helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

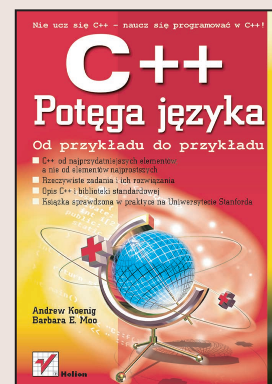
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Potęga języka. Od przykładu do przykładu

Autorzy: Andrew Koenig, Barbara E. Moo
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-374-9
Tytuł oryginału: Accelerated C++. Practical Programming by Example
Format: B5, stron: 422
Przykłady na ftp: 122 kB



Książka ta ma pomóc Czytelnikowi w szybkim nauczeniu się języka C++ poprzez pisanie w nim przydatnych programów. Ta strategia wydaje się oczywista, jednak jest odmienna od powszechnie przyjętej metodologii nauczania. Autorzy nie będą uczyć Cię języka C, choć wielu uważa, że jest to niezbędne. W prezentowanych przykładach od razu wykorzystane zostaną wysokopoziomowe struktury, a prezentacja sposobu ich zastosowania będzie często wyprzedzać omówienie ich fundamentów. Dzięki takiemu podejściu zaczniesz szybko pisać programy wykorzystujące idiomy C++.

Zastosowany w książce schemat autorzy wypróbowali podczas kursów prowadzonych na Uniwersytecie Stanforda, na których studenci uczą się pisać programy już na pierwszych zajęciach.

Poznaj:

- Podstawowe cechy C++
- Operacje na ciągach
- Pętle i liczniki
- Przetwarzanie danych „porcja po porcji”
- Organizację programów i danych
- Kontenery sekwencyjne i analiza ciągów tekstowych
- Algorytmy biblioteki standardowej
- Kontenery asocjacyjne
- Funkcje uogólnione i definiowanie własnych typów
- Zarządzanie pamięcią i niskopoziomowymi strukturami danych
- Półautomatyczne zarządzanie pamięcią
- Programowanie zorientowane obiektowo

Andrew Koenig jest członkiem działu badającego systemy oprogramowania w Shannon Laboratory firmy AT&T oraz redaktorem projektu komitetów standaryzacyjnych języka C++. Jako programista z trzydziestoletnim stażem (piętnaście lat poświęconych C++) opublikował ponad 150 artykułów i wygłosił mnóstwo odczytów o tym języku.

Barbara E. Moo jest konsultantką z dwudziestoletnim doświadczeniem programistycznym. Pracując przez 15 lat dla AT&T współtworzyła jeden z pierwszych komercyjnych produktów tworzonych w języku C++, zarządzała projektem pierwszego kompilatora C++ firmy AT&T i kierowała rozwojem uznanego systemu AT&T WorldNet Internet Service. Jest współautorką książki „Ruminations on C++”. Wykłada na całym świecie.

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

Spis treści

Przedmowa **9**

Wstęp **15**
Zaczynamy 15
W.1. Komentarze 16
W.2. Dyrektywa #include 16
W.3. Funkcja main 16
W.4. Nawiasy klamrowe 17
W.5. Wykorzystanie biblioteki standardowej do realizacji operacji wyjścia 17
W.6. Instrukcja return 18
W.7. Nieco głębsza analiza 19
W.8. Podsumowanie 21

Rozdział 1. Operacje na ciągach **25**
1.1. Wejście programu 25
1.2. Ozdabianie powitania 28
1.3. Podsumowanie 32

Rozdział 2. Pętle i liczniki **37**
2.1. Zadanie 37
2.2. Ogólna struktura programu 38
2.3. Wypisywanie nieznanej z góry liczby wierszy 38
2.4. Wypisywanie wierszy 43
2.5. Połączenie fragmentów programu 48
2.6. Zliczanie 52
2.7. Podsumowanie 54

Rozdział 3. Przetwarzanie porcji danych **61**
3.1. Obliczanie średniej ocen z egzaminów 61
3.2. Mediana zamiast średniej 68
3.3. Podsumowanie 77

Rozdział 4. Organizacja programów i danych **81**
4.1. Organizacja przetwarzania 82
4.2. Organizacja danych 93
4.3. Połączenie fragmentów programu 98
4.4. Podział programu wyznaczającego oceny semestralne 101
4.5. Ostateczna wersja programu 103
4.6. Podsumowanie 105

Rozdział 5. Kontenery sekwencyjne i analiza ciągów tekstowych 109
5.1. Dzielenie studentów na grupy 109
5.2. Iteratory 114
5.3. Wykorzystanie iteratorów w miejsce indeksów 118
5.4. Ulepszanie struktury danych pod kątem większej wydajności 120
5.5. Kontener typu list 121
5.6. Wracamy do ciągów typu string 124
5.7. Testowanie funkcji split 127
5.8. Kolekcjonowanie zawartości ciągu typu string 129
5.9. Podsumowanie 134

Rozdział 6. Korzystanie z algorytmów biblioteki standardowej 141
6.1. Analiza ciągów znakowych 142
6.2. Porównanie metod wyznaczania ocen 151
6.3. Klasyfikacja studentów — podejście drugie 159
6.4. Algorytmy, kontenery, iteratory 163
6.5. Podsumowanie 164

Rozdział 7. Stosowanie kontenerów asocjacyjnych 169
7.1. Kontenery obsługujące szybkie wyszukiwanie 169
7.2. Zliczanie wyrazów 171
7.3. Generowanie tabeli odsyłaczy 173
7.4. Generowanie prostych zdań 176
7.5. Słowo o wydajności 184
7.6. Podsumowanie 185

Rozdział 8. Pisanie funkcji uogólnionych 189
8.1. Czym jest funkcja uogólniona? 189
8.2. Niezależność od struktur danych 194
8.3. Iteratory wejściowe i wyjściowe 202
8.4. Wykorzystanie iteratorów dozwiększeniaelastycznościprogramów 204
8.5. Podsumowanie 205

Rozdział 9. Definiowanie własnych typów 209
9.1. Rewizja typu Student_info 209
9.2. Typy definiowane przez użytkownika 210
9.3. Ochrona składowych 214
9.4. Klasa Student_info 219
9.5. Konstruktory 219
9.6. Stosowanie klasy Student_info 222
9.7. Podsumowanie 223

Rozdział 10. Zarządzanie pamięcią i niskopoziomowymi strukturami danych 227
10.1. Wskaźniki i tablice 228
10.2. Literały łańcuchowe — powtórka 235
10.4. Argumenty funkcji main 238
10.5. Odczyt i zapis plików 239
10.6. Trzy tryby zarządzania pamięcią 242
10.7. Podsumowanie 245

Rozdział 11. Definiowanie abstrakcyjnych typów danych 249
11.1. Klasa Vec 249
11.2. Implementacja klasy Vec 250
11.3. Kontrola operacji kopiowania 258
11.4. Pamięć dynamiczna w klasie Vec 267
11.5. Elastyczne zarządzanie pamięcią 269
11.6. Podsumowanie 275

Rozdział 12. Obiekty typów definiowanych przez użytkownika jako wartości 279
12.1. Prosta klasa ciągów 280
12.2. Konwersje automatyczne 281
12.3. Operacje wykonywane na klasie Str 283
12.4. Ryzykowne konwersje 290
12.5. Operatory konwersji 292
12.6. Konwersje a zarządzanie pamięcią 293
12.7. Podsumowanie 295

Rozdział 13. Dziedziczenie i wiązanie dynamiczne 299
13.1. Dziedziczenie 299
13.2. Polimorfizm i funkcje wirtualne 305
13.3. Rozwiązanie zadania z użyciem dziedziczenia 310
13.4. Prosta klasa manipulatora 316
13.5. Stosowanie klasy manipulatora 321
13.6. Niuanse 323
13.7. Podsumowanie 324

Rozdział 14. Automatyczne (prawie) zarządzanie pamięcią 329
14.1. Manipulatory kopiujące obiekty docelowe 330
14.2. Manipulatory zliczające odwołania 337
14.3. Manipulatory z opcją współużytkowania danych 340
14.4. Ulepszanie manipulatorów 342
14.5. Podsumowanie 346

Rozdział 15. Obrazki znakowe — podejście drugie 347
15.1. Projekt 348
15.2. Implementacja 357
15.3. Podsumowanie 368

Rozdział 16. Co dalej? 371
16.1. Korzystaj z posiadanych narzędzi 371
16.2. Pogłębiaj wiedzę 373

Dodatek A Tajniki języka 375
A.1. Deklaracje 375
A.2. Typy 380
A.3. Wyrażenia 388
A.4. Instrukcje 391

Dodatek B Opis biblioteki 395
B.1. Wejście i wyjście 396
B.2. Kontenery i iteratory 399
B.3. Algorytmy 411

Skorowidz 417

Rozdział 6.
Korzystanie z algorytmów biblioteki standardowej

W rozdziale 5. widzieliśmy, że wiele operacji wykonywanych na kontenerach da się zastosować do więcej niż jednego typu kontenera. Przykładowo, typy `vector`, `string` i `list` pozwalają na realizację operacji wstawiania elementów za pośrednictwem metody `insert` i usuwania elementów metodą `erase`. Operacje te realizowane są za pośrednictwem identycznego we wszystkich trzech przypadkach interfejsu. Dzięki temu wiele z operacji charakterystycznych dla kontenerów da się wykonać również wobec obiektów typu `string`.

Każdy kontener — również klasa `string` — udostępnia skojarzone z danym typem kontenera klasy iteratorów, za pośrednictwem których można wybierać elementy z kontenera. Tu znów biblioteka standardowa troszczy się o to, aby każdy z udostępnianych iteratorów pozwalała na realizację pewnych operacji za pośrednictwem wspólnego dla wszystkich iteratorów interfejsu. Dla przykładu, operator ++ daje się zastosować wobec iteratora dowolnego typu i zawsze oznacza przesunięcie iteratora do następnego elementu kontenera; operator * służy z kolei do odwoływania się do elementu skojarzonego z iteratorem, niezależnie od typu tego ostatniego.

W bieżącym rozdziale zobaczymy, w jaki sposób biblioteka standardowa wykorzystuje koncepcję wspólnego interfejsu w udostępnianiu zbioru tak zwanych standardowych algorytmów. Korzystając z tych algorytmów, unikamy pisania i przepisywania wciąż tego samego kodu dla różnych struktur danych. Co ważniejsze, możemy dzięki nim pisać programy prostsze i mniejsze niż gdybyśmy wszystkie algorytmy implementowali sami — niekiedy różnica rozmiarów i prostoty oprogramowania w wyniku zastosowania algorytmów standardowych może być ogromna.

