

Indywidualizacja operacji wejścia i wyjścia

„Wyjaśnienia powinny być tak proste,
jak jest to możliwe,
ale nie prostsze.”

— Albert Einstein

W tym rozdziale skupimy się na adaptacji opisanej w rozdziale 10. ogólnej struktury strumieni wejścia i wyjścia do konkretnych zastosowań i upodobań. Wiąże się z tym mnóstwo skomplikowanych szczegółów odnoszących się do ludzkiej wrażliwości na to, co ludzie czytają, oraz praktyczne ograniczenia dotyczące używania plików. W ostatnim przykładzie przedstawimy projekt strumienia wejściowego, dla którego można zdefiniować zestaw separatorów.

11.1. Regularność i nieregularność

11.2. Formatowanie danych wyjściowych

- 11.2.1. Wysyłanie na wyjście liczb całkowitych
- 11.2.2. Przyjmowanie na wejściu liczb całkowitych
- 11.2.3. Wysyłanie na wyjście liczb
zmiennoprzecinkowych
- 11.2.4. Precyzja
- 11.2.5. Pola

11.3. Otwieranie plików i pozycjonowanie

- 11.3.1. Tryby otwierania plików
- 11.3.2. Pliki binarne
- 11.3.3. Pozycjonowanie w plikach

11.4. Strumienie łańcuchowe

11.5. Wprowadzanie danych wierszami

11.6. Klasyfikowanie znaków


11.7. Stosowanie niestandardowych separatorów

11.8. Zostało jeszcze tyle do poznania

11.1. Regularność i nieregularność


Biblioteka wejścia i wyjścia — odpowiadająca za przyjmowanie i wysyłanie danych część biblioteki standardowej C++ — stanowi jednolity i rozszerzalny szkielet dla technik przyjmowania i wysyłania na wyjście tekstu. Pod słowem „tekst” rozumiemy wszystko, co można zaprezentować w postaci sekwencji znaków. W tym kontekście liczba całkowita 1234 jest tekstem, ponieważ można ją zaprezentować jako sekwencję znaków 1, 2, 3 i 4.

Do tej pory wszystkie źródła danych wejściowych traktowaliśmy na równi. Czasami to nie wystarcza. Pliki na przykład różnią się tym od innych źródeł danych (np. połączeń komunikacyjnych), że można odwoływać się do ich poszczególnych bajtów. Analogicznie przyjęliśmy założenie, że typ obiektu w pełni definiuje układ jego danych wejściowych i wyjściowych. Nie do końca tak jest i nie byłoby to dobre. Często na przykład chcemy określić liczbę cyfr reprezentujących na wyjściu liczby zmiennoprzecinkowe (ich precyzję). W tym rozdziale opiszemy kilka technik dostosowywania wejścia i wyjścia do własnych potrzeb.



Jako programiści jesteśmy zwolennikami standardów. Dzięki traktowaniu wszystkich obiektów w pamięci i wszystkich danych wejściowych jednakowo oraz nałożeniu jednego standardu na sposób reprezentacji obiektów wchodzących do systemu i wychodzących z niego powstaje najbardziej przejrzysty, najprostszy, najłatwiejszy w utrzymaniu i często najwydajniejszy kod. Jednak nasze programy są po to, aby służyć ludziom, a ci mają bardzo jasno określone preferencje. Zatem jako programiści musimy dążyć do osiągnięcia balansu między złożonością programu a zaspokojeniem wymagań użytkowników.

11.2. Formatowanie danych wyjściowych



Ludzie są bardzo wrażliwi na punkcie, wydawałoby się, mało ważnych detali w danych wyjściowych, które mają czytać. Na przykład dla fizyka liczba 1,25 (zaokrąglona do dwóch cyfr po przecinku) może być całkiem inną wartością niż 1,24670477, a dla księgowego wartość (1,25) może różnić się od (1,2467) i jest kompletnie czymś innym niż 1,25 (w księgowości nawiasy czasami stosuje się do oznaczania strat, czyli wartości ujemnych). Naszym celem jako programistów jest sformatowanie danych w taki sposób, aby były jak najłatwiejsze do czytania i jak najlepiej odpowiadały użytkownikom. W strumieniach wyjściowych (ostream) znajdują się rozmaite narzędzia do formatowania wysyłanych na wyjście typów wbudowanych. W przypadku typów zdefiniowanych przez użytkownika to programista powinien zdefiniować odpowiednie operatory <<.

Wydaje się, że liczba możliwości dostosowywania danych wyjściowych jest nieskończona, a i dla wejścia jest niemała. Jako przykłady mogą posłużyć znaki stosowane do oddzielenia części całkowitej od dziesiętnej (najczęściej kropka lub przecinek), sposoby wyrażania wartości walutowych, sposoby reprezentacji na wyjściu prawdy jako słowa true (albo vrai lub sandt) zamiast liczby 1, sposoby obsługi znaków spoza zestawu ASCII (np. Unicode) oraz sposoby ograniczania liczby znaków wczytywanych do łańcucha. Wszystko to wydaje się mało interesujące, dopóki nie zaczniesz być potrzebne. Dlatego po ich opis odsyłamy do wyspecjalizowanych publikacji, tj. książki *Standard C++ IOSTreams and Locales* Angeliki Langer, rozdziałów 38. i 39. książki *Język C++* Bjarne Stroustrupa oraz rozdziałów 22. i 27. standardu ISO języka C++. Tutaj opiszemy tylko kilka najczęściej potrzebnych narzędzi i niektóre ogólne koncepcje.

11.2.1. Wysyłanie na wyjście liczb całkowitych

Wartości całkowitoliczbowe można wysyłać na wyjście w postaci liczb w systemie ósemkowym (system liczbowy o podstawie 8), dziesiętnym (system liczb o podstawie 10) lub szesnastkowym (system liczb o podstawie 16). Jeśli nie znasz tych systemów, najpierw przeczytaj punkt A.2.1.1 i dopiero potem dokończ czytać ten rozdział. W większości operacji wyjściowych używa się liczb dziesiętnych. System szesnastkowy często jest wykorzystywany w prezentowaniu informacji na temat sprzętu. Powodem tego jest fakt, że liczba szesnastkowa dokładnie reprezentuje czterobitową wartość. Zatem za pomocą dwóch cyfr szesnastkowych można zaprezentować wartość ośmiobitowego bajta, cztery takie cyfry określają wartość dwóch takich bajtów (to często jest pół słowa), a osiem cyfr szesnastkowych umożliwia reprezentację wartości czterech bajtów (często odpowiada to jednemu słowu lub rejestrowi). Gdy zaprojektowano przodka języka C++, język C w latach 70., system ósemkowy często wykorzystywano do reprezentowania wzorców bitowych, ale teraz używa się go już rzadko.

Można wysłać wartość 1234 (dziesiętną) na wyjście jako liczbę dziesiętną, szesnastkową lub ósemkową:

```
cout << 1234 << "\t(dziesiętny)\n"
    << hex << 1234 << "\t(szesnastkowy)\n"
    << oct << 1234 << "\t(ósemkowy)\n";
```

Sekwencja '\t' oznacza znak tabulacji. Te instrukcje wydrukują następujący wynik:

```
1234    (dziesiętny)
4d2     (szesnastkowy)
2322    (ósemkowy)
```

Notacje <<hex i <<oct nie wysyłają nic na wyjście, tylko informują, że znajdujące się dalej liczby całkowite mają zostać wyświetlone odpowiednio w systemie szesnastkowym lub ósemkowym. Na przykład:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n'; // System ósemkowy jest cały czas włączony
```

Wynik będzie następujący:

```
1234 4d2 2322
2322 // Liczby całkowite będą wyświetlane w systemie ósemkowym, aż zostanie to zmienione
```

Zauważ, że ostatnia liczba jest przedstawiona w systemie ósemkowym. Oznacza to, że oct, hex i dec (dla dziesiętnych) mają trwałe działanie — zmieniają reprezentację wszystkich wysyłanych na wyjście liczb całkowitych, dopóki nie zostanie napisane inaczej. Takie elementy jak hex i oct, które służą do zmienienia zachowania strumieni, nazywają się **manipulatorami**.

WYPRÓBUJ



Wyślij na wyjście swoją datę urodzenia w formacie dziesiętnym, szesnastkowym i ósemkowym. Każdej wartości nadaj odpowiednią etykietę. Ustaw w linii dane wyjściowe za pomocą znaku tabulatora. Następnie wyświetl swój wiek.

Wartości zapisane w innym systemie niż dziesiętny mogą sprawiać problemy. Jeśli na przykład nie zaznaczymy, że używamy innego systemu, każdy będzie myślał, że 11 reprezentuje dziesiętną wartość 11, a nie 9 (11 w systemie ósemkowym) ani 17 (11 w systemie szesnastkowym). Aby uniknąć takich nieporozumień, można zmusić strumień ostream, aby pokazał podstawę wszystkich wydrukowanych liczb całkowitych. Na przykład:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec; // pokazuje podstawy systemów liczbowych
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

Wynik:

```
1234 4d2    2322
1234 0x4d2 02322
```

Liczy dziesiętne nie mają prefiksu, przed liczbami ósemkowymi znajduje się 0, a przed szesnastkowymi 0x (lub 0X). Za pomocą takiej właśnie notacji zapisuje się literały całkowito-liczbowe w języku C++. Na przykład:

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

W formacie dziesiętnym wydruk będzie następujący:

```
1234 1234 1234
```

Jak zapewne zauważyłeś, manipulator showbase ma trwały skutek, jak oct i hex. Jego działanie odwraca manipulator noshowbase — przywraca standardowe ustawienie, polegające na wyświetlaniu liczb bez określnika systemu liczbowego.

Poniżej znajduje się tabela zawierająca zestawienie wyjściowych manipulatorów liczb całkowitych:

Modyfikacje wyjściowych liczb całkowitych	
oct	Włącza notację w systemie ósemkowym.
dec	Włącza notację w systemie dziesiętnym.
hex	Włącza notację w systemie szesnastkowym.
showbase	Dodaje przedrostek 0 do liczb ósemkowych i 0x do szesnastkowych.
noshowbase	Wyłącza stosowanie prefiksów.

11.2.2. Przyjmowanie na wejściu liczb całkowitych

Standardowo operator >> zakłada, że przekazywane do niego liczby całkowite są w systemie dziesiętnym. Można jednak zmusić go do wczytywania liczb ósemkowych i szesnastkowych:

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

Jeśli na wejściu pojawiają się poniższe liczby:

```
1234 4d2 2322 2322
```

program wydrukuje następujący wynik:

```
1234 1234 1234 1234
```

Należy zauważyć, że manipulatory oct, dec i hex w strumieniu wejściowym mają takie samo trwałe działanie, jak w strumieniu wyjściowym.

WYPRÓBUJ

Uzupełnij powyższy fragment kodu, aby stał się kompletnym programem. Wypróbuj podane wcześniej dane wejściowe, a następnie poniższe:

```
1234 1234 1234 1234
```

Wyjaśnij wyniki. Sprawdź, co się stanie po podaniu innych danych na wejściu.

Operator >> można zmusić, aby przyjmował i poprawnie interpretował przedrostki 0 i 0x. W tym celu należy usunąć wszystkie domyślne ustawienia. Na przykład:

```
cin.unsetf(ios::dec); // Nie zakładaj z góry, że liczby są dziesiętne (0x może oznaczać hex)
cin.unsetf(ios::oct); // Nie zakładaj z góry, że liczby są ósemkowe (12 może oznaczać dwanaście)
cin.unsetf(ios::hex); // Nie zakładaj z góry, że liczby są szesnastkowe (12 może oznaczać dwanaście)
```

Funkcja składowa strumienia o nazwie unsetf() usuwa znacznik (lub znaczniki) podany jako argument. Jeśli teraz napiszemy poniższy kod:

```
cin >>a >> b >> c >> d;
```

i na wejściu pojawiają się liczby:

```
1234 0x4d2 02322 02322
```

wynik będzie następujący:

```
1234 1234 1234 1234
```

11.2.3. Wysyłanie na wyjście liczb zmiennoprzecinkowych

Jeśli bezpośrednio pracujesz na urządzeniach, będziesz potrzebować notacji szesnastkowej (lub może ósemkowej). Analogicznie, każdy, kto wykonuje obliczenia naukowe, musi znać formatowanie liczb zmiennoprzecinkowych. Do ich obsługi służą manipulatory z biblioteki iostream, które działają bardzo podobnie do manipulatorów wartości całkowitych. Na przykład:

```
cout << 1234.56789 << "\t\t(defaultfloat)\n" // Znaki \t\t wyrównują kolumny
    << fixed << 1234.56789 << "\t(stały)\n"
    << scientific << 1234.56789 << "\t(naukowy)\n";
```

Wynik działania powyższego kodu będzie następujący:

```
1234.57          (ogólny)
1234.567890      (stały)
1.234568e+003    (naukowy)
```

Manipulatory `fixed` (stały), `scientific` (naukowy) i `defaultfloat` (domyślny zmiennoprzecinkowy) służą do włączania formatów zmiennoprzecinkowych. Manipulator `defaultfloat` to domyślny format (zwany też ogólnym). Teraz możemy napisać:

```
cout << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n'; // Format zmiennoprzecinkowy jest „trwały”
cout << defaultfloat << 1234.56789 << '\t' // Format domyślny dla liczb zmiennoprzecinkowych
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
```


Wynik będzie następujący:

```
1234.57 1234.567890 1.234568e+003
1.234568e+003 // Manipulator scientific jest „trwały”
1234.57 1234.567890 1.234568e+003
```

Poniżej znajduje się zestawienie podstawowych manipulatorów wyjściowych liczb zmiennoprzecinkowych:

Formaty liczb zmiennoprzecinkowych	
<code>fixed</code>	Notacja stałoprzecinkowa.
<code>scientific</code>	Notacja wykorzystująca mantysę i wykładnik. Mantysa mieści się w zakresie [1,10), tzn. przed przecinkiem (kropką dziesiętną) zawsze jest niezerowa cyfra.
<code>defaultfloat</code>	Wybierz <code>fixed</code> lub <code>scientific</code> , aby uzyskać jak najdokładniejszą reprezentację liczbową w precyzji <code>defaultfloat</code> .

11.2.4. Precyzja

 Domyślnie liczba zmiennoprzecinkowa jest reprezentowana za pomocą sześciu cyfr w formacie `defaultfloat`. Wybierany jest najlepszy możliwy format i liczba zostaje zaokrąglona do najbliższej jej wartości, którą można wydrukować przy użyciu sześciu cyfr (jest to domyślna precyzja formatu `defaultfloat`). Na przykład:

```
1234.567 zostanie wydrukowana jako 1234.57
1.2345678 zostanie wydrukowana jako 1.23457
```

Stosowane jest zaokrąglanie 4/5 — cyfry od 0 do 4 zaokrągla się w dół, a od 5 do 9 w górę. Zauważ, że formatowanie zmiennoprzecinkowe działa tylko na liczbach zmiennoprzecinkowych, a więc:

```
1234567 zostanie wydrukowana jako 1234567 (ponieważ jest to liczba całkowita)
1234567.0 zostanie wydrukowana jako 1.23457e+006
```

W drugim z przedstawionych przypadków klasa `ostream` uzna, że liczby `1234567.0` nie można wydrukować w formacie `fixed` za pomocą tylko sześciu cyfr, a więc przełącza się na format `scientific`, aby zachować jak najwyższą precyzję reprezentacji. Zasadniczo format `defaultfloat`

wybiera między formatami `fixed` i `scientific`, aby dostarczyć użytkownikowi jak najprecyzyjniejszą reprezentację wartości zmiennoprzecinkowej w swoich ramach, czyli domyślnie w postaci sześciu cyfr.

WYPRÓBUJ



Napisz program drukujący trzy razy liczbę 1234567.89, za pierwszym razem w formacie `defaultfloat`, potem `fixed` i na końcu `scientific`. Który wynik jest najbardziej precyzyjny? Dlaczego?

Precyzję można ustawić za pomocą manipulatora `setprecision()`. Na przykład:

```
cout << 1234.56789 << '\t'
      << fixed << 1234.56789 << '\t'
      << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(5)
      << 1234.56789 << '\t'
      << fixed << 1234.56789 << '\t'
      << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(8)
      << 1234.56789 << '\t'
      << fixed << 1234.56789 << '\t'
      << scientific << 1234.56789 << '\n';
```

Wynik jest następujący (zwróć uwagę na zaokrąglenia):

```
1234.57      1234.567890  1.234568e+003
1234.6 1234.56789      1.23457e+003
1234.5679    1234.56789000    1.23456789e+003
```

Poniżej znajduje się zestawienie definicji precyzji:

Precyzja liczb zmiennoprzecinkowych	
<code>defaultfloat</code>	Precyzję określa ogólna liczba cyfr.
<code>scientific</code>	Precyzję określa liczba miejsc po przecinku.
<code>fixed</code>	Precyzję określa liczba miejsc po przecinku.

Stosuj domyślny format (`defaultfloat` z precyzją 6), jeśli nie ma powodu, aby to zmienić. Najczęstszym powodem, aby to zmienić, jest: „Ponieważ potrzebuję większej precyzji w danych wyjściowych”.

11.2.5. Pola

Programista stosujący formaty stały i naukowy może kontrolować, ile dokładnie miejsca każda wartość będzie zajmować. Jest to bardzo przydatne przy drukowaniu tabel itp. Odpowiednik tej techniki dla liczb całkowitych nazywa się **polem** (ang. *field*). Można określić dokładną liczbę

miejsc na znaki, którą będzie zajmować wartość lub łańcuch. Służy do tego manipulator `setw()`. Na przykład:

```
cout << 123456                                // Bez pola
<< '|' << setw(4) << 123456 << '|'          // Liczba 123456 nie mieści się w polu o szerokości czterech znaków
<< setw(8) << 123456 << '|'                  // Ustawia szerokość pola na osiem znaków
<< 123456 << "|\n";                          // Szerokość pola nie jest „trwała”
```

Wynik będzie następujący:

