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++. Biblioteka IOStreams i lokalizacja programów



Autorzy: Angelika Langer, Klaus Kreft
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-739-6
Tytuł oryginału: Standard C++ IOStremas and Locales
Format: B5, stron: 608

Opanuj biblioteki strumieni i lokalizacji

- Poznaj budowę i architekturę biblioteki strumieni
- Zastosuj standardowe mechanizmy lokalizacji
- Stwórz własne usługi internacjonalizacji programów

Biblioteka strumieni (IOStreams) oraz biblioteka lokalizacji to jedne z najważniejszych komponentów języka C++, wykorzystywane przy dostowywaniu aplikacji do wymogów językowych krajów, w których są wydawane i wykorzystywane. Biblioteki te stosuje się do sterowania formatowaniem tekstów, definiowania własnych mechanizmów formatujących oraz tworzenia wersji językowych oprogramowania. Jednakże pomimo wielkiej wagi tych bibliotek, poświęcona im dokumentacja jest wyjątkowo skąpa – trudno znaleźć kompletne omówienie znajdujących się w nich klas oraz sposobów ich wykorzystania we własnych aplikacjach.

Książka „C++. Biblioteka IOStreams i lokalizacja programów" wypełnia tę lukę. Zawiera ona opis oraz dokumentację programistyczną klas biblioteki IOStreams i lokalizacji, a także prezentuje sposoby ich wykorzystywania i zaawansowane metody rozszerzania i dostosowywania do własnych potrzeb. Przedstawia zagadnienia związane z tworzeniem wersji językowych aplikacji i dostosowywania ich do lokalnych konwencji językowo-kulturowych.

- Formatowanie wejścia i wyjścia
- Operacje na strumieniach plikowych
- Synchronizowanie strumieni
- Architektura biblioteki IOStreams
- Internacjonalizacja i lokalizacja programów
- Szkielet lokalizacji
- Definiowanie własnych aspektów lokalizacji

Jeśli planujesz implementację własnych mechanizmów lokalizacji programów, ta książka będzie dla Ciebie doskonałym źródłem informacji.

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

Spis treści

Słowo wstępne 11
Wstęp 13
Wskazówki dla Czytelników 17

Część I Strumieniowe wejście-wyjście 23

1.
Podstawy biblioteki IOStreams 25

1.1. Wejście i wyjście 25
1.2. Formatowanie wejścia i wyjścia 33
1.2.1. Strumienie globalne biblioteki IOStreams 33
1.2.2. Operatory wejścia i wyjścia 34
1.2.3. Parametry formatowania strumienia 37
1.2.4. Manipulatory 43
1.2.5. Lokalizacja strumienia 47
1.2.6. Formatowanie wejścia i wyjścia — porównanie 48
1.2.7. Jeszcze o formatowaniu wejścia 49
1.3. Stan strumienia 51
1.3.1. Znaczniki stanu strumienia 52
1.3.2. Kontrola stanu strumienia 54
1.3.3. Przechwytywanie wyjątków strumieni 56
1.3.4. Zerowanie stanu strumienia 58
1.4. Operacje wejścia-wyjścia na strumieniach plikowych 59
1.4.1. Tworzenie, otwieranie, zamykanie i usuwanie strumieni plikowych 60
1.4.2. Tryby otwierania strumieni plikowych 61
1.4.3. Dwukierunkowe strumienie plikowe 65
1.5. Operacje wejścia-wyjścia na strumieniach w pamięci 67
1.6. Wejście i wyjście bez formatowania 69

1.7. Pozycjonowanie w strumieniu ... 71
1.8. Synchronizacja strumieni ... 73
1.8.1. Środki synchronizacji ... 74
1.8.1.1. Synchronizacja wywołaniami flush() i sync() ... 74
1.8.1.2. Synchronizacja za pośrednictwem znacznika formatu unitbuf ... 75
1.8.1.3. Synchronizacja przez wiązanie strumieni ... 75
1.8.2. Synchronizacja predefiniowanych strumieni standardowych ... 76
1.8.2.1. Synchronizacja pomiędzy predefiniowanymi strumieniami standardowymi ... 77
1.8.2.2. Synchronizacja predefiniowanego strumienia z jego analogiem w postaci standardowego pliku wejścia-wyjścia języka C ... 77
1.8.2.3. Synchronizacja predefiniowanego strumienia standardowego ze skojarzonym z nim urządzeniem zewnętrznym ... 78
1.8.2.4. Synchronizacja pomiędzy strumieniami znaków zwykłych i poszerzonych ... 78

2.

Architektura biblioteki IOStreams ... 79

2.1. Klasy strumieni ... 80
2.1.1. Hierarchia klas ... 81
2.1.1.1. Bazowe klasy strumieni ... 81
2.1.1.2. Ogólne klasy strumieni ... 84
2.1.1.3. Konkretne klasy strumieni ... 85
2.1.2. Zarządzanie buforem strumienia ... 86
2.1.3. Kopiowanie i przypisania obiektów strumieni ... 89
2.1.4. Zarządzanie lokalizacjami strumieni ... 92
2.1.5. Współpraca obiektów strumieni, buforów strumieni i lokalizacji ... 96
2.2. Klasy buforów strumieni ... 100
2.2.1. Hierarchia klas ... 101
2.2.2. Abstrakcja bufora strumienia ... 102
2.2.3. Bufory strumieni ciągów znaków ... 107
2.2.4. Bufory strumieni plikowych ... 113
2.3. Typy i cechy znakowe ... 122
2.3.1. Reprezentacje znaków ... 122
2.3.2. Cechy znakowe ... 127
2.3.2.1. Wymagania odnośnie typu cechy znakowej ... 127
2.3.2.2 Predefiniowane standardowe typy cech znakowych ... 131
2.3.3. Typy znakowe ... 132
2.3.3.1. Wymagania dla typów znakowych ... 132
2.4. Iteratory strumieni i buforów strumieni ... 135
2.4.1. Koncepcja iteratorów w bibliotece standardowej ... 136
2.4.2. Iteratory strumieni ... 138
2.4.2.1. Iterator strumienia wyjściowego ... 139
2.4.2.2. Iterator strumienia wejściowego ... 140
2.4.2.3. Iteratory strumieni a jednoprzebiegowość ... 144
2.4.3. Iteratory buforów strumieni ... 146
2.4.3.1. Iterator bufora strumienia wyjściowego ... 146
2.4.3.2. Iterator bufora strumienia wejściowego ... 147
2.5. Dodatkowa pamięć obiektu strumienia i wywołania zwrotne ... 150
2.5.1. Dodatkowa pamięć obiektu strumienia ... 151
2.5.2. Wywołania zwrotne strumieni ... 153

3.

