. Jeśli nic szczególnie paskudnego nie przydarzy się danym w czasie wykonywania operacji napełniania wektora danymi, funkcja ta nawet oferuje pełną gwarancję. Nie zmienia to jednak faktu, że kod `try-catch` nadal jest nieelegancki. Rozwiązanie jest oczywiste — trzeba jakoś zastosować podejście RAII. Inaczej mówiąc, musimy utworzyć obiekt, który będzie przechowywał ten wektor `vector<int>`, aby mógł go usunąć, gdy wystąpi wyjątek. Biblioteka standardowa udostępnia specjalny taki obiekt o nazwie `auto_ptr` w nagłówku `<memory>`:

```
vector<int>* make_vec()                          // Tworzy napełniony wektor
{
    unique_ptr<vector<int>> p {new vector<int>}; // Alokacja w pamięci wolnej
    // Napełnia wektor danymi. Tu może zostać zgłoszony wyjątek
    return p.release();                          // Zwraca wskaźnik przechowywany przez p
}
```

`unique_ptr` to obiekt przechowujący wskaźnik. Inicjuje się go bezpośrednio wskaźnikiem otrzymanym od operatora `new`. Można na jego rzecz stosować operatory `->` i `*` w dokładnie taki sam sposób jak na rzecz wskaźników (np. `p->at()` lub `(*p).at(2)`). Dlatego `unique_ptr` należy traktować jako rodzaj wskaźnika. Jednak `unique_ptr` jest właścicielem obiektu, który wskazuje, to znaczy jego usunięcie powoduje usunięcie także wskazywanego obiektu. Oznacza to, że jeśli podczas wstawiania wartości do struktury `vector<int>` wystąpi wyjątek lub przedwcześnie zakończy się działanie `make_vec`, ta struktura `vector<int>` zostanie poprawnie zniszczona. Wywołanie `p.release()` pobiera wskaźnik (do `vector<int>`) z `p`, aby można go było zwrócić, oraz zeruje `p` (`nullptr`), aby jego usunięcie (na przykład w wyniku zwrotu) nie spowodowało zniszczenia czegokolwiek.

Wskaźnik `unique_ptr` bardzo upraszcza `make_vec()`, do tego stopnia, że funkcja ta jest prosta jak naiwna, ale niebezpieczna wersja. Co ważne, mając `unique_ptr`, możemy powtórzyć nasze zalecenie, aby wszystkie wprost wyrażone bloki `try` traktować podejrzliwie. Większość z nich — tak jak w `make_vec()` — można zastąpić techniką RAII.

Wersja funkcji `make_vec()` wykorzystująca `unique_ptr` jest w porządku, tyle że nadal zwraca wskaźnik, przez co wciąż trzeba pamiętać, aby go kiedyś usunąć. Problem ten rozwiązałoby zwrócenie wskaźnika `unique_ptr`:

```
unique_ptr<vector<int>> make_vec()               // utworzenie pełnego wektora
{
    unique_ptr<vector<int>> p {new vector<int>}; // alokacja pamięci wolnej
    // . . . napełnienie wektora danymi; przy tym może wystąpić wyjątek. . .
    return p;
}
```

Obiekt `unique_ptr` pod wieloma względami przypomina zwykły wskaźnik, ale obowiązuje go istotne ograniczenie: nie można przypisać jednego `unique_ptr` do innego, aby otrzymać dwa wskaźniki `unique_ptr` do tego samego obiektu. Jest to konieczne, ponieważ inaczej nie byłoby wiadomo, który wskaźnik jest właścicielem wskazywanego obiektu i który odpowiada za jego usunięcie. Na przykład:

```
void no_good()
{
    unique_ptr<X> p { new X };
    unique_ptr<X> q {p}; // błąd: na szczęście
    // ...
}                                // tu zarówno p, jak i q usuwają X
```

Jeśli potrzebny jest „inteligentny" wskaźnik, który gwarantuje usunięcie i może być kopiowany, należy użyć `shared_ptr` (punkt B.6.5). Jednak ten wskaźnik jest „cięższy", ponieważ zawiera licznik pozwalający dopilnować, aby ostatnia kopia zniszczyła wskazywany obiekt.