```
123456|123456| 123456|123456|
```

Zwróć uwagę na dwie wiodące spacje znajdujące się przed trzecią liczbą 123456. Tak powinna wyglądać sześciocyfrowa liczba znajdująca się w ośmioznakowym polu. Jednak w przypadku pola czteroznakowego liczba 123456 nie została obcięta. Dlaczego? Wydaje się, że w polu o szerokości czterech znaków należałoby się spodziewać wartości `|1234|` lub `|3456|`. Skutkiem tego byłoby wydrukowanie całkiem innej wartości niż poprawna bez żadnego ostrzeżenia dla biednego użytkownika, który nie miałby pojęcia, że coś się nie udało. Strumień `ostream` nie robi tego, tylko łamie regułę formatu wyjściowego. Złe formatowanie jest prawie zawsze lepsze od „złych danych wyjściowych”. W większości przypadków użycia pól (np. przy drukowaniu tabeli) „przepełnienie” jest bardzo dobrze widoczne, dzięki czemu można nanieść korektę.

Pola można także wykorzystywać do drukowania liczb zmiennoprzecinkowych i łańcuchów. Na przykład:

```
cout << 12345 << '|' << setw(4) << 12345 << '|'
      << setw(8) << 12345 << '|' << 12345 << "|\n";
cout << 1234.5 << '|' << setw(4) << 1234.5 << '|'
      << setw(8) << 1234.5 << '|' << 1234.5 << "|\n";
cout << "asdfg" << '|' << setw(4) << "asdfg" << '|'
      << setw(8) << "asdfg" << '|' << "asdfg" << "|\n";
```

Wynik będzie następujący:

```
12345|12345| 12345|12345|
1234.5|1234.5| 1234.5|1234.5|
asdfg|asdfg| asdfg|asdfg|
```

Należy zauważyć, że szerokość pola nie jest „trwała”. We wszystkich trzech powyższych przypadkach pierwsza i ostatnia wartość są drukowane w domyślnym formacie, czyli dostają dokładnie tyle miejsc, ile potrzebują. Innymi słowy, jeśli szerokości pola nie ustawi się bezpośrednio przed wysłaniem danych na wyjście, pojęcie pola nie zostanie zastosowane.

WYPRÓBUJ



Utwórz prostą tabelę zawierającą Twoje imię, nazwisko, numer telefonu i adres e-mail oraz dane jeszcze przynajmniej pięciu innych osób. Wypróbuj różne szerokości pól, aż znajdziesz taką tabelę, która będzie się odpowiednio prezentować.

11.3. Otwieranie plików i pozycjonowanie

Z punktu widzenia języka C++ plik jest abstrakcją tego, co dostarcza system operacyjny. Zgodnie z podrozdziałem 10.3 plik jest sekwencją bajtów ponumerowanych od 0 w górę:



Pytanie, jak się dostać do tych bajtów? Jeśli użyje się strumienia `istream`, dostęp do bajtów będzie zależał od konkretnego strumienia. Rodzaj operacji, które można wykonywać na otwartym pliku, zależy od właściwości otwartego dla niego strumienia. Jeśli na przykład dla pliku zostanie otwarty strumień `istream`, można z niego odczytywać dane. Jeśli natomiast otworzy się strumień `ostream`, będzie można informacje zapisywać.

11.3.1. Tryby otwierania plików

Plik można otworzyć w jednym z kilku trybów. Strumień `ifstream` domyślnie otwiera plik w trybie do odczytu, a `ostream` w trybie do zapisu. To wystarcza do zaspokojenia większości potrzeb. Do wyboru jest jednak jeszcze kilka innych trybów:

Tryby otwierania strumieni plikowych	
<code>ios_base::app</code>	dołączanie (tzn. dodawanie danych na końcu pliku)
<code>ios_base::ate</code>	„at end” — otwiera i szuka końca
<code>ios_base::binary</code>	tryb binarny — uwaga na specyficzne zachowania systemów
<code>ios_base::in</code>	do odczytu
<code>ios_base::out</code>	do zapisu
<code>ios_base::trunc</code>	przycina plik do długości 0

Tryb otwarcia pliku można opcjonalnie określić za nazwą pliku. Na przykład:

```
ofstream of1 {n ame1}; // stosuje domyślny tryb ios_base::out
ifstream if1 {n ame2}; // stosuje domyślny tryb ios_base::in

ofstream ofs {name, ios_base::app}; // strumień ofstream domyślnie zawierają ios_base::out
fstream fs {"myfile", ios_base::in|ios_base::out}; // in i out
```

Znak `|` widoczny w ostatnim wierszu kodu to operator „bitowego OR” (punkt A.5.5), za pomocą którego można łączyć tryby w taki sposób, jak powyżej. Opcja `app` jest często wykorzystywana w zapisywaniu do dzienników, w których dane zawsze dodaje się na końcu.

Wynik zastosowania każdego z tych trybów zależy od systemu operacyjnego. Jeśli system nie może spełnić żądania otwarcia pliku w danym trybie, powstanie strumień, który będzie w innym stanie niż `good()`:

```
if (!fs) // Nie dało się otworzyć pliku w ten sposób
```

Najczęstszym powodem niepowodzenia otwarcia pliku w trybie do odczytu jest nieistnienie tego pliku (przynajmniej nie podaną nazwą):

```
ifstream ifs{"readings"};
if (!ifs) // Błąd: nie można otworzyć pliku „readings” w trybie do odczytu
```


W tym przypadku łatwo zgadnąć, że powodem problemu jest literówka.

Należy zauważyć, że system operacyjny najczęściej tworzy plik wyjściowy, jeśli taki nie istnieje. Na szczęście nie robi tego w przypadku otwierania plików wejściowych:

```
ofstream ofs{"nie-ma-takiego-pliku"}; // Utworzy nowy plik o nazwie nie-ma-takiego-pliku
ifstream ifs{"nie-ma-pliku-o-takiej-nazwie"}; // Błąd: strumień ifs będzie w innym stanie niż good()
```

Nie próbuj żadnych sprytnych sztuczek przy otwieraniu plików w różnych trybach, ponieważ systemy operacyjne różnie traktują „nietypowe” tryby. Kiedy to tylko możliwe, otwieraj pliki przy użyciu strumieni `istream` i zapisuj w plikach za pomocą strumieni `ostream`.

11.3.2. Pliki binarne

 Liczbę 123 można w pamięci reprezentować jako liczbę całkowitą lub łańcuch. Na przykład:

```
int n = 123;
string s = "123";
```

W pierwszym przypadku liczba 123 zostanie zapisana jako binarna liczba i zostanie jej przydzielona typowa dla typu `int` ilość miejsca (4 bajty, czyli 32 bity, w komputerach PC). Gdybyśmy zamiast tej użyli liczby 12345, wykorzystane zostałyby te same 4 bajty. W drugim przypadku wartość 123 została zapisana jako łańcuch znaków. Gdybyśmy użyli wartości łańcuchowej "12345", zostałyby wykorzystane miejsca dla pięciu znaków (plus stała ilość miejsca do zarządzania typem `string`). Można to zilustrować graficznie w następujący sposób (przy użyciu zwykłych reprezentacji liczb dziesiętnych i łańcuchów, a nie binarnej reprezentacji, która jest w rzeczywistości wykorzystywana przez komputer):

123 jako znaki:	<table><tr><td>1</td><td>2</td><td>3</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	?	?	?	?	?
1	2	3	?	?	?	?	?		
12345 jako znaki:	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	4	5	?	?	?
1	2	3	4	5	?	?	?		
123 w formacie binarnym:	<table><tr><td>123</td><td></td></tr></table>	123							
123									
12345 w formacie binarnym:	<table><tr><td>12345</td><td></td></tr></table>	12345							
12345									

Stosując reprezentację znakową, należy użyć jakiegoś znaku oznaczającego koniec liczby w pamięci, tak jak się to robi na papierze — 123456 jest jedną liczbą, a 123 456 to dwie liczby. Na „papierze” koniec liczby oznacza się spacją. W pamięci można zrobić to samo:

123456 jako znaki:	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td></td><td>?</td></tr></table>	1	2	3	4	5	6		?
1	2	3	4	5	6		?		
123 456 jako znaki:	<table><tr><td>1</td><td>2</td><td>3</td><td></td><td>4</td><td>5</td><td>6</td><td></td></tr></table>	1	2	3		4	5	6	
1	2	3		4	5	6			

Różnica między zapisywaniem reprezentacji binarnej o stałym rozmiarze (np. liczb typu `int`) a reprezentacji łańcuchowej o zmiennym rozmiarze (np. wartości typu `string`) jest także widoczna w plikach. Strumienie `istream` domyślnie operują na reprezentacjach znakowych. To znaczy, strumień `istream` wczytuje sekwencje znaków i zamienia je na obiekty odpowiedniego typu. Strumień `ostream` pobiera obiekt określonego typu i zamienia go na sekwencję znaków, którą wysyła na wyjście. Można jednak zmusić te strumienie do kopiowania bajtów do i z plików. Nazywa się to binarnymi operacjami wejścia i wyjścia i aby je wykonać, należy otworzyć plik w trybie `ios_base::binary`. Poniżej znajduje się przykładowy program, który wczytuje i zapisuje binarne pliki zawierające liczby całkowite. Fragmenty o kluczowym znaczeniu dla wykonania binarnych operacji zostały objaśnione dalej:

```
int main()
{
    // Otwieramy strumień istream w binarnym trybie odczytu pliku:
    cout << "Podaj nazwę pliku wejściowego:\n";
    string iname;
    cin >> iname;
    ifstream ifs {iname, ios_base::binary};           // Uwaga: tryb strumienia
    // „binary” oznacza dla niego, że ma się za bardzo nie ciekawić znaczeniem bajtów
    if (!ifs) error("Nie można otworzyć pliku wejściowego ", iname);

    // Otwieramy strumień ostream w binarnym trybie do zapisu:
    cout << "Podaj nazwę pliku wyjściowego:\n";
    string oname;
    cin >> oname;
    ofstream ofs {oname, ios_base::binary};           // Uwaga: tryb strumienia
    // „binary” oznacza dla niego, że ma się za bardzo nie ciekawić znaczeniem bajtów
    if (!ofs) error("Nie można otworzyć pliku wyjściowego ", oname);

    vector<int> v;

    // Odczyt z pliku binarnego:
    for(int x; ifs.read(as_bytes(x), sizeof(int)); ) // Uwaga: wczytywanie bajtów
        v.push_back(x);
    // ... operacje przy użyciu v ...