Algorytmy, podobnie jak iteratory i kontenery, również wykorzystują konwencję spójnego interfejsu. Spójność ta pozwala nam ograniczyć naukę do kilku algorytmów i na zastosowanie zdobytych doświadczeń w pracy z pozostałymi algorytmami. W tym rozdziale do rozwiązywania problemów związanych z przetwarzaniem ciągów typu `string` i ocen studentów wykorzystamy kilka standardowych algorytmów. Przy okazji nauczymy się korzystać z większości kluczowych koncepcji algorytmów biblioteki standardowej.

O ile w tekście nie będzie zaznaczone inaczej, wszystkie prezentowane w tym rozdziale algorytmy definiowane będą w nagłówku <algorithm>.

6.1. Analiza ciągów znakowych

W §5.8.2/132 wykorzystywaliśmy do konkatenacji dwóch obrazków znakowych następującą pętlę:

```
for (vector<string>::const)iterator it = bottom.begin();
     it != bottom.end(); ++it)
    ret.push_back(*it);
```

Stwierdziliśmy, że pętla ta odpowiada czynności wstawienia kopii elementów kontenera bottom na koniec kontenera ret, która to czynność w przypadku kontenerów typu vector może zostać zrealizowana bezpośrednio:

```
ret.insert(ret.end(), bottom.begin(), bottom.end());
```

Niniejsze zadanie da się jednak rozwiązać jeszcze bardziej ogólnie — można bowiem oddzielić kopiowanie elementów od ich wstawiania na koniec kontenera, jak poniżej:

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

W tym przykładzie copy jest algorytmem uogólnionym, a back_inserter jest przykładem adaptora.

Algorytm uogólniony to algorytm, który nie jest częścią definicji jakiegokolwiek konkretnego kontenera, a o sposobie dostępu do przetwarzanych danych decyduje na podstawie typów argumentów. Algorytmy uogólnione biblioteki standardowej przyjmują zwykle w liście argumentów wywołania iteratory, za pośrednictwem których odwołują się do manipulowanych elementów w kontenerach źródłowych. Dla przykładu, algorytm copy przyjmuje trzy iteratory, którym przypiszemy nazwy begin, end i out i kopiuje elementy w zakresie [begin, end) do miejsca wskazywanego przez out, w miarę potrzeby rozszerzając kontener docelowy. Innymi słowy, konstrukcja:

```
copy(begin, end, out);
```

jest co do efektu równoważna konstrukcji:

```
while (begin != end)
    *out++ = *begin++;
```

z tym, że powyższa instrukcja while zmienia wartości iteratorów, a algorytm copy tego nie robi.

Zanim przejdziemy do omawiania adaptorów iteratorów, powinniśmy zwrócić uwagę na wykorzystanie w powyższej pętli operatora inkrementacji w postaci ***przyrostkowej***. Operatory przyrostkowe różnią się od przedrostkowych tym, że wyrażenie begin++ zwraca kopię oryginalnej wartości begin, a zwiększenie wartości begin stanowi efekt uboczny operacji. Innymi słowy, kod:

```
it = begin++;
```

jest równoważny kodowi:

```
it = begin;
++begin;
```

Operator inkrementacji ma priorytet identyczny z priorytetem operatora *; oba te operatory są też operatorami łącznymi prawostronnie, co oznacza, że `*out++` powinno być interpretowane jako `*(out++)`. Stąd zapis:

```
*out++ = *begin++;
```

odpowiada zapisowi:

```
{ *out = *begin; ++out; ++begin; }
```

Wróćmy teraz do ***adaptorów iteratorów***, które są, zasadniczo rzecz biorąc, funkcjami zwracającymi iteratory, których właściwości są związane z argumentami wywołania. Adaptory iteratorów są definiowane w nagłówku `<iterator>`. Najpowszechniej wykorzystywanym adaptorem jest `back_inserter`, który przyjmuje jako argument wywołania kontener i zwraca iterator, który zastosowany w operacji wstawiania wskazuje na miejsce kolejnego elementu. Przykładowo więc `back_inserter(ret)` jest iteratorem, który wykorzystany jako iterator elementu docelowego operacji wstawiania spowoduje wstawienie elementu na koniec kontenera. Stąd:

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

kopiuje wszystkie elementy kontenera `bottom`, dołączając ich kopie na koniec kontenera `ret`. Po zakończeniu działania funkcji rozmiar kontenera `ret` zostanie zatem zwiększony o wielkość `bottom.size()`.

Zauważmy, że nie można wykonać wywołania:

```
// błąd — ret nie jest iteratorem
copy(bottom.begin(), bottom.end(), ret);
```

a to dlatego, że trzecim parametrem funkcji `copy` musi być iterator, a w wywołaniu w miejsce odpowiedniego argumentu przekazano kontener. Podobnie niemożliwe jest wykonanie kodu:

```
// błąd — brak elementu pod wskazaniem end.ret()
copy(bottom.begin(), bottom.end(), ret.end());
```

Tutaj błąd jest mniej oczywisty, ponieważ dzięki zgodności typów argumentów wywołanie takie zostanie przyjęte przez kompilator. Błąd ujawni się dopiero przy wykonywaniu powyższego kodu — na początek funkcja `copy` spróbuje przypisać skopiowaną wartość do elementu wskazywanego iteratorem `ret.end()`, podczas gdy wiemy, że `ret.end()` to odniesienie do nieistniejącego elementu za ostatnim elementem kontenera. Sposób działania programu w takiej sytuacji jest całkowicie niezdefiniowany.

Skąd taka definicja funkcji `copy`? Otóż oddzielenie koncepcji kopiowania od zwiększania rozmiaru kontenera pozwala programistom na dokonywanie wyboru operacji w zależności od potrzeb. Przydaje się to, na przykład, w sytuacji, kiedy zachodzi potrzeba skopiowania elementów do kontenera bez zmieniania jego rozmiaru (a więc z nadpisaniem bieżącej zawartości kontenera). W innym przykładzie, prezentowanym zresztą w §6.2.2/154, dzięki adaptorowi `back_inserter` można dołączać do kontenera elementy, które nie są prostymi kopiami elementów innego kontenera.

6.1.1. Inna implementacja funkcji split

Inną funkcją, którą moglibyśmy zmodyfikować pod kątem wykorzystania algorytmów standardowych, jest funkcja `split`, prezentowana w §5.6/125. Najtrudniejsze w pierwotnej wersji funkcji było kontrolowanie indeksów ograniczających kolejne wyrazy ciągu wejściowego. Indeksy te możemy zastąpić iteratorami i zaprząc do pracy algorytmy biblioteki standardowej:

```
// zwraca true, jeżeli argument jest znakiem odstępu; w przeciwnym przypadku zwraca false
bool space(char c)
{
    return isspace(c);
}

// zwraca false, jeżeli argument jest znakiem odstępu; w przeciwnym przypadku zwraca true
bool not_space(char c)
{
    return isspace(c);
}

vector<string> split(const string& str)
{
    typedef string::const_iterator iter;
    vector<string> ret;

    iter i = str.begin();
    while (i != str.end()) {

        // pomiń wprowadzające znaki odstępu
        i = find_if(i, str.end(), not_space);

        // znajdź koniec bieżącego wyrazu
        iter j = find_if(i, str.end(), space);

        // kopiuj znaki z zakresu [i, j)
        if (i != str.end())
            ret.push_back(string(i, j));
        i = j;
    }
    return ret;
}
```

W kodzie nowej wersji funkcji widać sporo nowości wymagających objaśnienia. W gruncie rzeczy jednak powyższa implementacja stanowi nieco inne wcielenie dotychczasowego algorytmu, wykorzystującego `i` i `j` do wyznaczania granic odszukanego wyrazu ciągu `str`, zgodnie z wytycznymi z §5.6/125. Po znalezieniu słowa, jego kopia jest dołączana do kontenera `ret`.

Tym razem jednak `i` oraz `j` są iteratorami, a nie indeksami. Do wygodnego posługiwania się typem iteraotra wykorzystaliśmy definicję `typedef`, dzięki czemu później można było korzystać z nazwy typu `iter` w miejsce nazwy `string::const_iterator`. I choć typ `string` nie obsługuje wszystkich operacji charakterystycznych dla kontenerów, obsługuje iteratory, dzięki czemu możemy w stosunku do znaków umieszczonych w ciągu `string` wykorzystywać algorytmy biblioteki standardowej, podobnie, jak czyniliśmy to w odniesieniu do elementów kontenera typu `vector`.

Wykorzystywanym w tym wcieleniu funkcji algorytmem jest `find_if`. Pierwszymi dwoma argumentami wywołania tego algorytmu są iteratory ograniczające analizowany ciąg. Trzeci argument jest predykatem sprawdzającym przekazany argument i zwracającym wartość `true` bądź `false`. Funkcja `find_if` wywołuje predykat dla każdego elementu we wskazanym zbiorze, przerywając działanie w momencie napotkania pierwszego elementu spełniającego predykat.

Biblioteka standardowa udostępnia funkcję `isspace`, która sprawdza, czy przekazany argumentem znak należy do grupy znaków odstępu. Funkcja ta jest jednak przeciążona pod kątem obsługi języków takich jak, na przykład, japoński, korzystających z rozszerzonego typu znaku `wchar_t` (patrz §1.3/32). Nie jest łatwo przekazać funkcję przeciążoną jako bezpośredni argument szablonu funkcji. Kłopot z takim przekazaniem polega na tym, że kompilator nie „wie", której wersji funkcji przeciążonej użyć w wywołaniu szablonu funkcji; nie może tego rozstrzygnąć na podstawie argumentów wywołania funkcji przeciążonej, ponieważ w wywołaniu argumentów tych nie ma. Z tego względu udostępniliśmy własny predykat dla algorytmu `find_if`, a konkretnie dwa takie predykaty: `space` i `not_space`; dzięki nim staje się jasne, której wersji funkcji `isspace` ma użyć kompilator.

Pierwsze wywołanie funkcji `find_if` wyszukuje znak niebędący znakiem odstępu rozpoczynający kolejny wyraz. Pamiętajmy, że znaki odstępów mogą znajdować się również przed pierwszym z wyrazów. Tych znaków nie chcemy wyprowadzać na wyjście.