Wskaźnik `unique_ptr` ma tę ciekawą właściwość, że nie wiąże się z dodatkowym narzutem w porównaniu do zwykłego wskaźnika.

19.5.5. Zwrot przez przeniesienie

Technika zwracania dużych ilości informacji przez umieszczenie ich w pamięci wolnej i zwrócenie wskaźnika do nich cieszy się dużą popularnością. Jest też źródłem wielu komplikacji i licznych błędów związanych z zarządzaniem pamięcią: kto usuwa wskaźnik do pamięci wolnej zwrócony przez funkcję? Czy można mieć pewność, że wskaźnik do obiektu w pamięci wolnej zostanie poprawnie usunięty w przypadku wystąpienia wyjątku? Jeśli nie będziemy bardzo skrupulatnie zarządzać wskaźnikami (albo używać „inteligentnych" wskaźników, takich jak `unique_ptr` i `shared_ptr`), na to pytanie odpowiemy: „chyba tak", a to za mało.

Na szczęście w przypadku wektorów problem ten rozwiązaliśmy przez dodanie operacji przenoszenia: wystarczy użyć konstruktora przenoszącego, aby przenieść prawo własności elementów poza funkcję. Na przykład:

```
vector<int> make_vec() // tworzy pełny wektor
{
    vector<int> res;
    // … wstawia dane do wektora; przy tym może wystąpić wyjątek…
    return res;          // konstruktor przenoszący efektywnie przenosi własność
}
```

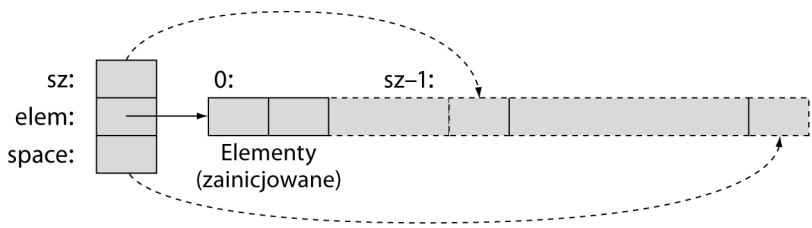
Ta wersja funkcji `make_vec()` jest najprostsza z dotychczasowych i możemy ją polecić. Rozwiązanie z wykorzystaniem przenoszenia można zastosować we wszystkich kontenerach, a także do wszystkich uchwytów do zasobów. Strumień `fstream` na przykład przy jej użyciu śledzi uchwyty do plików. Jest to proste i ogólne rozwiązanie. Uchwyty do zasobów upraszczają kod i eliminują poważne źródło błędów. Powodowany przez nie narzut czasu wykonywania w porównaniu z bezpośrednim użyciem wskaźników jest nieistotny lub bardzo mały i przewidywalny.

19.5.6. Technika RAII dla wektora

Użycie inteligentnego wskaźnika, jak `unique_ptr`, może wydawać się trochę słabo przemyślanym rozwiązaniem. Skąd można mieć pewność, że zauważyło się wszystkie wskaźniki, które wymagają ochrony? Skąd można mieć pewność, że wszystkie wskaźniki do obiektów, które nie powinny zostać usunięte na końcu zakresu, zostały uwolnione? Rozważmy funkcję `reserve()` z punktu 19.5.6:

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return; // Nigdy nie zmniejszaj obszaru alokacji
    T* p = alloc.allocate(newalloc);                    // Alokuje nową przestrzeń

    for (int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]); // Kopiuje

    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);        // Usuwa

    alloc.deallocate(elem,space);                       // Dealokuje starą przestrzeń
    elem = p;
    space = newalloc;
}
```

Należy zauważyć, że operacja kopiowania starego elementu, `alloc.construct(&p[i],elem[i])`, może zgłosić wyjątek. Zatem `p` jest przykładem problemu, przed którym ostrzegaliśmy w punkcie 19.5.1. Au! Moglibyśmy zastosować nasze rozwiązanie z użyciem wskaźnika `unique_ptr`. Lepszym rozwiązaniem jest przemyśleć wszystko dokładnie i zdać sobie sprawę, że „pamięć dla wektora" jest rodzajem zasobu. Oznacza to, że można zdefiniować klasę `vector_base` do reprezentowania podstawowej koncepcji, z której korzystaliśmy cały czas. Poniższy rysunek przedstawia trzy elementy definiujące wykorzystanie pamięci przez wektor:



Realizacja tego w kodzie jest następująca (po dodaniu dla uzupełnienia alokatora):

