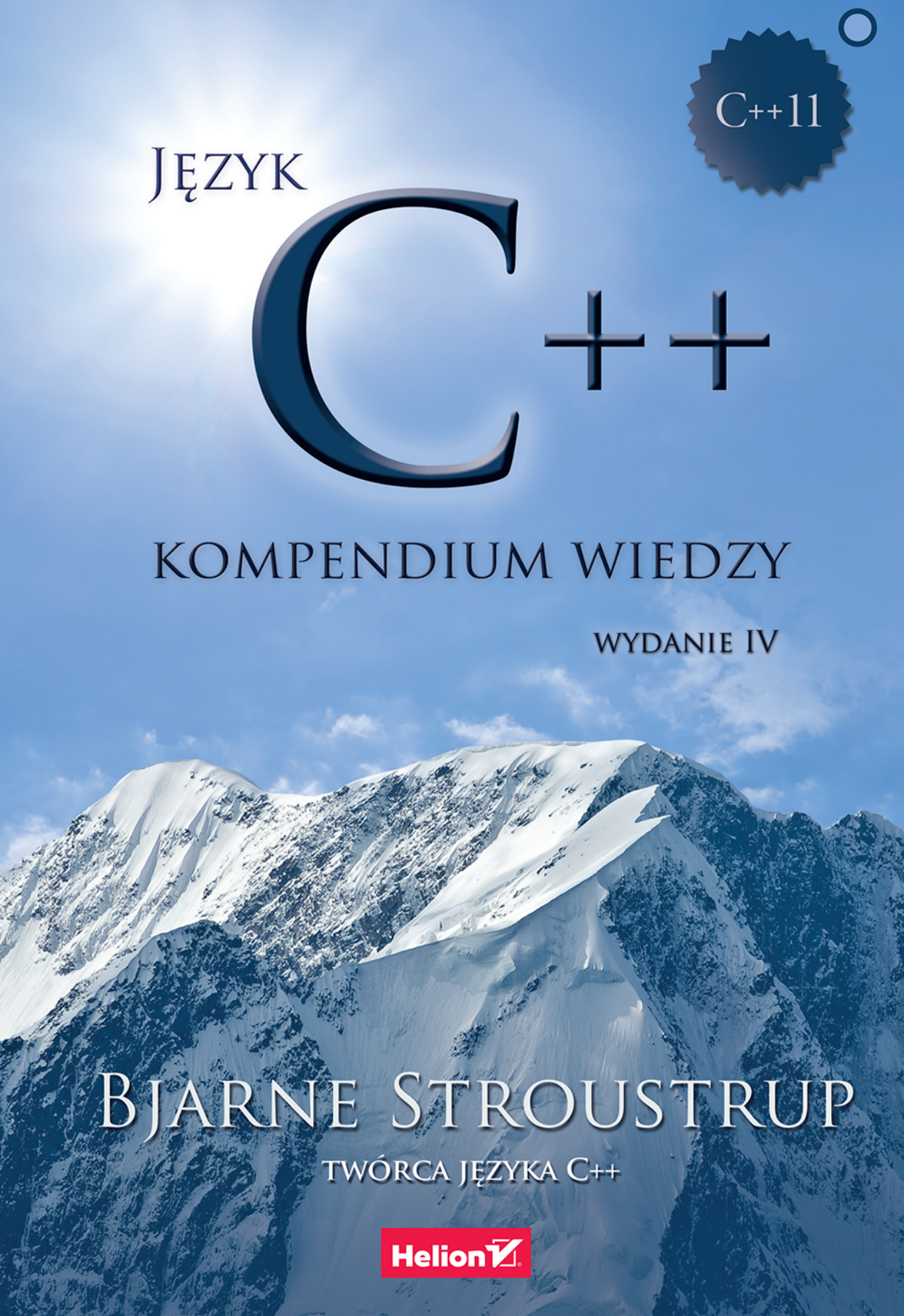

C++11
Język
C++
KOMPENDIUM WIEDZY
WYDANIE IV
BJARNE STROUSTRUP
TWÓRCA JĘZYKA C++
Helion

Tytuł oryginału: The C++ Programming Language, 4th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-8329-6

Authorized translation from the English language edition, entitled:
THE C++ PROGRAMMING LANGUAGE, FOURTH EDITION;
ISBN 0321563840; by Bjarne Stroustrup; published by Pearson Education, Inc, publishing as Addison Wesley.

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

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

Printed in Poland.

Spis treści

10

Wyrażenia

Programowanie jest jak seks:
może dawać konkretne wyniki,
ale nie po to się to robi
— przeprosiny dla Richarda Feynmana

- Wprowadzenie
- Kalkulator
  Parser; Wejście; Wejście niskopoziomowe; Obsługa błędów; Sterownik; Nagłówki; Argumenty wiersza poleceń; Uwaga na temat stylu
- Zestawienie operatorów
  Wyniki; Kolejność wykonywania działań; Kolejność wykonywania operatorów; Obiekty tymczasowe
- Wyrażenia stałe
  Stałe symboliczne; const w wyrażeniach stałych; Typy literałowe; Argumenty referencyjne; Wyrażenia stałe adresowe
- Niejawna konwersja typów
  Promocje; Konwersje; Typowe konwersje arytmetyczne
- Rady

10.1. Wprowadzenie

W tym rozdziale znajduje się szczegółowy opis właściwości wyrażeń. W języku C++ wyrażeniem jest przypisanie, wywołanie funkcji, utworzenie obiektu, jak również wiele innych operacji wykraczających daleko poza konwencjonalne obliczanie wyrażeń arytmetycznych. Aby pokazać, w jaki sposób używa się wyrażeń, a także przedstawić je w odpowiednim kontekście, opisałem budowę niewielkiego programu — prostego kalkulatora. Dalej przedstawiłem zestawienie operatorów oraz zwięźle opisałem sposób ich działania dla typów wbudowanych. Operatory wymagające bardziej szczegółowego omówienia są opisane w rozdziale 11.

10.2. Kalkulator

Wyobraź sobie prosty kalkulator służący do wykonywania czterech podstawowych działań arytmetycznych reprezentowanych przez operatory infiksowe działające na liczbach zmiennoprzecinkowych. Na przykład dla poniższych danych:

```
r = 2.5
area = pi*r*r
```

(wartość `pi` jest zdefiniowana standardowo) program ten zwróci taki wynik:

```
2.5
19.635
```

`2.5` to wynik dla pierwszej linijki danych wejściowych, a `19.635` to wynik dla drugiej.

Cztery główne elementy budowy kalkulatora to: parser, funkcja przyjmowania danych, tablica symboli oraz sterownik. W istocie jest to miniaturowy kompilator, w którym parser wykonuje analizę składniową, funkcja wejściowa pobiera dane i wykonuje analizę leksykalną, tablica symboli przechowuje informacje stałe, a sterownik obsługuje inicjację, wyjście i błędy. Kalkulator ten można by było wzbogacić o wiele dodatkowych funkcji, ale jego kod i tak już jest długi, a takie dodatki w żaden sposób nie przyczyniłyby się do lepszego poznania sposobu używania języka C++.

10.2.1. Parser

Poniżej znajduje się gramatyka języka rozpoznawanego przez parser:

```
program:
    end                 // end oznacza koniec danych wejściowych
    expr_list end

expr_list:
    expression print    // print jest znakiem nowego wiersza lub średnikiem
    expression print expr_list

expression:
    expression + term
    expression – term
    term

term:
    term / primary
    term*primary
    primary

primar y:
    number              // number jest literałem zmiennoprzecinkowym
    name                // name jest identyfikatorem
    name = expression
    – primar y
     ( expression )
```

Innymi słowy, program jest sekwencją wyrażeń oddzielonych średnikami. Podstawowymi jednostkami wyrażenia są liczby, nazwy i operatory `*`, `/`, `+`, `-` (zarówno jedno-, jak i dwuargumentowy) oraz `=` (przypisanie). Nazw nie trzeba deklarować przed użyciem.

Rodzaj analizy, który stosuję, nazywa się **zstępowaniem rekurencyjnym** (ang. *recursive descent*). Jest to popularna i prosta technika przetwarzania kodu od góry do dołu. W językach programowania takich jak C++, w których wywołania funkcji są względnie mało kosztowne, metoda ta jest dodatkowo bardzo wydajna. Dla każdej produkcji w gramatyce istnieje funkcja wywołująca inne funkcje. Symbole terminalne (np. `end`, `number`, `+` i `-`) są rozpoznawane przez

analizator leksykalny, a symbole nieterminalne rozpoznają funkcję analizy leksykalnej: `expr()`, `term()` i `prim()`. Gdy oba argumenty wyrażenia lub podwyrażenia są znane, następuje obliczenie wartości. W prawdziwym kompilatorze mógłby to być moment wygenerowania kodu.

Do obsługi danych wejściowych wykorzystywana jest klasa `Token_stream` zawierająca operacje wczytywania znaków oraz składania z nich tokenów. Innymi słowy, klasa `Token_stream` zamienia strumienie znaków, takie jak `123.45`, w tokeny (`Token`). Tokeny to pary {rodzaj-tokenu, wartość}, jak np. {`number`,`123.45`}, gdzie literał `123.45` jest zamieniony na wartość zmiennoprzecinkową. Główne części parsera muszą tylko znać nazwę strumienia tokenów (`Token_stream`), `ts`, oraz móc pobrać z niego tokeny. W celu wczytania kolejnego tokenu wywoływana jest funkcja `ts.get()`. Aby pobrać ostatnio wczytany (bieżący) token, wywoływana jest funkcja `ts.current()`. Dodatkową funkcją klasy `Token_stream` jest ukrywanie prawdziwego źródła znaków. Zobaczysz, że mogą one pochodzić wprost od użytkownika wpisującego je za pomocą klawiatury do strumienia `cin`, z wiersza poleceń lub z jakiegoś innego strumienia wejściowego (10.2.7).

Definicja tokenu jest następująca:

```
enum class Kind : char {
    name, number, end,
    plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};

struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

Reprezentowanie każdego tokenu przez wartość całkowitoliczbową jego znaku jest wygodnym i wydajnym rozwiązaniem oraz może być pomocne dla programistów używających debugera. Metoda ta działa, pod warunkiem że żaden znak pojawiający się na wejściu nie ma wartości użytej jako enumerator — a żaden znany mi zestaw znaków nie zawiera żadnego drukowalnego znaku w postaci jednocyfrowej liczby całkowitej.

Interfejs klasy `Token_stream` wygląda tak:

```
class Token_stream {
public:
    Token get();              // wczytuje i zwraca następny token
    const Token& current(); // ostatnio wczytany token
    //...
};
```

Implementacja jest przedstawiona w sekcji 10.2.2.

Każda funkcja parsera pobiera argument typu `bool` (6.2.2), o nazwie `get`, wskazujący, czy konieczne jest wywołanie funkcji `Token_stream::get()` w celu pobrania następnego tokenu. Każda funkcja parsera oblicza wartość „swojego” wyrażenia i ją zwraca. Funkcja `expr()` obsługuje dodawanie i odejmowanie. Zawiera jedną pętlę znajdującą wyrazy do dodania lub odjęcia:

```
double expr(bool get) // dodaje i odejmuje
{
    double left = term(get);

    for (;;) { // wieczność
        switch (ts.current().kind) {
        case Kind::plus:
```