Po zakończeniu pierwszego wywołania funkcji `find_if` `i` wskazuje pierwszy niebędący odstępem znak ciągu `str`. Wskazanie to wykorzystywane jest w drugim wywołaniu funkcji `find_if`, która z kolej wyszukuje pierwszy znak odstępu za pozycją `i` i przed końcem ciągu. Jeżeli funkcji nie uda się znaleźć znaku spełniającego predykat, zwraca drugi argument, który w naszym przypadku będzie miał wartość `str.end()`. Tak więc `j` zostanie zainicjalizowane wartością iteratora odnoszącego się do pierwszego znaku za znakiem `i` będącym znakiem spacji, ewentualnie wartością `str.end()`, jeżeli takiego znaku w ciągu nie było.

W tej fazie wykonania `i` i `j` ograniczają kolejny znaleziony wyraz ciągu `str`. Zostało już tylko skorzystanie z wartości iteratorów do skopiowania podciągu z ciągu `str` do kontenera `ret`. W poprzedniej wersji funkcji `split` do tego zadania wytypowana została funkcja `string::substr`. Ponieważ jednak w nowej wersji funkcji `split` nie mamy już indeksów, a jedynie iteratory, a funkcja `substr` nie jest przeciążona dla argumentów tego typu, musimy skonstruować podciąg typu `string` jako kopię ograniczonego iteratorami ciągu `str`. Realizujemy to w wyrażeniu `string(i, j)`, które wygląda podobnie do definicji zmiennej `spaces` pokazanej w §1.2/30. W tym przykładzie obiekt typu `string` inicjalizowany jest kopią znaków w zakresie `[i, j)`. Obiekt ten jest następnie za pośrednictwem metody `push_back` kierowany do kontenera `ret`.

Warto wskazać, że nowa wersja programu pomija testowanie indeksu `i` pod kątem zrównania się z `str.size()`. Nie ma też odpowiedników tych testów porównujących iteratory z `str.end()`. Przyczyną tego stanu rzeczy jest fakt, że algorytmy biblioteki standardowej są zaprojektowane i zaimplementowane z uwzględnieniem obsługi wywołań z przekazaniem zakresu pustego. Jeżeli, na przykład, w którymś momencie pierwsze z wywołań `find_if` ustawi wartość iteratora `i` na `str.end()`, wtedy nie ma konieczności

wykrywania tego faktu przed kolejnym wywołaniem funkcji `find_if` z tym iteratorem — `find_if` wykryje fakt przekazania ciągu pustego `[i, str.end())` i zwróci w takim przypadku `str.end()`, sygnalizując jednocześnie brak spełnienia predykatu.

6.1.2. Palindromy

Kolejnym zadaniem związanym z manipulowaniem ciągami, do którego realizacji możemy wykorzystać algorytmy biblioteki standardowej, jest sprawdzanie, czy zadany wyraz jest palindromem. Palindromy to wyrazy, czytane tak samo w przód i wspak. Przykłady palindromów to: „kajak”, „oko”, „Anna”, „potop” czy „radar”.

Oto rozwiązanie postawionego zadania:

```
bool is_palindrome(const string&)
{
    return equal(s.begin(), s.end(), s.rbegin());
}
```

Wyrażenie wartości zwracanej w powyższej funkcji korzysta z wywołania `equal` i metody `rbegin`; obu tych funkcji nigdy dotąd nie wykorzystywaliśmy.

Podobnie jak metoda `begin`, `rbegin` zwraca iterator, tyle że iterator ten rozpoczyna wskazanie od ostatniego elementu kontenera, a jego zwiększanie powoduje cofanie wskazania do poprzednich elementów kontenera.

Funkcja `equal` porównuje dwie sekwencje znaków i określa, czy sekwencje te zawierają identyczne znaki. Pierwsze dwa argumenty wywołania to iteratory ograniczające pierwszą z sekwencji. Trzeci argument to iterator odnoszący się do początku sekwencji drugiej. Funkcja `equal` zakłada, że sekwencja numer dwa ma rozmiar identyczny z rozmiarem sekwencji pierwszej, stąd brak konieczności określania iteratora kończącego drugą sekwencję. Jako że trzecim argumentem wywołania, a więc początkiem drugiej sekwencji porównania, jest `s.rbegin()`, efektem wywołania jest porównanie znaków z początku kontenera (ciągu) ze znakami końca kontenera. Funkcja `equal` porównuje kolejno pierwszy znak ciągu `s` z ostatnim znakiem, następnie drugi znak ciągu ze znakiem przedostatnim i tak dalej. Działanie takie idealnie spełnia warunki rozwiązania postawionego zadania.

6.1.3. Wyszukiwanie w tekście adresów URL

W ramach ostatniego przykładu przetwarzania ciągów znakowych napiszemy funkcję, która w przekazanym ciągu typu `string` wyszuka adresy zasobów sieci WWW, zwane też identyfikatorami URL. Funkcję taką można wykorzystać do przeglądania dokumentu (po ich uprzednim skopiowaniu do zmiennej typu `string`) w poszukiwaniu zawartych w tym dokumencie identyfikatorów URL. Funkcja powinna odnaleźć wszystkie identyfikatory URL występujące w tekście dokumentu.

Identyfikator URL jest ciągiem znaków postaci:

nazwa-protokołu`://`*nazwa-zasobu*

gdzie *nazwa-protokołu* może zawierać wyłącznie litery, gdy *nazwa-zasobu* może składać się z liter, cyfr oraz rozmaitych znaków przestankowych i specjalnych. Nasza funkcja powinna przyjmować ciąg typu `string` i wyszukiwać w nim wystąpienia sekwencji znaków `://`. W przypadku odnalezienia takiego zestawu znaków funkcja powinna odszukać poprzedzającą go nazwę protokołu i następującą po nim nazwę zasobu.

Jako że projektowana funkcja powinna odnaleźć wszystkie identyfikatory URL występujące w tekście, wartością zwracaną powinien być kontener typu `vector<string>`, w którym każdy z elementów reprezentowałby pojedynczy identyfikator URL. Działanie funkcji opiera się na przesuwaniu iteratora wzdłuż przekazanego ciągu i wyszukiwanie w tym ciągu znaków `://`. W momencie odnalezienia ciągu `://` funkcja wyszukuje wstecz wyraz będący nazwą protokołu i w przód ciąg będący nazwą zasobu URL:

```
vector<string> find_urls(const string& s)
{
    vector<string> ret;
    typedef string::const_iterator iter;
    iter b = s.begin(), e = s.end();

    // przejrzyj cały ciąg wejściowy
    while (b != e) {

        // poszukaj jednego lub kilku znaków, po których znajduje się podciąg ://
        b = url_beg(b, e);

        // jeśli udało się znaleźć podciąg
        if (b != e) {
            // pobierz resztę URL-a
            iter after = url_end(b, e);

            // zapamiętaj znaleziony URL
            ret.push_back(string(b, e));

            // zwiększ b i szukaj następnych identyfikatorów
            b = after;
        }
    }
    return ret;
}
```

Funkcja zaczyna się od zadeklarowania `ret` jako kontenera typu `vector`, przechowującego docelowo znalezione identyfikatory URL, oraz od zdefiniowania iteratorów ograniczających wejściowy ciąg typu `string`. Musimy jeszcze napisać funkcje `url_beg` i `url_end`. Pierwsza z nich będzie wyszukiwać początek każdego z identyfikatorów URL przekazanego ciągu i ewentualnie zwracać iterator ustawiony na początek podciągu *nazwa-protokołu*; jeżeli w ciągu wejściowym nie uda się znaleźć URL-a, funkcja `url_beg` powinna zwrócić drugi argument wywołania (tutaj `e`), sygnalizując w ten sposób niepowodzenie.

Gdy `url_beg` wyszuka początek identyfikatora URL, resztą zajmuje się `url_end`. Funkcja ta przeszukuje ciąg od wskazanej pozycji początkowej aż do osiągnięcia końca ciągu wejściowego albo znalezienia znaku, który nie może być częścią identyfikatora URL. Funkcja zwraca iterator ustawiony na pozycję następną za pozycją ostatniego znaku identyfikatora.

Po wywołaniu funkcji url_beg i url_end iterator b powinien odnosić się do początku znalezionego identyfikatora URL, a iterator after powinien wskazywać pozycję następną za pozycją ostatniego znaku identyfikatora, jak na rysunku 6.1.

Rysunek 6.1.
Podział obowiązków w zadaniu wyszukiwania identyfikatora URL

Ze znaków znajdujących się we wskazanym iteratorami zakresie konstruowany jest nowy ciąg typu string umieszczany w kontenerze ret.

W tym momencie można już tylko zwiększyć wartość iteratora b i przejść do wyszukiwania następnego identyfikatora. Ponieważ identyfikatory nie mogą na siebie nachodzić, b można ustawić na znak następny za ostatnim znakiem właśnie znalezionego identyfikatora URL; pętlę while należy kontynuować aż do przetworzenia wszystkich znaków ciągu wejściowego. Po wyjściu z tej pętli zwracany przez funkcję kontener ret zawierać będzie wszystkie URL-e występujące w ciągu wejściowym.

Pora na zastanowienie się nad implementacją funkcji url_beg i url_end. Na pierwszy ogień pójdzie funkcja prostsza, a więc url_end:

```
string::const_iterator url_end(string::const_iterator b, string::const_iterator e)
{
    return find_if(b, e, not_url_char);
}
```

Funkcja przegląda kolejne znaki wskazanego zakresu, poddając je analizie za pośrednictwem funkcji find_if, którą poznaliśmy w §6.1.1/144. Predykat przekazany w wywołaniu funkcji find_if, not_url_char, musimy napisać własnoręcznie. Powinien on zwracać wartość true dla znaku, który nie może znaleźć się w identyfikatorze URL:

```
bool not_url_char(char c)
{
    // znaki (poza znakami alfanumerycznymi) dozwolone w identyfikatorach URL
    static const string url_ch = "~;/?:@=&$-_+!'(),";

    // sprawdź, czy c jest dozwolonym znakiem URL
    return !(isalnum(c) || find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
}
```

Powyższa funkcja, choć niepozorna, wykorzystuje kilka nowych elementów. Po pierwsze, w definicji ciągu znaków dozwolonych w identyfikatorze URL pojawiło się słowo static. Zmienne lokalne deklarowane jako statyczne (właśnie ze słowem static) są zachowywane pomiędzy wywołaniami funkcji. Zmienna typu string o nazwie url_ch będzie więc konstruowana i inicjalizowana jedynie podczas realizacji pierwszego wywołania funkcji not_url_char. Kolejne wywołania zastaną gotową wartość typu string. Dodatkowo słowo const zapewnia niezmienność wartości zmiennej url_ch po jej inicjalizacji.