```
template<typename T, typename A>
struct vector_base {
    A alloc;    // alokator
    T* elem;    // początek alokacji
    int sz;     // liczba elementów
    int space; // ilość alokowanej przestrzeni

    vector_base(const A& a, int n)
        : alloc{a}, elem{alloc.allocate(n)}, sz{n}, space{n}{ }
    ~vector_base() { alloc.deallocate(elem,space); }
};
```

Należy zauważyć, że `vector_base` operuje na pamięci, a nie obiektach (o określonym typie). W naszym wektorze możemy wykorzystać tę klasę do przechowywania obiektów pożądanego typu. W zasadzie `vector` jest wygodnym interfejsem do `vector_base`:

```
template<typename T, typename A = allocator<T> >
class vector : private vector_base<T,A> {
public:
    // ...
};
```

Następnie można uprościć i poprawić funkcję `reserve()`:

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=this->space) return;          // Nigdy nie zmniejszaj obszaru dealokacji,
    vector_base<T,A> b(this->alloc,newalloc);   // Alokacja nowej przestrzeni
    for (int i=0; i<this->sz; ++i) this->alloc.construct(&b.elem[i],this->elem[i]);
                                                // Kopiuje
    for (int i=0; i<this->sz; ++i) this->alloc.destroy(&this->elem[i]); // Usuwa stary
    swap< vector_base<T,A> >(*this,b);          // Zamienia reprezentacje
}
```

Kopie elementów wektora `b` utworzyliśmy za pomocą funkcji `uninitialized_copy` z biblioteki standardowej, ponieważ poprawnie obsługuje ona wyjątki zgłaszane przez konstruktor kopiujący oraz prościej jest wywołać funkcję niż napisać pętlę. Po wyjściu z funkcji `reserve()` stara alokacja zostaje automatycznie zwolniona przez destruktor klasy `vector_base` — jeśli operacja kopiowania się powiodła. Jeżeli natomiast wyjście to zostanie spowodowane zgłoszeniem wyjątku przez operację kopiowania, nowa alokacja zostanie zwolniona. Funkcja `swap()` to algorytm z biblioteki standardowej (nagłówek `<algorithm>`), który zamienia wartości dwóch obiektów. Zastosowaliśmy instrukcję `swap< vector_base<T,A>> ↪(*this,b)` zamiast prostszej `swap(*this,b)`, ponieważ `*this` i `b` są różnych typów (odpowiednio `vector` i `vector_base`), przez co musieliśmy dokładnie określić, która specjalizacja algorytmu zamieniającego nas interesuje. Analogicznie, aby odwołać się do składowej klasy bazowej `vector_base<T,A>` ze składowej klasy pochodnej `vector<T,A>`, np. `vector_base<T,A>::reserve()`, musimy jawnie użyć operatora `->`. Analogicznie musimy wprost użyć `this–>` przy odwołaniu do składowej klasy bazowej `vector_base<T,A>` ze składowej klasy pochodnej `vector<T,A>`, na przykład `<T,A>::reserve()`.

WYPRÓBUJ

Zmodyfikuj `reserve`, aby wykorzystywała `unique_ptr`. Pamiętaj o uwolnieniu wskaźnika przed zwrotem. Porównaj swoje rozwiązanie z rozwiązaniem, w którym wykorzystano `vector_base`. Przemyśl, którą wersję łatwiej napisać i którą łatwiej napisać poprawnie.