```
            left += term(true);
            break;
        case Kind::minus:
            left -= term(true);
            break;
        default:
            return left;
        }
    }
}
```

Funkcja ta sama niewiele robi. Jak typowa wysokopoziomowa funkcja w każdym większym programie ogranicza się do wywoływania innych funkcji.

Instrukcja `switch` (2.2.4, 9.4.2) porównuje swój warunek, który jest wpisany w nawiasie za słowem kluczowym `switch`, ze zbiorem stałych. Instrukcje `break` służą do wychodzenia z instrukcji `switch`. Jeśli porównywana wartość nie pasuje do żadnej z klauzul `case`, wybierana jest klauzula `default`. Programista nie musi definiować tej klauzuli.

Zwróć uwagę, że wyrażenia takie jak `2-3+4` są zgodnie z gramatyką obliczane jako `(2-3)+4`.

Ta dziwna instrukcja `for(;;)` to pętla nieskończona (9.5). Alternatywnie można też używać pętli `while(true)` w tym samym celu. Wykonanie instrukcji `switch` jest powtarzane, dopóki nie pojawi się coś innego niż `+` lub `-`, w którym to przypadku następuje przejście do klauzuli `default` i wykonanie instrukcji `return`.

Operatory `+=` i `-=` służą do obsługi dodawania i odejmowania. Zamiast nich można by było bez znaczenia dla działania programu użyć wyrażeń `left=left+term(true)` i `left=left-term(true)`. Jednak wyrażenia `left+=term(true)` i `left-=term(true)` nie dość, że są krótsze, to na dodatek bardziej bezpośrednio odzwierciedlają zamiar programisty. Każdy operator przypisania jest osobnym tokenem leksykalnym, a więc `a + = 1;` jest błędem składniowym z powodu spacji między operatorami `+` i `=`.

W języku C++ dla operatorów binarnych dostępne są operatory przypisania:

```
+    -    *    /    %    &    |    ^    <<    >>
```

Można więc używać następujących operatorów przypisania:

```
=    +=    -=    *=    /=    %=    &=    |=    ^=    <<=    >>=
```

Operator `%` oznacza resztę z dzielenia. Operatory `&`, `|` oraz `^` to bitowe operacje logiczne i, lub oraz lub wykluczające. Operatory `<<` i `>>` to operacje przesunięcia w lewo i prawo. Zestawienie operatorów i opis ich działania znajduje się w podrozdziale 10.3. Dla binarnego operatora `@` zastosowanego do argumentów typu wbudowanego wyrażenie `x@=y` oznacza `x=x@y`, z tym że wartość `x` jest obliczana tylko raz.

Funkcja `term()` obsługuje mnożenie i dzielenie w taki sam sposób jak `expr()` dodawanie i odejmowanie:

```
double term(bool get) // mnoży i dzieli
{
    double left = prim(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::mul:
            left*= prim(true);
            break;
```

```
            case Kind::div:
                if (auto d = prim(true)) {
                    left /= d;
                    break;
                }
                return error("Dzielenie przez 0");
            default:
                return left;
            }
        }
    }
```

Wynik dzielenia przez zero jest niezdefiniowany i najczęściej operacja taka ma katastrofalne skutki. Dlatego przed wykonaniem dzielenia sprawdzamy, czy dzielnik nie jest zerem, i w razie potrzeby wywołujemy funkcję `error()`. Jej opis znajduje się w sekcji 10.2.4.

Zmienna `d` została wprowadzona do programu dokładnie w miejscu, w którym jest potrzebna, i od razu jest zainicjowana. Zakres dostępności nazwy wprowadzonej w warunku obejmuje instrukcję kontrolowaną przez ten warunek, a powstała wartość jest wartością tego warunku (9.4.3). W konsekwencji dzielenie i przypisanie `left/=d` są wykonywane tylko wtedy, gdy `d` nie równa się zero.

Funkcja `prim()` obsługująca *wyrażenia pierwotne* jest podobna do funkcji `expr()` i `term()`, ale w odróżnieniu od nich znajduje się na nieco niższym poziomie hierarchii wywołań i wykonuje trochę realnej pracy oraz nie zawiera pętli:

```
double prim(bool get)   // obsługuje wyrażenia pierwotne
{
    if (get) ts.get(); // wczytuje następny token

    switch (ts.current().kind) {
    case Kind::number: // stała zmiennoprzecinkowa
    { double v = ts.current().number_value;
      ts.g et();
      return v;
    }
    case Kind::name:
    { double& v = table[ts.current().string_value];       // znajduje odpowiednik
      if (ts.get().kind == Kind::assign) v = expr(true); // znaleziono operator =: przypisanie
      return v;
    }
    case Kind::minus: // jednoargumentowy minus
        return -prim(true);
    case Kind::lp:
    { auto e = expr(true);
      if (ts.current().kind != Kind::rp) return error("Oczekiwano ')'");
      ts.get();       // zjada ')'
      return e;
    }
    default:
        return error("Oczekiwano wyrażenia pierwotnego");
    }
}
```

Gdy zostanie znaleziony `Token` będący liczbą (tzn. literałem całkowitoliczbowym lub zmiennoprzecinkowym), jego wartość jest umieszczana w składowej `number_value`. Analogicznie, gdy zostanie napotkany `Token` będący nazwą (`name`) — jakakolwiek jest jej definicja; zobacz 10.2.2 i 10.2.3 — jego wartość jest umieszczana w składowej `string_value`.

Zwróć uwagę, że funkcja `prim()` zawsze wczytuje o jeden token więcej, niż potrzebuje do analizy swojego wyrażenia pierwotnego. Powodem tego jest fakt, że *musi* to robić w niektórych przypadkach (aby np. sprawdzić, czy do nazwy jest coś przypisywane), więc dla zachowania spójności robi to zawsze. W przypadkach, w których funkcja parsera ma tylko przejść do następnego tokenu, nie musi używać wartości zwrotnej funkcji `ts.get()`. Nie ma problemu, bo wynik można pobrać z `ts.current()`. Gdyby przeszkadzało mi ignorowanie wartości zwrotnej funkcji `get()`, to albo dodałbym funkcję `read()`, która tylko by aktualizowała `current()` bez zwracania wartości, albo jawnie „wyrzucałbym" wynik: `void(ts.get())`.

Zanim kalkulator w jakikolwiek sposób użyje nazwy, najpierw musi sprawdzić, czy jest coś do niej przypisywane, czy też jest ona tylko odczytywana. W obu przypadkach trzeba sięgnąć do tablicy symboli. Tablica ta jest mapą (4.4.3, 31.4.3):

```
map<string,double> table;
```

To znaczy, że mapa `table` jest indeksowana typem `string`, dla którego zwracane są wartości typu `double`. Jeśli na przykład użytkownik wpisze:

```
radius = 6378.388;
```

kalkulator dojdzie do klauzuli `case Kind::name` i wykona:

```
double& v = table["radius"];
// ... expr() oblicza wartość do przypisania...
v = 6378.388;
```

Referencja `v` jest użyta jako uchwyt do wartości `double` związanej z `radius`, podczas gdy funkcja `expr()` oblicza wartość `6378.388` ze znaków wejściowych.

W rozdziałach 14. i 15. znajdują się wskazówki na temat organizacji programu jako zbioru modułów. Ale w tym kalkulatorze deklaracje można uporządkować tak, że (z jednym wyjątkiem) każda wystąpi tylko raz, przed samym jej użyciem. Wyjątkiem jest funkcja `expr()`, która wywołuje funkcję `term()`, która wywołuje funkcję `prim()`, która z kolei wywołuje funkcję `expr()`. Ten krąg wywołań trzeba jakoś rozerwać. Można na przykład umieścić deklarację:

```
double expr(bool);
```

przed definicją funkcji `prim()`.

10.2.2. Wejście

Mechanizm wczytywania danych wejściowych to często najbardziej skomplikowana część programu, bo trzeba wziąć pod uwagę humory, przyzwyczajenia i rozmaite błędy popełniane przez ludzi. Próby zmuszenia użytkownika do zachowywania się w sposób bardziej odpowiadający maszynie są zwykle (słusznie) uważane za niegrzeczne. Zadaniem niskopoziomowej procedury wejściowej jest wczytanie znaków i złożenie z nich tokenów, którymi następnie posługują się procedury działające na wyższym poziomie. W opisywanym programie niskopoziomowa procedura obsługi wejścia nazywa się `td.get()`. Pisanie takich funkcji nie jest częstym zadaniem, bo wiele systemów zawiera standardowe funkcje tego rodzaju.

Najpierw jednak musimy zobaczyć kompletny kod klasy `Token_stream`:

```
class Token_stream {
public:
    Token_stream(istream& s) : ip{&s}, owns{false} { }
    Token_stream(istream*p) : ip{p}, owns{true} { }

    ~Token_stream() { close(); }

    Token get();       // wczytuje token i go zwraca
    Token& current(); // ostatnio wczytany token

    void set_input(istream& s) { close(); ip = &s; owns=false; }
    void set_input(istream*p) { close(); ip = p; owns = true; }

private:
    void close() { if (owns) delete ip; }

    istream*ip;              // wskaźnik do strumienia wejściowego
    bool owns;               // czy Token_stream jest właścicielem strumienia istream?
    Token ct {Kind::end} ; // bieżący token
};
```

Obiekt klasy `Token_stream` inicjujemy strumieniem wejściowym (4.3.2, rozdział 38.), z którego pobierane są znaki. Klasa `Token_stream` jest zaimplementowana w taki sposób, że staje się właścicielem (i ostatecznie też niszczycielem — 3.2.1.2, 11.2) strumienia `istream` przekazanego jako wskaźnik, ale nie strumienia `istream` przekazanego jako referencja. W tak prostym programie może nie trzeba stosować aż tak wyszukanego rozwiązania, ale jest to bardzo przydatna i ogólna technika wykorzystywana do budowy klas przechowujących wskaźniki do zasobów, które kiedyś trzeba usunąć.

Klasa `Token_stream` zawiera trzy wartości: wskaźnik do swojego strumienia wejściowego (`ip`), wartość logiczną (`owns`) określającą własność strumienia wejściowego oraz bieżący token (`ct`).

Zmiennej `cp` nadałem wartość domyślną, bo wydawało mi się, że niezrobienie tego byłoby nie w porządku. Wprawdzie nie powinno się wywoływać funkcji `current()` przed `get()`, ale jeśli ktoś to zrobi, to otrzyma poprawny token. Jako wartość początkową `ct` wybrałem `Kind::end`, aby w przypadku niewłaściwego użycia funkcji `current()` program nie otrzymał żadnej wartości, która nie pojawiła się w strumieniu wejściowym.

Funkcję `Token_stream::get()` przedstawię w dwóch odsłonach. Najpierw pokażę złudnie prostą wersję, która będzie mniej przyjazna dla użytkownika. A następnie zmodyfikuję ją do mniej eleganckiej, ale za to o wiele łatwiejszej w użyciu postaci. Ogólnie zadaniem funkcji `get()` jest wczytanie znaku, zdecydowanie, do jakiego rodzaju tokenu należy go użyć, oraz w razie potrzeby wczytanie większej ilości znaków i zwrócenie tokenu reprezentującego te wczytane znaki.

Początkowe instrukcje wczytują pierwszy niebiały znak z `*ip` (strumienia wejściowego wskazywanego przez `ip`) do `ch` i sprawdzają, czy operacja odczytu się powiodła:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end}; // przypisanie i zwrot
```

Domyślnie operator >> pomija białe znaki (spacje, tabulatory, znaki nowego wiersza itd.) i pozostawia wartość ch niezmienioną, jeśli operacja przyjmowania danych się nie powiedzie. W konsekwencji ch==0 oznacza koniec wprowadzania danych.

Przypisanie jest operatorem, a jego wynik jest wartością zmiennej, której dotyczy to przypisanie. Dzięki temu mogę przypisać wartość Kind::end do curr_tok i zwrócić ją w tej samej instrukcji. Jedna instrukcja zamiast dwóch ułatwia konserwację kodu. Gdybym rozdzielił przypisanie i zwrot, to inny programista mógłby zmienić coś w jednej części i zapomnieć odpowiednio dostosować drugą.

Zwróć też uwagę na sposób użycia notacji listowej {} (3.2.1.3, 11.3) po prawej stronie przypisania. Jest to wyrażenie. Powyższą instrukcję return można by było napisać również tak:

```
ct.kind = Kind::end; // przypisanie
return ct;           // zwrot
```

Uważam jednak, że przypisanie kompletnego obiektu {Kind::end} jest bardziej zrozumiałe niż posługiwanie się poszczególnymi składowymi ct. Zapis {Kind::end} jest równoważny z {Kind::end,0,0}. To dobrze, jeśli interesują nas dwie ostatnie składowe tokenu, i źle, jeśli zależy nam na wydajności. Nas w tym przypadku nie dotyczy ani pierwsze, ani drugie, ale ogólnie rzecz biorąc, posługiwanie się kompletnymi obiektami stwarza mniej okazji do popełnienia błędu niż grzebanie przy poszczególnych składowych. Poniżej przedstawiona jest implementacja tej drugiej strategii.

