

27

Język C

„C to język programowania ze ścisłą kontrolą
i luźnym sprawdzaniem typów.”

— Dennis Ritchie

Ten rozdział zawiera zwięzły przegląd właściwości języka C i jego biblioteki standardowej dla osób znających język C++. Zawiera listę rzeczy, które są w C++, a nie ma w C, oraz przykłady rozwiązań związanych z tym problemami. Zostały w nim opisane płaszczyzny niezgodności między językami C i C++ oraz obszary, na których te języki współpracują. Jako przykłady przedstawiono operacje wejścia i wyjścia, manipulacje listami, zarządzanie pamięcią oraz przetwarzanie łańcuchów.

27.1. C i C++ to rodzeństwo

- 27.1.1. Zgodność języków C i C++
- 27.1.2. Co jest w języku C++, czego nie ma w C
- 27.1.3. Biblioteka standardowa języka C

27.2. Funkcje

- 27.2.1. Brak możliwości przeciążania nazw funkcji
- 27.2.2. Sprawdzanie typów argumentów funkcji
- 27.2.3. Definicje funkcji
- 27.2.4. Wywoływanie C z poziomu C++ i C++ z poziomu C
- 27.2.5. Wskaźniki na funkcje

27.3. Mniej ważne różnice między językami

- 27.3.1. Przestrzeń znaczników struktur
- 27.3.2. Słowa kluczowe
- 27.3.3. Definicje
- 27.3.4. Rzutowanie w stylu języka C
- 27.3.5. Konwersja typu void*
- 27.3.6. Typ enum
- 27.3.7. Przestrzenie nazw

27.4. Pamięć wolna

27.5. Łańcuchy w stylu języka C

- 27.5.1. Łańcuchy w stylu języka C i const
- 27.5.2. Operacje na bajtach
- 27.5.3. Przykład — funkcja strcpy()
- 27.5.4. Kwestia stylu

27.6. Wejście i wyjście — nagłówek stdio

- 27.6.1. Wyjście
- 27.6.2. Wejście
- 27.6.3. Pliki

27.7. Stałe i makra

27.8. Makra

- 27.8.1. Makra podobne do funkcji
- 27.8.2. Makra składniowe
- 27.8.3. Kompilacja warunkowa

27.9. Przykład — kontenery intruzyjne

27.1. C i C++ to rodzeństwo



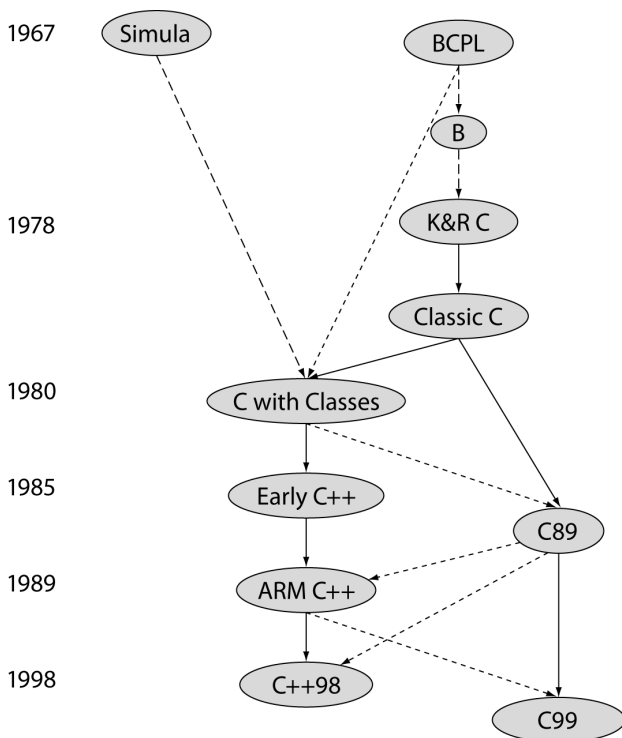
Język programowania C zaprojektował i pierwszy zaimplementował Dennis Ritchie w Bell Labs. Do jego popularyzacji przyczyniła się książka *Język ANSI C. Programowanie*, którą napisali Brian Kernighan i Dennis Ritchie (potocznie nazywa się ją K&R). To prawdopodobnie wciąż najlepsze wprowadzenie do języka C jest jedną z najlepszych książek o programowaniu w ogóle (punkt 22.2.5). Tekst pierwotnej definicji języka C++ był przeredagowaną wersją napisanego przez Dennisa Ritchiego tekstu definicji języka C z 1980 roku. Od tego czasu języki te cały czas ewoluowały. Język C, podobnie jak C++, doczekał się już standardu ISO.

Język C zasadniczo traktuje się jako podzbiór języka C++. Dlatego problematyka opisu języka C z punktu widzenia języka C++ sprowadza się do dwóch rzeczy:

- Opis obszarów, w których język C nie jest podzbiorem języka C++.
- Opis narzędzi języka C++, których nie ma w C, oraz czym można je zastąpić.



Z historycznych powodów aktualne wersje języków C i C++ są rodzeństwem. Oba są bezpośrednimi potomkami klasycznego języka C — dialektu języka C spopularyzowanego przez pierwsze wydanie książki Ritchiego i Kernighana *Język ANSI C. Programowanie* wzbogaconego o przypisywanie struktur i wyliczenia:



Obecnie najczęściej używa się wersji języka C o nazwie C89 (zgodnie z drugim wydaniem książki K&R) i tę właśnie wersję opiszemy tutaj. W użyciu są jeszcze wersje klasyczna i C99, ale nie powinny sprawić problemów nikomu, kto zna języki C++ i C89.

Języki C i C++ „narodziły się” w centrum badań informatycznych Bell Labs w Murray Hill w stanie New Jersey (przez jakiś czas moje pomieszczenie znajdowało się po przeciwnej stronie korytarza w odległości kilku drzwi od pomieszczeń Dennisa Ritchiego i Briana Kernighana):



Oba języki są obecnie zdefiniowane i kontrolowane przez komisje standaryzacyjne ISO. W użyciu jest wiele implementacji każdego z nich. Wiele implementacji obsługuje oba te języki. Wyboru jednego z nich dokonuje się za pomocą przełącznika w kompilatorze lub rozszerzenia pliku źródłowego. Oba są dostępne na większej liczbie platform niż jakikolwiek inny język. Oba zostały zaprojektowane do trudnych zadań programistycznych i są wykorzystywane w takich dziedzinach jak programowanie:



- jąder systemów operacyjnych,
- sterowników urządzeń,
- systemów wbudowanych,
- kompilatorów,
- systemów komunikacyjnych.

Nie ma żadnej różnicy pod względem wydajności między dwoma odpowiadającymi sobie programami w językach C i C++.

Podobnie jak C++, język C jest szeroko stosowany. Społeczność programistów skupiona na tych dwóch językach jest największą społecznością programistyczną na świecie.

27.1.1. Zgodność języków C i C++

Nierzadko spotyka się odwołania do „C/C++”. Oczywiście nie ma języka o takiej nazwie i występowanie frazy „C/C++” jest zwykle oznaką ignorancji. My używamy określenia „C/C++” tylko w kontekście mowy o zgodności i dużej społeczności, która korzysta z obu tych języków.



Język C++ jest w dużym stopniu (ale nie w pełni) nadzbiorem języka C. Z bardzo rzadkimi wyjątkami konstrukcje, które są zgodne zarówno z C, jak i C++, mają takie samo znaczenie (semantykę) w obu językach. Język C++ został zaprojektowany, „aby był tak bliski językowi C, jak to możliwe, ale ani trochę bliższy”:



- Aby ułatwić przejście z jednego na drugi.
- Aby umożliwić im współistnienie.

Większość niezgodności ma związek ze ściślejszą kontrolą typów w języku C++.

Przykładem programu poprawnego w języku C, a niepoprawnego w C++ jest taki, w którym użyto jako identyfikatora słowa kluczowego języka C++, którego nie ma w C (punkt 27.3.2):

```
int class(int new, int bool); /* C, ale nie C++ */
```

Trudniej jest znaleźć przykład konstrukcji, która jest poprawna w obu językach, ale ma inną semantykę. Poniżej znajduje się jeden taki przykład:

```
int s = sizeof('a'); /* sizeof(int), w C zwykle 4, a w C++ 1 */
```

Literał znakowy, np. 'a', w języku C ma typ int, a w C++ char. Jednak dla zmiennej typu char o nazwie ch w obu językach sizeof(ch)=1.

Informacje na temat podobieństw i różnic między językami nie są zbyt fascynujące. Nie można się nauczyć żadnych ciekawych nowych technik. Może spodobać Ci się funkcja printf() (podrozdział 26.7), ale poza tym potencjalnym wyjątkiem (i kilkoma nieudolnymi próbami wniesienia trochę technicznego humoru) ten rozdział jest suchy jak pieprz. Jego zadanie jest proste — pozwolić Ci rozumieć i pisać kod w języku C, jeśli będziesz tego potrzebować. To wymaga objaśnienia pułapek, które są oczywiste dla doświadczonych użytkowników języka C, ale całkowicie niespodziewane przez programistów języka C++. Mamy nadzieję, że nauczysz się ich unikać po jak najmniejszej liczbie wpadek.

Większość programistów języka C++ wcześniej czy później będzie miała do czynienia z kodem w języku C i odwrotnie. Wiele opisywanych przez nas tu rzeczy jest znanych większości programistów języka C, chociaż niektóre będą uważane za zaawansowane. Powód tego jest prosty — nie ma zgody, co należy uważać za zaawansowane, a my opisujemy to, co można spotkać w realnym kodzie. Możliwe, że zrozumienie zawłości związanych ze zgodnością języków może być tanim sposobem na uzyskanie sobie niezasłużonej opinii „eksperta”. Pamiętaj — bycie ekspertem polega na posługiwaniu się w zaawansowany sposób językiem programowania (w tym przypadku C), a nie na znajomości dziwacznych zasad języka (które ujawniają się przy rozważaniu zagadnień związanych ze zgodnością).

Źródła

ISO/IEC 9899:1999, *Programming Languages — C*. Definicja języka C99; większość implementacji implementuje język C89 (zwykle z kilkoma rozszerzeniami).

ISO/IEC 9899:2011, *Programming Languages — C*. Opisuje język C11.

ISO/IEC 14882:2011, *Programming Languages — C++*.

Kernighan Brian W., Ritchie Dennis M., *Język ANSI C. Programowanie. Wydanie II*, Helion, 2010.

Stroustrup Bjarne, *Learning Standard C++ as a New Language*, „C/C++ Users Journal”, maj 1999.

Stroustrup Bjarne, *C and C++: Siblings, C and C++: A Case for Compatibility* oraz *C and C++: Case Studies in Compatibility*, „C/C++ Users Journal”, lipiec, sierpień i październik 2002.

Artykuły Stroustrupa najłatwiej znaleźć na jego stronie internetowej.

27.1.2. Co jest w języku C++, czego nie ma w C

Z perspektywy języka C++ językowi C (tj. C89) brakuje wielu rzeczy:

- Klas i funkcji składowych:
 - Należy używać struktur i funkcji globalnych.
- Klas pochodnych i funkcji wirtualnych:
 - Należy używać struktur, funkcji globalnych i wskaźników na funkcje (punkt 27.2.3).
- Szablonów i funkcji wstawianych (ang. *inline function*):
 - Należy używać makr (podrozdział 27.8).
- Wyjątków:
 - Należy używać kodów błędów, wartości zwrrotnych oznaczających błąd itp.
- Przeciążania funkcji:
 - Należy każdej funkcji nadawać unikatową nazwę.
- Operatorów `new` i `delete`:
 - Należy używać funkcji `malloc()` i `free()` oraz osobnego kodu inicjalizującego i czyszczącego.
- Referencji:
 - Należy używać wskaźników.
- Słów kluczowych `const` i `constexpr` oraz funkcji w wyrażeniach stałych:
 - Należy używać makr.
- Typu `bool`:
 - Należy używać typu `int`.
- Rzutowań `static_cast`, `reinterpret_cast` oraz `const_cast`:
 - Należy stosować rzutowanie w stylu języka C, np. `(int)a` zamiast `static<int>(a)`.

W języku C napisano mnóstwo wartościowego kodu, dlatego powyższa lista powinna stanowić przypomnienie, że żadne narzędzie językowe nie jest bezwzględnie potrzebne. Większość z nich (także w języku C) służy tylko jako udogodnienie dla programisty. Przecież przy wystarczającej ilości czasu, inteligencji i cierpliwości każdy program można by było napisać w assemblerze. Warto zauważyć, że dzięki temu iż języki C i C++ posługują się modelem, który jest bardzo bliski rzeczywistemu sprzętowi, dobrze nadają się do imitowania różnych stylów programowania.