Ćwiczenia

1. Zdefiniuj strukturę szablonową `template<typename T> struct S {T val;};`.
2. Dodaj konstruktor, aby można było dokonywać inicjalizacji przy użyciu `T`.
3. Zdefiniuj zmienne typów `S<int>`, `S<char>`, `S<double>`, `S<string>` oraz `S<vector<int>>`. Zainicjuj je dowolnymi wartościami.
4. Odczytaj te wartości i wydrukuj je.
5. Dodaj funkcję szablonową `get()` zwracającą referencję do `val`.
6. Umieść definicję funkcji `get()` poza strukturą.
7. Uczyń `val` składową prywatną.
8. Wykonaj ćwiczenie 4. jeszcze raz przy użyciu funkcji `get()`.
9. Dodaj funkcję szablonową `set()` umożliwiającą zmianę wartości `val`.
10. Zastąp funkcję `set()` funkcją `S<T>::operator=(const T&)`. Podpowiedź: o wiele prościej niż w punkcie 19.2.5.
11. Utwórz stałą i niestałą wersję funkcji `get()`.
12. Zdefiniuj funkcję `template<typename T> read_val(T& v)` wczytującą dane ze strumienia `cin` do `v`.
13. Użyj funkcji `read_val()` do wczytania wszystkich zmiennych z ćwiczenia 3. z wyjątkiem `S<vector<int>>`.
14. Bonus: zdefiniuj operatory wejścia i wyjścia (`>>` i `<<`) dla struktur `vector<T>`. Zarówno na wejściu, jak i na wyjściu używaj formatu `{ val, val, val }`. To pozwoli funkcji `read_val()` obsługiwać także zmienne typu `S<vector<int>>`.

Pamiętaj o przeprowadzeniu testów po każdym etapie.

Powtórzenie

1. Po co miałoby się zmieniać rozmiar wektora?
2. Do czego mogłaby się przydać możliwość przechowywania elementów różnych typów w różnych wektorach?
3. Dlaczego nie można po prostu zawsze definiować wektora o wystarczająco dużych rozmiarach, aby wystarczył do wszelkich zastosowań?
4. Ile zapasowej przestrzeni alokuje się dla nowego wektora?
5. Kiedy konieczne jest skopiowanie elementów wektora w nowe miejsce?
6. Które operacje wektora mogą zmienić rozmiar wektora po jego utworzeniu?
7. Co jest wartością wektora po skopiowaniu?
8. Jakie dwie operacje definiują kopiowanie w wektorze?
9. Jak odbywa się domyślne kopiowanie obiektów klas?
10. Co to jest szablon?
11. Jakie są dwa najbardziej użyteczne typy argumentów szablonowych?
12. Co to jest programowanie ogólne?
13. Czym różni się programowanie ogólne od obiektowego?
14. Czym różni się tablica (`array`) od wektora (`vector`)?
15. Czym różni się typ `array` od wbudowanej tablicy?

16. Czym różni się funkcja `resize()` od funkcji `reserve()`?
17. Co to jest zasób? Podaj definicję i kilka przykładów.
18. Co to jest wyciek zasobów?
19. Co to jest RAII? Jaki problem rozwiązuje?
20. Do czego służy wskaźnik `unique_ptr`?

Terminologia

`#define`	ponowne zgłoszenie wyjątku	`this`
`at()`	`push_back()`	`throw;`
gwarancje	RAII	uchwyt
konkretyzacja	`resize()`	`unique_ptr`
makro	samoprzypisanie	właściciel
parametr szablonu	`shared_ptr`	wyjątek
pełna gwarancja	specjalizacja	zasób
podstawowa gwarancja	szablon	

Praca domowa

Do każdego ćwiczenia utwórz i przetestuj (z wyświetleniem wyniku) kilka obiektów zdefiniowanej klasy, aby pokazać, że opracowany projekt i jego implementacja rzeczywiście robią to, co Twoim zdaniem powinny. W przypadkach, w których mogą wystąpić wyjątki, może być konieczne dokładne przemyślenie, gdzie mogą wystąpić błędy.