Zaawansowane zastosowania biblioteki IOStreams 155

3.1. Wejście i wyjście dla typów definiowanych przez użytkownika 155
3.1.1. Sygnatury inserterów i ekstraktorów 156
3.1.2. Pierwsze własne insertery i ekstraktory 156
3.1.3. Udoskonalenia 159
3.1.3.1. Sterowanie formatowaniem 160
3.1.3.2. Operacje wstępne i likwidacyjne 161
3.1.3.3. Sygnalizowanie błędów 162
3.1.3.4. Umiędzynarodowienie 163
3.1.3.5. Operacje wejścia-wyjścia 165
3.1.4. Udoskonalone wersje insertera i ekstraktora daty 165
3.1.4.1. Umiędzynarodawianie daty 165
3.1.4.2. Czynności wstępne i likwidacyjne 167
3.1.4.3. Sterowanie formatowaniem 168
3.1.4.4. Sygnalizacja błędów 169
3.1.4.5. Zastosowanie udoskonalonego insertera i ekstraktora 176
3.1.5. Ogólne wersje insertera i ekstraktora 177
3.1.6. Wersja prosta a udoskonalona 182
3.2. Manipulatory definiowane przez użytkownika 184
3.2.1. Manipulatory bezparametrowe 185
3.2.2. Manipulatory sparametryzowane 187
3.2.2.1. Proste implementacje manipulatorów z parametrami 187
3.2.2.2. Wersja uogólniona — szablon klasy bazowej manipulatora 188
3.2.2.3. Warianty implementacji manipulatora 192
3.2.2.4. Udoskonalenia 194
3.2.2.5. Typ bazowy manipulatorów standardowych — smanip 198
3.3. Rozszerzanie funkcji strumienia 199
3.3.1. Własne zastosowania dodatkowej pamięci strumienia — iword, pword i xalloc 199
3.3.1.1. Inicjalizacja i utrzymywanie indeksu pary iword-pword 202
3.3.1.2. Implementacja insertera dat 202
3.3.1.3. Implementacja manipulatora 204
3.3.1.4. Zarządzanie pamięcią za pośrednictwem wywołań zwrotnych strumieni 206
3.3.1.5. Sygnalizowanie błędów funkcji wywołań zwrotnych 211
3.3.1.6. Stosowanie nowych funkcji 213
3.3.1.7. Ocena metody wykorzystującej pola iword-pword 213
3.3.2. Wyprowadzanie nowych klas strumieni 215
3.3.2.1. Wyprowadzanie nowej klasy strumienia 216
3.3.2.2. Implementacja insertera i manipulatora 223
3.3.2.3. Podsumowanie 226
3.3.3. Porównanie techniki iword-pword i rozbudowy hierarchii klas 227
3.4. Rozszerzanie funkcji bufora strumienia 229
3.4.1. Wyprowadzanie z klasy bazowej bufora strumienia 230
3.4.1.1. Rdzenne funkcje buforów strumieni — transport znaków 230
3.4.1.2. Opcjonalne funkcje buforów strumieni 246
3.4.1.3. Udostępnianie nowych klas strumieni z nowymi klasami buforów strumieni 248
3.4.2. Wyprowadzanie pochodnych konkretnych klas buforów strumieni 248

Część II Internacjonalizacja 251

4.

Wprowadzenie do internacjonalizacji i lokalizacji 253

4.1. Internacjonalizacja a lokalizacja 253
4.2. Konwencje kulturowe 254
4.2.1. Język 255
4.2.2. Wartości liczbowe 255
4.2.3. Wartości pieniężne 255
4.2.4. Godziny i daty 256
4.2.5. Porządek alfabetyczny 257
4.2.6. Komunikaty 258
4.2.7. Kodowanie znaków 258
4.2.7.1. Pojęcia i definicje 258
4.2.7.2. Zestawy znaków 259
4.2.7.3. Metody kodowania znaków 260
4.2.7.4. Zastosowania metod kodowania wielobajtowego i znaków poszerzonych 263
4.2.7.5. Konwersje kodowania 264

5.

Lokalizacja 267

5.1. Tworzenie obiektów lokalizacji 269
5.1.1. Nazwane obiekty lokalizacji 269
5.1.2. Lokalizacje kombinowane 270
5.1.3. Lokalizacja globalna 272
5.2. Pobieranie aspektów z lokalizacji 273
5.2.1. Funkcja has_facet 273
5.2.2. Funkcja use_facet 274

6.

Aspekty standardowe 277

6.1. Aspekty językowe i alfabetyczne 278
6.1.1. Klasyfikacja znaków 278
6.1.1.1. Klasyfikacja znaków 278
6.1.1.2. Konwersja znaków liter małych i wielkich 279
6.1.1.3. Konwersja znaków pomiędzy typami charT i char 280
6.1.1.4. Właściwości specjalne aspektu ctype<char> 281
6.1.2. Porządkowanie ciągów 281
6.1.3. Konwersja kodowania 283
6.1.4. Katalogi komunikatów 287
6.2. Aspekty formatowania i analizy leksykalnej 287
6.2.1. Wartości liczbowe i logiczne 288
6.2.1.1. Aspekt numpunct 288
6.2.1.2. Aspekt num_put 289
6.2.1.3. Aspekt num_get 290

6.2.2. Wartości pieniężne....291
6.2.2.1. Aspekt moneypunct....292
6.2.2.2. Aspekt money_put....293
6.2.2.3. Aspekt money_get....294
6.2.3. Wartości daty i czasu....296
6.2.3.1. Aspekt time_put....297
6.2.3.2. Aspekt time_get....298
6.3. Grupowanie aspektów standardowych w obiektach lokalizacji....300
6.3.1. Rodziny aspektów standardowych....300
6.3.1.1. Szablony klas bazowych aspektów standardowych....300
6.3.1.2. Aspekty byname....301
6.3.1.3. Zachowanie klasy bazowej aspektów....302
6.3.1.4. Obowiązkowe typy aspektów....303
6.3.2. Kategorie lokalizacji....305
6.3.3. Diagram kategorii aspektów....305
6.4. Zaawansowane zastosowania aspektów standardowych....307
6.4.1. Użycie aspektu za pośrednictwem obiektu strumienia....307
6.4.2. Użycie aspektu za pośrednictwem obiektu lokalizacji....308
6.4.3. Użycie aspektu wprost, niezależnie od obiektu lokalizacji....310

7.
Architektura szkieletu lokalizacji....313

7.1. Hierarchia klas....313
7.2. Identyfikacja i wyszukiwanie aspektów w lokalizacjach....313
7.2.1. Identyfikacja aspektów....314
7.2.2. Wyszukiwanie aspektów....317
7.2.2.1. Pobieranie aspektu z obiektu lokalizacji....317
7.2.2.2. Umieszczanie aspektów w obiekcie lokalizacji....321
7.2.2.3. Uzasadnienie dla zastosowania polimorfizmu dwufazowego....321
7.3. Zarządzanie pamięcią aspektów w obiektach lokalizacji....322
7.3.1. Licznik odwołań aspektu....322
7.3.2. Niezmienność aspektów w obiekcie lokalizacji....327

8.
Aspekty definiowane przez użytkownika....329

8.1. Dodawanie własnego aspektu do istniejącej rodziny aspektów....329
8.2. Definiowanie nowej rodziny aspektów....333

Dodatki 345

A
Podręcznik programisty....347

Lokalizacja....350
Plik nagłówkowy <locale>....350
Funkcje globalne....352
codecvt<internT, externT, stateT>....354
codecvt_base....359
codecvt_byname<internT, externT, stateT>....360
collate<charT>....362