W pozostałej części tego rozdziału objaśniamy pisanie programów bez użycia tych narzędzi. Oto nasze podstawowe rady dla użytkowników języka C:

- Naśladuj techniki programistyczne, które są obsługiwane przez narzędzia dostępne w języku C++ za pomocą narzędzi języka C.
- Jeśli piszesz w języku C, poruszaj się w obszarze będącym podzbiorem języka C++.



- Wykorzystuj te poziomy ostrzeżeń kompilatora, które zapewniają sprawdzanie argumentów funkcji.
- Analizuj duże programy za pomocą narzędzia lint (punkt 27.2.2).

Wiele szczegółów dotyczących zgodności języków C i C++ to zawile techniczne zagadnienia. Aby jednak pisać kod w języku C, nie trzeba większości tych rzeczy pamiętać:

- Kompilator ostrzeże Cię, że używasz narzędzia języka C++, którego nie ma w C.
- Jeśli podporządkujesz się powyższym zasadom, istnieje bardzo małe prawdopodobieństwo, że napotkasz cokolwiek, co w języku C znaczy coś innego niż w języku C++.

Przy braku tyłu narzędzi języka C++ większego znaczenia nabierają niektóre narzędzia języka C:

- tablice i wskaźniki,
- makra,
- typedef (odpowiednik w C i C++98 deklaracji using — podrozdział 20.5 i punkt A.16),
- sizeof,
- rzutowanie.

W rozdziale tym przedstawimy kilka przykładów ich użycia w takiej roli.

Wprowadziłem do języka C++ komentarze w stylu `//` (po przodku języka C języku BCPL), ponieważ miałem już dość pisania komentarzy `/* .. */`. Komentarze `//` akceptuje większość dialektów języka C włącznie z C99 i C11, a więc ich stosowanie wydaje się bezpieczne. W tym rozdziale w przykładach kodu w języku C będziemy używać tylko komentarzy `/* .. */`. W języku C99 i C11 wprowadzono kilka narzędzi więcej z języka C++ (a także kilka rzeczy, które nie są zgodne z językiem C++), ale tutaj skoncentrujemy się na języku C89, który jest znacznie bardziej popularny.



27.1.3. Biblioteka standardowa języka C



Oczywiście narzędzia biblioteki C++, które wymagają klas i szablonów, nie są dostępne w języku C. Należą do nich:

- typ `vector`;
- typ `map`;
- typ `set`;
- typ `string`;
- algorytmy STL, np. `sort()`, `find()`, `copy()`;
- strumienie wejścia i wyjścia;
- `regex`.

Ich brak w bibliotekach języka C można zwykle zrekompensować tablicami, wskaźnikami i funkcjami. Główne części standardowej biblioteki C to:

- `<stdlib.h>` — narzędzia ogólnego przeznaczenia (np. funkcje `malloc()` i `free()` — podrozdział 27.4).
- `<stdio.h>` — standardowe wejście i wyjście (podrozdział 27.6).
- `<string.h>` — przetwarzanie łańcuchów i zarządzanie pamięcią w stylu języka C (podrozdział 27.5).
- `<math.h>` — standardowe zmiennoprzecinkowe funkcje matematyczne (podrozdział 24.8).
- `<errno.h>` — kody błędów dla nagłówka `<math.h>` (podrozdział 24.8).
- `<limits.h>` — rozmiary typów całkowitoliczbowych (podrozdział 24.2).
- `<time.h>` — daty i godziny (punkt 26.6.1).
- `<assert.h>` — asercje wspomagające debugowanie (podrozdział 27.9).
- `<ctype.h>` — klasyfikacja znaków (podrozdział 11.6).
- `<stdbool.h>` — makra logiczne.

Kompletny opis znajdziesz w dobrym podręczniku do języka C, np. K&R. Wszystkie te biblioteki (i pliki nagłówkowe) są obecne także w języku C++.

27.2. Funkcje

W języku C:

- Może być tylko jedna funkcja o określonej nazwie.
- Sprawdzanie typu argumentów funkcji nie jest obowiązkowe.
- Nie ma referencji (i przez to przekazywania przez referencję).
- Nie ma funkcji składowych.
- Nie ma funkcji wstawianych (ang. *inline function*) — z wyjątkiem języka C99.
- Istnieje alternatywna składnia do definiowania funkcji.

Poza wymienionymi rzeczami reszta jest podobna do języka C++. Sprawdźmy, co to znaczy.

27.2.1. Brak możliwości przeciążania nazw funkcji

Rozważmy:

```
void print(int); /* drukuje liczbę typu int */
void print(const char*); /* drukuje łańcuch */ /* Błąd! */
```


Druga deklaracja jest błędna, ponieważ nie może być dwóch funkcji o takich samych nazwach. Trzeba w związku z tym wymyślić odpowiednią parę nazw:

```
void print_int(int); /* drukuje liczbę typu int */
void print_string(const char*); /* drukuje łańcuch */
```

Niektórzy każą traktować to jako zaletę, ponieważ nie można teraz użyć niewłaściwej funkcji do wydrukowania liczby typu `int`! Oczywiście nie dajemy się na to nabrać. Niemożność przeciążania funkcji sprawia, że trudno elegancko zaimplementować programowanie ogólne, ponieważ jego zasada opiera się na semantycznie podobnych funkcjach, które mają takie same nazwy.

27.2.2. Sprawdzanie typów argumentów funkcji

Rozważmy:

```
int main()
{
    f(2);
}
```



Kompilator C zgodzi się na takie coś — nie trzeba deklarować funkcji przed jej użyciem (choć można i powinno się). Definicja funkcji `f()` może gdzieś się znajdować. Może być w innej jednostce translatablejnej, ale jeśli jej nie będzie, konsolidator zgłosi błąd.

Niestety ta definicja w innym pliku źródłowym może wyglądać następująco:

```
/* other_file.c: */
int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

Konsolidator nie zgłosi tego błędu. Wystąpi błąd czasu wykonywania lub program zwróci jakiś losowy wynik.



Jak radzić sobie z tego rodzaju problemami? Praktycznym rozwiązaniem jest spójne stosowanie plików nagłówkowych. Jeśli wszystkie wywoływane lub definiowane funkcje są zadeklarowane w nagłówku, który jest zawsze dołączany, gdy jest potrzebny, mamy sprawdzanie. W dużych programach jest jednak trudno to osiągnąć. Dlatego większość kompilatorów języka C ma opcje ostrzegające o niezadeklarowanych funkcjach — zalecamy korzystanie z nich. Dodatkowo od samego początku istnienia języka C istnieją programy do wyszukiwania wszelkich problemów ze spójnością. Zwykle nazywa się je narzędziami typu **lint**. Używaj tych narzędzi we wszystkich niebanalnych programach. Odkryjesz, że narzędzie `lint` będzie Cię pchać w stronę takiego stylu używania języka C, który przypomina używanie podzbioru języka C++. Jednym ze spostrzeżeń, które doprowadziły do opracowania języka C++, było to, że kompilator mógł z łatwością sprawdzić wiele rzeczy (ale nie wszystkie), które sprawdza narzędzie `lint`.

Można poprosić o sprawdzanie typów argumentów funkcji w języku C. W tym celu należy w deklaracji funkcji określić typy jej argumentów (tak jak w C++). Deklaracja taka nazywa się **prototypem funkcji**. Należy uważać na deklaracje funkcji, w których nie ma określonych argumentów. Nie są to prototypy funkcji i nie wymuszają sprawdzania argumentów funkcji:

```
int g(double); /* Prototyp — jak deklaracja funkcji w języku C++ */
int h();       /* Nie prototyp — nieokreślone typy argumentów */

void my_fct()
{
    g();        /* Błąd: brak argumentu */
    g("asdf"); /* Błąd: zły typ argumentu */
    g(2);       /* Dobrze: wartość 2 zostanie przekonwertowana na 2.0 */
    g(2,3);     /* Błąd: o jeden argument za dużo */

    h();        /* Dobrze wg kompilatora! Może spowodować niespodziewane wyniki */
    h("asdf"); /* Dobrze wg kompilatora! Może spowodować niespodziewane wyniki */
    h(2);       /* Dobrze wg kompilatora! Może spowodować niespodziewane wyniki */
    h(2,3);    /* Dobrze wg kompilatora! Może spowodować niespodziewane wyniki */
}
```

W deklaracji funkcji `h()` nie podano typów argumentów. To nie oznacza, że funkcja ta nie przyjmuje argumentów, tylko że „przyjmuje każdy zestaw argumentów z nadzieją, że są odpowiednie dla wywołania”. Dobry kompilator ostrzeże o tym, a narzędzie `lint` znajdzie ten problem.



C++	Odpowiednik w C
<code>void f() // preferowany</code>	<code>void f(void)</code>
<code>void f(void)</code>	<code>void f(void)</code>
<code>void f(...) // przyjmuje wszystkie argumenty</code>	<code>void f() /* przyjmuje wszystkie argumenty */</code>

Istnieje specjalny zestaw reguł dla konwersji argumentów, jeśli w zasięgu nie ma prototypu funkcji. Na przykład typy `char` i `short` są konwertowane na `int`, a `float` na `double`. Jeśli chcesz dowiedzieć się, co się dzieje np. z typem `long`, sięgnij po dobry podręcznik do języka C. Nasze zalecenie jest proste — nie wywołuj funkcji bez prototypów.

Należy zauważyć, że mimo iż kompilator pozwala na przekazanie argumentu niewłaściwego typu, jak np. `char*` zamiast `int`, jest to błędem. Jak powiedział Dennis Ritchie: „C to język programowania ze ścisłą kontrolą i luźnym sprawdzaniem typów”.

27.2.3. Definicje funkcji

Funkcje można definiować dokładnie tak samo, jak w języku C++, i definicje te są prototypami funkcji:

```
double square(double d)
{
    return d*d;
}
```



```

void ff()
{
    double x = square(2);      /* Dobrze: konwersja 2 na 2.0 i wywołanie */
    double y = square();      /* brak argumentu */
    double y = square("Witaj"); /* Błąd: nieprawidłowy typ argumentu */
    double y = square(2,3);    /* Błąd: za dużo argumentów */
}

```

Definicja funkcji bez argumentów nie jest prototypem funkcji:

```

void f() { /* coś robi */ }

void g()
{
    f(2); /* Dobrze w C; błąd w C++ */
}

```

Fakt, że definicję typu:

```
void f(); /* nie określono typu argumentów */
```

oznaczały: „funkcja `f()` może przyjąć dowolną liczbę argumentów dowolnego typu”, wydawał mi się dziwny. Dlatego opracowałem nową notację, w której „nic” trzeba bezpośrednio oznaczyć słowem kluczowym `void` (*void* po angielsku znaczy „pusty”):

```
void f(void); /* nie przyjmuje żadnych argumentów */
```



Szybko jednak tego żałowałem, ponieważ to też wygląda dziwnie i na dodatek to jedno słowo więcej jest całkiem zbędne, gdy sprawdzanie typów jest ujednolicone. Co gorsza, Dennis Ritchie (ojciec języka C) i Doug McIlroy (wyrocznia w tematach wyczucia stylu w centrum badań komputerowych Bell Labs — punkt 22.2.5) nazwali to „obrzydlistwem”. Niestety to obrzydlistwo szybko zyskało popularność wśród programistów języka C. Nie należy jednak stosować tego w języku C++, w którym nie tylko jest brzydactwem, ale jest też logicznie zbędne.

Język C obsługuje też drugi styl definiowania funkcji zaczerpnięty z języka Algol60. Typy parametrów można opcjonalnie określać osobno od ich nazw:

```

int old_style(p,b,x) char* p; char b;
{
    /* ... */
}

```



Ten „stary styl definiowania” jest sporo starszy od języka C++ i nie jest prototypem. Zgodnie z konwencją argument, który nie ma określonego typu, jest typu `int`. W związku z tym w funkcji `old_style()` parametr `x` jest typu `int`. Funkcję `old_style()` można wywołać w następujący sposób:

```

old_style();          /* Dobrze: nie ma żadnego argumentu */
old_style("witaj", 'a', 17); /* Dobrze: wszystkie argumenty mają prawidłowe typy */
old_style(12, 13, 14); /* Dobrze: 12 jest niewłaściwego typu, ale */
                        /* może funkcja old_style() nie użyje p */

```

Kompilator powinien zaakceptować powyższe wywołania (ale mamy nadzieję, że dla pierwszego i ostatniego zgłosi ostrzeżenie).

Oto nasze zalecenia dotyczące sprawdzania argumentów funkcji:

- Konsekwentnie stosuj prototypy funkcji (używaj plików nagłówkowych).
- Tak ustaw poziom ostrzeżeń kompilatora, aby zgłaszał błędy typów argumentów.
- Używaj jakiegoś narzędzia typu lint.

W wyniku powstanie kod, który będzie poprawny także w języku C++.

27.2.4. Wywoływanie C z poziomu C++ i C++ z poziomu C

Można połączyć pliki skompilowane kompilatorem języka C z plikami skompilowanymi kompilatorem języka C++, pod warunkiem że te dwa kompilatory zostały do tego przystosowane. Można na przykład połączyć pliki obiektowe wygenerowane z kodu źródłowego w językach C i C++ przy użyciu kompilatora GNU C i C++ (GCC). To samo można zrobić z plikami wygenerowanymi przez kompilator C i C++ firmy Microsoft (MSC++). Jest to często stosowana i przydatna technika, która pozwala korzystać z większej liczby bibliotek, niż jest dostępna tylko w jednym języku.

W języku C++ jest ściślejsza kontrola typów niż w C. Kompilator i konsolidator C++ sprawdza, czy funkcje `f(int)` i `f(double)` są jednolicie zdefiniowane i konsekwentnie używane — nawet jeśli znajdują się w różnych plikach źródłowych. Konsolidator języka C nie sprawdza takich rzeczy. Aby wywołać funkcję zdefiniowaną w języku C z poziomu języka C++ i odwrotnie, należy poinformować kompilator, co się robi:



// Wywoływanie funkcji zdefiniowanej w języku C z poziomu języka C++:

```
extern "C" double sqrt(double); // Dołącza jako funkcję w języku C

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

Instrukcja `extern "C"` nakazuje kompilatorowi zastosować konwencje konsolidatora języka C. Poza tym cały ten kod jest normalny z punktu widzenia języka C++. W istocie standardowa funkcja `sqrt(double)` w języku C++ jest funkcją `sqrt(double)` z biblioteki standardowej języka C. W programie w języku C nie jest nic wymagane, aby umożliwić wywołanie funkcji z języka C++ w taki sposób. C++ przystosuje się do konwencji łączenia konsolidatora C.

Za pomocą instrukcji `extern "C"` można także wywołać funkcję C++ z poziomu języka C:

// Wywołanie funkcji C++ z poziomu języka C:

```
extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```


Teraz w programie w języku C można pośrednio wywołać funkcję składową `f()`:

/ Wywołanie funkcji C++ z poziomu języka C: */*

```
int call_f(S* p, int i);
struct S* make_S(int, const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p, i);
    /* ... */
}
```

Nie trzeba (a nawet nie da się) wspominać nic o języku C++, aby ten kod działał.

Zalety takiej współpracy języków są oczywiste — można pisać mieszankę kodu C i C++. W programach w języku C++ można wykorzystywać biblioteki napisane w języku C i odwrotnie. Ponadto większość języków (zwłaszcza Fortran) ma interfejsy do współpracy z językiem C.

W powyższych przykładach przyjęliśmy, że języki C i C++ mogą wspólnie korzystać z obiektu klasy wskazywanego przez `p`. W przypadku większości obiektów klas jest to prawdą. W szczególności jeśli ma się taką klasę jak poniższa:

```
// WC++:
class complex {
    double re, im;
public:
    // Wszystkie typowe operacje
};
```

Można bez kłopotów przekazywać wskaźniki na obiekty do i z języka C. Można nawet uzyskać dostęp do składowych `re` i `im` w programie C, posługując się deklaracją:

```
/* WC: */
struct complex {
    double re, im;
    /* brak operacji */
};
```



Zasady dotyczące układu elementów mogą być skomplikowane w każdym języku programowania, a na gruncie współpracy między językami jest jeszcze gorzej. Między językami C i C++ można przesyłać typy wbudowane i klasy (`struct`) bez funkcji wirtualnych. Jeśli klasa zawiera funkcje wirtualne, należy przekazywać tylko wskaźniki do jej obiektów i wszelkie operacje na nich wykonywać w kodzie C++. Przykładem tego była funkcja `call_f()` — funkcja `f()` mogłaby być wirtualna i wówczas tamten przykład ilustrowałby sposób wywoływania funkcji wirtualnej w języku C.

Najprostszym i najbezpieczniejszym sposobem na współdzielenie typów między językami C i C++ jest, obok stosowania typów wbudowanych, definiowanie struktur we wspólnych plikach nagłówkowych. To jednak znacznie ogranicza możliwości używania języka C++, a więc nie ograniczamy się tylko do tej metody.

27.2.5. Wskaźniki na funkcje

Co można zrobić, jeśli chce się w języku C zastosować techniki obiektowe (podrozdziały 14.2 – 14.4)? Zasadniczo potrzebne jest jakieś zastępstwo dla funkcji wirtualnych. Większości ludzi jako pierwszy do głowy przychodzi pomysł użycia struktury ze specjalnym polem określającym, jaki rodzaj figury dany obiekt reprezentuje. Na przykład:

```
struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* ... */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
    case circle:
        /* Rysuje koło */
        break;
    case rectangle:
        /* Rysuje prostokąt */
        break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* ... */
}
```

Technika ta działa, ale ma dwie wady:

- Dla każdej „pseudowirtualnej funkcji” (np. `draw()`) trzeba napisać osobną instrukcję `switch`.
- Za każdym razem, gdy dodaje się nową figurę, trzeba zmodyfikować każdą „pseudowirtualną funkcję”, dodając do instrukcji `switch` klauzulę `case`.

Drugi z tych problemów jest szczególnie uciążliwy, ponieważ uniemożliwia umieszczanie takich „pseudowirtualnych funkcji” w bibliotekach, gdyż użytkownicy dość często muszą je modyfikować. Najbardziej efektywną alternatywą są wskaźniki na funkcje:

```
typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* ... */
};

void draw(struct Shape2* p)
{
```



```

    (p->draw)(p);
}
void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}

```

Tego typu Shape2 można używać tak samo, jak Shape1.

```

int f(struct Shape2* pp)
{
    draw(pp);
    /* ... */
}

```

Przy odrobinie dodatkowej pracy obiekt nie musi zawierać jednego wskaźnika na funkcję dla każdej pseudowirtualnej funkcji. Może zamiast tego zawierać wskaźnik na tablicę wskaźników na funkcje (podobnie są implementowane funkcje wirtualne w języku C++). Największym problemem w realnych programach z tą techniką jest to, że trudno dobrze zainicjować wszystkie te wskaźniki na funkcje.

27.3. Mniej ważne różnice między językami

W tym podrozdziale zostaną opisane mniej ważne różnice między językami C i C++, których nieznamość może wciągnąć w pułapkę. Niewiele z nich ma duży wpływ na programowanie, ponieważ większość tych różnic można obejść.

27.3.1. Przestrzeń znaczników struktur

W języku C nazwy struktur (struct — w języku tym nie ma słowa kluczowego class) należą do innej przestrzeni nazw niż pozostałe identyfikatory. Dlatego przed nazwą każdej struktury (nazywaną **znacznikiem struktury** — ang. *struct tag*) musi znajdować się przedrostek struct. Na przykład:

```

struct pair { int x,y; };
pair p1;          /* Błąd: nie ma w zasięgu identyfikatora pair */
struct pair p2; /* Dobrze */
int pair = 7;     /* Dobrze: znacznik struktury pair nie jest w tym zakresie */
struct pair p3; /* Dobrze: znacznik struktury pair nie został przesłonięty przez liczbę int */
pair = 8;         /* Dobrze: identyfikator pair odnosi się do liczby int */

```

Co zaskakujące, dzięki chytrej sztuczce związanej ze zgodnością to zadziała także w języku C++. Tworzenie zmiennych (lub funkcji) o takich samych nazwach jak struktury jest często spotykane w języku C, chociaż my tego nie zalecamy.



Jeśli nie chcesz przed nazwą każdej struktury pisać słowa struct, użyj typedef (podrozdział 20.5). Często spotyka się coś takiego, jak poniżej:

```

typedef struct { int x,y; } pair;
pair p1 = { 1, 2 };

```


Zasadniczo w języku C często spotyka się instrukcje z typedef, ponieważ nie ma w nim możliwości definiowania nowych typów ze skojarzonymi z nimi operacjami.

W języku C nazwy zagnieżdżonych struktur znajdują się w tym samym zakresie dostępności, co nazwy struktur, w których są zagnieżdżone. Na przykład:



```
struct S {
    struct T { /*... */ };
    /*... */
};

struct T x; /*Dobrze w C (ale nie w C++) */
```

W języku C++ trzeba by było napisać:

```
S::T x; // Dobrze w C++ (ale nie w C)
```

Jeśli tylko jest to możliwe, należy unikać zagnieżdżania struktur w języku C. Zasady dotyczące zakresu ich dostępności są inne, niż większość ludzi naiwnie (i rozsądnie) myśli.

27.3.2. Słowa kluczowe

Wiele słów, które są kluczowe w języku C++, nie jest kluczowych w języku C (ponieważ nie ma w nim określonej przez nie funkcjonalności), a więc można w języku C używać ich jako identyfikatorów:

Słowa kluczowe języka C++, które nie są słowami kluczowymi w języku C					
alignas	and_eq	asm	bitand	bitor	bool
alignof					
and					
catch	class	compl	const_cast	delete	dynamic_cast
char16_t		concept	constexpr		
char32_t					
explicit	export	false	friend	inline	mutable
				nullptr	
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	reinterpret_cast	static_cast
		thread_local		requires	
				static_assert	
template	this	throw	true	try	typeid
typename	using	virtual	wchar_t	xor	xor_eq

Nie należy tych nazw używać w kodzie języka C jako identyfikatorów, jeśli chce się, aby był zgodny z językiem C++. Jeśli którejkolwiek z nich użyje się w pliku nagłówkowym, cały ten plik będzie bezużyteczny w języku C++.



Niektóre słowa kluczowe języka C++ są w języku C makrami:

Słowa kluczowe C++, które są makrami w języku C							
and	and_eq	bitand	bitor	bool	compl	false	
not	not_eq	or	or_eq	true	wchar_t	xor	xor_eq

W języku C ich definicje znajdują się w nagłówkach `<iso646.h>` i `<stdbool.h>` (`bool`, `true`, `false`). Nie wykorzystuj faktu, że są to makra w języku C.

27.3.3. Definicje

W języku C++ definicje można umieszczać w większej liczbie różnych miejsc niż w języku C89. Na przykład:

```
for (int i = 0; i<max; ++i) x[i] = y[i]; // Definicja i niedozwolona w C
while (struct S* p = next(q)) {          // Definicja p niedozwolona w C
    /*... */
}

void f(int i)
{
    if (i< 0 || max<=i) error("Błąd zakresu.");
    int a[max]; // Błąd: w języku C za instrukcją nie może znajdować się deklaracja
    /*... */
}
```

W języku C (C89) deklaracje nie mogą występować jako inicjalizatory w instrukcjach `for`, jako warunki ani za instrukcjami w blokach. Trzeba pisać kod podobny do poniższego:

```
int i;
for (i = 0; i<max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /*... */
}

void f(int i)
{
    if (i< 0 || max<=i) error("Błąd zakresu.");
    {
        int a[max];
        /*... */
    }
}
```

W języku C++ niezainicjowana deklaracja jest definicją. W języku C jest to tylko deklaracja, a więc może być ich więcej niż jedna:

```
int x;
int x; /* W języku C definiuje lub deklaruje liczbę całkowitą o nazwie x. W języku C++ to jest błąd */
```


W języku C++ wszystko może być zdefiniowane tylko raz. Ciekawiej będzie, gdy powyższe dwie liczby `int` umieści się w dwóch różnych jednostkach translacyjnych:

```
/* W pliku x.c: */
int x;
```

```
/* W pliku y.c: */
int x;
```

Żaden kompilator C czy C++ nie znajdzie w plikach `x.c` i `y.c` jakiegokolwiek błędu. Jeśli jednak skompiluje się je w kompilatorze języka C++, konsolidator zgłosi błąd „podwójnej definicji”. Przy kompilacji kompilatorem języka C konsolidator zaakceptuje ten program (który jest poprawny według zasad języka C) i uzna, że jest jedna zmienna `x` współdzielona przez kod w tych dwóch plikach. Jeśli chcesz, aby globalna zmienna `x` była współdzielona, poinformuj o tym bezpośrednio:

```
/* W pliku x.c: */
int x = 0; /* definicja */
```

```
/* W pliku y.c: */
extern int x; /* deklaracja — nie definicja */
```

Lepiej jednak użyć pliku nagłówkowego:

```
/* W pliku x.h: */
extern int x; /* deklaracja — nie definicja */
```

```
/* W pliku x.c: */
#include "x.h"
int x = 0; /* definicja */
```

```
/* W pliku y.c: */
#include "x.h"
/* Deklaracja x znajduje się w nagłówku */
```

Jeszcze lepiej jest unikać globalnych zmiennych.

27.3.4. Rzutowanie w stylu języka C

W języku C (i C++) można jawnie przekonwertować wartość `v` na typ `T` przy użyciu następującej minimalnej notacji:

`(T)v`

Ten styl rzutowania języka C (albo tzw. stary styl rzutowania) jest uwielbiany przez osoby słabo piszące na klawiaturze i nie lubiące zbyt dużo myśleć, ponieważ jest minimalny i nie wymaga wiedzy, na czym polega konwersja `v` na typ `T`. Z drugiej strony, stylu tego jak ognia boją się programiści zajmujący się utrzymaniem kodu, ponieważ nie wynika z niego w żaden sposób, co miał na myśli implementator. Rzutowania w stylu języka C++ (**rzutowanie w nowym stylu** lub **rzutowanie w stylu szablonowym** — punkt A.5.7) zostało po to wprowadzone, aby jawne konwersje typów były łatwe do zauważenia (brzydkie) i specyficzne. W języku C nie ma wyboru:




```

int* p = (int*)7; /* Reinterpretacja wzorca bitowego: reinterpret_cast<int*>(7) */
int x = (int)7.5; /* Przycięcie typu double: static_cast<int>(7.5) */

typedef struct S1 { /*... */ } S1;
typedef struct S2 { /*... */ } S2;
S2 a;
const S2 b;      /* W języku C dozwolone jest używanie niezainicjowanych stałych */

S1* p = (S1*)&a; /* Reinterpretacja wzorca bitowego: reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b; /* Rzutowanie stałej: const_cast<S2*>(&b) */
S1* r = (S1*)&b; /* Pozbycie się stałości i zmiana typu — pewnie błąd */

```

Mamy obawy, czy zalecać stosowanie makr (podrozdział 27.8) nawet w języku C, ale mogą być przydatne w wyrażaniu takich rzeczy, jak poniżej:

```

#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);

```

Nie jest w takim przypadku zapewnione sprawdzanie typów, jak przy użyciu rzutowań `reinterpret_cast` czy `const_cast`, ale sprawia, że te z natury brzydkie operacje są dobrze widoczne i wyraźnie pokazują intencje programisty.

27.3.5. Konwersja typu `void*`

W języku C (przeciwieństwie do C++) typu `void*` można używać po prawej stronie przypisania lub jako inicjalizatora zmiennej dowolnego typu wskaźnikowego. Na przykład:

```

void* alloc(size_t x); /* Alokuje x bajtów */

void f (int n)
{
    int* p = alloc(n*sizeof(int)); /* Dobrze w C. Błąd w C++ */
    /*... */
}

```


W tym kodzie wynik typu `void*` zwrócony przez funkcję `alloc()` został przekonwertowany na typ `int*`. W języku C++ wiersz ten trzeba by było napisać tak:

```

int* p = (int*)alloc(n*sizeof(int)); /* Dobrze w C i C++ */

```

Zastosowaliśmy rzutowanie w stylu języka C (punkt 27.3.4), aby było poprawne zarówno w języku C, jak i C++.

 Dlaczego niejawną konwersję typu `void*` na `T*` jest w języku C++ zabroniona? Ponieważ konwersje tego typu mogą być niebezpieczne:

```

void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;

```



```

void* q = p;
int* pp = q; /* Niebezpieczne — dozwolone w C, błąd w C++ */
*pp = -1;    /* Nadpisuje pamięć, zaczynając od miejsca &i */
}

```

Ten kod nie daje nawet pewności, które miejsce w pamięci zostanie nadpisane. Może j i część p? Może jakiś fragment używany do obsługi wywołania funkcji f() (ramka stosu funkcji f())? Bez względu na to, jakie dane są tu nadpisywane, wywołanie funkcji f() to zła wiadomość.

Należy zauważyć, że odwrotna operacja konwersji T* na void* jest w pełni bezpieczna — nie da się napisać takiego wrednego przykładu kodu dla tej sytuacji. Takie konwersje są dozwolone zarówno w C, jak i C++.

Niestety niejawne konwersje typu void* na T* w języku C są często spotykane i stanowią jeden z największych problemów związanych ze zgodnością realnego kodu w językach C i C++ (podrozdział 27.4).

27.3.6. Typ enum

W języku C można przypisać typ int do enum bez rzutowania. Na przykład:

```

enum color { red, blue, green };
int x = green;    /* Dobrze w C i C++ */
enum color col = 7; /* Dobrze w C, błąd w C++ */

```

Jedną z implikacji tego stanu rzeczy jest to, że zmienne typu wyliczeniowego w języku C można inkrementować (++) i dekrementować (--). To może być wygodne, ale wiąże się z tym pewne ryzyko:

```

enum color x = blue;
++x; /* x zamienia się w green; błąd w C++ */
++x; /* x zamienia się w 3; błąd w C++ */

```

„Wypadnięcie za brzeg” wyliczenia może, ale nie musi, być tym, co zamierzaliśmy.

Należy zauważyć, że nazwy wyliczeń, podobnie jak struktur, należą do własnej przestrzeni nazw, a więc przy ich używaniu zawsze należy stawiać przed nimi przedrostek enum:

```

color c2 = blue; /* Błąd w C: color nie jest w zasięgu. Dobrze w C++ */
enum color c3 = red; /* dobrze */

```

27.3.7. Przestrzeń nazw

W języku C nie ma przestrzeni nazw (w takim sensie jak w C++). Co w takim razie robi się, aby uniknąć konfliktów nazw w dużych programach w tym języku? Zwykle stosuje się przedrostki lub przyrostki. Na przykład:

```

/* W bs.h: */
typedef struct bs_string { /*... */ } bs_string; /* Łańcuch Bjarnego */
typedef int bs_bool ; /* Typ logiczny Bjarnego */

/* W pete.h: */
typedef char* pete_string; /* Łańcuch Pete'a */
typedef char pete_bool ; /* Typ logiczny Pete'a */

```

Technika ta jest tak popularna, że zwykle lepiej stosować dłuższe przedrostki i przyrostki niż dwuliterowe.

27.4. Pamięć wolna

W języku C nie ma operatorów `new` i `delete` do obsługi obiektów. Aby użyć pamięci wolnej, trzeba skorzystać z odpowiednich funkcji. Najważniejsze takie funkcje są zdefiniowane w „ogólnym” standardowym nagłówku `<stdlib.h>`:

```
void* malloc(size_t sz);           /* Alokuje sz bajtów */
void free(void* p);               /* Dealokuje pamięć wskazywaną przez p */
void* calloc(size_t n, size_t sz); /* Alokuje n*sz bajtów zainicjowanych wartością 0 */
void* realloc(void* p, size_t sz); /* Realokuje pamięć wskazywaną przez p
                                   na przestrzeń o rozmiarze sz */
```



W nagłówku `<stdlib.h>` znajduje się też definicja typu bez znaku `typedef size_t`.

Dlaczego funkcja `malloc()` zwraca typ `void*`? Ponieważ nie wie, jakiego typu obiekt zostanie umieszczony w zarezerwowanej przez nią pamięci. Inicjalizacja to problem programisty. Na przykład:

```
struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"jabłko", 78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair)); /* alokacja */
pp->p = "gruszka"; /* inicjalizacja */
pp->val = 42;
```

Należy zauważyć, że nie można napisać czegoś takiego:

```
*pp = {"gruszka", 42}; /* Błąd: nie C i nie C++ */
```

W języku C++ jednak można by było zdefiniować konstruktor dla typu `Pair` i napisać taki kod:

```
Pair* pp = new Pair("gruszka", 42);
```

W języku C (ale nie w C++ — punkt 27.3.4) można opuścić rzutowanie przed funkcją `malloc()`, chociaż nie zalecamy robienia tego:

```
int* p = malloc(sizeof(int)*n); /* Unikaj tego */
```

Takie opuszczanie rzutowania jest dość popularne, ponieważ jest dzięki temu trochę mniej pisania i pozwala wykryć rzadko spotykany błąd niedołączenia nagłówka `<stdlib.h>` przed użyciem funkcji `malloc()`. To może jednak oznaczać pozbycie się widocznej wskazówki, że źle obliczono rozmiar:

```
p = malloc(sizeof(char)*m); /* Pewnie błąd — nie alokuje miejsca dla m liczb typu int */
```



Nie należy używać funkcji `malloc()` i `free()` w programach w języku C++. Operatory `new` i `delete` nie wymagają rzutowania, radzą sobie z inicjalizowaniem (konstruktorami) i czyszczeniem (destruktorami), zgłaszają błędy alokacji pamięci (jako wyjątki) oraz są tak samo szybkie. Nie należy usuwać za pomocą operatora `delete` obiektów alokowanych przez funkcję `malloc()` ani zwalniać za pomocą funkcji `free()` obiektów alokowanych przez operator `new`. Na przykład:


```
int* p = new int[200];
// ...
free(p); // błąd

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q; // błąd
```

Ten kod może działać, ale jest nieprzenośny. Poza tym, jeśli obiekty mają konstruktory lub destruktory, mieszanie sposobów zarządzania pamięcią w stylu C i C++ jest proszeniem się o kłopoty.

Do rozszerzania buforów zwykle używa się funkcji `realloc()`:

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) { /* Czytaj: ignoruj znaki w wierszu końca pliku */
    if (count==max-1) {      /* Trzeba rozszerzyć bufor */
        max += max;         /* Podwaja rozmiar bufora */
        p = (char*)realloc(p,max);
        if (p==0) quit();
    }
    p[count++] = c;
}
```

Objaśnienie operacji wejściowych języka C znajduje się w punktach 27.6.2 i B.11.2.

Funkcja `realloc()` może, ale nie musi, przenieść stare alokowane dane do nowego obszaru alokacji. Nawet nie myśl o użyciu jej na pamięci alokowanej przez operator `new`.