Najpierw przeanalizuj kilka z poniższych klauzul case osobno, a dopiero potem zastanów się nad działaniem całej funkcji. Znak oznaczający koniec wyrażenia (;), nawiasy oraz operatory są obsługiwane poprzez zwykłe zwracanie ich wartości:

```
case ';': // koniec wyrażenia; drukowanie
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
    return ct={static_cast<Kind>(ch)};
```

Operacja static_cast (11.5.2) jest w tym przypadku niezbędna, ponieważ nie istnieje niejawna konwersja typu char na Kind (8.4.1). Tylko niektóre znaki odpowiadają wartościom Kind, więc musimy zagwarantować to dla ch.

Liczby są obsługiwane w następujący sposób:

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
↪case '8': case '9': case '.':
ip->putback(ch);         // wstawia pierwszą cyfrę (albo .) z powrotem do strumienia wejściowego
*ip >> ct.number_value; // wczytuje liczbę do ct
ct.kind=Kind::number;
return ct;
```

Wypisanie wszystkich klauzul case w jednej linii zamiast każdej w osobnym wierszu nie jest dobrym pomysłem, bo taki kod jest mniej czytelny. Z drugiej strony, wpisywanie każdej cyfry w osobnej linijce jest żmudne. Dzięki temu, że operator >> już obsługuje wczytywanie liczb zmiennoprzecinkowych do zmiennych typu double, ten kod jest bardzo prosty. Najpierw pierwszy znak (cyfra lub kropka) jest umieszczany w cin. Następnie wartość zmiennoprzecinkowa może zostać wczytana do ct.number_value.

Jeśli token nie jest końcem danych wejściowych ani operatorem, znakiem interpunkcyjnym czy też liczbą, to musi być nazwą. Nazwy są obsługiwane podobnie do liczb:

```
default: // name, name = lub błąd
    if (isalpha(ch)) {
        ip->putback(ch);        // umieszcza pierwszy znak z powrotem w strumieniu wejściowym
        *ip>>ct.string_value; // wczytuje łańcuch do ct
        ct.kind=Kind::name;
        return ct;
}
```

W końcu może też wystąpić błąd. Proste a zarazem całkiem efektywne rozwiązanie na poradzenie sobie z błędami to napisanie funkcji `error()` i zwrócenie tokenu `print` w wartości zwrotnej tej funkcji:

```
error("Niepoprawny token");
return ct={Kind::print};
```

Funkcja `isalpha()` z biblioteki standardowej (36.2.1) została użyta po to, aby uniknąć konieczności wymieniania każdego znaku w osobnej klauzuli `case`. Jeśli operator `>>` zostanie zastosowany do łańcucha (w tym przypadku `string_value`), wczytuje znaki do momentu napotkania białego znaku. W związku z tym użytkownik musi po nazwie wpisać biały znak przed operatorem używającym tej nazwy jako argumentu. Jest to bardzo słabe rozwiązanie i będziemy musieli jeszcze do tego wrócić w sekcji 10.2.3.

Poniżej znajduje się ukończona funkcja wejściowa:

```
Token Token_stream::g et()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end}; // przypisanie i zwrot
    case ';': // koniec wyrażenia; drukuje
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return ct={static_cast<Kind>(ch)};

    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
    ↪case '8': case '9': case '.':
        ip->putback(ch);          // wstawia pierwszą cyfrę (albo .) z powrotem do strumienia wejściowego
        *ip >> ct.number_value;   // wczytuje liczbę do ct
        ct.kind=Kind::number;
        return ct;
    default: // name, name = lub błąd
        if (isalpha(ch)) {
            ip->putback(ch);      // umieszcza pierwszy znak z powrotem w strumieniu wejściowym
            *ip>>ct.string_value; // wczytuje łańcuch do ct
            ct.kind=Kind::name;
```

```
            return ct;
        }

        error("Niepoprawny token");
        return ct={Kind::print};
    }
}
```

Konwersja operatora na token jest bardzo prosta, bo rodzaje operatorów zostały zdefiniowane jako wartości całkowitoliczbowe (10.2.1).

10.2.3. Wejście niskopoziomowe

W kalkulatorze w obecnej postaci można znaleźć kilka niedogodności. Trzeba pamiętać o dodawaniu średnika na końcu wyrażeń, aby otrzymać wynik, oraz po nazwie zawsze trzeba wpisać biały znak, co również jest uciążliwe. Na przykład `x+7` jest tylko identyfikatorem, a nie identyfikatorem `x`, operatorem `=` i liczbą `7`. Aby ten zapis został zinterpretowany tak, jak byśmy tego normalnie chcieli, trzeba by było dodać biały znak za `x`: `x =7`. Oba problemy można rozwiązać poprzez zamianę zorientowanych na typy domyślnych operacji w funkcji `get()` na kod wczytujący pojedyncze znaki.

Najpierw zrównamy znaczenie znaku nowego wiersza ze średnikiem oznaczającym koniec wyrażenia:

```
Token Token_stream::get()
{
    char ch;

    do { // pomija białe znaki oprócz '\n'
        if (!ip->get(ch)) return ct={Kind::end};
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::print};
```

W kodzie tym użyłem instrukcji `do`, która różni się od pętli `while` tylko tym, że kontrolowane przez nią instrukcje są wykonywane przynajmniej raz. Wywołanie `ip->get(ch)` wczytuje jeden znak ze strumienia wejściowego `*ip` do `ch`. Domyślnie funkcja `get()` nie pomija białych znaków tak jak operator `>>`. Test `if (!ip->get(ch))` kończy się pomyślnie, gdy nie można wczytać żadnego znaku z `cin`. W tym przypadku sesję kalkulatora kończy zwrócenie wartości `Kind::end`. Operator `!` (nie) został użyty, bo funkcja `get()` zwraca `true` w przypadku powodzenia.

Do sprawdzania, czy znak jest znakiem białym, można używać funkcji `isspace()` z biblioteki standardowej (36.2.1). Wywołanie `isspace(c)` zwraca wartość niezerową, jeśli `c` jest białym znakiem, a zero w przeciwnym przypadku. Test ten jest zaimplementowany w formie przeszukiwania tablicy, dzięki czemu użycie funkcji `isspace()` jest wydajniejszym rozwiązaniem niż samodzielne sprawdzanie poszczególnych znaków. Istnieją podobne funkcje do sprawdzania, czy znak jest cyfrą (`isdigit()`), literą (`isalpha()`) oraz cyfrą lub literą (`isalnum()`).

Po pominięciu białego znaku brany jest następny znak w celu określenia, jaki pojawi się token leksykalny.

Problem powodowany przez wczytywanie przez operator `>>` znaków do łańcucha do momentu napotkania białego znaku jest rozwiązany przez wczytywanie po jednym znaku aż do znalezienia znaku nie będącego literą ani cyfrą:

```
default:            // name, name= lub błąd
    if (isalpha(ch)) {
        ct.string_value = ch;
        while (ip->get(ch))
            if (isalnum(ch))
                ct.string_value += ch;      // dołącz ch na końcu string_value
            else {
                ip->putback(ch);
                break;
            }
        ct.kind = Kind::name;
        return ct;
    }
```

Na szczęście oba udoskonalenia można zaimplementować poprzez wprowadzenie zmian w jednym lokalnym miejscu kodu. Konstruowanie programów w taki sposób, aby można je było usprawniać poprzez wprowadzanie zmian lokalnych, jest ważną częścią projektowania.

Ktoś może się obawiać, że dodawanie znaków po jednym do łańcucha to bardzo niewydajna operacja. W przypadku bardzo długich łańcuchów rzeczywiście by tak było, ale wszystkie nowoczesne implementacje typu `string` zawierają „mechanizm optymalizacji małych łańcuchów" (19.3.3). To oznacza, że obsługa łańcuchów, jakie mogą pojawić się jako nazwy w naszym kalkulatorze (a nawet w kompilatorze), nie wymaga wykonywania kosztownych operacji. W szczególności użycie krótkiego łańcucha nie wymaga korzystania z pamięci wolnej. Maksymalna liczba znaków w krótkim łańcuchu jest zależna od implementacji, ale myślę, że można powiedzieć, iż wynosi około 14.

10.2.4. Obsługa błędów

Nigdy nie należy zaniedbywać wykrywania i zgłaszania błędów, chociaż w tym programie wystarczy prosta strategia obsługi błędów. Funkcja `error()` liczy błędy, drukuje informację o błędzie i dokonuje zwrotu:

```
int no_of_errors;

double error(const string& s)
{
    no_of_errors++;
    cerr << "Błąd: " << s << '\n';
    return 1;
}
```

Strumień `cerr` to niebuforowany strumień wyjściowy służący do zgłaszania błędów (38.1).

Powodem, dla którego zwracana jest wartość, jest to, że błędy najczęściej występują w trakcie obliczania wartości wyrażeń, więc należy całkiem przerwać tę operację albo zwrócić wartość, która nie spowoduje kolejnych błędów. To drugie rozwiązanie jest właściwe w przypadku naszego kalkulatora. Gdyby funkcja `Token_stream::get()` zapamiętywała numery wierszy, funkcja `error()` mogłaby informować, gdzie w przybliżeniu wystąpił błąd. Informacja taka byłaby przydatna przy używaniu kalkulatora w nieinteraktywny sposób.

Można by było opracować bardziej elegancki i ogólny mechanizm obsługi błędów, w którym wykrywanie błędów byłoby oddzielone od procedur odzyskiwania sprawności po ich wystąpieniu. Można do tego celu użyć wyjątków (2.4.3.1, rozdział 13.), ale zastosowane obecnie rozwiązanie jest w zupełności wystarczające jak dla kalkulatora składającego się z około 180 wierszy kodu.

10.2.5. Sterownik

Mamy już praktycznie wszystkie części programu i potrzebujemy jeszcze tylko sterownika, aby go uruchomić. Zdecydowałem się użyć dwóch funkcji: `main()` do konfiguracji i zgłaszania błędów oraz `calculate()` do wykonywania samych obliczeń:

```
Token_stream ts {cin}; // użycie danych z cin

void calculate()
{
    for (;;) {
        ts.g et();
        if (ts.current().kind == Kind::end) break;
        if (ts.current().kind == Kind::print) continue;
        cout << expr(false) << '\n';
    }
}

int main()
{
    table["pi"] = 3.1415926535897932385; // standardowo zdefiniowane nazwy
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

Funkcja `main()` tradycyjnie zwraca zero, jeżeli działanie programu zakończy się w normalny sposób, a wartość różną od zera w pozostałych przypadkach (2.2.1) — dobrym pomysłem jest zwracanie liczby błędów. W tym programie jedyne inicjacje, jakie należy wykonać, to wprowadzenie standardowych nazw do tablicy symboli.

Najważniejszym zadaniem pętli głównej (w funkcji `calculate()`) jest wczytywanie wyrażeń i drukowanie wyników. Służy do tego poniższy wiersz:

```
cout << expr(false) << '\n';
```

Argument `false` informuje funkcję `expr()`, że nie trzeba wywoływać funkcji `ts.get()` w celu wczytania tokenu do obróbki.

Test dotyczący `Kind::end` gwarantuje poprawne wyjście z pętli, gdy funkcja `ts.get()` napotka błąd wejściowy albo koniec pliku. Instrukcja `break` powoduje wyjście z najbliższego otaczającego bloku `switch` lub pętli (9.5). Test dotyczący `Kind::print` (tzn. znaków `'\n'` i `';'`) zwalnia funkcję `expr()` z obowiązku obsługi pustych wyrażeń. Instrukcja `continue` jest równoznaczna z przejściem na koniec pętli.

10.2.6. Nagłówki

Do budowy kalkulatora zostały użyte narzędzia z biblioteki standardowej. Aby więc działał, trzeba dołączyć odpowiednie nagłówki:

```
#include<iostream> // wejście i wyjście
#include<string>   // łańcuchy
#include<map>      // mapa
#include<cctype>   // isalpha() itp.
```

Narzędzia z tych nagłówków są dostępne w przestrzeni nazw `std`, a więc żeby posługiwać się nazwami tych narzędzi, trzeba stosować kwalifikator `std::` lub przenieść je do globalnej przestrzeni nazw:

```
using namespace std;
```

Aby nie mieszać kwestii dotyczących wyrażeń z modułowością, zdecydowałem się na drugie z wymienionych rozwiązań. W rozdziałach 14. i 15. znajduje się opis technik podziału tego programu na moduły przy użyciu przestrzeni nazw oraz jego organizacji w kilku plikach.