collate_byname<charT> ... 364
ctype<charT> ... 366
ctype<char> ... 370
ctype_base ... 373
ctype_byname<charT> ... 374
locale ... 376
messages<charT> ... 381
messages_base ... 383
messages_byname<charT> ... 384
money_base ... 386
money_get<charT, InputIterator> ... 387
moneypunct<charT, Inter> ... 390
moneypunct_byname<charT, Inter> ... 394
money_put<charT, OutputIterator> ... 396
num_get<charT, InputIterator> ... 398
numpunct<charT> ... 403
numpunct_byname<charT> ... 406
num_put<charT, OutputIterator> ... 408
time_base ... 412
time_get<charT, InputIterator> ... 413
time_get_byname<charT, InputIterator> ... 417
time_put<charT, OutputIterator> ... 418
time_put_byname<charT, OutputIterator> ... 421
tm ... 423
Cechy znakowe ... 425
Plik nagłówkowy <string> ... 425
char_traits<charT> ... 427
char_traits<char> ... 428
char_traits<wchar_t> ... 431
IOStreams ... 434
Plik nagłówkowy <iosfwd> ... 434
Plik nagłówkowy <iostream> ... 437
Plik nagłówkowy <ios> ... 438
Plik nagłówkowy <streambuf> ... 440
Plik nagłówkowy <istream> ... 441
Plik nagłówkowy <ostream> ... 442
Plik nagłówkowy <iomanip> ... 443
Plik nagłówkowy <sstream> ... 444
Plik nagłówkowy <fstream> ... 445
Globalne definicje typów ... 446
Obiekty globalne ... 447
basic_filebuf<charT, traits> ... 449
basic_fstream<charT, traits> ... 453
basic_ifstream<charT, traits> ... 455
basic_ios<charT, traits> ... 457
basic_iostream<charT, traits> ... 461
basic_istream<charT, traits> ... 462
basic_istringstream<charT, traits, Allocator> ... 471
basic_ostream<charT, traits> ... 473
basic_ostream<charT, traits> ... 475
basic_ostringstream<charT, traits, Allocator> ... 483
basic_streamgbuf<charT, traits> ... 485
basic_stringbuf<charT, traits, Allocator> ... 492
basic_stringstream<charT, traits, Allocator> ... 495
fpos<stateT> ... 497

ios_base ... 498
Manipulatory ... 508
Iteratory strumieni ... 509
Plik nagłówkowy <iterator> ... 509
istreambuf_iterator<charT, traits> ... 511
istream_iterator<T, charT, traits, Distance> ... 515
iterator<Category, T, Distance, Pointer, Reference> ... 518
Znaczniki kategorii iteratorów ... 519
ostreambuf_iterator<charT, traits> ... 520
ostream_iterator<T, charT, traits, Distance> ... 522
Inne operacje wejścia-wyjścia ... 524
bitset<N> ... 524
complex<T> ... 525
basic_string<charT, traits, Allocator> ... 526

B

Analiza leksykalna i wyodrębnianie wartości liczbowych i logicznych ... 529

B.1. Analiza leksykalna wartości liczbowych ... 530
B.2. Analiza leksykalna wartości logicznych ... 531
B.3. Specyfikatory konwersji i modyfikatory długości ... 532

C

Formatowanie wartości liczbowych i logicznych ... 535

C.1. Formatowanie wartości liczbowych ... 535
C.2. Formatowanie wartości logicznych ... 538
C.3. Specyfikatory formatowania, kwalifikatory i modyfikatory długości ... 538

D

Specyfikatory formatu funkcji strftime() ... 541

E

Podobieństwa elementów biblioteki IOStreams do biblioteki języka C ... 543

E.1. Tryby otwierania plików ... 543
E.2. Pozycjonowanie w strumieniu ... 544

F

IOStreams — różnice pomiędzy implementacją klasyczną a standardową ... 545

F.1. Parametryzacja klas IOStreams ... 546
F.2. Podział klasy bazowej ios ... 547
F.3. Sygnalizowanie błędów ... 548
F.4. Umiędzynarodowienie biblioteki IOStreams ... 549
F.5. Brak klas _withassign ... 549
F.6. Brak deskryptorów plików ... 550
F.7. Strumienie ciągów znaków — stringstream zamiast strstream ... 551
F.8. Zmiany w klasach buforów strumieni ... 551
F.9. Zmiany pomniejsze ... 554

G

Powiązania mechanizmów lokalizacji w C i C++ 555

G.1. Kategorie lokalizacji w C i C++ 555
G.2. Globalne obiekty lokalizacji w C i C++ 555

H

Nowe elementy i idiomy języka C++ 559

H.1. Typy masek bitowych 559
H.2. POD 560
H.3. Konstruktory jawne 560
H.4. Specjalizacje szablonu 562
H.5. Domyślne argumenty szablonów 568
H.6. Jawna specyfikacja argumentu szablonu 571
H.7. Słowo typename 572
H.8. Rzutowanie dynamiczne 575
H.9. Funkcyjne bloki try 579
H.10. Wyjątki standardowe 582
H.11. Ograniczenia zakresów liczbowych 583
H.12. Ciągi języka C++ 583

Bibliografia 585

Skorowidz 587

4

Wprowadzenie do internacjonalizacji i lokalizacji

Współczesne oprogramowanie, jeśli ma się dobrze sprzedawać, powinno być tak użyteczne, jak i atrakcyjne dla użytkowników na całym świecie. Oczywiście użytkownicy komputerów w poszczególnych krajach z pewnością chcieliby, aby program komunikował się z nimi w ich ojczystym języku i zgodnie z przyjętymi lokalnie konwencjami kulturowymi. Programista produktu przeznaczonego na zróżnicowane pod tym kątem rynki musi wbudować w swoje dzieło możliwość adaptacji do owych konwencji.

Niniejszy rozdział stanowić będzie wprowadzenie do internacjonalizacji przeznaczone dla tych Czytelników, którzy chcieliby umiędzynarodowić swoje programy pisane w języku C++, ale nie zetknęli się jeszcze z zagadnieniem internacjonalizacji jako takim. Rozdział ten jest więc świetnym źródłem podstawowych informacji o rzeczonym problemie.

Na początek winniśmy Czytelnikowi wyjaśnienie pojęć *internacjonalizacja* i *lokalizacja*, ich wzajemnych koligacji i różnic pomiędzy nimi. Następnie wskażemy na różnice kulturowe. Różnice te decydują o złożoności problemu internacjonalizacji aplikacji. Do różnic kulturowych zaliczylibyśmy (między innymi) przyjęty lokalnie sposób zapisu liczb, symboli walut, wartości pieniężnych, czasu i daty, kolejności słów, kodowania znaków. Wszystkie te konwencje postaramy się zaprezentować na przykładach. Poświęcimy też spory podrozdział kodowaniu znaków, ponieważ zagadnienie to ma dla programistów szczególne znaczenie. Różnice w sposobie kodowania znaków będą dla nas tym ważniejsze, że wpływają na sposób realizacji operacji wejścia-wyjścia biblioteki IOStreams.

4.1. Internacjonalizacja a lokalizacja

Dwoma najczęściej wykorzystywanymi w programowaniu aplikacji dla międzynarodowych rynków zbytu terminami są internacjonalizacja i lokalizacja. Czasami oba pojęcia są podciągane pod pojęcie internacjonalizacji[1], jednak, aby rozróżnić niuanse znaczeniowe, powinniśmy na potrzeby dalszego omówienia sprecyzować i rozdzielić *internacjonalizację* od *lokalizacji*[2].

[1] Słowo internacjonalizacja doczekało się w języku angielskim zabawnego akronimu w postaci I18N. Został on utworzony z pierwszej (*I*) i ostatniej (*N*) litery słowa *internationalization* oraz liczby liter pomiędzy literami skrajnymi (*18*).

[2] Zaprezentowane dalej rozróżnienie nie jest naszym wynalazkiem — lokalizacja i internacjonalizacja to popularne terminy, posiadające w dziedzinie inżynierii oprogramowania dobrze zdefiniowane znaczenie.