Mniej więcej równoważny powyższemu kod napisany przy użyciu biblioteki standardowej C++ wygląda następująco:

```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

Bardziej szczegółowy opis operacji wejściowych i technik alokacji znajduje się w artykule *Learning Standard C++ as a New Language* (zobacz źródła do podrozdziału 27.1).

27.5. Łańcuchy w stylu języka C

W języku C łańcuch (w tekstach na temat języka C++ często nazywany łańcuchem C lub łańcuchem w stylu języka C) jest zakończoną zerem tablicą znaków. Na przykład:

```
char* p = "asdf";
char s[ ] = "asdf";
```

p:

9

 →

'a'	's'	'd'	'f'	0
-----	-----	-----	-----	---

s:

'a'	's'	'd'	'f'	0
-----	-----	-----	-----	---

W języku C nie można w strukturach pisać funkcji składowych, przeciążać funkcji ani definiować operatorów (np. ==). Z tego powodu potrzebujemy zestawu (nie składowych) funkcji do przetwarzania łańcuchów w stylu C. Funkcje takie w językach C i C++ znajdują się w nagłówku `<string.h>`:

```
size_t strlen(const char* s);           /* Liczy znaki */
char* strcat(char* s1, const char* s2); /* Kopiuje s2 na koniec s1 */
int strcmp(const char* s1, const char* s2); /* Porównanie leksykograficzne */
char* strcpy(char* s1, const char* s2); /* Kopiuje s2 do s1 */

char* strchr(const char *s, int c);      /* Znajduje c w s */
char* strstr(const char *s1, const char *s2); /* Znajduje s2 w s1 */

char* strncpy(char*, const char*, size_t n); /* strcpy, maks. n znaków */
char* strncat(char*, const char, size_t n); /* strcat z maks. n znaków */
int strncmp(const char*, const char*, size_t n); /* strcmp z maks. n znaków */
```

Nie jest to pełna lista, ale te funkcje są najbardziej przydatne. Pokażemy krótkie przykłady ich użycia.



Łańcuchy można porównywać. Operator równości (==) porównuje wartości wskaźnikowe. Standardowa funkcja `strcmp()` porównuje łańcuchy w stylu C:

```
const char* s1 = "asdf";
const char* s2 = "asdf";

if (s1==s2) { /* Czy s1 i s2 wskazują tę samą tablicę? */
    /* (zwykle nie o to chodzi) */
}

if (strcmp(s1,s2)==0) { /* Czy s1 i s2 przechowują takie same znaki? */
}
```

Funkcja `strcmp()` porównuje swoje dwa argumenty trytorowo. Biorąc powyższe wartości `s1` i `s2`, `strcmp(s1,s2)` zwróci wartość 0, jeśli są one identyczne. Jeśli `s1` jest w porządku leksykograficznym przed `s2`, funkcja ta zwróci wartość ujemną, a jeśli jest odwrotnie — zwróci wartość dodatnią. **Porządek leksykograficzny** (ang. *lexicographical order*) oznacza mniej więcej „tak jak w słowniku”. Na przykład:

```
strcmp("pies", "pies")==0
strcmp("małpa", "dodo")>0 /* „małpa” występuje w słowniku za „dodo” */
strcmp("krowa", "świnia")<0 /* „świnia” występuje w słowniku za słowem „krowa” */
```

Nie ma gwarancji, że wynikiem porównywania wskaźników `s1==s2` będzie 0 (false). W implementacji może zostać podjęta decyzja o użyciu tej samej pamięci do przechowywania wszystkich kopii literału znakowego i wówczas otrzymalibyśmy wynik 1 (true). Zwykle funkcja `strcmp()` jest dobrym wyborem do porównywania łańcuchów w stylu języka C.

Długość łańcucha w stylu języka C można sprawdzić następująco:

```
int lgt = strlen(s1);
```


Należy pamiętać, że funkcja `strlen()` nie wlicza do liczby znaków końcowego zera. W tym przypadku `strlen(s1)==4`, a na zapisanie łańcucha "asdf" potrzeba pięciu bajtów. Ta mała różnica jest często przyczyną pomyłki o jeden.

Łańcuch w stylu C można skopiować do innego łańcucha (razem z końcowym zerem):

```
strcpy(s1,s2); /* Kopiuje znaki z s2 do s1 */
```

Zapewnienie odpowiedniej ilości miejsca w docelowym łańcuchu (tablicy) należy do programisty.

Funkcje `strncpy()`, `strncat()` oraz `strncmp()` to wersje funkcji `strcpy()`, `strcat()` oraz `strcmp()`, które przyjmują trzeci argument `n` określający maksymalną liczbę znaków. Należy zauważyć, że jeśli w łańcuchu źródłowym jest więcej niż `n` znaków, funkcja `strncpy()` nie skopiuje końcowego zera, przez co wynikiem nie będzie poprawny łańcuch w stylu języka C.

Funkcje `strchr()` i `strstr()` znajdują wartość swojego drugiego argumentu w łańcuchu przekazanym im jako pierwszy argument i zwracają wskaźnik na pierwszy znak dopasowanego ciągu. Podobnie jak `find()`, funkcje te przeszukują łańcuchy od lewej do prawej.

Zadziwiające, jak dużo można zrobić przy użyciu tych prostych funkcji i jak łatwo przy tym popełnić mały błąd. Rozważmy proste zadanie konkatencji nazwy użytkownika z adresem i umieszczenia między nimi znaku `@`. Jeśli użyje się typu `std::string`, można to zrobić w następujący sposób:

```
string s = id + '@' + addr;
```

Przy użyciu standardowej funkcji operującej na łańcuchach w stylu języka C można to napisać tak:

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz-1]=0;
    return res;
}
```

Dobrze to zrobiliśmy? Jak zostanie zwolniony (`free()`) łańcuch zwrócony przez funkcję `cat()`?

WYPRÓBUJ



Przetestuj funkcję `cat()`. Czemu? Zostawiliśmy w niej błąd odbijający się na wydajności na poziomie początkującego ucznia. Znajdź go i usuń. „Zapomnieliśmy” napisać komentarzy do kodu. Napisz komentarze odpowiednie dla kogoś, kto zna standardowe funkcje operujące na łańcuchach w stylu języka C.

27.5.1. Łańcuchy w stylu języka C i const

Rozważmy:

```
char* p = "asdf";
p[2] = 'x';
```



Jest to poprawny kod C, ale nie C++. W języku C++ literal łańcuchowy jest stałą, a więc nie podlegającą modyfikacji wartością. Dlatego instrukcja `p[2]='x'` (mająca zmienić wskazywaną wartość na "asxf") jest błędem. Niestety niewiele kompilatorów wykryje to problematyczne przypisanie do `p`. Jeśli masz szczęście, wystąpi błąd czasu działania, ale lepiej na to nie liczyć. Lepiej zamiast tego napisać:

```
const char* p = "asdf"; // Teraz nie można zapisywać w "asdf" poprzez p
```

To zalecenie dotyczy zarówno C, jak i C++.

Funkcja języka C `strchr()` ma podobną, ale trudniejszą do wykrycia wadę. Rozważmy:



```
char* strchr(const char* s, int c); /* Znajduje c w stałej s (nie C++) */
```

```
const char aa[] = "asdf";          /* aa jest tablicą stałych */
char* q = strchr(aa, 'd');          /* Znajduje 'd' */
*q = 'x';                          /* Zmienia 'd' w aa na 'x' */
```

To jest niedozwolone zarówno w C, jak i C++, ale kompilatory języka C tego nie znajdują. Czasami nazywa się takie coś **transmutacją** — zamiana stałej na niestałą, co jest pogwałceniem rozsądnych założeń dokonanych w kodzie.

W języku C++ problem ten został rozwiązany poprzez zmodyfikowanie deklaracji funkcji `strchr()`:

```
char const* strchr(const char* s, int c); // Znajduje c w stałej s
char* strchr(char* s, int c);             // Znajduje c w s
```

To samo dotyczy funkcji `strstr()`.

27.5.2. Operacje na bajtach

W zamierzonych czasach (wczesnych latach 80.), przed wynalezieniem typu `void*`, programiści języka C (i C++) do manipulowania bajtami używali operacji łańcuchowych. Teraz podstawowe standardowe funkcje do manipulowania bajtami przyjmują parametry typu `void*` i zwracają typy ostrzegające użytkowników o bezpośredniej manipulacji pamięcią, która zasadniczo nie ma żadnego typu:

```
/* Kopiuje n bajtów z s2 do s1 (jak strcpy()): */
void* memcpy(void* s1, const void* s2, size_t n);
```

```
/* Kopiuje n bajtów z s2 do s1, ([s1,s1+n) może się częściowo pokrywać z [s2,s2+n)): */
void* memmove(void* s1, const void* s2, size_t n);
```

```
/* Porównuje n bajtów z s2 z bajtami z s1 (jak strcmp()): */
int memcmp(const void* s1, const void* s2, size_t n);
```



```
/* Znajduje c (po konwersji na typ char bez znaku) w pierwszych n bajtach s: */
void* memchr(const void* s, int c, size_t n);
```

```
/* Kopiuje c (po konwersji na typ char bez znaku)  
do każdego z n pierwszych bajtów, na które wskazuje s: */  
void* memset(void* s, int c, size_t n);
```

Nie używaj tych funkcji w języku C++. Zwłaszcza funkcja `memset()` zwykle koliduje z gwarancjami oferowanymi przez konstruktory.

27.5.3. Przykład — funkcja `strcpy()`

Definicja funkcji `strcpy()` jest zarazem sławnym i niesławnym przykładem tego, jak lapidarny może być kod w języku C (i C++ też):

```
char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}
```

Pozostawiamy Czytelnikowi do samodzielnego rozszyfrowania, czemu powyższy kod rzeczywiście kopiuje łańcuch w stylu C q do p. Postinkrementacja jest opisana w punkcie A.5 — wartością `p++` jest wartość p przed inkrementacją.

WYPRÓBUJ

Czy ta implementacja funkcji `strcpy()` jest poprawna? Wyjaśnij swoją odpowiedź.

Jeśli nie potrafisz tego wyjaśnić, nie jesteś według naszych standardów programistą języka C (bez względu na Twoje kompetencje w programowaniu w innych językach). Każdy język ma swoje kruczki, a to jest jeden z kruczków języka C.

27.5.4. Kwestia stylu

Niepostrzeżenie zajęliśmy stanowisko w ciągnącej się od lat, często ostrej i w dużym stopniu niepotrzebnej dyskusji na temat stylu. Wskaźniki deklarujemy następująco:

```
char* p; // p jest wskaźnikiem na obiekt typu char
```

Nie robi się tego tak:

```
char *p; /* p to coś, z czego można wyluskać znak */
```

Dla kompilatora nie ma znaczenia w którym miejscu znajduje się spacja, ale dla programistów tak. Nasz styl (powszechny w języku C++) podkreśla typ deklarowanej zmiennej, podczas gdy ten drugi styl (bardziej rozpowszechniony w języku C) podkreśla użycie tej zmiennej. Przypominamy, że nie zalecamy deklarowania wielu zmiennych za pomocą jednej deklaracji:

```
char c, *p, a[177], *f(); /* Dozwolone, ale mylące */
```


Takie deklaracje w starszym kodzie nie są rzadkością. Lepiej rozbić to na kilka wierszy i dodatkową przestrzeń wykorzystać na napisanie komentarzy i inicjalizatorów:

```
char c = 'a'; /* Znak oznaczający koniec danych wejściowych pobieranych przez funkcję f() */
char* p = 0; /* Ostatni znak wczytany przez funkcję f() */
char a[177]; /* bufor wejściowy */
char* f(); /* Wczytuje do bufora a; zwraca wskaźnik na pierwszy wczytany znak */
```

Należy także dobierać coś znaczące nazwy.

27.6. Wejście i wyjście — nagłówek stdio

W języku C nie ma strumieni iostream. Z tego powodu należy używać standardowych wejść i wyjść zdefiniowanych w nagłówku <stdio.h>, który zwykle nazywa się stdio. Odpowiedniki strumieni cin i cout w tym nagłówku to stdin i stdout. W jednym programie można używać zarówno nagłówka stdio, jak i strumieni iostream (dla tych samych strumieni wejścia i wyjścia), ale nie zalecamy robienia tego. Jeśli musisz to zrobić, poczytaj na temat stdio i strumieni iostream (zwłaszcza ios_base::sync_with_stdio()) w podręczniku dla zaawansowanych. Zobacz też punkt B.11.

27.6.1. Wyjście

Najpopularniejszą i najbardziej przydatną funkcją nagłówka stdio jest printf(). Najbardziej podstawowym zastosowaniem tej funkcji jest drukowanie łańcuchów w stylu języka C:

```
#include<stdio.h>

void f(const char* p)
{
    printf("Witaj, świecie!\n");
    printf(p);
}
```

To nie jest zbyt interesujące. Ciekawe jest to, że funkcja printf() może przyjąć dowolną liczbę argumentów, a pierwszy z nich zdecyduje o tym, czy i jak pozostałe argumenty zostaną wydrukowane. Deklaracja funkcji printf() w języku C jest następująca:

```
int printf(const char* format, ... );
```

Trzykropek oznacza „dowolną liczbę opcjonalnych argumentów”. Funkcję printf() można wywołać następująco:

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

Tutaj ciąg %g oznacza: „Drukuj liczbę zmiennoprzecinkową w formacie ogólnym”, %s oznacza: „Wydrukuj łańcuch w stylu języka C”, %d oznacza: „Wydrukuj liczbę całkowitą cyframi dziesiętnymi”, a %c oznacza: „Wydrukuj znak”. Każdy z tych specyfikatorów formatu modyfikuje

następny jeszcze nieużywany argument, a więc %g drukuje d, %s drukuje s, %d drukuje i, a %c drukuje ch. Pełna lista formatów funkcji printf() została zamieszczona w punkcie B.11.2.



Niestety funkcja printf() nie jest bezpieczna ze względu na typy:

```
char a[] = { 'a', 'b' };          /* Brak końcowego zera */
void f2(char* s, int i)
{
    printf("goof %s\n", i);        /* Nieprzechwycony błąd */
    printf("goof %d: %s\n", i);    /* Nieprzechwycony błąd */
    printf("goof %s\n", a);        /* Nieprzechwycony błąd */
}
```

Ciekawy jest wynik działania ostatniego z powyższych wywołań funkcji printf() — wydrukuje wszystkie bajty z pamięci znajdujące się za a[1] do napotkania pierwszego 0. To może być mnóstwo znaków.