1. Napisz funkcję szablonową `f()` dodającą element jednego wektora `vector<T>` do elementów innego. Na przykład wywołanie `f(v1,v2)` powinno wykonać operację `v1[i]+=v2[i]` dla każdego elementu wektora `v1`.
2. Napisz funkcję szablonową pobierającą jako argumenty `vector<T> vt` i `vector<U> vu` i zwracającą sumę wszystkich działań `vt[i]*vu[i]`.
3. Napisz klasę szablonową `Pair` mogącą przechowywać pary wartości dowolnego typu. Użyj jej do zaimplementowania prostej tablicy symboli, jak ta, której użyliśmy w implementacji kalkulatora (punkt 7.8).
4. Zmodyfikuj klasę `Link` z punktu 17.9.3, aby była szablonem z typem wartości jako argumentem szablonowym. Następnie jeszcze raz wykonaj zadanie 13. z rozdziału 17. przy użyciu `Link<God>`.
5. Zdefiniuj klasę `Int` mającą jedną składową klasy `int`. Zdefiniuj konstruktory, przypisania oraz operatory `+`, `-`, `*` i `/`. Przeprowadź testy i wprowadź niezbędne poprawki (np. zdefiniuj operatory `<<` i `>>`, aby wygodnie wykonywać operacje wejścia i wyjścia).
6. Powtórz poprzednie zadanie z klasą `Number<T>`, gdzie `T` może być dowolnym typem numerycznym. Spróbuj dodać operator `%` i sprawdź, co się stanie, jeśli zostanie on użyty na rzecz typów `Number<double>` i `Number<int>`.
7. Wypróbuj swoje rozwiązanie zadania 2. na kilku obiektach typu `Number`.
8. Zaimplementuj alokator (punkt 19.3.7) przy użyciu podstawowych funkcji alokacyjnych `malloc()` i `free()` (punkt B.11.4). Opracuj kilka prostych przypadków testowych przy użyciu wektora z końca podrozdziału 19.4. Podpowiedź: poszukaj informacji na temat „placement new” i „jawnego wywołania konstruktora” w kompletnym opisie języka C++.

9. Ponownie zaimplementuj funkcję `vector::operator=()` (punkt 19.2.5) przy użyciu alokatora (punkt 19.3.7), aby dobrze zarządzała pamięcią.
10. Zaimplementuj prosty wskaźnik `unique_ptr` obsługujący tylko konstruktor, destruktor, `->`, `*` oraz `release()`. W szczególności nie próbuj zaimplementować przypisania i konstruktora kopiującego.
11. Zaprojektuj i zaimplementuj wskaźnik `counted_ptr<T>` będący typem przechowującym wskaźnik na obiekt typu `T` i wskaźnik na „licznik użyć" (liczba typu `int`) współdzielony przez wszystkie wskaźniki `counted_ptr` na ten sam obiekt typu `T`. Wskaźnik użyć powinien przechowywać liczbę wskaźników `counted_ptr` wskazujących na dany typ `T`. Niech konstruktor `counted_ptr` alokuje obiekt `T` i wskaźnik użyć w pamięci wolnej. Niech konstruktor `counted_ptr` przyjmuje argument służący jako wartość początkowa elementów `T`. Gdy ostatni wskaźnik `counted_ptr` dla `T` zostanie usunięty, destruktor `counted_ptr` powinien usunąć (`delete`) `T`. Nadaj `counted_ptr` operacje pozwalające używać go jako wskaźnika. Jest to przykład inteligentnego wskaźnika, który gwarantuje, że obiekt nie zostanie usunięty, dopóki jest używany choćby przez jednego użytkownika. Napisz zestaw przypadków użycia `counted_ptr`, używając go jako argumentu w wywołaniach, elementu kontenerów itp.
12. Zdefiniuj klasę `File_handle` z konstruktorem przyjmującym argument łańcuchowy, otwierający plik w konstruktorze i zamykający go w destruktorze.
13. Napisz klasę `Tracer`, której konstruktor i destruktor drukują łańcuchy. Podaj te łańcuchy jako argumenty konstruktora. Użyj jej, aby zobaczyć, gdzie obiekty zarządzania RAII się sprawdzą (tzn. poeksperymentuj z obiektami klasy `Tracer` jako obiektami lokalnymi, składowymi, globalnymi, alokowanymi za pomocą operatora `new` itd.). Następnie dodaj konstruktor kopiujący i przypisanie kopiujące, aby móc używać obiektów klasy `Tracer` do sprawdzania, kiedy jest wykonywane kopiowanie.
14. Utwórz GUI i elementy graficzne dla gry w polowanie na Wumpusa z zadań pracy domowej w rozdziale 18. Przyjmuj dane za pomocą pola tekstowego i wyświetlaj w oknie mapę części jaskini, która jest znana graczowi.
15. Zmodyfikuj program z poprzedniego zadania, aby umożliwić użytkownikowi oznaczanie pomieszczeń zgodnie z wiedzą faktyczną i przewidywaną, np. „może być nietoperz" i „otchłań".
16. Czasami dobrze jest, aby pusty wektor był jak najmniejszy. Na przykład ktoś może często korzystać z wektorów `vector<vector<vector<int>>>`, w których większość elementów wektorowych jest pusta. Zdefiniuj taki wektor, aby `sizeof(vector<int>)==sizeof(int*)`, tzn. aby sam wektor zawierał tylko wskaźnik do reprezentacji zawierającej elementy, liczbę elementów i wskaźnik `space`.