Internacjonalizacja. *Internacjonalizacja* (umiędzynarodawianie) to wysiłek związany z wbudowaniem do oprogramowania potencjału globalnej użyteczności. Wymaga ona wprowadzenia do oprogramowania środków adaptacji do lokalnych konwencji kulturowych. Projektanci i programiści oprogramowania umiędzynarodowionego muszą uwzględnić możliwość tej adaptacji już na etapie projektu. Unikają więc „sztywnego" kodowania elementów, które wymagałyby potencjalnie lokalizacji. Na przykład umiędzynarodowione oprogramowanie nigdy nie zawiera tekstów komunikatów w kodzie źródłowym — wszelkie napisy wykorzystywane w komunikacji z użytkownikiem są przechowywane poza właściwym programem, na przykład w katalogu komunikatów. Taki rozdział kodu źródłowego od napisów wyświetlanych w czasie działania programu umożliwia podmianę owego tekstu na różne jego wersje językowe bez konieczności modyfikowania samego programu. Ponadto programiści nie powinni nigdy czynić założeń co do konwencji formatowania wartości liczbowych czy pieniężnych bądź sposobów wyświetlania dat i czasu. Reguły formatowania (i analizy leksykalnej) takich elementów powinny być wyodrębnione poza właściwy kod programu, tak aby były od niego maksymalnie niezależne.

Lokalizacja. *Lokalizacja* to proces przystosowywania oprogramowania do danego obszaru geograficzno-kulturowego. Obejmuje tłumaczenie tekstów przechowywanych w katalogu komunikatów i tworzenie tabel opisujących przyjęte lokalnie konwencje kulturowe. Lokalizacja jest zazwyczaj zadaniem osób przygotowujących dany produkt programowy do dystrybucji na danym rynku regionalnym. Rzeczone katalogi komunikatów i tabele powinny być następnie udostępnione użytkownikom oprogramowania umiędzynarodowionego. Udostępnianie to leży zwykle w gestii administratorów systemów komputerowych, którzy do lokalizacji wykorzystują cechy i funkcje systemu operacyjnego. Użytkownicy umiędzynarodowionego oprogramowania powinni móc wybrać preferowany język komunikacji z programem i preferowane konwencje lokalne albo przed uruchomieniem aplikacji, albo już w trakcie jej używania.

Podsumowując, internacjonalizacja to proces polegający na udostępnieniu w oprogramowaniu środków i mechanizmów umożliwiających ewentualną adaptację tego oprogramowania do odmiennych obszarów kulturowo-geograficznych. W niniejszej książce skupimy się właśnie na internacjonalizacji, ignorując kwestie samej lokalizacji jako wtórne.

Umiędzynarodawianie oprogramowania można realizować na wiele sposobów. W językach C i C++ dostępne są standaryzowane komponenty internacjonalizacji wchodzące w skład bibliotek standardowych tych języków i towarzyszące każdemu systemowi wykonawczemu. Omówimy te z owych komponentów, które są dostępne w bibliotece standardowej języka C++, a które noszą miana *lokalizacji* i *aspektów*. Zanim jednak zagłębimy się w detalach owych komponentów, powinniśmy najpierw zyskać pogląd na istotę i zakres różnić kulturowych, na które wciąż się powołujemy. Kolejne ilustrujące je przykłady nie będą wyczerpywać tematu — przy umiędzynarodawianiu oprogramowania należy wziąć pod uwagę również wiele nieporuszanych w nich kwestii, jak choćby orientację tekstu, rozmiary i pozycjonowanie tekstu na ekranie, wyświetlanie pionowe, wybór tabel czcionek, obsługę międzynarodowych klawiatur i wiele innych. Poniżej ograniczamy się do tych zagadnień, które znalazły swoje odzwierciedlenie w komponentach biblioteki standardowej języka C++.

4.2. Konwencje kulturowe

Ludzie żyjący w różnych regionach świata porozumiewają się w różnych językach i pozostają pod wpływem różnych konwencji kulturowych, na przykład inaczej zapisują liczby czy daty, jak również przyjmują odmienne porządki alfabetyczne. Wszystkie te czynniki wpływają na kształt umiędzynarodowionego oprogramowania.

4.2.1. Język

Różne grupy etniczne posługują się różnymi *językami*, a oczywiście język jest najbardziej podstawowym wyróżnikiem kultur. Nawet w granicach administracyjnych jednego kraju mogą obowiązywać różne języki. Dotyczy to choćby Szwajcarów, którzy posługują niemieckim, francuskim i włoskim.

Języki różnią się wykorzystywanymi alfabetami. Oto kilka przykładów różnych języków i odpowiadających im alfabetów:

Amerykański angielski:	znaki a do z, A do Z i znaki przestankowe.
Niemiecki:	znaki a do z, A do Z, znaki przestankowe oraz znaki diakrytyczne (äöü ÄÖÜ ß).
Grecki:	znaki α do ω, A do Ω i znaki przestankowe.
Japoński:	znaki alfabetu języka amerykańskiego angielskiego oraz dziesięć tysięcy znaków alfabetów Kanji, Hiragana i Katakana.

4.2.2. Wartości liczbowe

Również reprezentacja liczb podlega konwencjom kulturowym. Na przykład symbol wykorzystywany do oddzielania części całkowitej od części ułamkowej liczby, zwany znakiem *przecinka dziesiętnego*[3], może w różnych krajach i językach wyglądać inaczej. W języku angielskim znakiem tym jest kropka; w większości pozostałych języków europejskich (w tym w niemieckim[4]) w tej samej roli występuje przecinek. Analogiczne różnice pojawiają się w obrębie symbolu oddzielającego grupy cyfr w liczbach ponadtrzycyfrowych — symbol ów, znany jako separator tysięczny, w języku angielskim ma postać przecinka, a w większości języków europejskich — kropki[5].

Różni się nawet sposób grupowania cyfr w liczbach wielocyfrowych. W języku amerykańskim angielskim cyfry są grupowane trójkami, ale na przykład w Nepalu przyjęło się, że pierwsza grupa cyfr to grupa trójkowa, lecz wszystkie kolejne zawierają już tylko po dwie cyfry:

Stany Zjednoczone:	10,000,000.00
Francja:	10.000.000,00
Polska:	10 000 000,00
Nepal:	1,00,00,000.00

4.2.3. Wartości pieniężne

W różnych regionach rozmaicie prezentują się wartości pieniężne i symbole walut. Symbol waluty może różnić się położeniem względem wartości pieniężnej. Różne są też formaty zapisu ujemnych wartości pieniężnych.

---

[3] W języku amerykańskim angielskim symbol ten (z angielska *radix character*) — ponieważ jest reprezentowany kropką — zwany jest również znakiem kropki dziesiętnej. W języku polskim z analogicznych powodów nazywamy go przecinkiem dziesiętnym — *przyp. tłum.*

[4] Oraz polskim — *przyp. tłum.*

[5] W języku polskim w tej roli występuje spacja — *przyp. tłum.*

Tę samą ilość gotówki w walucie amerykańskiej można wyrazić następująco:

wewnętrznie:	$ 99.99
międzynarodowo:	USD 99.99

Kolejny przykład pokaże, że w wartościach pieniężnych różnie umiejscowiony jest symbol waluty:

Niemcy (przed wprowadzeniem wspólnej waluty europejskiej):	49,99 DM
Japonia:	¥ 100
Wielka Brytania (przed wprowadzeniem miar dziesiętnych):	£13 18s. 5d.

A oto, jak mogą się różnić reprezentacje wartości ujemnych:

Hong Kong:	HK$0.95	(HK$0.95)
Niemcy:	0,95 DM	–0,95 DM
Austria:	ÖS 0,95	–ÖS 0,95
Szwajcaria:	SFr. 0.95	SFr. –0.95

4.2.4. Godziny i daty

Jak można się spodziewać, od konwencji kulturowych zależny jest również sposób zapisywania godzin i dat. Jedną (ale nie jedyną) z różnic stanowią skróty nazw dni tygodni i miesięcy. Ponadto w niektórych krajach stosuje się format 24-godzinny, w innych zaś 12-godzinny. Różnią się nawet kalendarze — bazują one na zdarzeniach historycznych, astronomicznych i na porach roku. Oficjalny kalendarz japoński jest na przykład oparty na wydarzeniach historycznych, z których najważniejszym jest początek panowania aktualnego cesarza. W wielu krajach, szczególnie w świecie zachodnim, wykorzystywany jest kalendarz gregoriański.