To niebezpieczeństwo ze względu na typy jest powodem, dla którego wolimy strumienie iostream niż bibliotekę stdio, mimo że ta druga działa identycznie zarówno w C, jak i C++. Drugi powód to brak możliwości rozszerzania funkcji stdio — nie można rozszerzyć funkcji printf(), aby drukowała wartości zdefiniowanych przez programistę typów. Nie da się na przykład zdefiniować własnego formatu %Y, aby drukował jakąś strukturę Y.

Istnieje przydatna wersja funkcji printf(), która pobiera deskryptor pliku jako pierwszy argument:

```
int fprintf(FILE* stream, const char* format, ... );
```

Na przykład:

```
fprintf(stdout, "Witaj, świecie!\n"); // Dokładnie jak printf("Witaj, świecie!\n");
FILE* ff = fopen("My_file", "w");    // Otwiera plik My_file do zapisu
fprintf(ff, "Witaj, świecie!\n");    // Zapisuje "Witaj, Świecie!\n" w pliku My_file
```

Opis uchwytów do plików znajduje się w punkcie 27.6.3.

27.6.2. Wejście

Do najpopularniejszych funkcji biblioteki stdio należą:

```
int scanf(const char* format, ... ); /* Odczytuje dane z stdin przy użyciu odpowiedniego
formatu */
int getchar(void);                  /* Pobiera znak ze stdin */
int getc(FILE* stream);             /* Pobiera znak ze strumienia */
char* gets(char* s);                /* Pobiera znaki ze stdin */
```

Najprostszym sposobem na wczytanie łańcucha znaków jest użycie funkcji gets(). Na przykład:

```
char a[12];
gets(a); /* Wczytuje dane do tablicy znaków wskazywanej przez a aż do pojawienia się znaku '\n' */
```

Nigdy tego nie rób! Wyobraź sobie, że funkcja gets() została zatruta. Funkcja ta razem ze swoją bliską krewną funkcją scanf("%s") jest powodem około jednej czwartej wszystkich udanych ataków hakerów. Nadal stanowi poważne zagrożenie. Skąd w powyższym banalnym kodzie



wiadomo, czy przed znakiem nowego wiersza zostanie wprowadzonych dokładnie 11 innych znaków? Nie wiadomo. Dlatego funkcja `gets()` prawie na pewno spowoduje uszkodzenia pamięci (bajtów znajdujących się za buforem), a uszkodzanie pamięci to jedno z najważniejszych narzędzi włamywaczy komputerowych. Nie myśl, że potrafisz zgadnąć maksymalny rozmiar bufora, który wystarczy wszystkim użytkownikom. Może po drugiej stronie znajduje się nie człowiek, a maszyna, która nie odpowiada Twoim kryteriom dotyczącym rozsądnosci.

Funkcja `scanf()` wczytuje dane przy użyciu określonego formatu w taki sam sposób, jak funkcja `printf()` je drukuje. Podobnie jak `printf()` może być też bardzo wygodna w użyciu:

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* Wczytuje dane do zmiennych przekazywanych jako wskaźniki: */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s pomija znajdujący się na początku biały znak i kończy się na białym znaku */
}
```



Funkcja `scanf()`, podobnie jak `printf()`, nie jest bezpieczna ze względu na typy. Znaki formatu i argumenty (wszystkie wskaźniki) muszą sobie dokładnie odpowiadać albo zaczną dziać się dziwne rzeczy w czasie działania programu. Należy też zauważyć, że wczytanie `%s` do `s` może doprowadzić do przepełnienia. Nigdy nie używaj funkcji `gets()` ani `scanf("%s")`!



Jak w takim razie bezpiecznie wczytać znaki? Można wykorzystać pewien rodzaj `%s` ograniczający liczbę możliwych znaków do wczytania. Na przykład:

```
char buf[20];
scanf("%19s", buf);
```

Potrzebne jest miejsce dla końcowego zera (dostarczanego przez funkcję `scanf()`), a więc maksymalną liczbę znaków do wczytania do `buf` ustawiliśmy na 19. Nie rozwiązuje to jednak problemu sytuacji, w której ktoś wpisze więcej niż 19 znaków. Wszystkie „niepotrzebne” znaki zostaną w strumieniu wejściowym, aby znalazły je następne operacje wejściowe.

Wady funkcji `scanf()` sprawiają, że często lepiej i łatwiej jest użyć funkcji `getchar()`. Poniżej znajduje się przykład typowego kodu do wczytywania znaków za pomocą funkcji `getchar()`:

```
while((x=getchar())!=EOF) {
    /* ... */
}
```

`EOF` to znajdujące się w bibliotece `stdio` makro oznaczające koniec pliku — zobacz też podrzdział 27.4.

Odpowiedniki funkcji `scanf("%s")` i `gets()` w bibliotece standardowej C++ nie mają tych wad:

```
string s;
cin >> s;           // Wczytuje słowo
getline(cin,s);      // Wczytuje wiersz
```


27.6.3. Pliki

W języku C (i C++) do otwierania plików można używać funkcji `fopen()`, a do zamykania funkcji `fclose()`. Funkcje te wraz z reprezentacją uchwytu do pliku, `FILE`, oraz makrem `EOF` (koniec pliku) znajdują się w nagłówku `<stdio.h>`:

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

Oto podstawowy przykład kodu wykorzystującego pliki:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r"); /* Otwiera fn do odczytu */
    FILE* fo = fopen(fn2, "w"); /* Otwiera fn2 do zapisu */

    if (fi == 0) error("Nie udało się otworzyć pliku wejściowego.");
    if (fo == 0) error("Nie udało się otworzyć pliku wyjściowego.");

    /* Wczytuje dane z pliku przy użyciu funkcji wejściowych biblioteki stdio, np. getc() */
    /* Zapisuje dane w pliku przy użyciu funkcji wyjściowych biblioteki stdio, np. fprintf() */

    fclose(fo);
    fclose(fi);
}
```

Pomyśl: skoro w języku C nie ma wyjątków, to jak sprawić, że pliki zostaną zamknięte bez względu na to, jaki rodzaj błędu wystąpi?

27.7. Stałe i makra

W języku C obiekt `const` nigdy nie jest stały w czasie kompilacji:

```
const int max = 30;
const int x; /* Niezainicjowana stała. Dobrze w C (błąd w C++) */

void f(int v)
{
    int a1[max]; /* Błąd: granica tablicy nie jest stałą (dobrze w C++) */
                    /* (max nie może pojawiać się w stałych wyrażeniach!) */
    int a2[x]; /* Błąd: granica tablicy nie jest stałą */

    switch (v) {
    case 1:
        /* ... */
        break;
    case max: /* Błąd: etykieta case nie jest stałą (dobrze w C++) */
        /* ... */
        break;
    }
}
```



W języku C (ale nie w C++) techniczny powód tego jest taki, że obiekty `const` są dostępne z poziomu innych jednostek translacji:

```
/* Plik x.c: */
const int x; /* inicjalizacja w innym miejscu */


/* Plik xx.c: */
const int x = 7; /* Tutaj znajduje się prawdziwa definicja */
```

W języku C++ byłyby to dwa różne obiekty o nazwie `x` w dwóch różnych plikach. Programiści języka C zamiast słowa kluczowego `const` do reprezentowania symbolicznych stałych często wykorzystują makra. Na przykład:

```
#define MAX 30
void f(int v)
{
    int a1[MAX]; /* dobrze */
    switch (v) {
    case 1:
        /* ... */
        break;
    case MAX: /* dobrze */
        /* ... */
        break;
    }
}
```

 Nazwa makra `MAX` zostanie zastąpiona znakami `30`, które stanowią jego wartość. To znaczy, że liczba elementów w `a1` wynosi `30` i wartość w drugiej etykietce `case` również wynosi `30`. Zgodnie z konwencją nazwę makra zapisaliśmy wielkimi literami. Zwyczaj ten pozwala zminimalizować liczbę błędów powodowanych przez makra.

27.8. Makra

 Uważaj na makra — w języku C nie ma efektywnych sposobów na ich unikanie, ale należy pamiętać, że stosowanie makr bez poszanowania typowych zasad dotyczących zakresu i typów w języku C (i C++) powoduje poważne skutki uboczne. Makra są rodzajem techniki podmieniania tekstu. Zobacz też punkt A.17.2.

Jak można chronić się przed potencjalnymi skutkami wykorzystania makr, poza unikaniem ich (i używaniem czegoś innego w języku C++)?



- Nazwy wszystkich makr pisz WIELKIMI_LITERAMI.
- Nie pisz WIELKIMI_LITERAMI innych nazw niż makr.
- Nigdy nie nadawaj makrom krótkich lub „milutkich” nazw typu `max` czy `min`.
- Miej nadzieję, że wszyscy inni stosują się do tych prostych i powszechnych zasad.

Oto lista najważniejszych zastosowań makr:

- definiowanie „stałych”,
- definiowanie konstrukcji podobnych do funkcji,

- „poprawianie” składni,
- sterowanie kompilacją warunkową.

Poza wymienionymi jest jeszcze wiele innych, mniej popularnych zastosowań.

Makra są naszym zdaniem mocno nadużywane, ale w języku C nie ma dobrego zastępstwa dla nich. Czasami nawet trudno ich uniknąć w programach w C++ (zwłaszcza jeśli pisze się programy, które muszą odpowiadać bardzo starym kompilatorom lub działać na platformach o bardzo nietypowych ograniczeniach).

Należą się przeprosiny osobom, które uważają opisane poniżej techniki za „paskudne sztuczki” i woleliby, aby o nich nie wspominać w kulturalnym towarzystwie. Jednak naszym zdaniem programowanie to czynność, którą wykonuje się w rzeczywistym świecie, w którym te (bardzo delikatne) przykłady użycia i nieprawidłowego zastosowania makr mogą pozwolić zaoszczędzić początkującemu programiście wiele godzin nerwów. Ignorancja w dziedzinie makr nikomu nie wyjdzie na dobre.

27.8.1. Makra podobne do funkcji

Poniżej znajduje się przykład typowego makra podobnego do funkcji:

```
#define MAX(x, y) ((x)>=(y)?(x):(y))
```

Napisałiśmy nazwę MAX wielkimi literami, aby odróżnić ją od wielu funkcji o nazwie max (stosowanych w różnych programach). Oczywiście kod ten bardzo różni się od funkcji — nie ma określonych typów argumentów, bloku, instrukcji return itd. Po co są te wszystkie nawiasy? Rozważmy:


```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```


To można rozwinąć do następującej postaci:

```
int aa = ((1)>=(2)?(1):(2));
double dd = ((aa++)>=(2)?(aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

Gdyby nie było tych wszystkich nawiasów, ostatnie rozwinięcie wyglądałoby tak:

```
char cc = dd>=aa?dd:aa+2;
```

Zmienna cc mogłaby z łatwością otrzymać inną wartość, niż można by się było spodziewać, patrząc na jej definicję. Pamiętaj, aby zawsze przy definiowaniu makra każdy argument użyty jako wyrażenie umieszczać w nawiasach. 

Z drugiej strony nawet wszystkie nawiasy tego świata nie uratowałyby drugiego rozwinięcia. Parametr makra x otrzymał wartość aa++, a ponieważ x jest w MAX użyty dwa razy, wartość a może zostać zwiększona dwukrotnie. Nie przekazuj do makr argumentów z efektami ubocznymi. 

Jak to czasem bywa, znalazł się geniusz, który zdefiniował makro w taki sposób i umieścił je w popularnym pliku nagłówkowym. Niestety na domiar złego nazwał je max zamiast MAX, przez co gdy standardowy nagłówek C++ zawiera taką definicję:

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```


max zostaje rozwinięte z argumentami T a i T b i kompilator widzi to:

```
template<class T> inline T ((T a)>=(T b)?(T a):(T b)) { return a<b?b:a; }
```

Komunikaty o błędach zgłaszane przez kompilator są „ciekawe” i niezbyt pomocne. W nagłej sytuacji można „oddefiniować” makro:

```
#undef max
```

Na szczęście makro to nie było bardzo ważne. Jednak w popularnych plikach nagłówkowych znajdują się dziesiątki tysięcy makr. Nie można ich wszystkich oddefiniować, nie powodując chaosu.