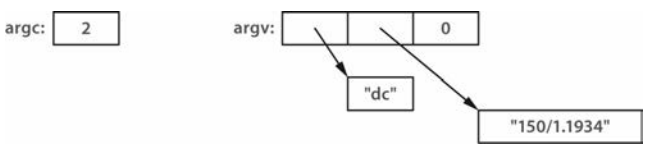
10.2.7. Argumenty wiersza poleceń

Po przetestowaniu programu zauważyłem, że jego uruchamianie, wpisywanie wyrażeń i w końcu zamykanie jest kłopotliwe. Najczęściej służył mi do sprawdzania wartości pojedynczych wyrażeń. Gdyby te wyrażenia dało się wpisywać jako argumenty wiersza poleceń, można by było zaoszczędzić trochę stukania w klawiaturę.

Program rozpoczyna działanie od wywołania funkcji `main()` (2.2.1, 15.4). Funkcji tej przekazywane są dwa argumenty, z których jeden określa liczbę argumentów (o nazwie `argc`) przekazanych przy wywołaniu, a drugi jest tablicą przechowującą te argumenty (o nazwie `argv`). Argumenty te są łańcuchami w stylu języka C (2.2.5, 7.3), a więc `argv` ma typ `char*[argc+1]`. Nazwę programu (w postaci występującej w wierszu poleceń) przekazuje się jako `argv[0]`, więc argument `argc` nigdy nie ma wartości mniejszej od zera. Lista argumentów jest zakończona zerem, tzn. `argv[argc]==0`. Na przykład dla polecenia:

```
dc 150/1.1934
```

argumenty mają następujące wartości:



Jako że metoda wywoływania funkcji `main()` jest w C++ taka sama jak w C, używane są tablice i łańcuchy w stylu języka C.

Chodzi o to, aby wczytywać dane z łańcucha wiersza poleceń w taki sam sposób jak ze strumienia wejściowego. Strumień wczytujący dane z łańcucha nazywa się `istringstream` (38.2.2). Aby więc obliczyć wartość wyrażeń podanych w wierszu poleceń, należy tylko sprawić, aby klasa `Token_stream` pobierała dane z odpowiedniego strumienia `istringstream`:

```
Token_stream ts {cin};

int main(int argc, char*argv[])
{
    switch (argc) {
    case 1: // odczyt ze standardowego strumienia wejściowego
        break;
    case 2: // odczyt z łańcucha argumentów
        ts.set_input(new istringstream{argv[1]});
        break;
    default:
        error("Zbyt wiele argumentów ");
```

```
        return 1;
    }

    table["pi"] = 3.1415926535897932385; // standardowe nazwy
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

Aby móc korzystać ze strumienia `istringstream`, należy dołączyć do programu nagłówek `<sstream>`.

Łatwo można zmodyfikować funkcję `main()`, aby przyjmowała kilka argumentów z wiersza poleceń, ale wydaje się to niepotrzebne, zwłaszcza że przecież można przekazać kilka wyrażeń w jednym argumencie:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Używam cudzysłowu, bo w systemach uniksowych, z których korzystam, średnik jest znakiem oddzielania poleceń. W innych systemach mogą obowiązywać inne konwencje przekazywania argumentów do uruchamianego programu.

Mimo swojej prostoty argumenty `argc` i `argv` często są źródłem drobnych, choć irytujących błędów. Aby ich uniknąć, a przede wszystkim żeby móc łatwiej posługiwać się argumentami programu, zazwyczaj używam prostej funkcji tworzącej wektor `vector<string>`:

```
vector<string> arguments(int argc, char*argv[])
{
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res;
}
```

Często spotyka się również bardziej wyszukane funkcje parsujące.

10.2.8. Uwaga na temat stylu

Dla programisty nie obeznanego z tablicami asocjacyjnymi użycie mapy ze standardowej biblioteki w roli tablicy symboli wygląda prawie jak oszustwo. Ale nim nie jest. Biblioteka standardowa i inne biblioteki są po to, by ich używać. Implementacji i projektowaniu wielu składników bibliotek poświęcono tak dużo czasu i uwagi, że żaden programista nie mógłby tak dopracować jakiegokolwiek fragmentu kodu do użytku w tylko jednym programie.

Patrząc na kod kalkulatora, zwłaszcza jego pierwszą wersję, można zauważyć, że nie ma w nim zbyt wiele tradycyjnego niskopoziomowego kodu w stylu języka C. Wiele zazwyczaj sprawiających trudności problemów rozwiązano poprzez użycie klas z biblioteki standardowej, takich jak `ostream`, `string` i `map` (4.3.1, 4.2, 4.4.3, 31.4, rozdział 36., rozdział 38.).

Zwróć uwagę na względnie niewielką ilość pętli, działań arytmetycznych i przypisań. Tak właśnie powinien wyglądać kod źródłowy programu, który nie operuje bezpośrednio na zasobach sprzętowych ani nie implementuje niskopoziomowych abstrakcji.

10.3. Zestawienie operatorów

W tym podrozdziale znajduje się zestawienie wyrażeń i trochę przykładów. Dla każdego operatora podana jest przynajmniej jedna nazwa i przykład użycia. W poniższych tabelach:

- *Nazwa* jest identyfikatorem (np. `sum`, `map`), nazwą operatora (np. `operator int`, `operator+` i `operator"" km`) lub nazwą specjalizacji szablonu (np. `sort<Record>` i `array<int,10>`) z ewentualnym kwalifikatorem `::` (np. `std::vector` i `vector<T>::operator[]`).
- *Nazwa-klasy* to po prostu nazwa klasy (wliczając `decltype(expr)`, gdzie `expr` oznacza klasę).
- *Składowa* to nazwa składowej (wliczając nazwy destruktorów i szablony będące składowymi).
- *Obiekt* to wyrażenie zwracające obiekt klasy.
- *Wskaźnik* to wyrażenie zwracające wskaźnik (wliczając `this` i obiekt tego typu, który obsługuje tę operację wskaźnikową).
- *Wyrażenie* to wyrażenie, wliczając literał (np. `17`, `"mysz"`, `true`).
- *Lista-wyrażeń* to po prostu lista wyrażeń, która może też być pusta.
- *Lvalue* to wyrażenie oznaczające obiekt, który można modyfikować (6.4.1).
- *Typ* może być w pełni ogólną nazwą typu (z `*`, `()` itd.), tylko gdy znajduje się w nawiasie. W pozostałych przypadkach obowiązują pewne ograniczenia (iso.A).
- *Deklarator-lambdy* to lista (może być pusta) parametrów oddzielanych przecinkami, po której znajduje się opcjonalny specyfikator `mutable`, po nim może znajdować się opcjonalny specyfikator `noexcept`, a po nim może się znajdować opcjonalne określenie typu zwrotnego (11.4).
- *Lista-zmiennych-lokalnych* to lista (może być pusta) określająca zależności kontekstowe (11.4).
- *Lista-instrukcji* to potencjalnie pusta lista instrukcji (2.2.4, rozdział 9.).

Składnia wyrażeń jest niezależna od typów argumentów. Sposób działania podany w tym podrozdziale dotyczy tylko sytuacji, gdy argumenty są typów wbudowanych (6.2.1). Można też definiować nowe znaczenie operatorów dla argumentów typów zdefiniowanych przez użytkownika (2.3, rozdział 18.).

W tabeli da się przedstawić gramatykę języka tylko w przybliżeniu. Szczegółowe informacje można znaleźć w iso.5 i iso.A.

Zestawienie operatorów		
Wyrażenie w nawiasie	(*wyrażenie*)	
Lambda	[*lista-zmiennych-lokalnych*] *deklarator-lambdy* { *lista-instrukcji* }	11.4
Określanie zakresu	*nazwa-klasy* :: *składowa*	16.2.3
Określanie zakresu	*nazwa-przestrzeni-nazw* :: *składowa*	14.2.1
Globalny	:: *nazwa*	14.2.1
Wybór składowej	*obiekt* . *składowa*	16.2.3
Wybór składowej	*wskaźnik* -> *składowa*	16.2.3
Indeksowanie	*wskaźnik* [*wyrażenie*]	7.3
Wywołanie funkcji	*wyrażenie* (*lista-wyrażeń*)	12.2
Tworzenie wartości	*typ* { *lista-wyrażeń* }	11.3.2
Konwersja typów w stylu funkcyjnym	*typ* { *lista-wyrażeń* }	11.5.4

Zestawienie operatorów — *ciąg dalszy*		
Postinkrementacja	`lvalue++`	11.1.4
Postdekrementacja	`lvalue--`	11.1.4
Identyfikacja typu	`typeid( typ )`	22.5
Identyfikacja typu w czasie działania programu	`typeid( wyrażenie )`	22.5
Konwersja kontrolowana w czasie działania programu	`dynamic_cast< typ > (wyrażenie)`	22.2.1
Konwersja kontrolowana w czasie kompilacji	`static_cast< typ > (wyrażenie)`	11.5.2
Niekontrolowana konwersja	`reinterpret_cast< typ > (wyrażenie)`	11.5.2
Konwersja `const`	`const_cast < typ > ( wyrażenie )`	11.5.2
Rozmiar obiektu	`sizeof wyrażenie`	6.2.8
Rozmiar typu	`sizeof (typ)`	6.2.8
Rozmiar pakietu parametrów	`sizeof... nazwa`	28.6.2
Układ typu	`alignof ( typ )`	6.2.9
Preinkrementacja	`++lvalue`	11.1.4
Predekrementacja	`--lvalue`	11.1.4
Dopełnienie	`~ wyrażenie`	11.1.2
Nie	`! wyrażenie`	11.1.1
Jednoargumentowy minus	`- wyrażenie`	2.2.2
Jednoargumentowy plus	`+ wyrażenie`	2.2.2
Adres	`& lvalue`	7.2
Dereferencja	`* wyrażenie`	7.2
Tworzenie (alokacja)	`new typ`	11.2
Tworzenie (alokacja i inicjacja)	`new typ ( lista-wyrażeń )`	11.2
Tworzenie (alokacja i inicjacja)	`new typ { lista-wyrażeń }`	11.2
Tworzenie (umieszczenie)	`new ( lista-wyrażeń ) typ`	11.2.4
Tworzenie (umieszczenie i inicjacja)	`new ( lista-wyrażeń ) typ ( lista-wyrażeń )`	11.2.4
Tworzenie (umieszczenie i inicjacja)	`new ( lista-wyrażeń ) typ ( lista-wyrażeń )`	11.2.4
Usuwanie (dealokacja)	`delete wskaźnik`	11.2
Usuwanie tablicy	`delete [] wskaźnik`	11.2.2
Czy wyrażenie może zgłosić wyjątek?	`noexcept ( wyrażenie )`	13.5.1.2
Rzutowanie (konwersja typów)	`( typ ) wyrażenie`	11.5.3
Wybór składowej	`obiekt .* wskaźnik-do-składowej`	20.6
Wybór składowej	`wskaźnik ->* wskaźnik-do-składowej`	20.6

Każda część tabeli zawiera operatory o tym samym priorytecie. Operatory znajdujące się wyżej mają wyższy priorytet. Na przykład `N::x.m` oznacza `(N::x).m`, a nie `N::(x.m)`.

Na przykład przyrostkowy operator ++ ma wyższy priorytet od jednoargumentowego operatora *, więc *p++ oznacza *(p++), *nie* (*p)++.

Zestawienie operatorów (kontynuacja)		
Mnożenie	*wyrażenie* * *wyrażenie*	10.2.1
Dzielenie	*wyrażenie* / *wyrażenie*	10.2.1
Modulo (reszta z dzielenia)	*wyrażenie* % *wyrażenie*	10.2.1
Dodawanie	*wyrażenie* + *wyrażenie*	10.2.1
Odejmowanie	*wyrażenie* - *wyrażenie*	10.2.1
Przesunięcie w lewo	*wyrażenie* << *wyrażenie*	11.1.2
Przesunięcie w prawo	*wyrażenie* >> *wyrażenie*	11.1.2
Mniejszość	*wyrażenie* < *wyrażenie*	2.2.2
Mniejszy lub równy	*wyrażenie* =< *wyrażenie*	2.2.2
Większość	*wyrażenie* > *wyrażenie*	2.2.2
Większy lub równy	*wyrażenie* >= *wyrażenie*	2.2.2
Równość	*wyrażenie* == *wyrażenie*	2.2.2
Nierówny	*wyrażenie* != *wyrażenie*	2.2.2
Bitowe i	*wyrażenie* & *wyrażenie*	11.1.2
Bitowe lub wykluczające	*wyrażenie* ^ *wyrażenie*	11.1.2
Bitowe lub niewykluczające	*wyrażenie* | *wyrażenie*	11.1.2
Logiczne i	*wyrażenie* && *wyrażenie*	11.1.1
Logiczne lub niewykluczające	*wyrażenie* || *wyrażenie*	11.1.1
Wyrażenie warunkowe	*wyrażenie* ? *wyrażenie* : *wyrażenie*	11.1.3
Lista	{ *lista-wyrażeń* }	11.3
Zgłoszenie wyjątku	throw *wyrażenie*	13.5
Proste przypisanie	*lvalue* = *wyrażenie*	10.2.1
Mnożenie i przypisanie	*lvalue* *= *wyrażenie*	10.2.1
Dzielenie i przypisanie	*lvalue* /= *wyrażenie*	10.2.1
Modulo i przypisanie	*lvalue* %= *wyrażenie*	10.2.1
Dodawanie i przypisanie	*lvalue* += *wyrażenie*	10.2.1
Odejmowanie i przypisanie	*lvalue* -= *wyrażenie*	10.2.1
Przesunięcie w lewo i przypisanie	*lvalue* <<= *wyrażenie*	10.2.1
Przesunięcie w prawo i przypisanie	*lvalue* =>> *wyrażenie*	10.2.1
Bitowe i oraz przypisanie	*lvalue* &= *wyrażenie*	10.2.1
Bitowe niewykluczające lub i przypisanie	*lvalue* |= *wyrażenie*	10.2.1
Bitowe lub wykluczające i przypisanie	*lvalue* ^= *wyrażenie*	10.2.1
Przecinek (do oznaczania sekwencji)	*wyrażenie*, *wyrażenie*	10.3.2

Na przykład `a+b*c` oznacza `a+(b*c)`, a nie `(a+b)*c`, ponieważ operator * ma wyższy priorytet niż +.

Operatory jednoargumentowe i operatory przypisania mają łączność prawostronną, a wszystkie pozostałe lewostronną. Na przykład `a=b=c` znaczy `a=(b=c)`, podczas gdy `a+b+c` znaczy `(a+b)+c`.

Kilku zasad gramatyki nie da się wyrazić za pomocą reguł priorytetów operatorów (zwanych także siłą wiązania) i łączności. Na przykład `a=b<c?d=e` znaczy `a=((b<c)?(d=e):(f=g))`, ale żeby o tym wiedzieć, trzeba zapoznać się z gramatyką (iso.A).

Zanim zastosowane zostaną zasady gramatyki, najpierw ze znaków układane są tokeny leksykalne. Do utworzenia tokenu wybierana jest najdłuższa możliwa sekwencja znaków. Na przykład `&&` jest pojedynczym operatorem, a nie dwoma operatorami `&`. Podobnie `a+++1` znaczy `(a++) + 1`. Czasami nazywa się to **regułą Maksa Muncha** (ang. *Max Munch rule*).

Zestawienie tokenów		
Klasa tokenów	**Przykład**	**Więcej informacji**
Identyfikator	`vector`, `foo_bar`, `x3`	6.3.3
Słowo kluczowe	`int`, `for`, `virtual`	6.3.3.1
Literał znakowy	`'x'`, `\n`, `'U'\UFADEFADE'`	6.2.3.2
Literał całkowitoliczbowy	`12`, `012`, `0x12`	6.2.4.1
Literał zmiennoprzecinkowy	`1.2`, `1.2e-3`, `1.2L`	6.2.5.1
Literał łańcuchowy	`"Witaj,!"`, `R"("Świecie"!)"`	7.3.2
Operator	`+=`, `%`, `<<`	10.3
Znak interpunkcyjny	`;`,`,`,`{`, `}`, `(`, `)`	
Notacja preprocesora	`#`, `##`	12.6

Białe znaki (np. spacja, tabulator i nowy wiersz) mogą być separatorami tokenów (np. `int count` to słowo kluczowe i identyfikator, a nie `intcount`), ale poza tym są ignorowane.

Wpisywanie niektórych znaków z podstawowego zestawu znaków źródła (6.1.2), takich jak `|`, na niektórych klawiaturach jest uciążliwe. Ponadto dla niektórych programistów używanie symboli typu `&&` czy `~` do oznaczania podstawowych operacji logicznych jest dziwne. Dlatego dostępne są też zastępcze formy w postaci słów kluczowych:

Alternatywne reprezentacje (iso.2.12)										
`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`	`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`
`&&`	`&=`	`&`	`|`	`~`	`!`	`!=`	`||`	`|=`	`^`	`^=`

Na przykład:

```
bool b = not (x or y) and z;
int x4 = ~ (x1 bitor x2) bitand x3;
```

Powyższy kod jest równoznaczny z poniższym:

```
bool b = !(x || y) && z;
int x4 = ~(x1 | x2) & x3;
```

Zwróć uwagę, że `and=` to nie to samo co `&=`. Jeśli preferujesz słowa kluczowe, to musisz pisać `and_eq`.

10.3.1. Wyniki

Typy wyników działań arytmetycznych są określone przez zestaw reguł zwanych „typowymi konwencjami arytmetycznymi" (10.5.3). Ogólnie otrzymuje się wynik „największego" typu argumentu. Na przykład: jeżeli operatorowi binarnemu zostanie przekazany argument zmiennoprzecinkowy, to obliczenia zostaną wykonane wg zasad arytmetyki zmiennoprzecinkowej i wynik będzie typu zmiennoprzecinkowego. Analogicznie: jeżeli argument jest typu `long`, obliczenia zostaną wykonane wg zasad arytmetyki dla typu `long` i wynik będzie typu `long`. Argumenty typów mniejszych niż `int` (np. `bool` i `char`) są konwertowane na `int` przed zastosowaniem operatora.

Operatory relacyjne — `==`, `<=` itd. — zwracają wyniki logiczne. Sposób działania i typ wyniku operatorów zdefiniowanych przez użytkownika są określone przez ich deklaracje (18.2).

Gdzie jest to logicznie możliwe, wynik operatora przyjmującego jako argument wartość lewostronną jest wartością lewostronną oznaczającą ten argument lewostronny. Na przykład:

```
void f(int x, int y)
{
    int j = x = y;          // wartość x=y jest wartością x po przypisaniu
    int*p = &++x;           // p wskazuje x
    int*q = &(x++);         // błąd: x++ nie jest wartością lewostronną (nie jest wartością przechowywaną w x)
    int*p2 = &(x>y?x:y); // adres większej wartości
    int& r = (x<y)?x:1;  // błąd: 1 nie jest wartością lewostronną
}
```

Jeśli drugi i trzeci argument operatora `?:` są wartościami lewostronnymi i mają ten sam typ, to wynik jest tego typu i jest wartością lewostronną. Taki sposób zachowywania wartości lewostronnych sprawia, że można bardziej elastycznie posługiwać się operatorami. Jest to szczególnie przydatne przy pisaniu kodu, który musi być tak samo wydajny niezależnie od tego, czy działa na typach wbudowanych, czy zdefiniowanych przez użytkownika (np. przy pisaniu szablonów albo programów generujących kod C++).

Wynik działania operatora `sizeof` jest typu całkowitoliczbowego bez znaku o nazwie `size_t` zdefiniowanego w nagłówku `<cstddef>`. Wynik odejmowania wskaźników jest typu całkowitoliczbowego o nazwie `ptrdiff_t` zdefiniowanego w nagłówku `<cstddef>`.

Implementacje nie muszą pilnować przepełnienia arytmetycznego i rzadko to robią. Na przykład:

```
void f()
{
    int i = 1;
    while (0 < i) ++i;
    cout << "i ma teraz wartość ujemną!" << i << '\n';
}
```

Ten kod w końcu dojdzie do momentu, w którym zwiększy wartość `i` poza zakres typu `int`. Nie jest określone, co się w takiej sytuacji zdarzy, ale najczęściej następuje „zawinięcie" wartości do liczby ujemnej (u mnie np. jest to `-2147483648`). Także wynik dzielenia przez zero jest niezdefiniowany, ale działanie takie najczęściej powoduje nagłe przerwanie pracy programu. W szczególności niedopełnienie, przepełnienie i dzielenie przez zero nie powodują zgłoszenia standardowych wyjątków (30.4.1.1).

10.3.2. Kolejność wykonywania działań

Kolejność wykonywania podwyrażeń w wyrażeniach jest niezdefiniowana. W szczególności nie można zakładać, że wyrażenie zostanie wykonane od lewej. Na przykład:

```
int x = f(2)+g(3); // nie wiadomo, czy najpierw zostanie wywołana funkcja f(), czy g()
```

Brak ograniczeń dotyczących kolejności wykonywania działań pozwala na generowanie lepszego kodu. Jednakże może on prowadzić także do powstawania niezdefiniowanych wyników. Na przykład:

```
int i = 1;
v[i] = i++; // wynik niezdefiniowany
```

Przypisanie w tym kodzie może zostać zinterpretowane jako `v[1]=1` lub `v[2]=2` albo spowodować jakieś dziwne operacje. Kompilator może ostrzegać o takich niejednoznacznościach. Niestety większość kompilatorów tego nie robi, więc uważaj, by nie pisać wyrażeń odczytujących lub zapisujących obiekt więcej niż raz, chyba że przy użyciu jednego operatora nie pozostawiającego wątpliwości, np. `++` albo `+=`, lub bezpośrednio określając sekwencję przy użyciu operatora `,` (przecinek), `&&` lub `||`.

Operatory `,` (przecinek), `&&` (logiczne i) i `||` (logiczne lub) gwarantują, że najpierw zostanie obliczona wartość lewego argumentu, a potem prawego. Na przykład `b=(a=2,a+1)` przypisuje wartość 3 do `b`. Przykłady użycia operatorów `&&` i `||` znajdują się w sekcji 10.3.3. Dla typów wbudowanych wartość drugiego argumentu operatora `&&` jest sprawdzana tylko wtedy, gdy pierwszy argument ma wartość `true`. Podobnie w przypadku operatora `||` wartość drugiego argumentu jest sprawdzana, gdy pierwszy ma wartość `false`. Technikę tę nazywa się czasami **skróconym określaniem wartości** (ang. *short-circuit evaluation*). Zwróć uwagę, że operator sekwencji (przecinek) różni się pod względem logicznym od przecinka oddzielającego argumenty wywołania funkcji. Na przykład:

```
f1(v[i],i++);    // dwa argumenty
f2((v[i],i++)); // jeden argument
```

Funkcja `f1()` jest wywoływana z dwoma argumentami, `v[i]` oraz `i++`, a kolejność wykonywania wyrażeń w argumentach jest niezdefiniowana, więc należy tego unikać. Zależność od kolejności wyrażeń w argumentach wywołania funkcji jest oznaką bardzo słabego stylu programowania, w którym nie da się przewidzieć zachowania programu. Wywołanie funkcji `f2()` ma tylko jeden argument, wyrażenie z przecinkiem (`v[i],i++`), które jest równoważne z `i++`. Jest to niejasne i tego też należy się wystrzegać.

Do grupowania wyrażeń można używać nawiasów. Na przykład `a*b/c` oznacza `(a*b)/c`, więc jeśli chodzi nam o `a*(b/c)`, musimy użyć nawiasu. Wyrażenie `a*(b/c)` może zostać wykonane `(a*b)/c`, tylko gdy zmiana ta nie ma znaczenia dla użytkownika. Różnice między `a*(b/c)` i `(a*b)/c` są szczególnie wyraźne w przypadku działań na liczbach zmiennoprzecinkowych i dlatego kompilator takie wyrażenia wykonuje dokładnie tak, jak są zapisane.

10.3.3. Priorytety operatorów

Zasady dotyczące priorytetów i łączności operatorów są zgodne z praktyką. Na przykład:

```
if (i<=0 || max<i) //...
```

znaczy „jeśli `i` jest mniejsze lub równe `0` lub `max` jest mniejsze od `i`". Jest to równoważne z:

```
if ( (i<=0) || (max<i) ) //...
```

Natomiast poniższy kod jest poprawny, ale bezsensowny:

```
if (i <= (0||max) < i) //...
```

Jeśli są jakiekolwiek wątpliwości co do kolejności wykonywania działań, należy używać nawiasów. Im bardziej skomplikowane wyrażenie, tym więcej nawiasów można w nim znaleźć, a mimo to skomplikowane wyrażenia są częstym źródłem rozmaitych błędów. Dlatego jeżeli czujesz potrzebę użycia nawiasu, to zastanów się, czy nie lepiej podzielić wyrażenie na części, wprowadzając dodatkową zmienną.

Czasami kolejność wykonywania operatorów nie jest oczywista. Na przykład:

```
if (i&mask == 0) // ups! wyrażenie == jako argument operatora &
```

To wyrażenie nie oznacza, że maska zostanie zastosowana do `i`, a potem nastąpi sprawdzenie, czy wynik jest równy `0`. Operator `==` ma wyższy priorytet od `&`, więc wyrażenie zostanie zinterpretowane jako `i&(mask==0)`. Na szczęście kompilator może łatwo wykryć takie błędy i ostrzec o nich użytkownika. W tym przypadku nawiasy są bardzo ważne:

```
if ((i&mask) == 0) //...
```

Należy zauważyć, że poniższe wyrażenie nie zostanie zinterpretowane w taki sposób, jakiego oczekiwałby matematyk:

```
if (0 <= x <= 99) //...
```

Ten poprawny kod zostanie zinterpretowany jako `(0<=x)<=99`, gdzie wynik pierwszego porównywania to `true` albo `false`. Ta logiczna wartość jest następnie niejawnie konwertowana na `1` lub `0` i porównywana z `99`, wynikiem czego jest `true`. Aby sprawdzić, czy `x` mieści się w przedziale od `0` do `99`, można napisać:

```
if (0<=x && x<=99) //...
```

Początkujący programiści często przypadkowo używają operatora przypisania (`=`) zamiast równości (`==`):

```
if (a = 7) // ups! przypisanie stałe w warunku
```

Jest to naturalne, bo w wielu językach znak `=` oznacza „równa się”. Dla kompilatora wykrycie takich błędów jest łatwe i wiele kompilatorów ostrzega o ich występowaniu. Nie zalecam udziwniania stylu programowania tylko po to, by zrekompensować niedobory kompilatorów w zakresie ostrzegania. W szczególności nie należy robić czegoś takiego:

```
if (7 == a) // próba ochrony przed niepoprawnym użyciem operatora =; niezalecane
```

10.3.4. Obiekty tymczasowe

Kompilator często musi tworzyć obiekty tymczasowe do przechowywania pośrednich wyników wyrażeń. Na przykład dla wyrażenia `v=x+y*z` trzeba gdzieś przechować wynik podwyrażenia `y*z` przed dodaniem go do `x`. W przypadku typów wbudowanych wszystko dzieje się w taki sposób, że **obiekt tymczasowy** jest niewidoczny dla użytkownika. Jednak w przypadku typów zdefiniowanych przez użytkownika zawierających zasoby wiedza o cyklu istnienia obiektu tymczasowego może być dla programisty ważna. Obiekt tymczasowy, jeśli nie jest związany z referencją ani wykorzystywany do inicjacji obiektu mającego nazwę, jest usuwany na końcu pełnego wyrażenia, dla którego został utworzony. **Pełne wyrażenie** to wyrażenie nie będące częścią innego wyrażenia.

Typ `string` z biblioteki standardowej zawiera składową `c_str()` (36.3) zwracającą wskaźnik w stylu języka C do zakończonej zerem tablicy znaków (2.2.5, 43.4). Ponadto operator + dla tego typu oznacza łączenie łańcuchów. Jeśli te bardzo przydatne do pracy z łańcuchami narzędzia zostaną użyte razem, mogą powodować różne trudne do określenia błędy. Na przykład:

```
void f(string& s1, string& s2, string& s3)
{
    const char*cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // użycie cs
    }
}
```

Pewnie w pierwszej chwili wykrzyknąłeś: „Ale tak się nie robi!”. Zgoda. Niemniej jednak ludzie piszą taki kod, więc warto wiedzieć, jak go interpretować.

Do przechowywania wartości wyrażenia `s1+s2` zostanie utworzony obiekt tymczasowy. Następnie z obiektu tego zostanie pobrany wskaźnik do łańcucha w stylu C. Później — na końcu wyrażenia — obiekt tymczasowy zostanie usunięty. A przecież łańcuch zwrócony przez funkcję `c_str()` został alokowany jako część obiektu tymczasowego przechowującego wynik podwyrażenia `s1+s2` i nie ma gwarancji, że ta pamięć będzie nadal istnieć po usunięciu tego obiektu tymczasowego. W konsekwencji `cs` wskazuje dealokowaną pamięć. Operacja wyjściowa `cout<<cs` może zadziałać zgodnie z oczekiwaniami, ale byłoby to czyste szczęście. Kompilator może wykryć wiele rodzajów takich problemów.

Problem z instrukcją `if` jest nieco bardziej subtelny. Warunek zadziała tak jak powinien, bo pełne wyrażenie, w którym tworzony jest obiekt tymczasowy przechowujący wynik `s2+s3`, samo jest tym warunkiem. Jednak obiekt ten zostanie usunięty przed rozpoczęciem wykonywania instrukcji kontrolowanej, przez co jakiekolwiek posługiwanie się zmienną `cs` w tym kodzie jest ryzykowne.

Należy zauważyć, że zarówno w tym przypadku, jak i wielu innych problemy z obiektami tymczasowymi są spowodowane użyciem wysokopoziomowego typu danych w niskopoziomowy sposób. Gdyby zastosowano prostszy styl programowania, powstałby bardziej zrozumiały kod i problemu by nie było. Na przykład:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;
    if (s.length()<8 && s[0]=='a') {
        // użycie s
    }
}
```

Obiektu tymczasowego można użyć jako inicjatora dla referencji stałej lub obiektu nazwanego. Na przykład:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;
    g(s,ss); // tu można używać s i ss
}
```

Ten kod jest w porządku. Obiekt tymczasowy jest usuwany w miejscu zakończenia zakresu „jego" referencji lub obiektu nazwanego. Pamiętaj, że zwrot referencji do zmiennej lokalnej jest błędem (12.1.4) oraz że obiektu tymczasowego nie można związać z niestałą referencją lewostronną (7.7).

Obiekt tymczasowy można też utworzyć w wyrażeniu bezpośrednio poprzez wywołanie konstruktora (11.5.1). Na przykład:

```
void f(Shape& s, int n, char ch)
{
    s.move(string{n,ch}); // utworzenie łańcucha z n kopii ch do przekazania do Shape::move()
    //...
}
```

Takie obiekty tymczasowe są usuwane wg tych samych zasad co obiekty tworzone niejawnie.

10.4. Wyrażenia stałe

W języku C++ funkcjonują dwa pokrewne pojęcia „stałej":

- `constexpr` — wartość wyznaczana w czasie kompilacji (2.2.3);
- `const` — oznacza brak możliwości modyfikacji obiektu w określonym zakresie (2.2.3, 7.5).

Najkrócej mówiąc, słowo kluczowe `constexpr` ma umożliwić gwarancję obliczania wartości podczas kompilacji, a `const` służy do zapewniania niezmienności interfejsów. Głównym tematem tego podrozdziału jest pierwsza rola: obliczanie wartości w czasie kompilacji.

Wyrażenie stałe to takie, którego wartość może obliczyć kompilator. Nie może zawierać wielkości nieznanych w czasie kompilacji ani powodujących efekty uboczne. W związku z tym na początku wyrażenia stałego musi znajdować się wartość całkowitoliczbowa (6.2.1), zmiennoprzecinkowa (6.2.5) lub enumerator (8.4). Można dodatkowo używać operatorów i funkcji `constexpr`, które tworzą wartości. Ponadto w niektórych formach wyrażeń stałych można używać adresów. Dla uproszczenia kwestię tę opisałem osobno w sekcji 10.4.5.

Jest wiele powodów, dla których należy użyć nazwanej stałej zamiast literału lub zmiennej:

1. Dzięki nazwanym stałym kod jest łatwiej zrozumieć i utrzymywać.
2. Zmienna może się zmienić (co zmusza programistę do bardziej ostrożnego posługiwania się niż w przypadku stałej).
3. Zasady języka wymagają używania wyrażeń stałych do określania rozmiarów tablic, w etykietach `case` oraz jako argumentów wartości szablonów.
4. Programiści układów wbudowanych lubią zapisywać niezmienne dane w pamięci tylko do odczytu, bo jest ona tańsza niż pamięć dynamiczna (przy uwzględnieniu ceny i zużycia energii) i często jest jej więcej. Ponadto dane znajdujące się w pamięci tylko do odczytu są odporne na większość awarii systemowych.
5. Jeśli obiekt jest inicjowany w czasie kompilacji, to nie ma mowy o wyścigu do niego w systemach wielowątkowych.
6. Czasami wykonanie jakichś obliczeń raz w czasie kompilacji jest o wiele lepszym rozwiązaniem pod względem wydajności niż powtarzanie ich miliony razy w czasie działania programu.

Powody [1], [2], [5] oraz częściowo [4] są logiczne. Nie używa się wyrażeń stałych tylko z powodu obsesyjnej dbałości o wydajność. Często jest tak, że wyrażenie stałe po prostu bardziej bezpośrednio reprezentuje wymogi systemu.

Wyrażenie stałe użyte jako część definicji elementu danych (celowo unikam tu słowa „zmienna") wyraża konieczność obliczenia wartości w czasie kompilacji. Jeżeli wartości inicjatora wyrażenia stałego nie można określić w czasie kompilacji, kompilator zgłosi błąd. Na przykład:

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1;      // błąd: inicjator nie jest wyrażeniem stałym
constexpr int x4 = x2;      // OK

void f()
{
    constexpr int y3 = x1; // błąd: inicjator nie jest wyrażeniem stałym
    constexpr int y4 = x2; // OK
    //...
}
```

Sprytny kompilator powinien wydedukować, że wartość `x1` w inicjatorze `x3` wynosi `7`. Jednak lepiej jest nie polegać na sprycie kompilatora. W dużych programach określenie wartości zmiennych w czasie kompilacji jest zazwyczaj bardzo trudne albo niemożliwe.

Wielką zaletą wyrażeń stałych jest ich ekspresja. Można w nich używać wartości całkowitych, zmiennoprzecinkowych oraz wyliczeniowych. Można używać wszystkich operatorów, które nie zmieniają stanu (np. `+`, `?:` i `[]`, ale nie `=` ani `++`). Używając funkcji `constexpr` (12.1.6) i typów literałowych (10.4.3), można zapewnić wysoki poziom bezpieczeństwa pod względem typów. Jest wręcz nadużyciem porównywanie tych możliwości z tym, co zwykle robi się przy użyciu makr (12.6).