Funkcja not_url_char wykorzystuje wywołanie funkcji isalnum zdefiniowanej w nagłówku <cctype>. Funkcja ta sprawdza, czy przekazany znak jest znakiem alfanumerycznym (czyli czy jest literą lub cyfrą).

Wykorzystane w tej funkcji wywołanie find jest kolejnym algorytmem biblioteki standardowej. Jego działanie jest podobne do find_if z tym, że zamiast wywoływać predykat, funkcja find samodzielnie porównuje kolejne znaki z trzecim argumentem wywołania. Jeżeli w sekwencji określonej dwoma pierwszymi argumentami znajduje się wartość określona argumentem trzecim, funkcja zwraca iterator ustawiony na miejsce pierwszego wystąpienia zadanej wartości w sekwencji. W przypadku nieodnalezienia zadanej wartości funkcja find zwraca drugi argument.

Teraz widać, jak dokładnie działa funkcja not_url_char. Ponieważ wartość zwracana stanowi negację całego wyrażenia, not_url_char zwraca wartość false, jeżeli zadany znak okaże się literą, cyfrą bądź znakiem dozwolonym w identyfikatorze URL, a określonym w ciągu url_ch. Dla każdej innej wartości argumentu c funkcja zwraca wartość true.

Pora na najtrudniejsze — implementację funkcji url_beg. Funkcja ta jest nieco zagmatwana, ponieważ musi obsługiwać potencjalną sytuację zawierania się w ciągu wejściowym sekwencji :// w kontekście, który wyklucza istnienie wokół tej sekwencji identyfikatora URL. W praktyce implementacja takiej funkcji wykorzystywałaby zapewne zdefiniowaną uprzednio listę dozwolonych nazw protokołów. My, dla uproszczenia, ograniczymy się do zagwarantowania, że sekwencję :// poprzedza podciąg jednego lub więcej znaków i przynajmniej jeden znak znajduje się za tą sekwencją.

```
string::const_iterator url_beg(string::const_iterator b, string::const_iterator e)
{
    static const string sep = "://";

    typedef string::const_iterator iter;

    // i sygnalizuje odnalezienie separatora ://
    iter i = b;

    while ((i = search(i, e, sep.begin(), sep.end())) != e) {

        // upewnij się, że separator nie rozpoczyna się od początku ciągu
        if (i != b && i + sep.size() != e) {

            // beg zaznacza początek nazwy-protokołu
            iter beg = i;
            while (beg != b && isalpha(beg[-1]))
                --beg;

            // czy za i przed separatorem :// znajduje się przynajmniej po jednym znaku?
            if (beg != i && !not_url_char(i[sep.size()]))
                return beg;
        }
```

```
            // odnaleziony separator nie jest częścią URL-a; zwiększ i poza pozycję ostatniego
               znaku sepraratora
            i += sep.size();
        }
        return e;
    }
```

Najprostszą częścią tej funkcji jest jej nagłówek. Wiadomo, że w wywołaniu przekazane zostaną dwa iteratory określające zakres przetwarzanego ciągu i że funkcja ma zwrócić iterator ustawiony na początek pierwszego z odnalezionych identyfikatorów URL. Oczywista jest też deklaracja lokalnej zmiennej typu `string`, w której przechowywane są znaki składające się na wyróżniający identyfikator URL separator nazwy protokołu i nazwy zasobu. Podobnie, jak w funkcji `not_url_char` (§7.1.1/148), zmienna ta jest zmienną statyczną i stałą. Nie będzie więc można zmieniać w funkcji wartości zmiennej — zostanie ona ustalona przy pierwszym wykonaniu funkcji `url_beg`.

Działanie funkcji opiera się na ustaleniu wartości dwóch iteratorów w zakresie wyznaczanym iteratorami `b` i `e`. Iterator `i` ma wskazywać początek separatora URL, a iterator `beg` początek znajdującego się przed separatorem podciągu zawierającego nazwę protokołu (rysunek 6.2).

Rysunek 6.2.
Pozycjonowanie iteratorów w funkcji url_beg

Na początek funkcja odnajduje pierwsze wystąpienie separatora, wykorzystując wywołanie `search` (funkcję biblioteki standardowej), którego wcześniej nie stosowaliśmy. Funkcja `search` przyjmuje dwie pary iteratorów: pierwsza para odnosi się do sekwencji, w której następuje wyszukiwanie, druga określa sekwencję wyszukiwaną. Podobnie jak ma to miejsce w przypadku pozostałych funkcji biblioteki standardowej, jeżeli funkcji `search` nie uda się odnaleźć zadanego ciągu, zwraca iterator wskazujący na koniec przeszukiwanego ciągu. Po zakończeniu realizacji funkcji `search` iterator `i` może przyjąć dwie wartości — albo będzie wskazywał element następny za ostatnim ciągu wejściowego, albo zawierać będzie pozycję dwukropka rozpoczynającego separator `://`.

Po odnalezieniu separatora należy odszukać litery składające się na nazwę protokołu. Najpierw należy jednak sprawdzić, czy separator nie znajduje się na samym początku albo na samym końcu przeszukiwanego ciągu; w obu tych przypadkach mamy do czynienia z niepełnym identyfikatorem URL, ponieważ identyfikator taki powinien zawierać przynajmniej po jednym znaku z obu stron separatora. Jeżeli separator rokuje możliwości dopasowania identyfikatora URL, należy spróbować ustawić iterator `beg`. W wewnętrznej pętli `while` iterator `beg` jest przesuwany w tył, aż do momentu, gdy dotrze do początku ciągu albo do pierwszego znaku niebędącego znakiem alfanumerycznym. Widać tutaj zastosowanie dwóch nowych elementów kodu: pierwszym jest wykorzystanie faktu, że jeżeli kontener obsługuje indeksowanie, to podobną obsługę definiują iteratory tego kontenera. Innymi słowy, `beg[-1]` jest znakiem bezpośrednio poprzedzającym znak wskazywany przez `beg`. Zapis `beg[-1]` należy tu traktować jako skróconą wersję zapisu

`*(beg - 1)`. O tego typu iteratorach powiemy więcej w §8.2.6/200. Drugi nowy element to wykorzystanie funkcji `isalpha` zdefiniowanej w nagłówku `<cctype>` biblioteki standardowej; funkcja ta sprawdza, czy przekazany argument (znak) zawiera literę.

Jeżeli uda się cofnąć iterator choćby o jeden znak, można założyć, że znak ten tworzy nazwę protokołu. Przed zwróceniem wartości `beg` należy jednak jeszcze sprawdzić, czy przynajmniej jeden dopuszczalny znak znajduje się również po prawej stronie separatora `://`. Ten test jest nieco bardziej skomplikowany. Wiadomo, że w ciągu wejściowym znajduje się przynajmniej jeden znak za separatorem; wiadomo to stąd, że pozytywnie wypadł test różności `i + sep.size()` z wartością `e` wskazującą koniec ciągu. Do znaku tego możemy się odwołać za pośrednictwem `i[sep.size()]`, co odpowiada zapisowi `*(i + sep.size())`. Należy jeszcze sprawdzić, czy znak ten należy do znaków dozwolonych w identyfikatorze URL, przekazując ów znak do wywołania `not_url_char`. Funkcja ta zwraca wartość `true` dla znaków niedozwolonych; po zanegowaniu wartości zwracanej dowiemy się, czy przekazany znak jest dozwolony w identyfikatorze URL.

Jeżeli separator nie jest częścią poprawnego identyfikatora URL, funkcja zwiększa wartość iteratora `i` i kontynuuje wyszukiwanie.

W prezentowanym kodzie po raz pierwszy znalazł zastosowanie ***operator dekrementacji*** (zmniejszenia o jeden) wymieniony w tabeli operatorów §2.7/54. Operator ten działa podobnie jak operator inkrementacji, tyle że zamiast zwiększać, zmniejsza wartość operandu. Również ten operator można stosować jako operator przedrostkowy bądź przyrostkowy. Zastosowana tu wersja przedrostkowa zmniejsza wartość operandu i zwraca jego nową wartość.

6.2. Porównanie metod wyznaczania ocen

W §4.2/93 zaprezentowaliśmy metodę oceniania studentów w oparciu o ocenę pośrednią, ocenę z egzaminu końcowego i medianę ocen zadań domowych. Metoda ta może zostać nadużyta przez sprytnych studentów, którzy rozmyślnie mogą nie wykonywać niektórych prac domowych. W końcu połowa gorszych ocen nie wpływa na ostateczny wynik. Wystarczy bowiem odrobić dobrze połowę zadań domowych, aby uzyskać wynik identyczny jak osoba, która tak samo dobrze wykonała wszystkie prace domowe.

Z naszego doświadczenia wynika, że większość studentów nie będzie próbować wykorzystywać tej luki w metodzie oceniania. Mieliśmy jednak okazję prowadzić zajęcia w grupie, której studenci czynili to otwarcie. Zastanawialiśmy się wtedy, czy średnia ocen studentów którzy omijali prace domowe, może być zbliżona do średniej studentów, którzy sumiennie pracowali w czasie wolnym. Zastanawiając się nad odpowiedzią na to pytanie, uznaliśmy, że warto być może sprawdzić odpowiedź na to pytanie w kontekście dwóch różnych metod oceniania:

- Przy użyciu średniej w miejsce mediany (zadania nieodrobione oceniane są na 0 punktów).
- Przy użyciu mediany tylko tych zadań, które student odrobił.

Dla każdego z tych schematów oceny należało porównać medianę ocen studentów, którzy oddawali wszystkie prace domowe z medianą ocen studentów, którzy pominęli jedno lub więcej zadań samodzielnych. Mający pomóc w udzieleniu odpowiedzi program powinien realizować dwa zadania:

1. Wczytywać wpisy wszystkich studentów i wyodrębniać wpisy tych studentów, którzy oddali wszystkie prace domowe.
2. Zastosować obie metody oceniania do obu grup; wypisać na urządzeniu wyjściowym medianę ocen każdej z grup.