Poniżej prezentowanych jest kilka przykładów reprezentacji tej samej daty w kilku różnych krajach. Reprezentacje te różnią się kolejnością występowania dni, miesięcy i lat, znakami separatorów pomiędzy tymi elementami oraz zastosowaniem możliwości pominięcia elementów takich jak nazwa dnia tygodnia (jak w wydłużonym zapisie daty w języku węgierskim):

	Zapis skrócony	**Zapis wydłużony**
Stany Zjednoczone Ameryki:	10/14/97	Tuesday, October 14, 1997
Niemcy:	14.10.97	Dienstag, 14. Oktober 1997
Włochy:	14/10/97	martedi 14 ottobre 1997
Grecja:	14/10/1997	Τρίτ, 14 ηΟκτωβρίου 1997
Węgry:	1997.10.14	1997. október 14.

Ta sama godzina może w różnych językach być zapisywana zupełnie inaczej. Oto przykład zapisu tej samej godziny w formacie 24- i 12-godzinnym:

Niemcy:	17:55 Uhr
Stany Zjednoczone Ameryki:	5:55 PM

4.2.5. Porządek alfabetyczny

W różnych językach mamy do czynienia z różnymi regułami porządkowania alfabetycznego słów. Reguły te określane są mianem *porządku leksykograficznego* (ang. *collating sequence*). Wyznacza kolejność poszczególnych znaków i inne reguły porządkowania. W niektórych językach na przykład pewne grupy znaków są przy porządkowaniu traktowane jak jeden znak. W innych językach zdarza się z kolei, że jeden znak traktowany jest jak dwa kolejne znaki.

W programowaniu kolejność znaków jest często wyznaczana wartością liczbową bajta (bajtów) reprezentującego znak. Zilustrowaliśmy to w poniższym przykładzie w kolumnie *ASCII*. Tego rodzaju porządkowanie nie spełnia jednak wymagań żadnego z języków naturalnych.

Oto przykład różnic pomiędzy porządkiem ASCII a porządkiem leksykograficznym języka angielskiego:

Angielski	**ASCII**
alien	American
American	Zulu
zebra	alien
Zulu	zebra

W kodzie ASCII wartości liczbowe znaków liter wielkich są mniejsze od wartości liczbowych liter małych, przez co wszystkie słowa zawierające wielkie litery lądują na początku listy posortowanej według wartości kodów ASCII.

Oto przykład podobnych różnic w porządkowaniu dwuznaków:

Hiszpański	**ASCII**
año	año
cuchillo	chaleco
chaleco	cuchillo
dónde	dónde
lunes	llava
llava	lunes
maíz	maíz

Słowo `chuchillo` jest porządkowane przed słowem `chaleco`, ponieważ w języku hiszpańskim `ch` to dwuznak, co oznacza, że powinien być traktowany jako pojedynczy znak i porządkowany pomiędzy znakiem `c` a znakiem `d`. Zasadniczo dwuznaki są kombinacjami znaków zapisywanych oddzielnie, ale tworzących spójną jednostkę leksykalną. Innym przykładem dwuznaku w wymienionych wyrazach jest dwuznak `ll`, porządkowany po `l`, ale przed `m`.

Oto kolejny przykład, tym razem dotyczący języka niemieckiego:

Niemiecki	**ASCII**
Musselin	Musselin
Muße	Muster
Muster	Muße

Znak ß z niemieckiego alfabetu jest traktowany jak dwa znaki — *ss*. Z tego powodu w języku niemieckim jest porządkowany za *ss* i przed *st*.

4.2.6. Komunikaty

Program zawiera zazwyczaj wiele elementów tekstowych, które są w różnych momentach wykonania programu prezentowane użytkownikowi. Przykładami są choćby komunikaty o błędach, etykiety elementów interfejsu graficznego aplikacji czy napisy umieszczone w nagłówkach i stopkach drukowanych z poziomu aplikacji stron. Program umiędzynarodowiony nie powinien definiować żadnego z takich napisów w kodzie źródłowym programu. Wyodrębnienie tekstu komunikatów od kodu programu jest uzasadnione zależnością tych pierwszych od języka — jeśli program ma znaleźć nabywców na rynku globalnym, trzeba przetłumaczyć napisy na poszczególne języki narodowe.

Istnieje znana i uznana technika izolacji zależnego od języka interfejsu tekstowego od kodu źródłowego aplikacji — zakłada ona wykorzystanie *katalogu komunikatów*. Taki katalog może być plikiem albo bazą danych, którą można modyfikować niezależnie od kodu programu. Sam program zaś, zamiast definiować ciągi w kodzie źródłowym, może odwoływać się do katalogu i z niego pobierać komunikaty prezentowane użytkownikom.

4.2.7. Kodowanie znaków

Znaki grają pierwszorzędną rolę w przetwarzaniu tekstów. Wyjaśnijmy sobie zatem, czym jest znak i jakie znaczenie mają pojęcia zestawu znaków, kodowania znaków, znaku wielobajtowego i poszerzonego; przy okazji zastanowimy się nad sposobem obsługi znaków w programowaniu[6].

4.2.7.1. Pojęcia i definicje

Znak. Stanowi on w programowaniu pewną abstrakcję. Naturalne pojmowanie znaku identyfikuje go jako dający się zapisać symbol — ludzie przywykli do kojarzenia znaków ze znanymi im symbolami. Tak zdefiniowany znak dysponuje pewną reprezentacją graficzną i może pojawić się na ekranie albo zostać wydrukowany przez drukarkę. Należy przy tym pamiętać, że jeden i ten sam znak może mieć tutaj więcej niż jedną reprezentację graficzną. Trójkę znaków ABC możemy przecież zapisać jako ***ABC***, ABC czy **ABC**. Mówimy wtedy o czcionce znaku. Doszliśmy więc do tego, że jednym z aspektów znaku jest jego reprezentacja graficzna. Jednakże na potrzeby przetwarzania tekstu w procesie tworzenia oprogramowania znak musi również posiadać reprezentację w postaci ciągu bitów. Ciąg ten nazywamy *kodem znaku*.

Kod znaku. Jest to sekwencja bitów reprezentująca dany znak. Kod taki może mieć wiele reprezentacji. Znak litery „a” może na przykład być reprezentowany jako 0×61 (w kodzie ASCII[7]) czy 0×81 (w kodzie EBCDIC[8]) czy wreszcie jako 0×0061 (w standardzie Unicode[9]). Jak widać,

[6] Obsługę kodowania znaków w bibliotece IOStreams omawia podrozdział 2.3, „Typy i cechy znakowe”.

[7] ASCII to skrót od *American Standard Code for Information Exchange* (amerykański standard kodowania w wymianie informacji). Definiuje on zestaw siedmiobitowych kodów będący amerykańskim wariantem wykorzystywanego międzynarodowo zestawu IOS646.

[8] EBCDIC to skrót od *Extended Binary Coded Decimal Interchange Code*. Jest to kod 8-bitowy zdefiniowany przez IBM.

różnice w poszczególnych kodach dotyczą nie tylko liczby bitów kodujących znak, ale i ich wzorca bitowego; wzorzec bitowy reprezentujący literę „a” ma w standardach EBCDIC i ASCII rozmiar 8 bitów, a w Unicode — 17 bitów.

Zestaw znaków. To odwzorowanie jeden do jednego pomiędzy znakami a kodami znaków. Fragment zestawu znaków ASCII prezentowany jest w tabeli 4.1.