    // Zapisywanie do pliku binarnego:
    for(int x : v)
        ofs.write(as_bytes(x), sizeof(int));         // Uwaga: zapisywanie bajtów
    return 0;
}
```

Otwieramy pliki, stosując `ios_base::binary` jako tryb strumienia:

```
ifstream ifs {iname, ios_base::binary};
ofstream ofs {oname, ios_base::binary};
```

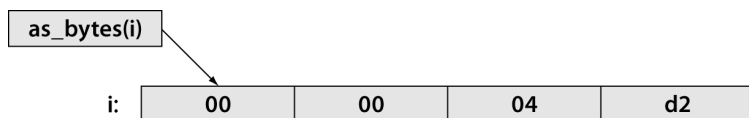
W obu przypadkach wybraliśmy trudniejszą, ale często zwięźlejszą, reprezentację binarną. Używając na wejściu i wyjściu reprezentacji binarnej, porzucamy typowe operatory `>>` i `<<`. Operatory te zamieniają wartości na sekwencje znaków, stosując się przy tym do domyślnych

konwencji (np. łańcuch "asdf" zostaje zamieniony na znaki a, s, d, f, a liczba 123 na znaki 1, 2 i 3). Gdybyśmy chcieli takiego czegoś, nie musielibyśmy używać słowa `binary` — wystarczyłyby ustawienia domyślne. Formatu binarnego używa się wówczas, gdy wiadomo, że sprawdzi się on lepiej w danym zastosowaniu niż zwykły format. Stosując ten format, nakazujemy strumieniowi, aby nie próbował robić z bajtami niczego szczególnego.

Co szczególnego można zrobić z bajtami liczby typu `int`? Można oczywiście zapisać cztero-bajtową liczbę typu `int` w 4 bajtach, tzn. można znaleźć reprezentację tej liczby w pamięci (sekwencję 4 bajtów) i wysłać te bajty do pliku. Później można w ten sam sposób wczytać te bajty z powrotem i odtworzyć liczbę:

```
ifs.read(as_bytes(i), sizeof(int))    // Uwaga: wczytywanie bajtów
ofs.write(as_bytes(v[i]), sizeof(int)) // Uwaga: zapisywanie bajtów
```

Funkcje `write()` strumienia `ostream` i `read()` strumienia `istream` pobierają adres (dostarczony tutaj przez funkcję `as_bytes()`) i liczbę bajtów (znaków), którą uzyskaliśmy za pomocą operatora `sizeof`. Adres ten powinien wskazywać pierwszy bajt miejsca w pamięci, w którym znajduje się wartość, jaką chcemy odczytać lub zapisać. Gdybyśmy na przykład mieli obiekt typu `int` z wartością 1234, otrzymalibyśmy następujące cztery bajty (w notacji szesnastkowej): 00, 00, 04, d2:



Funkcja `as_bytes()` zwraca adres pierwszego bajta reprezentacji obiektu. Można ją zdefiniować — przy użyciu nie opisywanych jeszcze narzędzi językowych (zobacz podrozdziały 17.8 i 19.3) — w następujący sposób:

```
template<class T>
char* as_bytes(T& i)                // Traktuje T jako sekwencję bajtów
{
    void* addr = &i;                // Pobiera adres pierwszego bajta
                                    // w pamięci, w której został zapisany obiekt
    return static_cast<char*>(addr); // Traktuje tę pamięć jako bajty
}
```

Niebezpieczna konwersja typów za pomocą operacji `static_cast` jest niezbędna, aby dostać się do „surowych bajtów” zmiennej. Pojęcie adresu zostanie szerzej opisane w rozdziałach 17. i 18. Naszym aktualnym celem jest pokazanie sposobu traktowania dowolnego obiektu w pamięci jako sekwencji bajtów do użytku z funkcjami `read()` i `write()`.

Binarna technika pobierania i wysyłania na wyjście danych jest mało elegancka, nieco skomplikowana i podatna na błędy. Niestety programista nie zawsze może wybrać format plików, dlatego czasami musimy używać formatu binarnego, ponieważ wybrał go ktoś dla plików, które musimy odczytać i zapisać. Może też istnieć dobry logiczny powód na zastosowanie reprezentacji innej niż znakowa. Typowym przykładem może być obraz lub plik z dźwiękiem, które nie mają sensownej reprezentacji znakowej — zdjęcie lub plik muzyczny to w zasadzie worek bitów.

Techniki wejścia i wyjścia dostępne domyślnie w bibliotece wejścia i wyjścia są przenośne, możliwe do odczytu przez człowieka i dobrze obsługiwane przez system typów. Używaj ich, jeśli możesz, i nie baw się z binarnymi formatami, jeśli nie musisz.

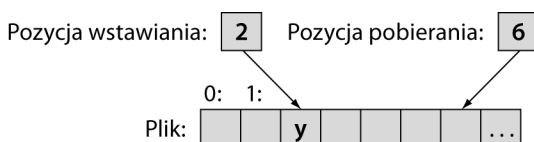


11.3.3. Pozycjonowanie w plikach

Jeśli się da, odczytuj i zapisuj pliki od początku do końca. Jest to najłatwiejszy i najmniej podatny na błędy sposób. Gdy konieczne jest wprowadzenie zmiany w pliku, często lepiej jest utworzyć nowy plik z uwzględnioną modyfikacją niż przerabiać istniejący.



W razie konieczności można jednak wybrać konkretne miejsce w pliku do odczytu lub zapisu za pomocą techniki pozycjonowania. Zasadniczo w każdym pliku otwartym do odczytu znajduje się „pozycja odczyt-pobieranie”, a w każdym pliku otwartym do zapisu znajduje się „pozycja zapis-wstawianie”:



Można to wykorzystać następująco:

```
fstream fs {name}; // Otwiera w trybie do odczytu i zapisu
if (!fs) error("Nie można otworzyć pliku ",name);
fs.seekg(5);        // Przesuwa pozycję odczytu (g od „get” — pobierz) do 5 (szósty znak)
char ch;
fs>>ch;             // Odczytuje i zwiększa indeks pozycji odczytu
cout << "znak [5] to " << ch << '(' << int(ch) << ")\n";

fs.seekp(1);        // Przesuwa pozycję zapisu (p od „put” — wstawianie) do 1
fs<<'y';            // Zapisuje i zwiększa numer pozycji zapisu
```

Funkcje seekg() i seekp() zwiększają numery swoich pozycji, a zatem rysunek przedstawia stan programu po wykonaniu.

Bądź ostrożny przy pozycjonowaniu — działanie mechanizmu pozycjonowania jest zależne od systemu operacyjnego, który rzadko wszystko dokładnie sprawdza lub dostarcza łatwych do zrozumienia komunikatów o błędach. W szczególności nie wiadomo, co się stanie, jeśli spróbujesz wyjść (za pomocą funkcji seekg() lub seekp()) za koniec pliku. Systemy operacyjne różnie się w takich sytuacjach zachowują.

11.4. Strumienie łańcuchowe

Łańcuch może być źródłem danych strumienia istream lub miejscem docelowym strumienia ostream. Strumień istream, który wczytuje dane z łańcucha, nazywa się istream, a strumień ostream, który wysyła dane do łańcucha, nazywa się ostream. Strumień istream wykorzystuje się na przykład do wydobywania wartości liczbowych z łańcuchów:



```
double str_to_double(string s)
    // Jeśli to możliwe, konwertuje znaki znajdujące się w s na wartości zmiennoprzecinkowe
{
    istringstream is{s}; // Tworzy strumień, aby można było odczytywać dane z s
    double d;
    is >> d;
    if (!is) error("Błąd formatu double: ",s);
    return d;
}


double d1 = str_to_double("12.4"); // testowanie
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("Dwanaście i trzy dziesiąte."); // Wywołanie funkcji error()
```

Próba odczytu danych spoza końca łańcucha strumienia `istringstream` zakończy się przełączeniem tego strumienia na stan `eof()`. Oznacza to, że do jego odczytu można użyć „zwykłej pętli wejściowej”. Strumień `istringstream` jest w istocie rodzajem strumienia `istream`.

Strumienia `ostringstream` można natomiast użyć do formatowania danych wyjściowych dla systemu, np. GUI (podrozdział 16.5) wymagającego prostego argumentu w postaci łańcucha. Na przykład:

```
void my_code(string label, Temperature temp)
{
    // ...
    ostringstream os; // Strumień do budowania komunikatu
    os << setw(8) << label << ": "
        << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    // ...
}
```

Funkcja składowa `str()` strumienia `ostringstream` zwraca łańcuch złożony przez operacje wyjściowe do strumienia `ostream`. Funkcja `c_str()` jest składową klasy `string` i zwraca łańcuch w stylu języka C, którego wymaga wiele interfejsów systemowych.



Strumieni `stringstream` używa się najczęściej po to, aby oddzielić operacje wejścia i wyjścia od przetwarzania. Na przykład argument typu `string` dla funkcji `str_to_double()` często pochodzi z pliku (np. dziennika sieciowego) lub zostaje podany za pośrednictwem klawiatury. Analogicznie złożony przez nas w funkcji `my_code()` komunikat w końcu zostanie wyświetlony gdzieś na ekranie. W podrozdziale 11.7 np. użyjemy strumienia `stringstream` do odfiltrowania z danych wprowadzonych przez użytkownika niepotrzebnych nam znaków. Strumień `stringstream` można traktować jako technikę dostosowywania wejścia i wyjścia do specyficznych wymagań i gustów.

Jednym z prostszych zastosowań strumienia `ostringstream` jest tworzenie łańcuchów poprzez konkatenaację. Na przykład:

```
int seq_no = get_next_number(); // Pobiera numer pliku dziennika
ostringstream name;
name << "myfile" << seq_no << ".log"; // np. plik17
ofstream lo gfile{name.str()}; // np. otwiera plik17
```

Najczęściej strumień `istream` inicjalizuje się łańcuchem i wczytuje znaki z tego łańcucha za pomocą operacji wejściowych. Natomiast strumień `ostream` zazwyczaj inicjalizuje się pustym łańcuchem, a następnie zapełnia go za pomocą operacji wyjściowych. Istnieje możliwość uzyskania bardziej bezpośredniego dostępu do znaków w strumieniu `stringstream` — wywołanie `ss.str()` zwraca kopię łańcucha `ss`, a `ss.str(s)` ustawia łańcuch `ss` na kopię `s`. W podrozdziale 11.7 przedstawimy przykład, w którym wywołanie `ss.str(s)` będzie miało kluczowe znaczenie.

11.5. Wprowadzanie danych wierszami

Operator `>>` wczytuje dane do obiektów odpowiedniego typu zgodnie ze standardowym formatem tego typu. Na przykład wczytując dane do obiektu typu `int`, operator `>>` zatrzyma się, gdy napotka coś, co nie jest cyfrą, a wczytując do typu `string`, zakończy, gdy napotka biały znak. Należąca do biblioteki standardowej biblioteka `istream` zawiera narzędzia umożliwiające wczytywanie pojedynczych znaków i całych wierszy znaków. Spójrz na poniższy kod:

```
string name;
cin >> name;           // Wejście: Dennis Ritchie
cout << name << '\n'; // Wyjście: Dennis
```

Co zrobić, jeśli będziemy chcieli wczytać cały wiersz na raz i później zdecydować, jak go sformatować? Można użyć funkcji `getline()`. Na przykład:

```
string name;
getline(cin,name);     // Wejście: Dennis Ritchie
cout << name << '\n'; // Wyjście: Dennis Ritchie
```

Teraz mamy cały wiersz. Po co mielibyśmy go chcieć? Jednym z powodów może być: „Ponieważ chcemy zrobić coś, czego nie może zrobić operator `>>`”. Często powód jest bardziej banalny: „Ponieważ użytkownik wpisał cały wiersz tekstu”. Jeśli to jest najlepsze, co możesz wymyślić, lepiej trzymaj się operatora `>>`, ponieważ po wczytaniu wiersza tekstu zazwyczaj trzeba go jakoś przeanalizować. Na przykład:

```
string first_name;
string second_name;
stringstream ss(name);
ss>>first_name; // Wczytuje Dennis
ss>>second_name; // Wczytuje Ritchie
```

Można w prostszy sposób wczytać dane bezpośrednio do zmiennych `first_name` i `second_name`.

Jednym z często występujących powodów, dla którego można chcieć wczytać cały wiersz tekstu, może być to, że definicja białego znaku nie zawsze jest odpowiednia. Czasami znak nowego wiersza chcemy traktować inaczej niż pozostałe białe znaki. Na przykład konsola komunikacyjna użytkownika z grą może traktować wiersz jako zdanie zamiast wykorzystywać konwencjonalne znaki interpunkcyjne:

```
idź w lewo aż zobaczysz obraz na ścianie po prawej stronie
zdejmij go i otwórz znajdujące się za nim drzwi. weź znajdującą się tam torbę
```

W takim przypadku wczytalibyśmy najpierw cały wiersz, a potem wydobyli z niego poszczególne słowa:

```
string command;
getline(cin,command);           // wczytuje wiersz

stringstream ss(command);
vector<string> words;
string s;
for (string s; ss>>s; )
    words.push_back(s); // wydobywa poszczególne słowa
```

Gdybyśmy jednak mieli wybór, najprawdopodobniej wolelibyśmy wykorzystać poprawnie użyte znaki interpunkcyjne niż znaki nowego wiersza.

11.6. Klasyfikowanie znaków



Zazwyczaj liczby całkowite i zmiennoprzecinkowe, słowa itp. wczytuje się zgodnie z konwencjonalnym formatem. Czasami jednak można — a nawet trzeba — zejść o jeden poziom abstrakcji niżej i wczytać pojedyncze znaki. Oznacza to więcej pracy, ale wczytując po jednym znaku, ma się pełną kontrolę nad tym, co się robi. Rozważ dzielenie wyrażenia na tokeny (punkt 7.8.2). Na przykład wyrażenie $1+4*x \leq y/z*5$ powinno zostać podzielone na jedenaście tokenów:

$$1 + 4 * x \leq y / z * 5$$

Można by było wczytać te liczby za pomocą operatora `>>`, ale gdybyśmy próbowali wczytać identyfikatory jako łańcuchy, `x<=y` zostałoby wczytane jako jeden łańcuch (ponieważ `<` i `=` nie są białymi znakami) i `z*` również zostałoby wczytane jako jeden łańcuch (ponieważ `*` także nie jest białym znakiem). Zamiast tego można napisać:

```
for (char ch; cin.get(ch); ) {
    if (isspace(ch)) { // Jeśli ch jest białym znakiem,
        // nic nie rób (tzn. pomiń go)
    }
    if (isdigit(ch)) {
        // Wczytuje liczbę
    }
    else if (isalpha(ch)) {
        // Wczytuje identyfikator
    }
    else {
        // Obsługa operatorów
    }
}
```

Funkcja `istream::get()` wczytuje do swojego argumentu jeden znak. Nie pomija białych znaków. Podobnie jak operator `>>`, funkcja `get()` zwraca referencję do swojego strumienia `istream`, dzięki czemu można sprawdzić jego stan.

Wczytując znaki pojedynczo, zazwyczaj chcemy je w jakiś sposób klasyfikować — np. określić, czy to jest znak czy cyfra, czy litera jest wielka czy mała itd. W bibliotece standardowej znajduje się zestaw funkcji, które służą do tych celów:

Klasyfikowanie znaków	
<code>isspace(c)</code>	Sprawdza, czy <code>c</code> jest białym znakiem (' ', '\t', '\n' itp.).
<code>isalpha(c)</code>	Sprawdza, czy <code>c</code> jest literą ('a', 'z', 'A', 'Z' itd., ale nie '_').
<code>isdigit(c)</code>	Sprawdza, czy <code>c</code> jest liczbą dziesiętną ('0'..'9').
<code>isxdigit(c)</code>	Sprawdza, czy <code>c</code> jest liczbą szesnastkową (cyfra dziesiętna lub litera 'a'..'f' lub 'A'..'F').
<code>isupper(c)</code>	Sprawdza, czy <code>c</code> jest wielką literą.
<code>islower(c)</code>	Sprawdza, czy <code>c</code> jest małą literą.
<code>isalnum(c)</code>	Sprawdza, czy <code>c</code> jest literą czy cyfrą dziesiętną.
<code>iscntrl(c)</code>	Sprawdza, czy <code>c</code> jest znakiem sterującym (kody ASCII 0..31 i 127).
<code>ispunct(c)</code>	Sprawdza, czy <code>c</code> nie jest literą, cyfrą, białym znakiem lub niewidocznym znakiem sterującym.
<code>isprint(c)</code>	Sprawdza, czy <code>c</code> jest znakiem drukowalnym (ASCII ' '..~').
<code>isgraph(c)</code>	Sprawdza, czy <code>isalpha(c)</code> lub <code>isdigit(c)</code> lub <code>ispunct(c)</code> (uwaga: nie spacja).

Zwróć uwagę na sposób łączenia funkcji klasyfikujących za pomocą operatora `||`. Na przykład `isalnum(c)` oznacza tyle, co `isalpha(c) || isdigit(c)`, tzn. „Czy `c` jest literą lub cyfrą?”.

Ponadto w bibliotece standardowej są dwie funkcje pozwalające pozbyć się różnic wielkości liter:

Wielkość litery	
<code>toupper(c)</code>	Zwraca oryginał <code>c</code> lub <code>c</code> jako wielką literę.
<code>tolower(c)</code>	Zwraca oryginał <code>c</code> lub <code>c</code> jako małą literę.

Funkcje te są przydatne, gdy trzeba pozbyć się różnic związanych z wielkością liter. W danych wprowadzanych przez użytkownika słowa `Dobrze`, `dobrze` i `dobrze` najprawdopodobniej oznaczają to samo (to ostatnie mogłoby powstać w wyniku przypadkowego naciśnięcia klawisza *Caps Lock*). Po przepuszczeniu każdego z tych łańcuchów przez funkcję `tolower()` w każdym przypadku otrzymałoby się wynik `dobrze`. Można to zrobić z dowolnym łańcuchem:

```
void tolower(string& s) // Zamienia litery łańcucha s na małe
{
    for (char& x : s) x = tolower(x);
}
```

Zastosowaliśmy przekazywanie przez referencję (punkt 8.5.5), aby zmienić przetwarzany łańcuch. Gdybyśmy chcieli zachować oryginał, moglibyśmy napisać funkcję tworzącą kopię łańcucha zawierającą małe litery. Lepiej jest stosować funkcję `tolower()` zamiast `toupper()`, ponieważ ta druga może sprawiać problemy w niektórych językach naturalnych, np. w alfabecie niemieckim nie każda mała litera ma wielki odpowiednik.



11.7. Stosowanie niestandardowych separatorów

W tym podrozdziale przedstawimy półrealistyczny przykład zastosowania strumieni `istream` w rozwiązywaniu realnego problemu. Słowa w łańcuchach zwyczajowo oddziela się spacjami. Niestety strumień `istream` nie umożliwia zdefiniowania znaków pełniących rolę spacji ani nie pozwala bezpośrednio zmienić działania operatora `>>`. Co zatem należy zrobić, gdy konieczna jest zmiana definicji spacji? Rozważmy przykład z punktu 4.6.3, w którym wczytywaliśmy i porównywaliśmy „słowa”. Tamte słowa były oddzielane spacjami, a więc wczytując poniższy łańcuch:

Goście przybyli, jak zaplanowano; potem

otrzymalibyśmy następujące „słowa”:

Goście
przybyli,
jak
zaplanowano;
potem

Takich słów nie znajdziemy w słowniku — „przybyli,” i „zaplanowano;” to nie są słowa. Są to słowa połączone z rozpraszającymi i niepotrzebnymi tutaj znakami interpunkcyjnymi. W większości przypadków znaki interpunkcyjne należy traktować jak białe znaki. Jak się ich pozbyć? Można by było wczytać wszystkie znaki, usunąć znaki interpunkcyjne — lub zamienić je na białe znaki — i ponownie wczytać „oczyszczone” dane:

```
string line;
getline(cin,line);           // Wczytuje do zmiennej line
for (char& ch : line)         // Zamienia znaki interpunkcyjne na spacje

    switch(ch) {
        case ';': case '.': case ',': case '?': case '!':
            ch = ' ';
    }

stringstream ss(line);       // Zmusza strumień istream ss do wczytania danych ze zmiennej line
vector<string> vs;
string word;
for (string word; ss>>word; ) // Wczytuje słowa bez znaków interpunkcyjnych
    vs.push_back(word);
```

Wczytując za pomocą tej funkcji wiersz tekstu, otrzymamy pożądaný rezultat:

Goście
przybyli
jak
zaplanowano
potem

Niestety powyższy kod nie jest elegancki i raczej mocno wyspecjalizowany. Co byśmy zrobili, gdyby trzeba było zmienić definicję znaków interpunkcyjnych? Opracujemy bardziej ogólną i przydatną technikę usuwania niechcianych znaków z łańcuchów. Jak to zrobić? Jakim kodem powinien posługiwać się nasz użytkownik, aby z tej techniki skorzystać? Może coś takiego:

```
ps.whitespace(" ; , . "); // Traktuje średniki, dwukropki, przecinki i kropki jako białe znaki
string word;
for (string word; ps>>word; )
    vs.push_back(word);
```

Jak zdefiniować strumień działający jak powyższy ps? Kluczem jest wczytanie słów z normalnego strumienia wejściowego i traktowanie zdefiniowanych przez użytkownika „białych znaków” jako białych znaków. Tzn. nie zwracamy użytkownikowi białych znaków, stosujemy je tylko do oddzielenia słów. Na przykład łańcuch:

```
nie.tak
```

powinien zostać podzielony na dwa słowa:

```
nie
tak
```

Można zdefiniować klasę, która będzie to robić. Powinna pobierać znaki ze strumienia istream i mieć operator >> działający prawie tak samo, jak w klasie istream, z tym wyjątkiem, że będzie mu można podać zestaw znaków do traktowania jako białe. Dla uproszczenia nie będziemy zajmować się zamienianiem istniejących białych znaków (spacji, znaków nowego wiersza itp.) jako niebiałych znaków. Umożliwimy tylko dodanie nowych „białych znaków”. Nie umożliwimy też całkowitego usuwania wyznaczonych znaków ze strumienia — zamienimy je tylko na białe znaki. Naszą klasę nazwiemy Punct_stream:

```
class Punct_stream {                                // Podobna do istream, ale umożliwia rozszerzanie
                                                    // zestawu białych znaków
public:
    Punct_stream(istream& is)
        : source{is}, sensitive{true} { }

    void whitespace(const string& s)                // s jest zbiorem białych znaków
        { white = s; }
    void add_white(char c) { white += c; } // Dodaje znak do zbioru białych znaków
    bool is_whitespace(char c);                    // Czy c należy do zbioru białych znaków?

    void case_sensitive(bool b) { sensitive = b; }
    bool is_case_sensitive() { return sensitive; }

    Punct_stream& operator>>(string& s);
    operator bool();
private:
    istream& source;                                // Źródło znaków
    istreambuf_iterator buffer;                     // Formatowanie wykonuje bufor
    string white;                                    // Znaki uznawane za „białe”
    bool sensitive;                                  // Czy strumień rozpoznaje wielkość liter?
};
```

Podstawową zasadą działania tej klasy — tak jak wcześniejszego kodu — jest wczytać jeden wiersz tekstu na raz ze strumienia istream, przekonwertować „białe znaki” na spacje i sformatować dane wyjściowe za pomocą strumienia istreambuf_iterator. Poza obsługą zdefiniowanych przez użytkownika białych znaków, ta klasa spełnia jeszcze jedno zadanie — jeśli zechcemy,

używając funkcji `case_sensitive()`, możemy przekonwertować dane wejściowe rozpoznające wielkość liter na dane bez rozpoznawania wielkości liter. Możemy na przykład zmusić klasę `Punct_stream` do wczytania tekstu:

Człowiek pogryzł psa!

jako:

```
człowiek
pogryzł
psa
```

Konstruktor klasy `Punct_stream` przyjmuje argument typu `istream`, który stanowi źródło znaków, i nadaje mu lokalną nazwę `source`. Ponadto konstruktor ten ustawia strumień na domyślny tryb działania z rozpoznawaniem wielkości znaków. Można zmusić klasę `Punct_stream`, aby wczytywała dane ze strumienia `cin`, traktując średniki, dwukropki i kropki jako białe znaki, oraz zamieniała wszystkie litery na małe:

```
Punct_stream ps{cin};           // ps wczytuje dane ze strumienia cin
ps.whitespace(";:.");           // Średniki, dwukropki i kropki są traktowane jako białe znaki
ps.case_sensitive(false);       // Nie rozpoznaje wielkości znaków
```

Oczywiście najciekawiej prezentuje się operator `>>`. Jest on także zdecydowanie najtrudniejszy do zdefiniowania. Nasza ogólna strategia zakłada wczytanie całego wiersza ze strumienia `istream` do łańcucha (o nazwie `line`). Później konwertujemy wszystkie „nasze” białe znaki na spacje (' '). Następnie wstawiamy tekst do strumienia `istringstream` o nazwie `buffer`. Teraz można odczytać dane ze strumienia `buffer` za pomocą zwykłego operatora `>>`, który rozpoznaje zwykłe spacje. Kod jest nieco bardziej skomplikowany, niż wynika z tego opisu, ponieważ próbujemy odczytać dane ze strumienia `buffer` oraz wstawiamy do niego dane tylko wówczas, gdy jest pusty:

```
Punct_stream& Punct_stream::operator>>(string& s)
{
    while (!(buffer>>s)) {           // Próba wczytania danych z bufora
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        string line;
        getline(source,line);        // Pobranie wiersza danych ze źródła (source)

        // Zastępowanie znaków zgodnie z potrzebą:
        for (char& ch : line)
            if (is_whitespace(ch))
                ch = ' ';             // Zamiana na spacje
            else if (!sensitive)
                ch = tolower(ch);     // Zamiana na małe litery

        buffer.str(line);             // Wstawianie łańcucha do strumienia
    }
    return *this;
}
```

Przeanalizujemy ten kod po kawałku. Najpierw przyjrzymy się dość niezwykłemu zapisowi:

```
while (!(buffer>>s)) {
```

Jeśli w strumieniu `istream` o nazwie `buffer` znajdują się znaki, operacja odczytu `buffer>>s` powiedzie się i `s` otrzyma oddzielone „spacją” słowo. Wówczas nie będzie nic więcej do zrobienia. Będzie się tak działo, dopóki w buforze będą znaki do wczytania. Jeśli jednak się to zmieni, tzn. nastąpi sytuacja `!(buffer>>s)`, trzeba będzie z powrotem napęlnić bufor danymi ze źródła. Zwróć uwagę, że instrukcja `buffer>>s` znajduje się w pętli. Po zakończeniu próby ponownego zapełnienia bufora musimy jeszcze raz spróbować odczytu:

```
while (!(buffer>>s)) { // Próba odczytu danych z bufora
    if (buffer.bad() || !source.good()) return *this;
    buffer.clear();