6.2.1. Przetwarzanie wpisów studentów

Pierwszym postawionym zadaniem jest wczytanie i klasyfikacja wpisów studentów. Na szczęście dysponujemy już pewnym kodem, który rozwiązywał podobny problem postawiony w jednym z wcześniejszych rozdziałów. Do wczytania wpisów studentów możemy bowiem wykorzystać typ `Student_info` z §4.2.1/94 i związaną z obsługą tego typu funkcję `read` (§4.2.2/94). Nie mamy natomiast funkcji, która sprawdzałaby liczbę oddanych prac domowych. Jej napisanie nie stanowi jednak problemu:

```
bool did_all_hw(const Student_info& s)
{
    return ((find(s.homework.begin(), s.homework.end(), 0)) == s.homework.end());
}
```

Funkcja ta sprawdza, czy kontener `homework` wpisu studenta `s` zawiera jakiekolwiek wartości zerowe. Przyjęliśmy tu, że sam fakt oddania pracy domowej jest punktowany, więc wartości zerowe odpowiadają pracom niezrealizowanym w ogóle. Wartość zwracana przez funkcję `find` porównywana jest z wartością `homework.end()`, jak bowiem pamiętamy, funkcja `find` zwraca w przypadku nieodnalezienia zadanej wartości drugi argument wywołania.

Dzięki dwóm wspomnianym funkcjom implementacja kodu wczytującego i klasyfikującego wpisy studentów jest znacznie uproszczona. Wystarczy bowiem wczytywać kolejne wpisy, sprawdzać, czy student oddał wszystkie zadania domowe, i w zależności od wyniku testu dołączyć wpis do jednego z dwóch kontenerów typu `vector`, o nazwach `did` (dla studentów sumiennych) i `didnt` (dla leniwych). Potem należy sprawdzić, czy któryś z kontenerów nie jest przypadkiem pusty, co uniemożliwiłoby dalszą analizę:

```
vector<Student_info> did, didnt;
Student_info student;

// wczytaj wszystkie wpisy, pogrupuj je według kryterium odrobienia prac domowych
while (read(cin, student)) {
    if (did_all_hw(student))
        did.push_back(student);
```

```
        else
            didnt.push_back(student);
    }

    // sprawdź, czy oba kontenery zawierają jakieś wpisy
    if (did.empty()) {
        cout << "Żaden ze studentów nie odrobił wszystkich prac domowych!";
        return 1;
    }
    if (didnt.empty()) {
        cout << "Wszyscy studenci odrobili wszystkie prace domowe!";
        return 1;
    }
```

Jedyną nowinką w powyższym kodzie jest wywoływanie metody `empty`, zwracającej wartość `true` dla kontenera pustego i `false` dla kontenera zawierającego jakieś elementy. Ten sposób sprawdzania zawartości kontenera jest lepszy niż porównywanie rozmiaru zwracanego przez `size` z zerem, ponieważ w przypadku niektórych rodzajów kontenerów sprawdzenie, czy w kontenerze w ogóle coś się znajduje może być realizowane dużo szybciej niż policzenie liczby znajdujących się w nim elementów.

6.2.2. Analiza ocen

Umiemy już wczytać wpisy studentów i sklasyfikować je jako przynależące do kontenera `did` bądź do kontenera `didnt`. Następne zadanie to analiza wpisów; przed przystąpieniem do takiej analizy wypadałoby zastanowić się nad jej założeniami.

Wiemy, że trzeba będzie wykonać trzy analizy, a każda z nich ma się składać z dwóch części, osobno dla studentów, którzy odrobili wszystkie zadania domowe i dla tych, którzy to zaniedbali. Ponieważ analizy obejmować będą dwa zbiory danych, najlepiej byłoby zaimplementować je w postaci osobnych funkcji. Jednakże niektóre operacje, takie jak wydruk zestawienia w ustalonym formacie, realizowane będą wobec par wyników analiz. Należy więc zaimplementować w postaci funkcji wydruk zestawienia analizy par zbiorów wartości.

Najgorsze, że docelowo program powinien wywoływać funkcję, która trzykrotnie (raz dla każdego rodzaju analizy) wypisywałaby na urządzeniu wyjściowym wyniki analizy. Każda z funkcji analitycznych powinna być wywołana dwukrotnie, raz dla kontenera `did`, raz dla kontenera `didnt`. Tyle że funkcja generująca zestawienie powinna za każdym razem wywoływać inną funkcję analityczną! Jak to zrobić?

Rozwiązaniem najprostszym jest zdefiniowanie trzech funkcji analitycznych i przekazywanie ich kolejno do funkcji generującej wydruk wyników. Argumentów typu funkcyjnego już używaliśmy, między innymi przy przekazywaniu funkcji `compare` do funkcji bibliotecznej `sort` (§4.2.2/96). W takim przypadku funkcja generująca zestawienia powinna przyjmować pięć argumentów:

- Strumień wyjściowy, do którego zapisywane byłyby ciągi wydruku.
- Ciąg `string` reprezentujący nazwę analizy.

- Funkcję analityczną.
- Dwa argumenty będące kontenerami typu vector zawierającymi dane wejściowe analizy.

Załóżmy, dla przykładu, że pierwsza analiza, wyznaczająca mediany ocen prac domowych, realizowana jest przez funkcję o nazwie median_analysis. Wyprowadzenie wyników tej analizy w stosunku do obu grup studentów oznaczałoby realizację następującego wywołania:

```
write_analysis(cout, "median", median_analysis, did, didnt);
```

Zanim zdefiniujemy funkcję write_anlysis powinniśmy zaimplementować funkcję median_analysis. Funkcja ta powinna przyjmować kontener (typu vector) wpisów studentów; na podstawie tych wpisów funkcja powinna obliczać oceny końcowe studentów zgodnie z przyjętą metodą oceniania studentów i zwrócić medianę obliczonych ocen. Funkcja taka mogłaby mieć postać:

```
// ta funkcja nie całkiem działa
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(), back_inserter(grades), grade);
    return median(grades);
}
```

Choć na pierwszy rzut oka funkcja może wydawać się skomplikowana, wprowadza tylko jedną nową funkcję — funkcję transform. Przyjmuje ona trzy iteratory i jedną funkcję. Pierwsze dwa iteratory definiują zakres elementów do przetworzenia; trzeci iterator to iterator kontenera docelowego wyników przetwarzania — w kontenerze tym zapisywane będą wyniki funkcji przetwarzającej, przekazywanej czwartym argumentem.

Wywołując funkcję transform, musimy zadbać o to, aby w kontenerze docelowym było dość miejsca na zachowanie wyników przetwarzania wejściowej sekwencji elementów. W tym przypadków nie jest to problemem, ponieważ kontener docelowy wskazujemy iteratorem back_inserter (patrz §6.1/142), powodującym automatyczne dołączanie wyników działania funkcji transform do kontenera grades; kontener ten będzie dzięki zastosowaniu tego iteratora rozszerzany w miarę dodawania kolejnych elementów potrzeby.

Czwartym argumentem wywołania funkcji transform jest funkcja przekształcająca (przetwarzająca) elementy wskazane zakresem wejściowym; jest ona wywoływana dla każdego elementu z tego zakresu, a wartości zwracane przez funkcję umieszczane są w kontenerze docelowym. W powyższym przykładzie wywołanie funkcji transform oznacza zastosowanie do elementów kontenera students funkcji grade; wyniki zwracane przez funkcję grade umieszczane są w kontenerze grades. Po skompletowaniu w kontenerze grades ocen końcowych studentów, wywołujemy funkcję median z §4.1.1/83, obliczającą medianę wartości przechowywanych w kontenerze grades.

Jest tylko jeden problem. Zgodnie z uwagą poprzedzającą kod, funkcja w tym kształcie niezupełnie działa.

Przyczyną takiego stanu rzeczy jest to, że mamy kilka przeciążonych wersji funkcji `grade`. Kompilator nie jest w stanie określić w wywołaniu `transform`, o jaką wersję funkcji `grade` może chodzić, ponieważ w wywołaniu tym nie ma żadnych argumentów funkcji `grade`. My wiemy, że chodzi o wersję zdefiniowaną w §4.2.2/95. Trzeba jeszcze znaleźć sposób na przekazanie tej informacji kompilatorowi.

Drugą niedoskonałością powyższego kodu jest to, że funkcja `grade` zgłasza wyjątek w przypadku, kiedy student nie ma żadnych ocen zadań domowych, tymczasem funkcja `transform` tych wyjątków w żaden sposób nie obsługuje. Jeżeli więc zdarzy się wyjątek, realizacja funkcji `transform` zostanie zatrzymana, a sterowanie przeniesione do funkcji `median_analysis`. Ponieważ funkcja `median_analysis` również nie przechwytuje ani nie obsługuje wyjątków, wyjątek będzie propagowany dalej w górę stosu wywołań funkcji. W efekcie również funkcja `median_analysis` zostanie przedwcześnie przerwana, przekazując sterowanie do wywołującego i tak dalej, aż wyjątek zostanie przechwycony odpowiednią klauzulą `catch`. Jeżeli klauzuli takiej zabraknie (z czym mielibyśmy do czynienia w proponowanej funkcji) przerywane jest działanie całego programu, a na urządzeniu wyjściowym wyświetlany jest komunikat o zgłoszeniu wątku (albo i nie, zależnie od implementacji).

Oba problemy możemy rozwiązać za pośrednictwem pomocniczej funkcji, która będzie realizować wywołanie `grade` wewnątrz instrukcji `try` oraz zajmie się obsługą ewentualnych wyjątków. Ponieważ w tej funkcji wywołanie funkcji `grade` będzie jawne, kompilator (na podstawie argumentów wywołania) nie będzie miał kłopotów z wytypowaniem wersji funkcji `grade` do realizacji wywołania:

```
double grade_aux(const Student_info& s)
{
    try {
        return grade(s);
    } catch (domain_error) {
        return grade(s,midterm, s.final, 0);
    }
}
```

Ta funkcja przechwytuje wyjątki zgłaszane w funkcji `grade`. W przypadku pojawienia się wyjątku zostanie on obsłużony w ramach klauzuli `catch`. Obsługa polega na wywołaniu funkcji `grade` z trzecim argumentem o wartości `0` — założyliśmy przecież, że brak pracy domowej oznacza zero punktów. Jeżeli więc dany student nie odrobił żadnej z prac, wtedy ogólna ocena pracy domowej będzie wynosiła również `0`. Do oceny końcowej liczone więc będą wyłącznie ocena z egzaminu końcowego i ocena pośrednia.