TABELA 4.1.
Fragment zestawu ASCII

Znak	…	?	@	A	B	C	D	E	F	…
Kod znaku (wyrażony szesnastkowo)	…	0×3F	0×40	0×41	0×42	0×43	0×44	0×45	0×46	…

Zwykle wszystkie kody znaków w danym zestawie mają równą liczbę bitów. Przykładami zestawów znaków są:

- tradycyjny 7-bitowy zestaw ASCII, w którym każdy ze znaków alfabetu języka amerykańskiego angielskiego jest kodowany na siedmiu bitach,
- IOS 8859-1, zestaw przyjęty dla krajów Europy Zachodniej, w którym każdy znak kodowany jest jednym bajtem,
- Unicode, w którym każdy ze znaków zajmuje dwa bajty.

Więcej informacji o zestawach znaków można znaleźć w podpunkcie 4.2.7.2, „Zestawy znaków”.

Metoda kodowania znaków. Jest to zestaw reguł wyznaczających sposób translacji sekwencji bajtowej do sekwencji znaków. Zdefiniowane z góry metody kodowania są niezbędne w przypadkach, kiedy do kodowania danego zbioru znaków wykorzystuje się różne zestawy znaków. Na przykład w alfabecie japońskim wyróżnia się cztery różne zbiory znaków. Znaki te są reprezentowane szeregiem różnych zestawów znaków. Niektóre z nich zawierają kody jednobajtowe, inne zaś dwubajtowe. Jest rzeczą oczywistą, że w przypadku tak zróżnicowanego zbioru zestawów znaków mieszającego reprezentację jedno- i dwubajtową należy ustalić reguły analizy leksykalnej ciągu bajtów, jeśli ma on zawierać znaki alfabetu japońskiego. W takim układzie interpretacja wartości bieżącego bajta zależna jest bowiem od kontekstu jego wystąpienia. Pojedynczy bajt może równie dobrze reprezentować pojedynczy znak, jak i pierwszy (albo drugi) z bajtów znaku dwubajtowego. Reguły rozstrzygające pomiędzy tymi przypadkami noszą nazwę metody kodowania znaków, w skrócie *kodowania*. Kodowanie znaków omówimy szczegółowo w podpunkcie 4.2.7.3, „Metody kodowania znaków”.

4.2.7.2. Zestawy znaków

Zestawy znaków mogą być klasyfikowane według liczby bitów reprezentujących pojedynczy znak.

7-bitowy zestaw ASCII jest zestawem znaków tradycyjnie już wykorzystywanym do kodowania znaków alfabetu języka angielskiego. 7-bitowa precyzja pozwala na zakodowanie 127 znaków, co wystarcza do jednoznacznej reprezentacji wszystkich znaków języka angielskiego wraz z szeregiem znaków pomocniczych i sterujących.

[9] Unicode to standard kodowania zakładający stałą, 16-bitową szerokość reprezentacji znaku, opracowany i promowany przez konsorcjum Unicode Consortium, niekomercyjną organizację zrzeszającą przedstawicieli przemysłu komputerowego.

Do przetwarzania tekstów w językach obowiązujących w większości krajów Europy (również Europy Wschodniej) i Azji wykorzystuje się *zestawy 8-bitowe*. Języki te korzystają z alfabetów obszerniejszych od alfabetu języka angielskiego. Niektóre z tych 8-bitowych zestawów są prostymi rozszerzeniami 7-bitowego ASCII — definiują wszystkie 7-bitowe kody ASCII, a dodatkowe znaki alfabetu języka narodowego rozmieszczają w bajtach o wartości powyżej 127. Tego rodzaju rozszerzenia — jednym z nich jest ISO 8859-1 — spełniają wymagania użytkowników komputerów w Europie Zachodniej[10] — zestaw ASCII jest w nich uzupełniany kodami znaków diakrytycznych i specjalnych charakterystycznych dla danego języka. Jednak dla niektórych języków tego regionu, jak choćby greckiego czy hebrajskiego, stosowane są zupełnie odmienne alfabety, niezawierające żadnych znaków języka angielskiego (czyli alfabetu łacińskiego). Aby zakodować znaki tych alfabetów zaprojektowane kolejne zestawy znaków: ISO 8859-5 (cyrylica), ISO 8859-6 (języki arabskie), ISO 8859-7 (grecki) czy ISO 8859-8 (hebrajski).

W *zestawach znaków poszerzonych* wszystkie każdy ze znaków jest reprezentowany więcej niż jednym bajtem. Zestawy takie okazują się niezbędne, kiedy dany alfabet lub grupa obsługiwanych w zestawie alfabetów mają łącznie więcej niż 256 znaków. Najbardziej spektakularnym przykładem jest alfabet japoński, składający się z tysięcy znaków. Japoński komitet standaryzacji przemysłu zdefiniował dwa zestawy znaków alfabetu japońskiego: JIS X 0208-1990[11], zawierający najczęściej wykorzystywane znaki, oraz JIS X 0212-1990[12], grupujący znaki rzadziej wykorzystywane. Innymi przykładami zestawów znaków poszerzonych są zestawy przyjęte jako międzynarodowe standardy, jak choćby Unicode, ISO 10646.UCS-2 czy ISO 10646.UCS-4[13]. Unicode i ISO 10646.UCS-2 są zestawami znaków dwubajtowych; ISO 10646.UCS-4 definiuje znaki czterobajtowe.

4.2.7.3. Metody kodowania znaków

Jak już wspomnieliśmy, metoda kodowania wyznacza reguły translacji sekwencji bajtowych na sekwencję kodów znaków.

Jeśli wszystkie znaki w tekście reprezentowane są kodami zestawu znaków o *stałej liczbie bajtów na znak*, translacja jest trywialna — mamy wtedy do czynienia z odwzorowaniem jeden do jednego pomiędzy znakiem a wzorcem bitowym zdefiniowanym dla danego znaku. W przypadku zestawów kodów jednobajtowych sekwencja bajtów przetwarzana jest porcjami jednobajtowymi — po odczytaniu kodu bajta z sekwencji interpretuje się ów bajt jako określony znak. W przypadku zestawów znaków poszerzonych przetwarzanie odbywa się porcjami wielobajtowymi odpowiednimi dla rozmiaru bajtowego znaku, ale zasadniczo nie różni się od przetwarzania sekwencji bajtów kodowania jednobajtowego. Jeśli więc tekst zawiera znaki zestawu znaków dwubajtowych, jednostka przetwarzania ma rozmiar dwóch bajtów; tekst składający się ze znaków czterobajtowych będzie przetwarzany czwórkami bajtów.

Translacja jest nieco bardziej złożona w przypadku *mieszanki zestawów znaków*. Jeśli znaki do przetworzenia pochodzą z wielu różnych zestawów znaków, nie można przyjąć strategii odwzorowania jeden do jednego. Należy wtedy wziąć pod uwagę reguły kodowania zdefiniowane w ramach metody kodowania. Na przykład w różnych zestawach znaków ten sam kod może reprezentować różne znaki. Aby prawidłowo odwzorować kody na znaki, należy uzupełnić przetwarzanie

[10] W Europie Wschodniej obowiązuje ISO 8859-2 — *przyp. tłum.*

[11] Oficjalną nazwą dla ISO X 0208-1990 jest *Code of the Japanese Graphic Character Seto for Information Exchange*.

[12] Oficjalną nazwą dla ISO X 0212-1990 jest *Code of the Japanese Graphic Character Seto for Information Exchange*.

[13] ISO 10646 to kodowanie zdefiniowane przez ISO, międzynarodową organizację standaryzacyjną. ISO 10646.USC-2 to ekwiwalent zestawu Unicode.

o informacje o kontekście mówiącym np. o tym, który z wielu zestawów kodowania powinien obowiązywać dla bieżącego znaku.