    // Ponowne napełnienie bufora
}
```

Jeśli bufor przejdzie w stan `bad()` lub wystąpi problem ze źródłem, poddajemy się. W innym przypadku czyścimy bufor i próbujemy jeszcze raz. Czyszczenie jest konieczne, ponieważ do tej „napełniającej się pętli” wchodzimy, tylko jeśli nie powiedzie się operacja odczytu, co najczęściej jest spowodowane napotkaniem końca pliku, tzn. w buforze nie ma więcej znaków. Praca ze strumieniami zawsze jest skomplikowana i często bywa źródłem subtelnych błędów, których wykrycie wymaga żmudnych poszukiwań. Na szczęście reszta napełniającej się pętli jest już bardzo prosta:

```
string line;
getline(source, line); // Pobiera wiersz danych ze źródła

// Zastępowanie znaków zgodnie z potrzebą:
for (char& ch : line)
    if (is_whitespace(ch))
        ch = ' '; // Zamiana na spacje
    else if (!isensitive)
        ch = tolower(ch); // Zamiana na małe litery
buffer.str(line); // Wstawianie łańcucha do strumienia
```

Wczytujemy wiersz danych do zmiennej `line`. Następnie sprawdzamy każdy znak, aby dowiedzieć się, czy nie trzeba go zmienić. Funkcja `is_whitespace()` jest składową klasy `Punct_stream`, którą zdefiniujemy później. Funkcja `tolower()` należy do biblioteki standardowej i zamienia wielkie litery, np. `A`, na małe, np. `a` (podrozdział 11.6).

Po pomyślnym przetworzeniu wiersza tekstu musimy wstawić go do strumienia `istream`. Robi to instrukcja `buffer.str(line)` — można ją przeczytać w następujący sposób: „Ustaw łańcuch strumienia `istream` o nazwie `buffer` na `line`”.

Zauważ, że „zapomnieliśmy” sprawdzić stan zmiennej `source` (źródła) po wczytaniu z niej danych za pomocą funkcji `getline()`. Nie musimy tego robić, ponieważ w końcu dojdziemy do znajdującego się na początku pętli testu `!source.good()`.

Jak zawsze, jako wynik operatora `>>` zwracamy referencję do samego strumienia, `*this` — podrozdział 17.10.

Sprawdzanie białych znaków jest łatwe. Wystarczy porównać dany znak z każdym znakiem łańcucha stanowiącego zbiór białych znaków:

```
bool Punct_stream::is_whitespace(char c)
{
    for (char w : white)
        if (c==w) return true;
    return false;
}
```

Pamiętaj, że strumień `istream` operuje z typową definicją białych znaków (np. rozpoznaje znaki nowego wiersza i spacje) w typowy sposób, a więc nie musimy z nimi robić nic specjalnego.

Pozostaje jedna tajemnicza funkcja:

```
Punct_stream::operator bool()
{
    return !(source.fail() || source.bad()) && source.good();
}
```

Strumień `istream` jest zwykle wykorzystywany do testowania wyniku operatora `>>`. Na przykład:

```
while (ps>>s) { /*... */ }
```

Oznacza to, że musimy znaleźć sposób na traktowanie wyniku `ps>>s` jako wartości logicznej. Wynik ten jest obiektem typu `Punct_stream`, a więc musimy znaleźć sposób na niejawnie zamienienie takiego obiektu na typ `bool`. Do tego właśnie służy funkcja-operator `bool()` klasy `Punct_stream`. Funkcja składowa o nazwie `bool()` definiuje konwersję na typ `bool`. Zwraca wartość `true`, jeśli operacja na typie `Punct_stream` zakończy się powodzeniem.

Teraz możemy napisać nasz program:

```
int main()
// Przyjmuje tekst. Tworzy posortowaną listę wszystkich słów w tym tekście,
// ignoruje interpunkcję i różnice wielkości liter oraz
// usuwa duplikaty z danych wyjściowych
{
    Punct_stream ps{cin};
    ps.whitespace(" ;, . ? ! ( ) \n { } < > / & $ @ # % ^ * | ~ " ); // Zauważ, że \" oznacza \" w łańcuchu
    ps.case_sensitive(false);

    cout << "Podaj słowa:\n";
    vector<string> vs;
    string word;
    for (string word; ps>>word; ) // Wczytuje słowa

        sort(vs.begin(),vs.end()); // Sortuje w porządku leksykograficznym
        for (int i=0; i<vs.size(); ++i) // Zapisuje słownik
            if (i==0 || vs[i]!=vs[i-1]) cout << vs[i] << '\n';
}
```

Ten program zwróci posortowaną listę słów podanych mu na wejściu. Poniższy test:

```
if (i==0 || vs[i]!=vs[i-1])
```

odpowiada za pozbycie się duplikatów. Jeśli programowi temu zostanie podany poniższy tekst:

Są tylko dwa rodzaje języków: takie, na które ludzie narzekają i takie, których nikt nie używa.

zwróci następujący wynik:

```
dwa
i
języków
które
których
ludzie
na
narzekają
nie
nikt
rodzaje
są
takie
tylko
używa
```

Ostrzeżenie: klasa `Punct_stream` w wielu przypadkach działa podobnie jak strumień `istream`, ale nie jest nim w rzeczywistości. Nie można na przykład sprawdzić jej stanu za pomocą funkcji `rdstate()`, nie ma definicji funkcji `eof()` ani operatora `>>` wczytującego liczby całkowite. Co ważne, nie można funkcji przyjmującej argumenty typu `istream` podać argumentu typu `Punct_stream`. Czy da się tak zdefiniować klasę `Punct_stream`, aby była strumieniem `istream`? Tak, ale nie mamy jeszcze wystarczająco dużo doświadczenia programistycznego, wiedzy programistycznej oraz za słabo znamy język programowania, aby dokonać takiego wyczynu (jeśli za jakiś — długi — czas zechcesz wrócić do tego problemu, musisz poszukać informacji o buforach strumieniowych w jakimś podręczniku dla ekspertów).



Czy łatwo Ci się czytało kod klasy `Punct_stream`? Bez problemu zrozumiałeś objaśnienia tego kodu? Myślisz, że napisałbyś taki kod samodzielnie? Jeśli do niedawna byłeś prawdziwym nowicjuszem, szczerą odpowiedź powinna brzmieć: „Nie, nie, nie!”, a nawet: „NIE, nie! Nieeee!! — zwariowałeś?”. Rozumiemy to, a odpowiedź na ostatnie pytanie brzmi: „Nie, przynajmniej tak mi się wydaje”. Celem tego przykładu jest:



- Pokazać w miarę realistyczny problem i jego rozwiązanie.
- Pokazać, co można uzyskać, dysponując względnie ograniczonymi środkami.
- Dostarczyć łatwego w użyciu rozwiązania dla pozornie łatwego problemu.
- Zilustrować różnicę między interfejsem a implementacją.

Aby stać się programistą, musisz czytać dużo kodu i to nie tylko starannie dopieszczonych rozwiązań uczelnianych problemów. To jest przykład. Za kilka dni lub tygodni tekst ten będzie dla Ciebie łatwy, a nawet zaczniesz szukać sposobów na ulepszenie tego kodu.



Przykład ten można porównać do nauczyciela, który w czasie kursu języka angielskiego wtrącił kilka zdań w prawdziwym potocznym języku angielskim, aby dodać nieco koloru i życia do lekcji.

11.8. Zostało jeszcze tyle do poznania



Wiedza na temat wejścia i wyjścia wydaje się nieprzebrana. Prawdopodobnie jest nieskończona, ponieważ jedynym jej ograniczeniem jest ludzka pomysłowość i kapryśność. Nie wzięliśmy na przykład pod uwagę komplikacji związanych z językami naturalnymi. Liczba 12.35 w języku angielskim w większości innych europejskich języków zostałaby zapisana jako 12,35. Oczywiście biblioteka standardowa C++ zawiera odpowiednie mechanizmy pozwalające poradzić sobie z tymi i wieloma innymi aspektami wejścia i wyjścia w różnych językach. Jak wpisać chińskie znaki? Jak porównywać łańcuchy zapisane za pomocą znaków języka malajalam? Są odpowiedzi na te pytania, ale znacznie wykraczają poza zakres tej książki. Jeśli chcesz je poznać, zajrzyj do bardziej specjalistycznej lub zaawansowanej książki (np. *Standard C++ I/O Streams and Locales* Angeliki Langer *Język C++* Bjarne Stroustrupa) oraz dokumentacji biblioteki i systemu. Szukaj słowa **locale**, które jest zazwyczaj używane w odniesieniu do pracy z językami naturalnymi.

Kolejnym źródłem komplikacji jest buforowanie — strumienie iostream biblioteki standardowej wykorzystują konstrukcję nazywaną streambuf. Są one niezastąpione w zaawansowanych pracach — związanych z poprawianiem wydajności lub funkcjonalności — nad strumieniami wejścia i wyjścia. Jeśli chcesz zdefiniować własne takie strumienie lub dostosować istniejące do nowych źródeł lub miejsc docelowych danych, przeczytaj rozdział 38. książki *Język C++* Stroustrupa i dokumentację swojego systemu.

Używając języka C++, można natknąć się na rodzinę standardowych funkcji wejścia i wyjścia języka C `printf()` i `scanf()`. W takim przypadku zajrzyj do podrozdziału 27.6 i punktu B.10.2 lub świetnej książki o języku C autorstwa Kernighana i Ritchiego (*Język C*) lub poszukaj informacji w niezmiernych zasobach internetu. Każdy język programowania posiada własne mechanizmy wejścia i wyjścia. Różnią się one między sobą, większość jest dziwaczna, ale też znaczna część z nich odzwierciedla (na różne dziwne sposoby) te same fundamentalne pojęcia, które opisaliśmy w rozdziałach 10. i 11. tej książki.

Narzędzia wejścia i wyjścia biblioteki standardowej zostały zwięźle przedstawione w dodatku B.

Związany z treścią tego rozdziału temat graficznych interfejsów użytkownika (GUI) został opisany w rozdziałach od 12. do 16.

Ćwiczenia

1. Utwórz program o nazwie *Test_output.cpp*. Zadeklaruj zmienną typu `int` o nazwie `birth_year` i przypisz jej rok swojego urodzenia.
2. Wyślij na wyjście swoją datę urodzenia w formacie dziesiętnym, ósemkowym i szesnastkowym.
3. Oznacz każdą wartość etykietą określającą jej format.
4. Wyrównałeś dane wyjściowe w kolumnach za pomocą tabulacji? Jeśli nie, zrób to.
5. Wyślij na wyjście swój wiek.
6. Napotkałeś jakiś problem? Co się stało? Przywróć dziesiętny format danych wyjściowych.

7. Wróć do ćwiczenia 2. i spraw, aby była wyświetlana informacja o podstawie systemu liczbowego wyświetlonych liczb.
8. Wypróbuj wczytywanie liczb w formacie ósemkowym, szesnastkowym itp.:

```
cin >> a >> oct >> b >> hex >> c >> d;  
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

Uruchom ten kod i podaj mu poniższe dane:
1234 1234 1234 1234
Wyjaśnij wyniki.
9. Napisz kod drukujący liczbę 1234567.89 w notacji defaultfloat, fixed i scientific. Który format prezentacji jest najbliższy oryginałowi? Wyjaśnij dlaczego.
10. Utwórz prostą tabelę zawierającą imię i nazwisko, numer telefonu oraz adres e-mail — Twoje i przynajmniej pięciu Twoich przyjaciół. Wypróbuj kilka różnych szerokości pól, aż tabela będzie wyglądała satysfakcjonująco.

Powtórzenie

1. Dlaczego obsługa wejścia i wyjścia jest trudna dla programisty?
2. Co znaczy notacja <<hex?
3. Do czego używa się w informatyce liczb szesnastkowych? Dlaczego?
4. Wymień kilka opcji, które można by było zaimplementować, aby umożliwić formatowanie wysyłanych na wyjście liczb całkowitych.
5. Co to jest manipulator?
6. Jaki przedrostek oznacza liczbę dziesiętną, ósemkową i szesnastkową?
7. Jaki jest domyślny format wyjściowy wartości zmiennoprzecinkowych?
8. Co to jest pole?
9. Wyjaśnij działanie funkcji setprecision() i setw().
10. Czemu służą tryby otwierania plików?
11. Który z poniższych manipulatorów nie ma trwałego działania: hex, scientific, setprecision(), showbase, setw?
12. Jaka jest różnica między wejściem i wyjściem znakowym a binarnym?
13. Podaj przykład sytuacji, w której mogłoby być korzystne użycie pliku binarnego zamiast tekstowego.
14. Podaj dwa przykłady zastosowania strumienia stringstream.
15. Co to jest pozycja w pliku?
16. Co się stanie, jeśli pozycja w pliku zostanie ustawiona za jego końcem?
17. Kiedy lepiej jest wczytać dane wiersz po wierszu zamiast według typów?
18. Co robi funkcja isalnum(c)?

Terminologia

binarny	manipulator	regularność
dziesiętny	nieregularność	scientific
fixed	niestandardowy separator	setprecision()
formatowanie danych wyjściowych	noshowbase	showbase
general	ósemkowy	szesnastkowy
klasyfikacja znaków	pozycjonowanie w pliku	wprowadzanie danych po wierszu

Praca domowa

1. Napisz program wczytujący plik tekstowy, konwertujący jego treść na małe litery i tworzący nowy plik zawierający skonwertowaną treść.
2. Napisz program, który pobiera nazwę pliku i słowo oraz zwraca każdy wiersz zawierający to słowo wraz z numerem wiersza. Podpowiedź: `getline()`.
3. Napisz program usuwający z pliku wszystkie samogłoski. Na przykład tekst Pewnego razu powinien zmienić się w Pwng rz. Co ciekawe, często taki tekst nadal da się przeczytać. Pokaż go znajomym.
4. Napisz program o nazwie *multi_input.cpp* proszący użytkownika o wpisanie kilku liczb całkowitych w formatach ósemkowym, dziesiętnym i szesnastkowym w dowolnej kolejności przy użyciu przyrostków 0 i 0x. Niech poprawnie interpretuje te liczby i konwertuje je na format dziesiętny. Na koniec niech wyświetla je w odpowiednio sformatowanych kolumnach, np.:

szesnastkowa	0x43	zamienia się w 67	w formacie dziesiętnym
ósemkowa	0123	zamienia się w 83	w formacie dziesiętnym
dziesiętna	65	zamienia się w 65	w formacie dziesiętnym
5. Napisz program wczytujący znaki i wyświetlający dla każdego z nich klasyfikację zgodną z funkcjami klasyfikującymi opisanymi w podrozdziale 11.6. Zauważ, że jeden znak można zaklasyfikować do kilku grup, np. x jest literą i znakiem alfanumerycznym.
6. Napisz program zamieniający znaki interpunkcyjne na białe znaki. Uwzględnij kropkę (.), średnik (;), przecinek (,), znak zapytania (?), łącznik (-) i apostrof ('). Nie zmieniaj cudzysłowów ("), np. tekst "- don't use the as-if rule." powinien zostać zamieniony na " don t use the as if rule".
7. Zmodyfikuj program z poprzedniego zadania, aby skrócony zapis wyrażeń typu don't i can't itp. zamieniał na do not i cannot, pozostawiał łączniki między wyrazami (aby zwracał tekst typu "do not use the as-if rule") oraz konwertował wszystkie znaki na małe.
8. Za pomocą programu z poprzedniego zadania utwórz słownik (jako alternatywę dla rozwiązania z podrozdziału 11.7). Przepuść przez niego wielostronicowy tekst, obejrzyj wynik i pomyśl, czy można coś poprawić w programie, aby tworzył lepsze słowniki.
9. Podziel program obsługujący binarne wejście i wyjście z punktu 11.3.2 na dwa programy: jeden konwertujący zwykły plik tekstowy na binarny i drugi, wczytujący plik binarny i konwertujący go na tekstowy. Przetestuj te programy, porównując oryginalny plik z otrzymanym w wyniku konwersji w jedną i drugą stronę.

10. Napisz funkcję `vector<string> split(const string& s)` zwracającą wektor oddzielanych białymi znakami podłańcuchów z argumentu `s`.
11. Napisz funkcję `vector<string> split(const string& s, const string& w)` zwracającą wektor oddzielanych białymi znakami podłańcuchów z argumentu `s`, gdzie białe znaki to „normalne znaki” i znaki zapisane w łańcuchu `w`.
12. Odwróć kolejność znaków w pliku tekstowym. Na przykład tekst `asdfghjkl` zamienia się na `lkjhgfdsa`. Ostrzeżenie: nie ma naprawdę dobrego i efektywnego sposobu wczytywania pliku od końca, który działałby tak samo skutecznie na każdej platformie.
13. Odwróć kolejność słów (zdefiniowanych jako oddzielane białymi znakami łańcuchy) w pliku. Na przykład `Norwegian Blue parrot` zamienia się na `parrot Blue Norwegian`. Możesz założyć, że wszystkie naraz łańcuchy z pliku zmieszczą się w pamięci.
14. Napisz program wczytujący zawartość pliku tekstowego i drukujący informację o liczbie znaków każdego rodzaju według klasyfikacji z podrozdziału 11.6.
15. Napisz program wczytujący plik liczb oddzielonych białymi znakami i zwracający plik z tymi liczbami w formacie naukowym i precyzji 8 w czterech polach po 20 znaków na wiersz.
16. Napisz program wczytujący plik oddzielanych białymi znakami liczb i drukujący te liczby w porządku od najmniejszej do największej, po jednej w wierszu. Każdą wartość wydrukuj tylko raz. Jeśli jakaś wartość będzie się powtarzać, wydrukuj obok niej, w tym samym wierszu, informację o liczbie powtórzeń. Na przykład dane wejściowe `7 5 5 7 3 117 5` powinny dać następujący wynik:

3

5 3

7 2

117

Podsumowanie

Obsługa wejścia i wyjścia jest problematyczna, ponieważ ludzkie gusta i konwencje nie są zgodne z prostymi i bezpośrednimi prawami matematyki. Jako programiści rzadko nakazujemy użytkownikom porzucić przyzwyczajenia, a kiedy już to robimy, nie powinniśmy być tak aroganccy, aby myśleć, że możemy dostarczyć prostą alternatywę dla konwencji, które ustalano przez wiele lat. W związku z tym musimy spodziewać się, akceptować i przyzwyczaić się do pewnego stopnia komplikacji obsługi wejścia i wyjścia, cały czas starając się pisać jak najprostszy kod — ale ani trochę bardziej prosty.