Możemy już przepisać funkcję analityczną tak, aby korzystała z pomocniczego wywołania `grade_aux`:

```
// ta wersja działa znakomicie
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(), back_inserter(grades), grade_aux);
    return median(grades);
}
```

Znając już postać funkcji analitycznej, możemy przejść do zdefiniowania funkcji write_analysis, która wykorzystuje funkcję analityczną do porównania dwóch grup studentów:

```
void write_analysis(ostream& out, const string& name,
                    double analysis(const vector<Student_info>&),
                    const vector<Student_info>& did,
                    const vector<Student_info>& didnt)
{
    out << name << " mediana(grupa studentów solidnych) = " << analysis(did)
                << " mediana(grupa studentów niesolidnych) = " << analysis(didnt)
                << endl;
}
```

Funkcja ta jest zaskakująco krótka, choć wprowadza kolejne nowości. Pierwszą z nich jest sposób definiowania parametru reprezentującego funkcję. Definicja parametru analysis wygląda identycznie jak deklaracja funkcji napisanej w §4.3/100 (co prawda w §10.1.2/230 przekonamy się, że różnice są większe, niż się wydaje, możemy je jednak pominąć jako nie mające wpływu na bieżące omówienie).

Druga nowinka to typ wartości zwracanej przez funkcję — ***void***. Zastosowanie typu void jest ograniczone między innymi właśnie do definiowania typu wartości zwracanej. Widząc funkcję zwracającą typ void, widzimy funkcję, która tak naprawdę nie zwraca żadnej wartości. Z takiej funkcji wychodzi się za pośrednictwem instrukcji return pozbawionej wartości, na przykład:

```
return;
```

Wykonanie funkcji zwracającej typ void może zostać zakończone również w wyniku dotarcia do klamry kończącej kod funkcji. W innych funkcjach takie zakończenie funkcji jest niedozwolone; w funkcjach niezwracających wartości możemy pominąć instrukcję return.

Teraz możemy napisać pozostały kod programu:

```
int main()
{
    // studenci, którzy nie odrobili wszystkich zadań domowych
    vector<Student_info> did, didnt;

    // wczytaj wszystkie wpisy, pogrupuj je według kryterium odrobienia prac domowych
    while (read(cin, student)) {
        if (did_all_hw(student))
            did.push_back(student);
        else
            didnt.push_back(student);
    }

    // sprawdź, czy oba kontenery zawierają jakieś wpisy
    if (did.empty()) {
        cout << "Żaden ze studentów nie odrobił wszystkich prac domowych!";
        return 1;
    }
    if (didnt.empty()) {
        cout << "Wszyscy studenci odrobili wszystkie prace domowe!";
        return 1;
    }
```

```
    // przystąp do analizy
    write_analysis(cout, "mediana", median_analysis, did, didnt);
    write_analysis(cout, "średnia", average_analysis, did, didnt);
    write_analysis(cout, "mediana prac oddanych", optimistic_analysis, did, didnt);

    return 0;
}
```

Zostało już tylko uzupełnić program o funkcje `average_analysis` i `optimistic_median_analysis`.

6.2.3. Wyznaczanie ocen na podstawie średniej ocen prac domowych

Funkcja `average_analysis` powinna obliczań oceny studentów przy uwzględnieniu nie mediany, ale wartości średniej ocen prac domowych. Logicznym początkiem implementacji powinno być napisanie funkcji obliczającej wartość średnią elementów kontenera typu `vector` i następnie wykorzystanie jej w miejsce funkcji `median` w funkcji obliczającej ocenę końcową (`grade`):

```
double average(const vector<double>& v)
{
    return accumulate(v.begin(), v.end(), 0.0) / v.size();
}
```

Funkcja ta wywołuje funkcję `accumulate`, która — w przeciwieństwie do dotychczas stosowanych algorytmów biblioteki standardowej — definiowana jest w nagłówki `<numeric>`. Jak wskazuje nazwa tego nagłówka, udostępnia on narzędzia wykorzystywane w obliczeniach numerycznych. Funkcja `accumulate` dodaje serię wartości z zakresu określonego wartościami dwóch pierwszych argumentów, zakładając początkową sumę o wartości równej wartości trzeciego argumentu.

Typ wartości zwracanej zależny jest od typu trzeciego argumentu; z tego względu nie wolno w miejsce `0.0` podać wartości `0` — ta ostatnia jest bowiem wartością typu `int`, co przy dodawaniu zaowocowałoby obcięciem części ułamkowych składników sumy.

Po uzyskaniu sumy wszystkich elementów kontenera dzielimy ją przez wartość `v.size()`, a więc przez liczbę elementów kontenera. Wynikiem dzielenia jest rzecz jasna średnia arytmetyczna wartości kontenera; średnia ta zwracana jest do wywołującego.

Dysponując funkcją `average` możemy wykorzystać ją w implementacji funkcji `average_grade` realizującej zmodyfikowaną metodę oceniania studentów według średnich ocen z zadań domowych:

```
double average_grade(const Student_info& s)
{
    return grade(s.midterm, s.final, average(s.homework));
}
```

Tutaj do obliczenia średniej ocen studenta z zadań domowych wywoływana jest funkcja average. Wartość zwracana przez tę funkcję przekazywana jest bezpośrednio do funkcji grade (patrz §4.1/82) wyliczającej ostateczną ocenę studenta.

Dysponując tak rozbudowaną infrastrukturą, w prosty sposób utworzymy funkcję average_analysis:

```
double average_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(),
              back_inserter(grades), average_grade);
    return median(grades);
}
```

Funkcję tę różni od funkcji median_analysis (§6.2.2/155) jedynie nazwa i wywołanie average_grade w miejsce grade_aux.

6.2.4. Mediana kompletu ocen zadań domowych

Ostatnia analiza, realizowana funkcją optimistic_analysis_scheme, wyznacza ocenę studenta w oparciu o optymistyczne założenie, że oceny studentów z tych prac domowych, których nie oddali, są identyczne z ocenami zadań, które zrealizowali. Przy takim założeniu będziemy obliczać medianę ocen tylko tych prac domowych, które zostały przez studenta oddane (przedłożone do oceny). Medianę taką nazwiemy medianą optymistyczną. Należy oczywiście uwzględnić ewentualność taką, że student nie oddał żadnej z prac domowych — w takim przypadku medianę jego ocen należałoby wyznaczyć jako 0:

```
// mediana optymistyczna niezerowej liczby prac domowych (0 dla zerowej liczby prac domowych)
double optimistic_median(const Student_info& s)
{
    vector<double> nonzero;
    remove_copy(s.homework.begin(), s.homework.end(),
                back_inserter(nonzero), 0);

    if (nonzero.empty())
        return grade(s.midterm, s.final, 0);
    else
        return grade(s,midterm, s.final. median(nonzero));
}
```

Powyższa funkcja wyodrębnia z ocen studenta niezerową liczbę ocen prac domowych i umieszcza je w nowym kontenerze typu vector o nazwie nonzero. Mając niezerową liczbę ocen z zadań domowych, wywołujemy dla nich i pozostałych ocen studenta wersję funkcji grade, zdefiniowaną w §4.1/82; wyliczana w miejscu przekazania argumentu mediana uwzględnia jedynie prace oddane.

Jedyną nowością jest w powyższym kodzie sposób wypełniania kontenera nonzero — wykorzystywana jest do tego funkcja remove_copy. Aby zrozumieć jej działanie, warto wiedzieć, że biblioteka standardowa udostępnia wersje „kopiujące” wielu typowych

algorytmów. Dla przykładu, algorytm `remove_copy` realizuje zasadniczo to, co algorytm `remove` (czyli usuwa elementy z kontenera), tyle że elementy nie są usuwane z kontenera, a jedynie kopiowane do kontenera wskazywanym trzecim argumentem.

Konkretnie, funkcja wyszukuje i „usuwa" z kontenera wszystkie elementy o wartości zgodnej z wartością zadaną jednym z argumentów wywołania. Wszystkie elementy sekwencji wejściowej, które nie zostały „usunięte", kopiowane są do kontenera docelowego. Niebawem dowiemy się, co znaczy w tym kontekście „usunięcie" elementu.

Funkcja `remove_copy` przyjmuje trzy iteratory i wartość typu zależnego od typu kontenera. Tradycyjnie już pierwsze dwa iteratory określają zakres elementów do przetworzenia. Trzeci iterator wskazuje miejsce w kontenerze docelowym. Implementacja algorytmu `remove_copy` zakłada, że kontener docelowy dysponuje miejscem wymaganym do umieszczania w nim kolejnych kopii przetwarzanych elementów. W prezentowanym wywołaniu o odpowiedni rozmiar kontenera docelowego troszczy się iterator `back_inserter`.

Efektem działania algorytmu `remove_copy` w postaci, w jakiej został tu wywołany jest skopiowanie do kontenera `nonzero` wszystkich niezerowych elementów kontenera `s.homework`. Po sprawdzeniu, czy kontener `nonzero` nie jest przypadkiem pusty, następuje wyznaczenie z niego mediany i przekazanie jej do wywołania funkcji `grade`, obliczającej ostateczną ocenę studenta. Jeżeli kontener `nonzero` jest pusty, do wywołania `grade` przekazywana jest — w miejsce mediany — wartość `0`.

Potrzebujemy jeszcze funkcji realizującej samą analizę z wykorzystaniem funkcji `optimistic_median`. Jej implementacja będzie jednak ćwiczeniem dla Czytelnika.

6.3. Klasyfikacja studentów — podejście drugie

W rozdziale 5. rozwiązywaliśmy zadanie kopiowania wpisów studentów, którzy nie zaliczyli kursu, do osobnego kontenera typu `vector`; wpisy tych studentów były też usuwane z kontenera źródłowego. Okazało się, że najprostsze rozwiązanie problemu klasyfikacji (dzielenia na grupy) cechuje się niedopuszczalną degradacją wydajności w miarę wzrostu rozmiaru danych wejściowych. Pokazaliśmy wtedy, jak problem małej wydajności wyeliminować przy użyciu kontenera typu `list`; obiecaliśmy sobie również, że do problemu wrócimy przy okazji omawiania algorytmów.