Podsumowanie

Szablony i wyjątki to niezwykle potężne narzędzia języka programowania. Wspierają bardzo elastyczne techniki programistyczne — głównie poprzez umożliwienie rozdzielenia poszczególnych problemów, a więc pracy nad jednym problemem na raz. Na przykład za pomocą szablonów można zdefiniować kontener, jak np. `vector`, oddzielnie od definicji typu elementów. Analogicznie za pomocą wyjątków można pisać kod wykrywający i zgłaszający błędy niezależnie od kodu, który je obsługuje. W podobnym świetle można spostrzegać trzeci ważny temat poruszony w tym rozdziale — zmianę rozmiaru wektora. Funkcje `push_back()`, `resize()` i `reserve()` pozwalają oddzielić definicję wektora od specyfikacji jego rozmiaru.

Programowanie w C++

Wykorzystaj wiedzę BJARNE STROUSTRUPA *i pisz profesjonalne programy w C++!*

Jeśli zależy Ci na tym, aby zdobyć rzetelną wiedzę i perfekcyjne umiejętności programowania z użyciem języka C++, powinieneś uczyć się od wybitnego eksperta i twórcy tego języka — Bjarne Stroustrupa, który zaprojektował i jako pierwszy zaimplementował C++. Podręcznik, który trzymasz w ręku, daje Ci szansę odkrycia wszelkich tajników tego języka, obszernie opisanego w międzynarodowym standardzie i obsługującego najważniejsze techniki programistyczne. C++ umożliwia pisanie wydajnego i eleganckiego kodu, a większość technik w nim stosowanych można przenieść do innych języków programowania.

Książka *Programowanie w C++. Teoria i praktyka* zawiera szczegółowy opis pojęć i technik programistycznych, a także samego języka C++ oraz przykłady kodu. Znajdziesz tu również omówienie zagadnień zaawansowanych, takich jak przetwarzanie tekstu i testowanie. Dowiesz się, na czym polega wywoływanie funkcji przeciążonych i dopasowywanie wyrażeń regularnych. Zobaczysz też, jaki powinien być standard kodowania. Poznasz sposoby projektowania klas graficznych i systemów wbudowanych, tajniki implementacji, wykorzystywania funkcji oraz indywidualizacji operacji wejścia i wyjścia. Dzięki temu przewodnikowi wprost od mistrza nauczysz się pisać doskonałe, wydajne i łatwe w utrzymaniu programy.

- *Techniki programistyczne*
- *Infrastruktura algorytmiczna*
- *Biblioteka standardowa C++*
- *Instrukcje sterujące i obsługa błędów*
- *Implementacja i wykorzystanie funkcji*
- *Kontrola typów*
- *Interfejsy klas*
- *Indywidualizacja operacji wejścia i wyjścia*
- *Projektowanie klas graficznych*
- *Wektory i pamięć wolna*
- *Kontenery i iteratory*
- *Programowanie systemów wbudowanych*
- *Makra*

DR BJARNE STROUSTRUP zaprojektował i jako pierwszy zaimplementował język C++. Kieruje katedrą informatyki w College of Engineering na Texas A&M University, jest członkiem amerykańskiej National Academy of Engineering oraz AT&T. Jest także założycielem i członkiem komisji standaryzacyjnej ISO do spraw języka C++.

ISBN 978-83-283-6312-0
9 788328 363120
Cena: 149,00 zł
Addison Wesley