Sprawy komplikują się jeszcze bardziej, kiedy zestaw znaków zawiera znaki kodowane na różnej liczbie bajtów. W takich przypadkach nie wiadomo nawet z góry, ile bajtów tworzy konstytuuje w danym tekście pojedynczy znak. Reguły kodowania określające sposób translacji tekstów znaków różnych rozmiarów będziemy określać mianem *metod kodowania wielobajtowego*. Sekwencje znaków wielobajtowych są zasadniczo przetwarzane porcjami jednobajtowymi, ponieważ bajt jest najmniejszą dającą się interpretować jednostką informacji w takiej sekwencji.

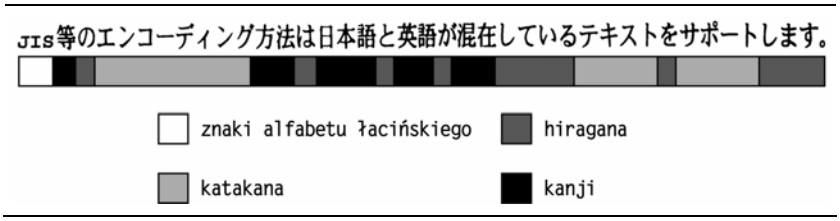
Zrozumienie metod kodowania wielobajtowego będzie łatwiejsze, kiedy metody te zostaną zaprezentowane na przykładach. Spójrzmy więc na kilka przykładów, bazujących na metodach kodowania przyjętych dla tekstów w języku japońskim.

4.2.7.3.1. Metody kodowania znaków wielobajtowych dla języka japońskiego

W języku japońskim tekst może zawierać znaki pochodzące z czterech różnych systemów zapisu:

- *kanji*, składającego się z tysięcy znaków, reprezentowanych ideogramami,
- *hiragana* i *katakana*, składające się z po około 80 znaków, reprezentujących sylaby,
- *alfabetu łacińskiego*, obejmującego 95 liter, cyfr i znaków przestankowych.

Przykład zdania w języku japońskim, zawierającego symbole z czterech systemów zapisu, widzimy na rysunku 4.1.



RYSUNEK 4.1. Zdanie w języku japońskim mieszające wiele przyjętych w tym kraju systemów zapisu

Zdanie to można przetłumaczyć jako: „Metody kodowania, jak JIS, pozwalają na mieszanie w tekście japońskiego i angielskiego".

Znaki pochodzące z czterech systemów zapisu można by reprezentować następująco:

(1) Za pomocą jednego olbrzymiego zestawu znaków poszerzonych obejmującego wszystkie znaki alfabetów japońskich. Taki model przyjęto w standardach międzynarodowych, jak Unicode czy ISO 10646.
(2) Za pomocą mieszanki kilku zestawów znaków, z których część definiuje znaki jednobajtowe, a inne zakładają kodowanie dwubajtowe. Każdy z zestawów znaków reprezentuje wtedy podzbiór znaków japońskich. Takie podejście zastosowano w standaryzacji na potrzeby przemysłu japońskiego realizowanej przez japoński przemysłowy komitet standaryzacyjny JISC (ang. *Japan Industry Standard Comitee*).

Rozważmy przypadek drugi, ponieważ na jego przykładzie można będzie zaobserwować zastosowanie kodowania wielobajtowego. Przemysł japoński wykorzystuje lokalne, narodowe standardy kodowania znaków. Znaki pochodzące z czterech systemów zapisu są reprezentowane szeregiem zestawów znaków, z których najpopularniejsze zostały wymienione w tabeli 4.2. Każdy z zestawów znaków zawiera wtedy podzbiór znaków języka japońskiego.

TABELA 4.2.
Zestawy znaków języka japońskiego

Zestaw znaków	Zawierane znaki	Rozmiar kodu
JIS X 0208-1990	Najczęściej wykorzystywane ideogramy kanji.	2 bajty
JIS X 0212-1990	Rzadziej wykorzystywane ideogramy kanji.	2 bajty
JIS_ROMAN[14]	Zestaw ASCII uzupełniony o znaki katakana.	1 bajt

Translacji sekwencji bajtów do postaci sekwencji znaków można w takim układzie dokonywać na wiele sposobów. Istnieje więc wiele metod kodowania dla języka japońskiego, przy czym żadna z nich nie jest uniwersalna. Stosuje się trzy metody kodowania wielobajtowego: JIS (ang. *Japanese Industrial Standard*), Shift-JIS oraz EUC (ang. *Extended UNIX Code*).

Wymienione metody kodowania podzieliły się zastosowaniami:

- Kodowanie *JIS* jest wykorzystywane przede wszystkim w transmisjach realizowanych elektronicznie, na przykład w wymianie poczty elektronicznej — chodzi o to, że koduje ona każdy znak tylko siedmioma bitami. To bardzo pożądana cecha, ponieważ niektóre protokoły sieciowe nie przewidują transmisji znaków 8-bajtowych. Do przełączania pomiędzy trybem jedno- i dwubajtowym i wybierania zestawów znaków służą specjalne sekwencje sterujące.
- Kodowanie *Shift-JIS* zostało opracowane przez firmę Microsoft na użytek jej produktów. Każdy bajt zawiera w nim informację o tym, czy jest znakiem jednobajtowym, czy też może pierwszym bajtem znaku dwubajtowego. Metoda Shift-JIS nie obsługuje aż tylu znaków co metody JIS i EUC.
- Kodowanie *EUC* zostało przewidziane do roli wewnętrznego kodowania większości platform uniksowych dostępnych na rynku japońskim. Pozwala ono na reprezentowanie znaków składających się z więcej niż dwóch bajtów i jest znacznie bardziej elastyczna niż Shift-JIS. EUC jest przy tym ogólną (dającą się adaptować również do języków innych niż japoński) metodą obsługi wielu zestawów znaków.

Przyjrzymy się teraz bliżej wszystkim trzem wymienionym metodom. Jak się później okaże, zasadniczą różnicą pomiędzy metodami jest pamięć stanu albo *bezstanowość*. Kodowanie JIS jest, na przykład, kodowaniem z pamięcią stanu, czyli *metodą modalną*; metody Shift-JIS oraz ECU są zaś *metodami bezstanowymi*.

Modalne metody kodowania wielobajtowego — kodowanie JIS

JIS, od *Japanese Industrial Standard*, definiuje szereg standardowych zestawów znaków japońskich; część z nich wymaga kodowania jednobajtowego, a część dwubajtowego. Zestawy te obejmują również sekwencje sterujące służące do przełączania pomiędzy trybami — jednobajtowym i dwubajtowym.

Sekwencje sterujące (ang. *escape sequences*) są sekwencjami *znaków sterujących*. Znaki te nie należą do żadnego z alfabetów. Są one tworami sztucznymi nieposiadającymi nawet reprezentacji graficznej. Uczestniczą one jednak w kodowaniu, służąc jako separatory różnych zestawów znaków i sygnalizując konieczność zmiany sposobu interpretowania sekwencji bajtowej. Sposób zastosowania sekwencji sterujących ilustruje rysunek 4.2.

[14] JIS_ROMAN to połączenie zestawu ASCII ze znakami katakana. Znaki katakana i hiragana są kodowane pojedynczymi bajtami za pomocą 128 kodów niezarezerwowanych dla zestawu ASCII.