Dzięki algorytmom biblioteki standardowej możemy zaproponować dwa kolejne rozwiązania problemu klasyfikacji. Pierwsze z nich jest nieco wolniejsze, ponieważ wykorzystuje dwa algorytmy i dwukrotnie odwiedza każdy z elementów. Przy zastosowaniu bardziej specjalizowanego algorytmu to samo zadanie można rozwiązać w jednym przebiegu, przeglądając każdy z elementów zbioru wejściowego jednokrotnie.

6.3.1. Rozwiązanie dwuprzebiegowe

W pierwszym podejściu wykorzystamy strategię podobną do tej zakładanej w §6.2.4/158, kiedy chcieliśmy wyodrębnić z kontenera te wpisy, które zawierały niezerową liczbę ocen zadań domowych. Nie chcieliśmy wtedy w żaden sposób modyfikować kontenera `homework`, więc do skopiowania wpisów o niezerowej liczbie ocen prac domowych do osobnego kontenera wykorzystaliśmy funkcję `remove_copy`. W naszym bieżącym zadaniu musimy już nie tylko skopiować, ale faktycznie usunąć z pierwotnego kontenera elementy mające być umieszczone w kontenerze `nonzero`:

```
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> fail;
    remove_copy_if(students.begin(), students.end(),
                   back_inserter(fail), pgrade);
    students.erase(remove_if(students.begin(), students.end(),
                             fgrade), students.end());

    return fail;
}
```

Interfejs programu jest identyczny z tym prezentowanym w §5.3/118, implementującym klasyfikację za pośrednictwem kontenerów typu `vector` i indeksów. Podobnie jak tam, tu również przekazany w wywołaniu kontener typu `vector` ma docelowo zawierać wpisy tych studentów, którzy kurs zaliczyli, pozostałe wpisy mają zostać skopiowane do kontenera `fail` (także typu `vector`). Na tym podobieństwa się kończą.

W oryginalnej wersji programu do przeglądania kolejnych elementów kontenera służył iterator `iter`; wskazywane przez niego elementy były kopiowane do kontenera `fail`, a następnie, za pośrednictwem składowej `erase`, usuwane z kontenera `students`. Tym razem kopiowanie wpisów studentów, którzy nie przekroczyli progu zaliczenia, odbywa się za pośrednictwem funkcji `remove_copy_if`. Funkcja ta działa podobnie jak `remove_copy` (§6.2.4/158) z tym, że w miejsce bezpośredniej wartości porównania tu wykorzystywany jest predykat. Predykatem tym jest funkcja odwracająca działanie funkcji `fgrade` (§5.1/110):

```
bool pgrade(const Student_info& s)
{
    return !fgrade(s);
}
```

Przekazując do funkcji `remove_copy_if` predykat, żądamy, aby funkcja „usunęła" każdy element, dla którego wywołanie predykatu zwróci wartość `true`. „Usuwanie" oznacza tu „niekopiowanie" — kopiowane są więc tylko te elementy, które *nie spełniają* predykatu (predykat daje dla nich wartość `false`). Predykat `pgrade` zaś jest skonstruowany tak, że wywołanie `remove_copy_if` kopiuje wpisy studentów, którzy oblali kurs.

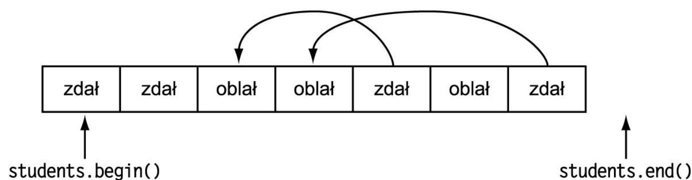
Następna instrukcja jest cokolwiek złożona. Po pierwsze mamy w niej wywołanie `remove_if`, „usuwające" elementy kontenera odpowiadające wpisom studentów z ocenami negatywnymi. Znowuż jednak cudzysłów otaczający słowo „usuwające" sugeruje, że tak naprawdę nie dochodzi do usunięcia elementów. Zamiast usuwać, funkcja `remove_if` kopiuje wszystkie te elementy, które nie spełniają predykatu (tu: wszystkie wpisy studentów, którzy kurs zaliczyli).

Wywołanie jest nieco karkołomne, ponieważ zakresem źródłowym i docelowym funkcji `remove_if` jest ten sam zakres tego samego kontenera. Funkcja ta kopiuje na początek kontenera elementy niespełniające predykatu. Załóżmy, przykładowo, że pierwotnie kontener zawierał wpisy siedmiu studentów z następującymi ocenami (rysunek 6.3):

Rysunek 6.3.
Przykładowa zawartość kontenera students

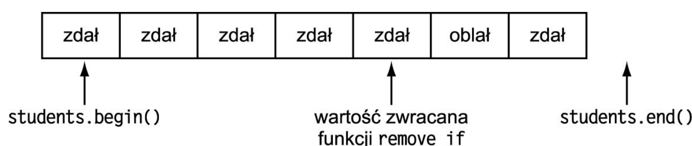
Funkcja `remove_if` pozostawi dwa pierwsze wpisy nietknięte, ponieważ znajdują się one w odpowiednich miejscach (na początku kontenera). Następne dwa wpisy zostaną „usunięte" w tym sensie, że zostaną potraktowane jako puste miejsca do wykorzystania przez następne wpisy, które powinny pozostać w kontenerze. Piąty wpis, reprezentujący studenta pozytywnie zaliczającego kurs, jest zatem kopiowany na pierwszą „wolną" pozycję kontenera, a więc na pozycję pierwszego „usuniętego" elementu kontenera (rysunek 6.4):

Rysunek 6.4.
Zawartość kontenera students po przetworzeniu go funkcją remove_if



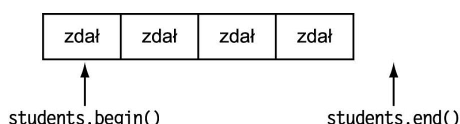
W takim układzie efektem wykonania funkcji `remove_if` będzie przesunięcie czterech wpisów studentów z ocenami pozytywnymi na początek kontenera — pozostałe wpisy pozostaną nietknięte. Aby wiadomo było, które z elementów kontenera należy wziąć pod uwagę w dalszym przetwarzaniu, funkcja `remove_if` zwraca iterator wskazujący element następny po ostatnim elemencie, który nie został „usunięty" (rysunek 6.5):

Rysunek 6.5.
Zawartość kontenera students a wartość zwracana przez funkcję remove_if



Po ułożeniu wpisów sumiennych studentów na początku kontenera, należy go obciąć, usuwając fizycznie pozostałe wpisy. Obcięcie realizowane jest za pomocą niewykorzystywanej dotąd wersji składowej `erase`. Przyjmuje ona dwa iteratory i usuwa wszystkie elementy z zakresu ograniczanego tymi iteratorami. Usuwając elementy pomiędzy iteratorem zwracanym przez funkcję `remove_if` i iteratorem `students.end()`, usuwamy z kontenera wszystkie wpisy studentów, którzy oblali kurs (rysunek 6.6):

Rysunek 6.6.
Zawartość kontenera students po wywołaniu składowej erase



6.3.2. Rozwiązanie jednoprzebiegowe

Przedstawione w poprzednim punkcie (§6.3.1/160) rozwiązanie działa znakomicie, ale nie oznacza to, że nie da się zadania rozwiązać jeszcze efektywniej. Widać bowiem, że kod prezentowany w §6.3.1/160 oblicza ocenę końcową każdego studenta dwukrotnie — raz w funkcji `remove_copy_if`, drugi raz w `remove_if`.

Choć nie istnieje algorytm biblioteki standardowej, który załatwiłby nasz problem w jednym wywołaniu, istnieje taki, którego wykorzystanie wiąże się z zupełnie innym podejściem do rozwiązania — algorytm ten przyjmuje na wejście sekwencję wartości i zmienia jej uporządkowanie tak, aby wartości spełniające zadany predykat znajdowały się przed wartościami niespełniającymi predykatu.

Algorytm ten dostępny jest w dwóch wersjach: `partition` i `stable_partition`. Różnica między nimi polega na tym, że `partition` dopuszcza zmianę wzajemnego uporządkowania wartości tej w ramach tej samej kategorii, podczas gdy `stable_partition` zachowuje wzajemne uporządkowanie wartości jednej kategorii. Gdyby, na przykład, kontener wpisów studentów zostałby wcześniej posortowany alfabetycznie, to do podziału studentów na grupy należałoby użyć algorytmu `stable_partition`.

Obie wersje algorytmu zwracają iterator reprezentujący pierwszy element drugiej grupy wartości. Dzięki nim możemy dokonać podziału studentów na dwie grupy w sposób następujący:

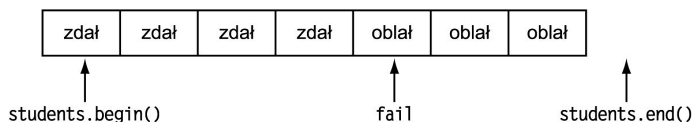
```
vector<Student_info> extract_file(vector<Student_info>& students)
{
    vector<Studetnt_info>::iterator iter =
        stable_partition(students.begin(), students.end(), pgrade);
    vector<Student_info> fail(iter, students.end());
    students.erase(iter, students.end());

    return fail;
}
```

Aby zrozumieć działanie powyższego kodu, najlepiej posłużyć się naszym hipotetycznym zbiorem danych (rysunek 6.7):

Rysunek 6.7. *Pierwotna zawartość kontenera students*

Po wywołaniu funkcji `stable_partition` otrzymujemy następujące uporządkowanie elementów kontenera (rysunek 6.8):



Rysunek 6.8. *Zawartość kontenera students po wywołaniu stable_partition*

Kontener `fail` został skonstruowany jako kopia wpisów studentów sklasyfikowanych jako oblewających kurs, czyli wpisów znajdujących się w zakresie `[iter, students.end())`. Elementy tego zakresu zostały następnie usunięte z kontenera `students`.

Uruchamiając nasze rozwiązania wykorzystujące algorytmy biblioteki standardowej, zauważymy, że wykonywane są one z wydajnością porównywalną z wydajnością naszego rozwiązania wykorzystującego kontenery typu `list`. Zgodnie z oczekiwaniami, wraz ze wzrostem liczby wpisów podawanych na wejście programu program wykorzystujący kontener typu list wyraźnie zwiększa przewagę nad swoim odpowiednikiem wykorzystującym kontener `vector`. Tymczasem obie wersje algorytmiczne są tak efektywne, że czas ich wykonania dla zbioru 75 000 wpisów jest w dużej mierze determinowany wydajnością operacji wejścia. Aby porównać efekty działania obu tych wersji, trzeba analizować osobno czas wykonania wyłącznie funkcji `extract_fails`. Pomiar czasu potwierdził, że implementacja jednoprzebiegowa działa prawie dwukrotnie szybciej od implementacji dwuprzebiegowej.