Nie wszystkie parametry makr są używane jako wyrażenia. Rozważmy:

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

Ten realny przykład może być bardzo przydatny w unikaniu błędów wynikających z niedopasowania planowanego typu alokacji do typu użytego jako argument funkcji sizeof:

```
double* p = malloc(sizeof(int)*10); /* prawdopodobnie błąd */
```

Niestety nie jest łatwo napisać makro wykrywające wyczerpanie pamięci. Poniższy kod może to robić, pod warunkiem że odpowiednio zdefiniuje się gdzieś indziej zmienną error_var i funkcję error():

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
    (error_var==0)\
    ?(error("Błąd alokacji pamięci."),0)\
    :error_var)
```

Znaki \ na końcach wierszy nie są objawem problemów z czcionką. W ten sposób dzieli się makra na kilka wierszy. W języku C++ wolimy używać operatora new.

27.8.2. Makra składniowe

Można definiować takie makra, które sprawiają, że kod źródłowy wygląda bardziej odpowiednio dla programisty. Na przykład:

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```



Zdecydowanie nie zalecamy robienia tego. **Wielu** programistów już próbowało wprowadzić ten pomysł w życie. Wszyscy oni (lub osoby zajmujące się utrzymaniem ich kodu) doszli do wniosku, że:

- Wielu ludzi nie podziela ich poglądów na temat tego, jak wygląda lepsza składnia.
- „Ulepszona” składnia jest niestandardowa i zaskakująca, co myli innych.
- Niektóre zastosowania „ulepszonej” składni powodują niejasne błędy kompilacji.
- Kompilator nie robi tego, co widać, i zgłasza błędy w znanym sobie stylu (i widzianym w kodzie), a nie w stylu programisty.

Nie należy pisać makr składniowych, aby tylko „polepszyć” wygląd kodu. Dla Ciebie i kilku Twoich przyjaciół kod ten może wyglądać bardzo ładnie, ale z doświadczenia wiadomo, że będziecie stanowić tylko małą grupkę w większej społeczności i w końcu ktoś będzie musiał ten kod napisać jeszcze raz (zakładając, że ten kod przetrwa).

27.8.3. Kompilacja warunkowa

Wyobraź sobie, że masz dwie wersje pliku nagłówkowego, np. dla Linuksa i systemu Windows. Jak sprawić, aby został wybrany ten, co trzeba? Oto typowy sposób:

```
#ifdef WINDOWS
    #include "my_windows_header.h"
#else
    #include "my_linux_header.h"
#endif
```

Jeśli ktoś zdefiniuje `WINDOWS`, zanim kompilator to znajdzie, wynik będzie następujący:

```
#include "my_windows_header.h"
```

W przeciwnym przypadku będzie to:

```
#include "my_linux_header.h"
```

Dla instrukcji `#ifdef WINDOWS` nie ma znaczenia, czym według definicji jest `WINDOWS`. Sprawdza ona tylko, czy definicja ta w ogóle jest.

Większość systemów (włącznie z wszystkimi rodzajami systemów operacyjnych) ma zdefiniowane makra do sprawdzania. Test sprawdzający, czy kod jest kompilowany jako język C++ czy C, wygląda następująco:

```
#ifdef __cplusplus
    // w C++
#else
    /* w C */
#endif
```

Istnieje podobna instrukcja zwana strażnikiem dołączania (ang. *include guard*), którą zwykle wykorzystuje się do zapobieżenia dwukrotnemu dołączeniu tego samego pliku:

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* Informacje nagłówka */
#endif
```

Instrukcja `#ifndef` sprawdza, czy coś nie zostało zdefiniowane, tj. `#ifndef` jest przeciwieństwem `#ifdef`. Pod względem logicznym makra stosowane do kontrolowania plików źródłowych znacznie się różnią od makr stosowanych do modyfikowania kodu. Przypadkiem tylko wykorzystują ten sam mechanizm.

27.9. Przykład — kontenery intruzyjne

Kontenery z biblioteki standardowej C++ są tzw. kontenerami nieintruzyjnymi (ang. *non-intrusive container*). Tzn. nie jest wymagane, aby w typach używanych jako ich elementy były jakiekolwiek dane. Dzięki temu właśnie można je uogólniać, aby współpracowały ze wszystkimi typami (wbudowanymi i zdefiniowanymi przez użytkownika), pod warunkiem że typy te dadzą się kopiować. Istnieją też **kontenery intruzyjne** (ang. *intrusive container*), które są popularne zarówno w C, jak i C++. Przedstawimy na przykładzie nieintruzyjnej listy zastosowanie struktur, wskaźników i pamięci wolnej w języku C.

Zdefiniujemy listę dwukierunkową z dziewięcioma operacjami:

```
void init(struct List* lst);    /* Inicjalizuje lst jako pustą listę */
struct List* create();         /* Tworzy nową pustą listę w pamięci wolnej */
void clear(struct List* lst);   /* Zwalnia wszystkie elementy listy */
void destroy(struct List* lst); /* Zwalnia wszystkie elementy listy, a na końcu samą listę */

void push_back(struct List* lst, struct Link* p); /* Dodaje p na końcu listy */
void push_front(struct List*, struct Link* p);    /* Dodaje p na początku listy */

/* Wstawia q przed p: */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p); /* Usuwa p z listy */

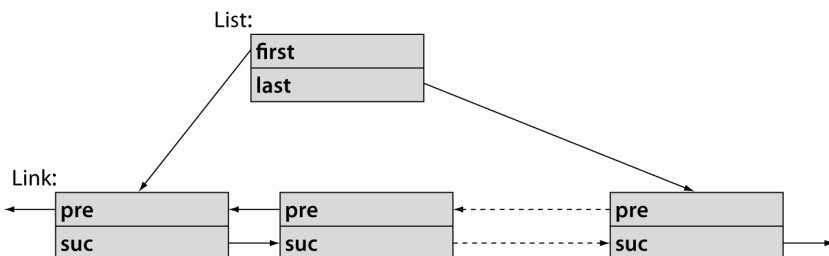
/* Zwraca ogniwo w odległości n przed lub za p: */
struct Link* advance(struct Link* p, int n);
```

Chodzi o to, by tak zdefiniować te operacje, aby ich użytkownicy musieli tylko używać typów `List*` i `Link*`. Dzięki temu można by było radykalnie zmienić implementację tych funkcji, nie wpływając w żaden sposób na tych użytkowników. Oczywiście styl nazw jest inspirowany biblioteką STL. Definicje struktur `List` i `Link` mogą być oczywiście banalne:

```
struct List {
    struct Link* first;
    struct Link* last;
};

struct Link { /* Ogniwo listy dwukierunkowej */
    struct Link* pre;
    struct Link* suc;
};
```

Poniżej znajduje się schematyczny rysunek listy:



Nie jest naszym celem pokazanie sprytnych technik reprezentacji ani algorytmów, a więc nic takiego nie da się tu znaleźć. Należy jednak zauważyć, że nie ma ani słowa o jakichkolwiek danych przechowywanych przez ogniwa (Link — elementy listy). Jeśli przyjrzeć się dostarczonym funkcjom, można zauważyć, że w istocie robimy coś bardzo podobnego do definiowania pary abstrakcyjnych klas Link i List. Dane dla ogniwa zostaną dostarczone później. Link* i List* czasami nazywa się uchwytami do typów nieprzejrzystych (ang. *opaque type*). Innymi słowy, przekazując funkcjom wskaźniki List* i Link*, można manipulować elementami listy, nie wiedząc nic na temat wewnętrznej struktury tej listy lub jej ogniwa.

Implementację naszych funkcji listy zaczniemy od dołączenia kilku nagłówków z biblioteki standardowej:

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

W języku C nie ma przestrzeni nazw, a więc można bez obaw stosować deklaracje i dyrektywy using. Z drugiej strony musimy wziąć pod uwagę, że wykorzystaliśmy bardzo często spotykane krótkie nazwy (Link, insert, init itp.), przez co tego zestawu funkcji w takim stanie, jak jest, nie można będzie użyć poza prostym programem).

Inicjacja jest banalna, chociaż warto zwrócić uwagę na użycie funkcji assert():

```
void init(struct List* lst) /* Inicjalizuje *lst jako pustą listę */
{
    assert(lst);
    lst->first = lst->last = 0;
}
```

Postanowiliśmy nie dodawać obsługi błędów czasu działania programu nieprawidłowych wskaźników na listy. Dzięki użyciu funkcji assert() zgłaszany będzie po prostu systemowy błąd czasu działania, jeśli wskaźnik na listę będzie pusty. W komunikacie o tym błędzie będzie zawarta nazwa pliku oraz wiersz, w którym znajduje się nieudana asercja. Funkcja assert() to makro zdefiniowane w nagłówku <assert.h>. Sprawdzanie jest włączone tylko w czasie debugowania. Przy braku wyjątków nie jest łatwo zdecydować się, co robić z nieprawidłowymi wskaźnikami.

Funkcja create() tworzy listę w pamięci wolnej. Jest to coś w rodzaju połączenia konstruktora (funkcja init() inicjalizuje) z operatorem new (funkcja malloc() alokuje pamięć):

```
struct List* create() /* Tworzy nową pustą listę */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

W funkcji clear() przyjęto założenie, że wszystkie ogniwa zostały utworzone w pamięci wolnej. Funkcja ta zwalnia je za pomocą funkcji free():

```
void clear(struct List* lst) /* Zwalnia wszystkie elementy listy lst */
{
    assert(lst);
    {
        struct Link* curr = lst->first;
```



```

        while(curr) {
            struct Link* next = curr->suc;
            free(curr);
            curr = next;
        }
        lst->first = lst->last = 0;
    }
}

```

Warto zwrócić uwagę na sposób przemierzania listy za pomocą składowej `suc` obiektu `Link`. Nie można uzyskać bezpiecznego dostępu do obiektu struktury po tym, jak obiekt ten został zwolniony przez funkcję `free()`. Dlatego wprowadziliśmy zmienną `next` do przechowywania informacji o naszym położeniu w liście podczas zwalniania ogniwa. Jeśli nie wszystkie ogniwa zostały alokowane w pamięci wolnej, lepiej nie wywoływać funkcji `clear()`, która w takim przypadku może narobić spustoszenia.

Funkcja `destroy()` to w zasadzie przeciwieństwo funkcji `create()`, tzn. jest to coś w rodzaju połączenia destruktor z operatorem `delete`:

```

void destroy(struct List* lst) /*Zwalnia wszystkie elementy listy, a na końcu samą listę */
{
    assert(lst);
    clear(lst);
    free(lst);
}

```

Należy zauważyć, że nie zapewniliśmy żadnych zabezpieczeń dla wywołania funkcji czyszczącej (destruktor) elementy reprezentowane przez obiekty typu `Link`. Projekt ten nie jest wierną imitacją technik czy ogólności języka C++ — nie mógłby i prawdopodobnie nie powinien nią być.

Kod funkcji `push_back()` (dodającej nowy obiekt typu `Link` na końcu listy) jest prosty:

```

void push_back(struct List* lst, struct Link* p) /*Dodaje p na końcu listy */
{
    assert(lst);
    {
        struct Link* last = lst->last;
        if (last) {
            last->suc = p; /*Dodaje p za ostatnim elementem */
            p->pre = last;
        }
        else {
            lst->first = p; /*p jest pierwszym elementem */
            p->pre = 0;
        }
        lst->last = p; /*p jest nowym ostatnim elementem */
        p->suc = 0;
    }
}

```

Nigdy by się nam jednak nie udało jej poprawnie napisać, gdybyśmy nie narysowali kilku prostokątów w swoim brudnopisie. Zauważ, że „zapomnieliśmy” wziąć pod uwagę przypadek, w którym argument `p` jest pusty. Wystarczy zamiast wskaźnika na obiekt typu `Link` przekazać 0

i powyższy kod marnie skończy działanie. Nie jest to nieuleczalnie zły kod, ale **nie** nadaje się do profesjonalnych zastosowań. Jego zadaniem jest ilustrować często stosowane i przydatne techniki (a w tym przypadku także powszechnie występującą słabość czy błąd).

Kod funkcji `erase()` może wyglądać następująco:

```
struct Link* erase(struct List* lst, struct Link* p)
/*
  Usuwa p z listy;
  zwraca wskaźnik na ogniwo znajdujące się za p.
*/
{
    assert(lst);
    if (p==0) return 0; /* Dobrze dla erase(0) */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc; /* Następnik staje się pierwszy */
            p->suc->pre = 0;
            return p->suc;
        }
        else {
            lst->first = lst->last = 0; /* Lista robi się pusta */
            return 0;
        }
    }
    else if (p == lst->last) {
        if (p->pre) {
            lst->last = p->pre; /* Poprzednik staje się ostatni */
            p->pre->suc = 0;
        }
        else {
            lst->first = lst->last = 0; /* Lista robi się pusta */
            return 0;
        }
    }
    else {
        p->suc->pre = p->pre;
        p->pre->suc = p->suc;
        return p->suc;
    }
}
```

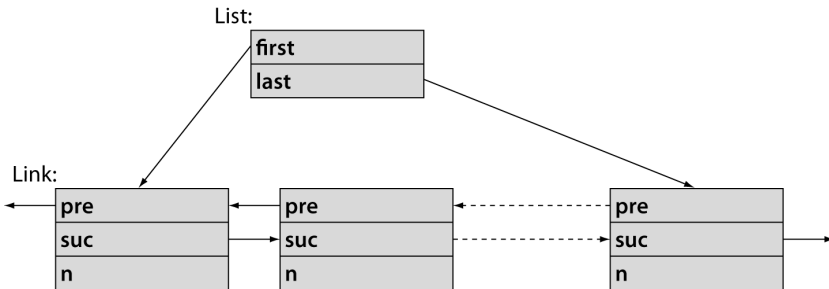
Implementację pozostałych funkcji pozostawiamy jako ćwiczenie do samodzielnego wykonania. I tak ich nie potrzebujemy w naszym (za) krótkim teście. Teraz musimy zmierzyć się z największą zagadką tego projektu — gdzie są dane elementów listy? Jak zaimplementować prostą listę nazw w postaci łańcuchów w stylu języka C. Rozważmy:

```
struct Name {
    struct Link lnk; /* Ogniwo wymagane przez operacje listy */
    char* p;        /* łańcuch nazwy */
};
```


Na razie wszystko dobrze, tylko nie wiemy, jak użyć tej składowej typu `Link`. Skoro wiadomo, że lista lubi, gdy jej ogniwa znajdują się w pamięci wolnej, napiszemy funkcję tworzącą nazwy (obiekty typu `Name`) w pamięci wolnej:

```
struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->p = n;
    return p;
}
```

W postaci schematu:



Wykorzystamy to:

```
int main()
{
    int count = 0;
    struct List names; /* Tworzy listę */
    struct Link* curr;
    init(&names);

    /* Tworzy kilka nazw i dodaje je do listy: */
    push_back(&names, (struct Link*)make_name("Nora"));
    push_back(&names, (struct Link*)make_name("Witalis"));
    push_back(&names, (struct Link*)make_name("Ryś"));

    /* Usuwa drugą nazwę (o indeksie 1): */
    erase(&names, advance(names.first, 1));
    curr = names.first; /* Drukuje wszystkie nazwy: */
    for (; curr!=0; curr=curr->suc) {
        count++;
        printf("Element %d: %s\n", count, ((struct Name*)curr)->p);
    }
}
```

I tak „naoszukiwaliśmy”. Za pomocą rzutowania zmusiliśmy program do traktowania typu `Name*` jako `Link*`. W ten sposób użytkownik dowiedział się o „bibliotecznym typie” `Link`. Jednak „biblioteka” nie zna „typu aplikacji” `Name`. Czy to dozwolone? Tak — w C (i C++) można traktować wskaźnik na strukturę jako wskaźnik na jej pierwszy element i odwrotnie.

Oczywiście ten przykład listy można bez żadnych zmian traktować jako kod w języku C++.

WYPRÓBUJ

Często programiści języka C++ twierdzą, że cokolwiek nie napisze się w języku C, oni mogą zrobić to samo, tylko lepiej. Napisz więc powyższą intruzyjną listę w C++, pokazując, jak ją skrócić i ułatwić jej użytkowanie bez spowalniania programu i zwiększania rozmiaru obiektów.

Ćwiczenia

1. Napisz program „Witaj, świecie!” w języku C. Skompiluj i uruchom go.
2. Zdefiniuj dwie zmienne przechowujące słowa „Witaj” i „świecie!”. Połącz je spacją i wydrukuj na wyjściu.
3. Zdefiniuj w języku C funkcję pobierającą parametr typu `char*` o nazwie `p` oraz parametr typu `int` o nazwie `x` i wydrukuj ich wartości w następującym formacie — `p` wynosi "foo" a `x` wynosi 7. Wywołaj ją z kilkoma parami argumentów.

Powtórzenie

W poniższym zestawie mamy na myśli standard ISO C89 języka C.

1. Czy język C++ jest podzbiorem języka C?
2. Kto wynalazł język C?
3. Podaj tytuł wysoce cenionej książki o języku C.
4. W jakiej organizacji wynaleziono języki C i C++?
5. Dlaczego język C++ jest prawie w pełni zgodny z C?
6. Dlaczego język C++ jest tylko **prawie** w pełni zgodny z C?
7. Wymień przynajmniej dziesięć narzędzi języka C++, których nie ma w C.
8. Do jakiej organizacji należą języki C i C++?
9. Wymień sześć składników biblioteki standardowej C++, których nie można używać w języku C.
10. Których składników biblioteki standardowej C można używać w C++?
11. Jak uzyskuje się sprawdzanie typów argumentów funkcji w języku C?
12. Jakich rzeczy związanych z funkcjami w języku C++ nie ma w języku C? Wymień przynajmniej trzy. Podaj przykłady.
13. Jak wywołuje się funkcje w języku C z poziomu języka C++?
14. Jak wywołuje się funkcje w języku C++ z poziomu języka C?
15. Które typy mają zgodne układy w językach C i C++? Podaj przykłady.
16. Co to jest znacznik struktury?
17. Wymień 20 słów kluczowych języka C++, które nie są kluczowe w języku C.
18. Czy `int x;` jest definicją w języku C++? A w C?
19. Na czym polega rzutowanie w stylu języka C i czemu jest niebezpieczne?
20. Co to jest `void*` i czym różni się `void*` w C od `void*` w C++?

21. Czym różnią się wyliczenia w języku C od wyliczeń w C++?
22. Co robi się w języku C, aby uniknąć problemów konsolidatora z popularnymi nazwami?
23. Wymień trzy najpopularniejsze funkcje języka C wykorzystywane do pracy z pamięcią wolną.
24. Podaj definicję łańcucha w stylu języka C.
25. Czym różni się działanie operatora == i funkcji strcmp(), jeśli chodzi o łańcuchy w stylu języka C?
26. Jak kopiuje się łańcuchy w stylu języka C?
27. Jak sprawdza się długość łańcucha w stylu języka C?
28. Jak należy kopiować duże tablice liczb typu int?
29. Jakie ma zalety funkcja printf()? Jakie ma wady?
30. Czemu nie należy używać funkcji gets()? Czym można ją zastąpić?
31. Jak w języku C otwiera się pliki do odczytu?
32. Czym różni się const w języku C od const w języku C++?
33. Czemu nie lubimy makr?
34. Do czego najczęściej używa się makr?
35. Co to jest strażnik dołączania?

Terminologia

#define	K&R	printf()
#ifdef	kompilacja warunkowa	przeciążanie
#ifndef	leksykograficzny	rzutowanie w stylu języka C
Bell Labs	łańcuch formatujący	strcpy()
Brian Kernighan	łańcuch w stylu języka C	typ nieprzejrzysty
C/C++	łączenie	void
Dennis Ritchie	makro	void*
FILE	malloc()	zgodność
fopen()	nieintryzyjny	znacznik struktury
intruzyjny	porównywanie trójkierunkowe	

Praca domowa

We wszystkich tych zadaniach dobrym pomysłem jest skompilowanie programów zarówno kompilatorem języka C, jak i C++. Jeśli będziesz używać tylko kompilatora C++, możesz przypadkiem użyć czegoś, czego nie ma w C. Jeśli użyjesz tylko kompilatora C, możesz pozostawić błędy związane z typami.

1. Zaimplementuj własne wersje funkcji strlen(), strcmp() i strcpy().
2. Dokończ przykład intruzyjnej listy z podrozdziału 27.9 i przetestuj wszystkie funkcje.
3. „Upiększ” listę intruzyjną z podrozdziału 27.9, jak tylko potrafisz, aby ułatwić korzystanie z niej. Znajdź i obsłuż jak najwięcej błędów. Możesz zmodyfikować definicje struktur, użyć makr itd.
4. Jeśli jeszcze tego nie zrobiłeś, napisz wersję C++ listy intruzyjnej z podrozdziału 27.9 i przetestuj jej wszystkie funkcje.

5. Porównaj wyniki 3. i 4. zadania.
6. Zmień reprezentacje obiektów `Link` i `List` z podrozdziału 27.9, nie zmieniając interfejsów użytkownika udostępnianych przez funkcje. Alokuj obiekty typu `Link` w tablicy i niech składowe `first`, `last`, `pre` oraz `suc` będą typu `int` (wskaźniki do tablicy).
7. Jakie zalety i wady mają intruzyjne kontenery w porównaniu ze standardowymi (nieintruzyjnymi) kontenerami C++? Sporządź listę zalet i wad.
8. Jak wygląda porządek leksykograficzny w Twoim komputerze? Wydrukuj wszystkie znaki swojej klawiatury wraz z ich wartościami liczbowymi. Następnie wydrukuj je w kolejności według tych wartości.
9. Przy użyciu tylko języka C wczytaj sekwencję słów ze strumienia `stdin` i wydrukuj je w strumieniu `stdout` w porządku leksykograficznym. Wskazówka: funkcja sortowania w języku C nazywa się `sort()`. Poszukaj gdzieś informacji na jej temat. Ewentualnie wstawiaj wczytywane słowa do listy uporządkowanej. Nie ma listy w bibliotece standardowej C.
10. Sporządź listę rzeczy przyjętych do języka C z języka C++ lub C with Classes (podrozdział 27.1).
11. Sporządź listę rzeczy, które są w języku C, a które nie zostały od niego przyjęte przez język C++.
12. Zaimplementuj (`string`, `int` w stylu C) tablicę wyszukiwania z operacjami `find(struct table*, const char*)`, `insert(struct table*, const char*, int)` oraz `remove(struct table*, const char*)`. Reprezentacją tej tablicy może być tablica dwóch struktur lub para tablic (`const char*[]` i `int*`). Wybierz też odpowiednie typy zwrotne dla swoich funkcji. Swoje decyzje projektowe opisz w dokumentacji.
13. Napisz program w języku C, który robi to samo, co instrukcje `string s; cin>>s;`. Tzn. zdefiniuj operację wejściową zapisującą dowolnie długą sekwencję znaków zakończoną białym znakiem w tablicy znaków zakończonej zerem.
14. Napisz funkcję pobierającą tablicę liczb typu `int` i znajdującą wśród nich najmniejszą i największą wartość oraz średnią i medianę. Niech zwraca strukturę (`struct`) zawierającą wyniki.
15. Zasymuluj pojedyncze dziedziczenie w języku C. Niech każda „klasa bazowa” zawiera wskaźnik na tablicę wskaźników na funkcje (aby zasymulować funkcje wirtualne jako wolne funkcje pobierające jako pierwszy argument wskaźnik na obiekt „klasy bazowej”) — zobacz punkt 27.2.3. Zaimplementuj „derywację klas”, czyniąc „klasę bazową” typem pierwszej składowej klasy pochodnej. Dla każdej klasy odpowiednio zainicjuj tablicę „funkcji wirtualnych”. W ramach testów zaimplementuj nową wersję „starego przykładu klasy `Shape`” z bazową i derywowaną funkcją `draw()` drukującą tylko nazwę swojej klasy. Używaj tylko narzędzi i biblioteki standardowej języka C.
16. Za pomocą makr zaciemnij (uproszcz notację) implementację z poprzedniego zadania.

Podsumowanie

Już pisaliśmy, że tematyka zgodności nie jest zbyt pasjonująca. Na świecie jest jednak mnóstwo kodu napisanego w języku C (miliardy wierszy) i ten rozdział przygotowuje Cię do jego czytania i pisania. Osobiście wolimy język C++, a niniejszy rozdział niech służy jako zbiór naszych argumentów. Nie oceniaj zbyt nisko opisanego w tym rozdziale przykładu listy intruzyjnej. Intruzyjne listy i typy nieprzejrzyste są ważną i potężną techniką (zarówno w C, jak i C++).