RYSUNEK 4.2. Przykład tekstu w języku japońskim kodowanego metodą JIS

W przypadku metod kodowania zawierających sekwencje sterujące, jak JIS, przy przetwarzaniu sekwencji konieczne jest utrzymywanie pamięci stanu. W przykładzie z rysunku 4.2 na początku przetwarzania znajdujemy się w pewnym domyślnym stanie początkowym. Tutaj jest to stan ASCII. Kolejno odczytywane bajty będą więc interpretowane jako znaki zestawu ASCII aż do rozpoznania sekwencji sterującej <ESC>$B. Reakcją na rozpoznanie tej sekwencji powinno być przełączenie stanu do trybu dwubajtowego definiowanego w JIS X 0208-1983. Sekwencja sterująca <ESC(B przełącza stan z powrotem do jednobajtowego trybu ASCII.

Bezstanowe metody kodowania wielobajtowego — Shift-JIS i EUC

Shift-JIS i EUC nie wykorzystują sekwencji sterujących. W tych metodach tryb kodowania określa wartość znaku bieżącego — każdy z bajtów sekwencji zawiera w sobie informację o tym, czy jest znakiem jednobajtowym czy może pierwszym bajtem znaku dwubajtowego.

Typ znaku jest określany przez rezerwację zbioru wartości bajtowych. W metodzie Shift-JIS:

- każdy bajt o wartości z przedziału 0×21 do 0×7E jest traktowany jak jednobajtowy znak ASCII,
- każdy bajt o wartości z zakresu od 0×A1 do 0×DF jest interpretowany jako jednobajtowy połówkowy znak katakana,
- każdy bajt o wartości z przedziału od 0×81 do 0×9F oraz od 0×E0 do 0×EF jest traktowany jako pierwszy znak dwubajtowego znaku z zestawu JIS X 0208-1990. Następny bajt powinien mieć przy tym wartość z przedziału od 0×40 do 0×7E lub od 0×80 do 0×FC.

Kodowanie to daje ciągi bajtowe bardziej zwarte niż JIS, nie może jednak reprezentować równie szerokiego zakresu znaków. W rzeczy samej Shift-JIS nie może reprezentować żadnego spośród znaków uzupełniającego zestawu JIS X 0212-1990 zawierającego ponad 600 dodatkowych znaków kanji.

Kodowanie EUC również polega na bieżącym interpretowaniu każdego kolejnego bajta celem określenia, czy reprezentuje on znak jednobajtowy, czy też może rozpoczyna sekwencję znaku dwubajtowego. Metoda EUC nie jest ograniczona do języka japońskiego — została opracowana jako uniwersalna metoda obsługi wielu zestawów znaków w jednym strumieniu tekstowym, japońskich czy dowolnych innych.

4.2.7.4. Zastosowania metod kodowania wielobajtowego i znaków poszerzonych

Znając już metody kodowania wielobajtowego, zastanówmy się nad ich rolą w procesie tworzenia oprogramowania. Wielobajtowe modalne metody kodowania są wykorzystywane przede wszystkim jako kod zewnętrzny — predestynuje je do tego zwartość, a co za tym idzie —

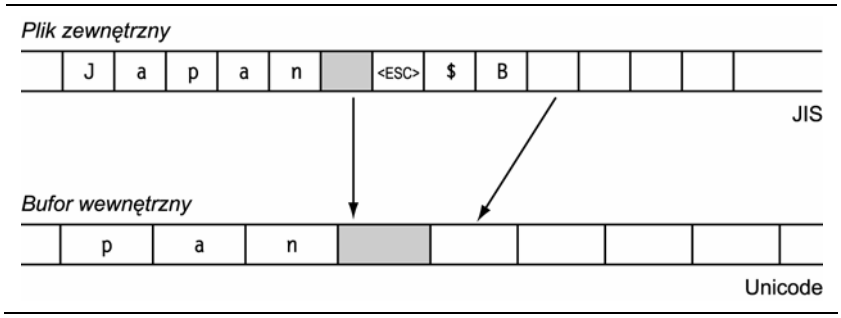
oszczędność pamięci. Jednak stosowane wewnętrznie w programie okazują się mało efektywne. Nie pozwalają, na przykład, na odwoływanie się do dowolnie wybranego znaku sekwencji znaków. Z tego względu wewnętrznie do reprezentacji przetwarzanych w programie łańcuchów tekstowych wykorzystywane są znaki poszerzone. Sposób zastosowania znaków poszerzonych zilustrujemy przykładem.

Załóżmy, że dysponujemy ciągiem reprezentującym nazwę pliku zawierającą również ścieżkę dostępu do tego pliku. Katalogi są w tej ścieżce separowane znakami ukośników, na przykład `/CC/include/locale.h`. Aby odnaleźć właściwą nazwę pliku w ciągu znaków jednobajtowych, rozpoczynamy przetwarzanie od końca ciągu i cofamy się w nim bajt po bajcie, wyszukując pierwszy znak ukośnika. Tekst pomiędzy tym znakiem a końcem ciągu jest szukaną nazwą pliku. Gdyby ten sam ciąg reprezentowany był znakami wielobajtowymi, musielibyśmy rozpocząć tę operację od początku ciągu, ponieważ zaczynając od końca tracimy wszelkie informacje o kontekście poszczególnych znaków. Gdyby jednak zamiast znaków wielobajtowych zastosować znaki poszerzone, moglibyśmy znów rozpocząć wyszukiwanie od końca, tyle że zamiast bajtami, musielibyśmy operować dwójkami bajtów.

Kodowanie wielobajtowe stanowi efektywną metodę przenoszenia ciągów znaków poza programami i ich wymiany pomiędzy programami a ich otoczeniem. Jednak wewnątrz programu łatwiej i efektywniej jest posługiwać się zestawami znaków poszerzonych, w których wszystkie znaki mają ten sam bajtowy rozmiar i format.

4.2.7.5. Konwersje kodowania

Ponieważ w wewnętrznych reprezentacjach znaków w programach stosowane są zazwyczaj zestawy znaków poszerzonych, w czasie operacji wejścia-wyjścia bardzo często konieczne jest podejmowanie konwersji kodowania pomiędzy kodowaniem wielobajtowym a kodowaniem zakładających stały bajtowy rozmiar znaku. Typowym przykładem takich operacji jest odczyt i zapis plików. Plik zewnętrzny może zawierać znaki wielobajtowe. Odczytując taki plik, należy dokonać ich konwersji na znaki poszerzone, lepiej nadające się do dalszego przetwarzania. Zapisując plik znaków wielobajtowych, należy zaś dokonać konwersji odwrotnej, zmieniając reprezentację z efektywnej szybkościowo do efektywnej pamięciowo (wielobajtowej). Sposób konwersji przy odczycie pliku ilustruje rysunek 4.3.



RYSUNEK 4.3. Konwersja kodowania wielobajtowego do kodowania wykorzystującego znaki poszerzone

Konwersja sekwencji znaków wielobajtowych na sekwencję znaków poszerzonych wymaga następujących trzech operacji:

- *Rozszerzenia znaków jednobajtowych do znaków poszerzonych (dwu- bądź czterobajtowych).* W przykładzie z rysunku 4.3 chodziłoby o „poszerzenie" znaków ASCII poprzedzających sekwencję sterującą `<ESC>$B` do ich dwubajtowych odpowiedników w zestawie Unicode.
- *Wyeliminowania ewentualnych sekwencji sterujących.* Sekwencja znaków `<ESC>$B` nie reprezentuje żadnego elementu właściwego tekstu, więc powinna zostać odrzucona.
- *Translacji* znaków wielobajtowych na ich odpowiedniki poszerzone. Znaki kanji znajdujące się w rozpatrywanym ciągu po sekwencji sterującej są konwertowane na ich odpowiedniki z zestawu Unicode. Konwersja nie zmienia rozmiaru znaku kanji, ponieważ tak w reprezentacji źródłowej, jak i docelowej znaki te mają rozmiar dwóch bajtów.