6.4. Algorytmy, kontenery, iteratory

Zrozumienie koncepcji kontenerów, iteratorów i algorytmów wymaga przyjęcia do wiadomości podstawowego założenia biblioteki standardowej wyrażanego zdaniem:

Algorytmy operują na elementach kontenerów — nie na samych kontenerach.

Funkcje `sort`, `remove` czy `partition` przesuwają elementy na nowe pozycje w ramach ich kontenera macierzystego, nie zmieniają jednak właściwości samego kontenera. Przykładowo, wywołanie `remove_if` nie zmienia rozmiaru kontenera, którego elementami operuje — ogranicza się do kopiowania elementów wewnątrz kontenera.

To rozróżnienie jest szczególnie istotne w zrozumieniu sposobu, w jaki algorytmy odwołują się do kontenerów określanych w tych algorytmach jako kontenery docelowe. Przyjrzyjmy się bliżej sposobowi użycia funkcji `remove_if` z punktu §6.3.1/160. Było to wywołanie o postaci:

```
remove_if(students.begin(), students.end(), fgrade);
```

Wywołanie to nie modyfikuje rozmiaru kontenera `students`. Kopiuje tylko elementy, dla których predykat `fgrade` daje wartość `false` na początek kontenera, pozostawiając resztę elementów nietkniętą. Jeżeli zachodzi potrzeba obcięcia kontenera typu `vector` w celu pozbycia się „usuniętych" elementów, musimy to zrobić samodzielnie.

W naszym kodzie przykładowym zapisaliśmy:

```
students.erase(remove_if(students.begin(), students.end(), fgrade),
students.end());
```

Tutaj składowa `erase` modyfikuje kontener, usuwając sekwencję określaną argumentami. Wywołanie to skraca kontener `students` tak, aby zawierał wyłącznie pożądane wpisy. Zauważmy, że `erase`, jako funkcja operująca nie tylko elementami kontenera, ale i nim samym (zmieniając jego rozmiar), musi być metodą obiektu kontenera.

Warto też uświadomić sobie kategorie interakcji pomiędzy iteratorami i algorytmami oraz pomiędzy iteratorami i operacjami na kontenerach. Przekonaliśmy się już (w §5.3/118 i §5.5.1/122) o tym, że operacje na kontenerach, jak `erase` czy `insert` unieważniają iteratory elementu usuwanego. Co ważniejsze, w przypadku typów `vector` i `string` operatory takie jak `insert` i `erase` powodują również unieważnienie wszelkich iteratorów odnoszących się do elementów znajdujących się *za* elementem wstawianym (usuwanym). Fakt potencjalnego unieważniania iteratorów w wyniku działania tych operacji zmusza do ostrożności przy ewentualnym zachowywaniu wartości iteratorów w zmiennych pętli.

Również wywołania funkcji takich jak `partition` czy `remove_if`, przemieszczających elementy wewnątrz kontenera, mogą doprowadzić do podmiany elementu wskazywanego przez iteratory kontenera — po wywołaniu jednej z takich funkcji nie można polegać na poprzednich wartościach iteratorów i odwoływać się za ich pomocą do elementów kontenera.

6.5. Podsumowanie

Modyfikatory typów:

`static` *typ zmienna*;

> Dla definicji lokalnych modyfikator `static` oznacza deklarację *zmiennej* przechowywanej statycznie. Wartość takiej zmiennej jest podtrzymywana pomiędzy wykonaniami zasięgu, w którym zmienna została zdefiniowana. Zmienna taka będzie też inicjalizowana przed pierwszym jej użyciem. Kiedy program opuszcza zasięg zmiennej, jej wartość jest zapamiętywana do czasu ponownego wkroczenia programu w jej zasięg. W §13.4/317 zobaczymy, że interpretacja modyfikatora `static` zmienia się w zależności od kontekstu.

Typy. Sposób wykorzystania wbudowanego typu `void` jest ograniczone; jednym z takich zastosowań jest określenie typu wartości zwracanej przez funkcję — funkcja zadeklarowana jako zwracająca wartość typu `void` nie zwraca żadnej wartości. Kod takiej funkcji należy kończyć instrukcją `return;`. Można też pominąć instrukcję `return` — funkcja zostanie zakończona po dotarciu do końca jej ciała.

Adaptory iteratorów to funkcje zwracające iteratory. Najpopularniejszymi takimi funkcjami są adaptory generujące iteratory `insert_iterator`, a więc iteratory, które dynamicznie zwiększają rozmiar skojarzonego z nimi kontenera. Iteratorów takich można używać jako określenia miejsca przeznaczenia w algorytmach kopiujących. Adaptory zdefiniowane są w nagłówku `<iterator>`. Wśród nich znajdziemy:

`back_inserter(c)`

> Zwraca iterator kontenera `c` pozwalający na dołączanie elementów na koniec kontenera `c`. Kontener musi obsługiwać metodę `push_back` (obsługują ją, miedzy innymi, kontenery typu `list`, `vector` i ciągi typu `string`).

`front_inserter(c)`

Zwraca iterator kontenera `c` pozwalający na wstawianie elementów na początek kontenera `c`. Kontener musi obsługiwać metodę `push_front` (obsługuje ją kontener typu `list`, ale nie `vector` i `string`).

`inserter(c, it)`

Jak `back_inserter`, tyle że wstawia elementy przed iterator `it`.

Algorytmy. O ile nie zaznaczono inaczej, algorytmy wykorzystywane w tekście rozdziału definiowane były w nagłówku `<algorithm>`. Znajdziemy tam następujące algorytmy:

`accumulate(b, e, t)`

Tworzy zmienną lokalną i inicjalizuje ją kopią wartości `t` (typu zgodnego z typem wartości `t`, co oznacza, że typ ten ma zasadnicze znaczenie dla sposobu działania algorytmu `accumulate`) zwiększoną o wartości wszystkich elementów z zakresu `[b, e)`. Algorytm zdefiniowany w nagłówku `<numeric>`.

```
find(b, e, t)
find_if(b, e, p)
search(b, e, b2, e2)
```

Algorytmy wyszukujące określoną wartość w sekwencji elementów z zakresu `[b, e)`. Algorytm `find` wyszukuje wartość `t`; `find_if` poddaje każdy z elementów testowi prawdziwości predykatu `p`; `search` wyszukuje w sekwencji `[b, e)` sekwencji `[b2, e2)`.

```
copy(b, e, d)
remove_copy(b, e, d, t)
remove_copy_if(b, e, d, p)
```

Algorytmy kopiujące sekwencję z zakresu `[b, e)` do miejsca docelowego wskazywanego wartością `d`. Algorytm `copy` kopiuje całość wskazanej sekwencji; `remove_copy` kopiuje wszystkie elementy różne od `t`; `remove_copy_if` kopiuje wszystkie elementy, dla których predykat `p` ma wartość `false`.

`remove_if(b, e, p)`

Zmienia uporządkowanie elementów w kontenerze w zakresie `[b, e)`, tak aby elementy niespełniające zadanego predykatu `p` znalazły się na początku kontenera. Zwraca iterator wskazujący element następny za ostatnim elementem przeniesionym na początek kontenera.

`remove(b, e, t)`

Podobna do `remove_if`, z tym, że w miejsce predykatu stosuje porównanie z wartością `t`.

`transform(b, e, f, d)`

Wykonuje funkcję `f` dla każdego elementu sekwencji `[b, e)`; wartość zwracaną przez funkcję `f` umieszcza w `d`.

```
partition(b, e, p)
stable_partition(b, e, p)
```

Grupuje elementy w zakresie `[b, e)` w zależności od predykatu `p`, tak aby elementy spełniające predykat znajdowały się przed elementami, dla których predykat daje wartość `false`. Zwraca iterator odnoszący się do pierwszego elementu grupy niespełniającej predykatu, ewentualnie zwraca `e`, jeżeli wszystkie elementy zakresu spełniały predykat. Wersja `stable_partition` zachowuje wzajemne uporządkowanie elementów w ramach grup.

Ćwiczenia

Ćwiczenie 6.0.

Skompiluj, uruchom i sprawdź działanie programów prezentowanych w tym rozdziale.

Ćwiczenie 6.1.

Zmodyfikuj funkcję `frame` i `hcat` z §5.8.1/130 i §5.8.3/132 tak, aby korzystały z iteratorów.

Ćwiczenie 6.2.

Napisz program testujący działanie funkcji `find_urls`.

Ćwiczenie 6.3.

Jak działa poniższy fragment kodu?

```
vector<int> u(10, 100);
vector<int> v;
copy(u.begin(), u.end(), v.begin());
```

Napisz program zawierający taki fragment, skompiluj go i uruchom.

Ćwiczenie 6.4.

Popraw program napisany w ramach poprzedniego ćwiczenia. Można to zrobić na, co najmniej, dwa sposoby. Zaimplementuj oba i opisz wzajemne zalety i wady obu rozwiązań.

Ćwiczenie 6.5.

Napisz funkcję analityczną korzystającą z funkcji `optimistic_analysis`.

Ćwiczenie 6.6.

Zauważ, że funkcja z poprzedniego ćwiczenia oraz funkcje z §6.2.2/155 i §6.2.3/158 realizują to samo zadanie. Połącz je w jedną funkcję analityczną.

Ćwiczenie 6.7.

Fragment programu analitycznego z §6.2.1/152, odpowiedzialny za wczytanie i klasyfikację wpisów studentów na podstawie liczby wykonanych prac domowych jest podobny do implementacji funkcji `extract_fails`. Napisz funkcję, która będzie rozwiązywała podobne podzadania.

Ćwiczenie 6.8.

Napisz pojedynczą funkcję, którą będzie można wykorzystać do klasyfikowania studentów w zależności od wskazanego kryterium. Sprawdź poprawność działania tej funkcji, stosując ją w miejsce funkcji `extract_fails`; wykorzystaj nową funkcję w programie analizującym oceny studentów.

Ćwiczenie 6.9.

Wykorzystaj algorytm biblioteki standardowej do konkatenacji wszystkich elementów kontenera typu `vector<string>`.