Do wyboru możliwości w wyrażeniach stałych służy operator wyrażenia warunkowego `?:`. Na przykład można obliczyć pierwiastek kwadratowy liczby całkowitej w czasie kompilacji:

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
    return isqrt_helper(1,3,x)/2 - 1;
}

constexpr int s1 = isqrt(9); // s1 ma wartość 3
constexpr int s2 = isqrt(1234);
```

Najpierw sprawdzana jest wartość warunku operatora `?:`, a następnie zostaje wykonane wybrane wyrażenie. Niewybrane wyrażenie nie jest wykonywane i może nawet nie być stałe. Analogicznie argumenty operatorów `&&` i `||`, które nie są obliczane, również nie muszą być wyrażeniami stałymi. Jest to najbardziej przydatne w funkcjach `constexpr`, które czasami są używane jako wyrażenia stałe, a czasami nie.

10.4.1. Stałe symboliczne

Najważniejszym zastosowaniem stałych (`constexpr` i `const`) jest tworzenie symbolicznych nazw dla wartości. Nazw takich należy używać po to, by uniknąć tzw. magicznych liczb. Literały rozsiane po całym kodzie programu są jednym z najgorszych koszmarów programisty zajmującego się konserwacją kodu. Jeśli jakaś numeryczna stała, jak np. określenie granicy tablicy, jest powtórzona w kodzie, to utrudnia poprawianie tego kodu, bo trzeba znaleźć jej wszystkie wystąpienia i je zmienić. Dzięki użyciu stałej można wszystkie informacje mieć w jednym miejscu. Stałe liczbowe często reprezentują jakieś założenia o programie. Na przykład liczba `4` może reprezentować liczbę bajtów w typie całkowitoliczbowym, `128` może być liczbą znaków mieszczących się w buforze, a `6.24` może oznaczać kurs wymiany walut między duńską koroną a amerykańskim dolarem. Jeśli takie wartości zostałyby umieszczone w kodzie, to osoba go czytająca mogłaby się nie domyślić, co oznaczają. Ponadto wiele tego typu wartości z czasem się zmienia. Łatwo je przeoczyć i problem gotowy, gdy program przeniesie się na inną platformę albo zmieni się coś, od czego one zależą. Najłatwiej tego typu trudności uniknąć poprzez reprezentowanie założeń jako stałych symbolicznych opatrzonych dobrym komentarzem.

10.4.2. const w wyrażeniach stałych

Słowa kluczowego `const` najczęściej używa się do budowy interfejsów (7.5), ale to nie znaczy, że nie może ono zostać użyte do wyrażania wartości stałych. Na przykład:

```
const int x = 7;
const string s = "asdf";
const int y = sqrt(x);
```

Stała `const` zainicjowana wyrażeniem stałym może być używana w wyrażeniach stałych. Różnica między `const` a `constexpr` jest taka, że stałą `const` można zainicjować czymś, co nie jest wyrażeniem stałym. W takim przypadku stałej `const` nie można używać jako wyrażenia stałego. Na przykład:

```
constexpr int xx = x;     // OK
constexpr string ss = s; // błąd: s nie jest wyrażeniem stałym
constexpr int yy = y;     // błąd: sqrt(x) nie jest wyrażeniem stałym
```

Błędy w dwóch ostatnich wierszach są spowodowane tym, że `string` nie jest typem literałowym (10.4.3), a `sqrt()` nie jest funkcją `constexpr` (12.1.6).

Często przy definiowaniu prostych stałych lepszym wyborem jest użycie `constexpr`, ale słowo to jest dostępne dopiero od C++11, więc w starszym kodzie nie można go spotkać. W wielu przypadkach dobrym zastępnikiem dla stałych `const` są enumeratory (8.4).

10.4.3. Typy literałowe

W wyrażeniu stałym można użyć prostego typu zdefiniowanego przez użytkownika. Na przykład:

```
struct Point {
    int x,y,z;
    constexpr Point up(int d) { return {x,y,z+d}; }
    constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
    // ...
};
```

Klasa zawierająca konstruktor `constexpr` nazywa się **typem literałowym** (ang. *literal type*). Aby spełniać wszystkie wymagania stawiane wyrażeniom stałym, konstruktor musi być pusty, a wszystkie składowe muszą być inicjowane potencjalnie stałymi wyrażeniami. Na przykład:

```
constexpr Point origo {0,0};
constexpr int z = origo.x;

constexpr Point a[] = {
    origo, Point{1,1}, Point{2,2}, origo.move(3,3)
};
constexpr int x = a[1].x;       // x ma wartość 1

constexpr Point xy{0,sqrt(2)}; // błąd: sqrt(2) nie jest wyrażeniem stałym
```

Podkreślę, że można też tworzyć tablice `constexpr` oraz uzyskiwać dostęp do elementów tablic i składowych obiektów.

Oczywiście można definiować funkcje `constexpr` pobierające argumenty typów literałowych. Na przykład:

```
constexpr int square(int x)
{
    return x*x;
}

constexpr int radial_distance(Point p)
{
    return isqrt(square(p.x)+square(p.y)+square(p.z));
}

constexpr Point p1 {10,20,30}; // domyślny konstruktor jest constexpr
constexpr p2 {p1.up(20)};      // Point::up() jest constexpr
constexpr int dist = radial_distance(p2);
```

Użyłem typu `int` zamiast `double`, bo nie miałem pod ręką funkcji `constexpr` do obliczania pierwiastka kwadratowego z liczby zmiennoprzecinkowej.

Dzięki temu, że funkcja składowa zdefiniowana jako `constexpr` implikuje `const`, nie musiałem pisać czegoś takiego:

```
constexpr Point move(int dx, int dy) const { return {x+dx,y+dy}; }
```

10.4.4. Argumenty referencyjne

Przy używaniu stałych `constexpr` należy pamiętać, że w wyrażeniach tych chodzi głównie o wartości. Nie ma tu obiektów, które mogą zmienić wartość albo mieć efekty uboczne: `constexpr` to coś w rodzaju miniaturowego funkcyjnego języka programowania interpretowanego w czasie kompilacji. Pewnie zgadujesz, że w wyrażeniach stałych nie można używać referencji. To prawda, ale tylko do pewnego stopnia, bo referencje `const` odnoszą się do wartości i mogą być używane. Weźmy na przykład specjalizację ogólnego typu `complex<T>` do `complex<double>` z biblioteki standardowej:

```
template<> class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
```

```
    explicit constexpr complex(const complex<long double>&);

    constexpr double real(); // odczytuje część rzeczywistą
    void real(double);       // ustawia część rzeczywistą
    constexpr double imag(); // odczytuje część urojoną
    void imag(double);       // ustawia część urojoną

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    // ...
};
```

Oczywiście operacje modyfikujące obiekty, takie jak `=` i `+=`, nie mogą być wyrażeniami stałymi. Natomiast operacje tylko odczytujące obiekty, takie jak `real()` i `imag()`, mogą być `constexpr` i mogą być obliczane w czasie kompilacji, jeśli jest podane wyrażenie stałe. Interesującą składową jest konstruktor szablonu z innego typu `complex`:

```
constexpr complex<float> z1 {1,2};     // uwaga: <float> nie <double>
constexpr double re = z1.real();
constexpr double im = z1.imag();
constexpr complex<double> z2 {re,im}; // z2 jest kopią z1
constexpr complex<double> z3 {z1};     // z3 jest kopią z1
```

Ten konstruktor kopiujący działa, bo kompilator rozpoznaje, że referencja (`const complex<float>&`) odnosi się do wartości stałej, i używamy po prostu tej wartości (zamiast próbować wymyślać jakieś skomplikowane lub niemądre rozwiązania oparte na referencjach i wskaźnikach).

Dzięki typom literałowym możliwe jest używanie typów do wykonywania obliczeń w czasie kompilacji. Kiedyś obliczenia czasu kompilacji w języku C++ były ograniczone do używania wartości całkowitoliczbowych (i bez funkcji). Przymus kodowania wszystkich informacji w liczbach całkowitych sprawiał, że trzeba było pisać nadmiernie skomplikowany kod, w którym łatwo było się pomylić. Przykładami tego są niektóre zastosowania metaprogramowania szablonów (rozdział 28.). Niektórzy programiści po prostu woleli korzystać z obliczeń w czasie działania programu, zamiast walczyć z ograniczeniami języka.

10.4.5. Wyrażenia stałe adresowe

Adres statycznie alokowanego obiektu (6.4.2), np. zmiennej globalnej, jest stałą. Jednakże jego wartość jest przypisywana przez konsolidator, a nie kompilator, więc kompilator nie może znać wartości takiej stałej adresowej. To ogranicza wachlarz zastosowań wyrażeń stałych typów wskaźnikowych i referencyjnych. Na przykład:

```
constexpr const char*p1 = "asdf";
constexpr const char*p2 = p1;    // OK
constexpr const char*p2 = p1+2; // błąd: kompilator nie zna wartości p1
constexpr char c = p1[2];        // OK, c=='d'. Kompilator zna wartość wskazywaną przez p1
```

10.5. Niejawna konwersja typów

Typy całkowitoliczbowe i zmiennoprzecinkowe (6.2.1) można dowolnie mieszać w przypisaniach i wyrażeniach. Gdy to możliwe, ich wartości są konwertowane tak, aby nie nastąpiła utrata informacji. Niestety także pewne zawężające (powodujące utratę części informacji)

konwersje są wykonywane w sposób niejawny. Konwersja nie powoduje utraty informacji, jeżeli wartość przekonwertuje się na inny typ, a potem z powrotem na poprzedni i w wyniku tych działań otrzyma się identyczną wartość jak na początku. Jeśli te warunki nie są spełnione, to mamy do czynienia z **konwersją zawężającą** (10.5.2.6). W tej sekcji znajduje się opis zasad konwersji oraz związanych z nimi problemów i metod ich rozwiązywania.

10.5.1. Promocje

Niejawne konwersje zachowujące wartości nazywają się **promocjami**. Zanim zostanie wykonane działanie arytmetyczne, stosowana jest **promocja całkowitoliczbowa** w celu zamiany na `int` krótszych typów całkowitych. Analogicznie typ `float` jest zamieniany na `double` w wyniku **promocji zmiennoprzecinkowej**. Należy podkreślić, że w wyniku tych promocji *nie* następuje zamiana na typ `long` (chyba że argument jest typu `char16_t`, `char32_t`, `wchar_t` lub jest wyliczeniem przekraczającym rozmiar typu `int`) ani `long double`. Stanowi to odzwierciedlenie pierwotnego przeznaczenia promocji jeszcze w języku C, w którym miały za zadanie sprowadzać argumenty do „naturalnego” rozmiaru dla działań arytmetycznych.

Promocje całkowitoliczbowe to:

- Typy `char`, `signed char`, `unsigned char`, `short int` oraz `unsigned short int` są zamieniane na `int`, jeśli typ ten może zostać użyty do reprezentowania wszystkich wartości typu źródłowego. W przeciwnym razie następuje konwersja na typ `unsigned int`.
- Typy `char16_t`, `char32_t`, `wchar_t` (6.2.3) oraz zwykłe wyliczenia (8.4.2) są konwertowane na pierwszy z następujących typów, który może zostać użyty do reprezentowania ich wartości: `int`, `unsigned int`, `long`, `unsigned long`, `unsigned long long`.
- Pola bitowe (8.2.7) są konwertowane na typ `int`, jeżeli typ ten może reprezentować wszystkie wartości tego pola. W przeciwnym razie następuje konwersja na typ `unsigned int`, jeśli jest on odpowiedni. Jeżeli nie, to promocja całkowitoliczbowa nie jest wykonywana.
- Typ `bool` jest konwertowany na `int`. Wartość `false` zamienia się na `0`, a `true` na `1`.

Promocje są używane w ramach zwykłych konwersji arytmetycznych (10.5.3).

10.5.2. Konwersje

Typy podstawowe mogą być konwertowane niejawnie na wiele sposobów (iso.4). Uważam, że zezwala się na zbyt wiele konwersji. Na przykład:

```
void f(double d)
{
    char c = d; // uwaga: konwersja double na char
}
```

Pisząc kod, należy unikać wszelkich niezdefiniowanych zachowań i konwersji powodujących niepostrzeżoną utratę części informacji (zawężających).

Na szczęście o wielu budzących wątpliwości konwersjach może nas ostrzec kompilator.

Konwersje zawężające eliminuje składnia inicjacji z użyciem {} (6.3.5). Na przykład:

```
void f(double d)
{
    char c {d}; // błąd: konwersja double na char
}
```

Jeśli nie da się uniknąć potencjalnie zawężającej konwersji, należy rozważyć możliwość użycia konwersji kontrolowanej w czasie działania programu, np. `narrow_cast<>` (11.5).

10.5.2.1. Konwersje całkowitoliczbowe

Liczba typu całkowitego może zostać przekonwertowana na inny typ całkowity. Także wartość wyliczeniową (zwykłe wyliczenie) można przekonwertować na typ całkowitoliczbowy (8.4.2).

Jeżeli typ docelowy jest bez znaku (`unsigned`), to wynik konwersji zawiera tyle bitów z wartości źródłowej, ile się zmieści w wartości docelowej (w razie potrzeby usuwane są bardziej znaczące bity). Mówiąc dokładniej, wynik jest najmniejszą liczbą całkowitą bez znaku przystającą do źródłowej liczby całkowitej modulo `2` do `n`, gdzie `n` jest liczbą bitów użytych do reprezentacji tego typu bez znaku. Na przykład:

```
unsigned char uc = 1023; // binarna liczba 1111111111: uc ma wartość 11111111, czyli 255
```

Jeśli typ docelowy ma znak, to wartość pozostaje bez zmian, jeżeli może być reprezentowana przez ten typ. W przeciwnym przypadku wszystko zależy od implementacji:

```
signed char sc = 1023;    // zależy od implementacji
```

Możliwe wyniki to `127` i `-1` (6.2.3).

Wartość logiczna i wyliczeniowa (zwykłe wyliczenie) może być niejawnie przekonwertowana na swój odpowiednik całkowitoliczbowy (6.2.2, 8.4).

10.5.2.2. Konwersje zmiennoprzecinkowe

Wartość zmiennoprzecinkową można przekonwertować na inny typ zmiennoprzecinkowy. Jeżeli wartość źródłową można dokładnie przedstawić w typie docelowym, to wynikiem jest oryginalna wartość liczbowa. Jeśli wartość źródłowa mieści się między dwiema przystającymi wartościami docelowymi, to wynik jest jedną z nich. W pozostałych przypadkach wynik jest niezdefiniowany. Na przykład:

```
float f = FLT_MAX;    // największa wartość typu float
double d = f;         // OK: d == f

double d2 = DBL_MAX; // największa wartość typu double
float f2 = d2;        // niezdefiniowane, jeśli FLT_MAX<DBL_MAX

long double ld = d2; // OK: ld = d3
long double ld2 = numeric_limits<long double>::max();
double d3 = ld2;      // niezdefiniowane, jeśli sizeof(long double)>sizeof(double)
```

Definicje `DBL_MAX` i `FLT_MAX` znajdują się w nagłówku `<climits>`. Definicja `numeric_limits` znajduje się w nagłówku `<limits>` (40.2).

10.5.2.3. Konwersje wskaźników i referencji

Każdy wskaźnik na typ obiektowy może zostać niejawnie przekonwertowany na `void*` (7.2.1). Wskaźnik (referencja) do klasy pochodnej może być niejawnie przekonwertowany na wskaźnik (referencję) do dostępnej i jednoznacznej klasy bazowej (20.2). Wskaźnik do funkcji lub składowej nie może zostać niejawnie przekonwertowany na `void*`.

Wyrażenie stałe (10.4) o wartości `0` może zostać niejawnie przekonwertowane na wskaźnik pusty dowolnego typu wskaźnikowego. Podobnie wyrażenie stałe o wartości `0` może zostać niejawnie przekonwertowane na typ wskaźnika do składowej (20.6). Na przykład:

```
int*p = (1+2)*(2*(1−1)); // w porządku, chociaż dziwne
```

Lepiej użyć `nullptr` (7.2.2).

Wskaźnik `T*` można niejawnie przekonwertować na `const T*` (7.5). Podobnie referencję `T&` można niejawnie przekonwertować na `const T&`.

10.5.2.4. Konwersje wskaźników do składowych

Wskaźniki i referencje do składowych mogą być niejawnie konwertowane, a opis tych konwersji znajduje się w sekcji 20.6.3.

10.5.2.5. Konwersje wartości logicznych

Wartości wskaźnikowe, całkowitoliczbowe i zmiennoprzecinkowe mogą być niejawnie konwertowane na typ `bool` (6.2.2). Wartości niezerowe są zamieniane na `true`, a zero na `false`. Na przykład:

```
void f(int*p, int i)
{
    bool is_not_zero = p; // true, jeśli p!=0
    bool b2 = i;          // true, jeśli i!=0
    // ...
}
```

Zamiana wskaźników na wartości logiczne jest przydatna w warunkach, ale w innych miejscach może powodować nieporozumienia:

```
void fi(int);
void fb(bool);

void ff(int*p, int*q)
{
    if (p) do_something(*p);           // OK
    if (q!=nullptr) do_something(*q); // OK, ale rozwlekłe
    // ...
    fi(p);                             // błąd: nie można konwertować wskaźnika na int
    fb(p);                             // OK: konwersja wskaźnika na bool (niespodzianka!?)
}
```

Trzeba mieć nadzieję, że w przypadku `fb(p)` kompilator wyświetli ostrzeżenie.

10.5.2.6. Konwersje między typami zmiennoprzecinkowymi i całkowitymi

Przy konwersji wartości zmiennoprzecinkowej na całkowitą następuje utrata części ułamkowej. Innymi słowy, konwersja typu zmiennoprzecinkowego na całkowitoliczbowy powoduje skrócenie wartości. Na przykład wartość wyrażenia `int(1.6)` to `1`. Jeśli skróconej wartości nie można reprezentować przy użyciu typu docelowego, to wynik takiej operacji jest niezdefiniowany. Na przykład:

```
int i = 2.7;      // i ma wartość 2
char b = 2000.7; // niezdefiniowane dla 8-bitowych znaków: 2000 nie można reprezentować w 8-bitowej
                 // zmiennej typu char
```

Konwersje typów zmiennoprzecinkowych na całkowite są na tyle dokładne matematycznie, na ile pozwala sprzęt. Gdy wartość całkowita nie może być reprezentowana dokładnie jako wartość zmiennoprzecinkowa, następuje utrata precyzji. Na przykład:

```
int i = float(1234567890);
```

W komputerze, w którym typy `int` i `float` mają 32 bity, wartość `i` wyniesie `1234567936`.

Oczywiście najlepiej jest unikać niejawnych konwersji mogących spowodować utratę części informacji. Niektóre oczywiste niebezpieczeństwa tego typu, takie jak konwersja liczby zmiennoprzecinkowej na całkowitą albo `long int` na `char`, mogą być wykrywane przez kompilatory. Mimo to należy uważać, bo poleganie na kompilatorze w tej kwestii jest niepraktyczne. Jeśli sama ostrożność to za mało, programista może zastosować jawną konwersję kontrolowaną. Na przykład:

```
char checked_cast(int i)
{
    char c = i; // ostrzeżenie: nieprzenośne (10.5.2.1)
    if (i != c) throw std::runtime_error{"Nieudana konwersja int na char"};
    return c;
}

void my_code(int i)
{
    char c = checked_cast(i);
    // ...
}
```

Bardziej ogólne podejście do wyrażania konwersji kontrolowanych jest opisane w sekcji 25.2.5.1.

Aby skrócenie wartości nie niweczyło przenośności, należy użyć `numeric_limits` (40.2). W inicjacjach skrócenia można wyeliminować, stosując notację {} (6.3.5).

10.5.3. Typowe konwersje arytmetyczne

Opisane poniżej konwersje są wykonywane na argumentach operatorów binarnych w celu sprowadzenia ich do wspólnego typu, który następnie jest używany jako typ wyniku działania:

1. Jeśli jeden z argumentów jest typu `long double`, drugi również jest konwertowany na `long double`.
   - W przeciwnym przypadku, jeśli jeden argument jest typu `double`, drugi również jest konwertowany na `double`.
   - W przeciwnym przypadku, jeśli jeden argument jest typu `float`, drugi również jest konwertowany na `float`.
   - W przeciwnym przypadku na obu argumentach jest wykonywana operacja promocji całkowitoliczbowej (10.5.1).
2. Jeżeli jeden z argumentów jest typu `unsigned long long`, to drugi zostaje również przekonwertowany na typ `unsigned long long`.
   - W przeciwnym przypadku, jeśli jeden argument jest typu `long long int`, a drugi `unsigned long int`, to jeżeli typ `long long int` może reprezentować wszystkie wartości typu `unsigned long int`, typ `unsigned long int` jest konwertowany na `long long int`. W przeciwnym przypadku oba argumenty są konwertowane na typ `unsigned long long int`. W przeciwnym przypadku, jeżeli jeden z argumentów jest typu `unsigned long long`, drugi również zostaje przekonwertowany na `unsigned long long`.

- Jeżeli jeden z argumentów jest typu `long int`, drugi jest `unsigned int`, a typ `long int` może reprezentować wszystkie wartości typu `unsigned int`, to wartość typu `unsigned int` jest konwertowana na `long int`. W przeciwnym przypadku oba argumenty są konwertowane na typ `unsigned long int`.
- Jeśli jeden z argumentów jest typu `long`, drugi również jest konwertowany na `long`.
- Jeśli jeden z argumentów jest typu `unsigned`, drugi również jest konwertowany na `unsigned`.
- W pozostałych przypadkach oba argumenty są typu `int`.

Z tych zasad wynika, że wynik konwersji wartości całkowitej bez znaku na wartość ze znakiem ewentualnie większego rozmiaru jest zależny od implementacji. Jest to kolejny powód, aby nie mieszać liczb całkowitych ze znakiem z liczbami całkowitymi bez znaku.

10.6. Rady

1. Wybieraj narzędzia z biblioteki standardowej zamiast innych bibliotek i własnego kodu — 10.2.8.
2. Pobieraj znaki na wejściu tylko wtedy, gdy nie możesz tego zrobić inaczej — 10.2.3.
3. Przy wczytywaniu danych zawsze zabezpiecz się przed możliwością pojawienia się niepoprawnych danych — 10.2.3.
4. Stosuj odpowiednie abstrakcje (klasy, algorytmy itd.) zamiast bezpośredniego używania narzędzi języka (np. `int` i instrukcji) — 10.2.8.
5. Unikaj używania skomplikowanych wyrażeń — 10.3.3.
6. W razie wątpliwości co do kolejności wykonywania działań używaj nawiasów — 10.3.3.
7. Unikaj wyrażeń z nieokreśloną kolejnością wykonywania działań — 10.3.2.
8. Unikaj konwersji zawężających — 10.5.2.
9. Definiuj stałe symboliczne, aby wyeliminować „stałe magiczne” — 10.4.1.
10. Unikaj konwersji zawężających — 10.5.2.

Skorowidz

Wyróżnia się dwa rodzaje wiedzy. Można wiedzieć coś na dany temat albo można wiedzieć, gdzie szukać o tym informacji
— Samuel Johnson

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Poznaj nowości C++ 11!

C++ dzielnie broni swojej pozycji na rynku języków programowania. Pomimo silnego naporu języka Java oraz platformy .NET wciąż jest niezastąpiony w wielu dziedzinach. Jeżeli tylko wymagana jest najwyższa wydajność, dostęp do sprzętu oraz przewidywalny czas wykonania, programiści najczęściej wybierają właśnie język C++. Ostatnia wersja standardu — oznaczona numerem 11 — pozwala na jeszcze łatwiejsze pisanie kodu oraz tworzenie szybszych i wydajniejszych programów.

Najnowsze wydanie tej cenionej książki zostało poprawione i uzupełnione o nowości z tej właśnie wersji standardu języka C++. Dowiesz się, jak korzystać ze wskaźników, liczb losowych oraz udoskonalonych kontenerów. Ponadto poznasz najlepsze zastosowanie wyrażeń lambda czy szablonów. Oprócz omówienia nowości znajdziesz tu również szczegółowy przegląd klasycznych elementów języka C++. Pętle, zmienne, tablice, instrukcje warunkowe — to tylko niektóre z omawianych zagadnień. Książka stanowi doskonały podręcznik dla początkujących programistów i świetne rozwinięcie dla programujących w języku C++ na co dzień. Przekonaj się, jak łatwo i przyjemnie możesz opanować ten popularny język oprogramowania.

Dzięki tej książce:

- poznasz nowości wprowadzone w C++ 11
- zaznajomisz się z elementami biblioteki standardowej
- opanujesz podstawowy model pamięci języka C++
- zrozumiesz model pamięci języka C++

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
INFORMATYKA W NAJLEPSZYM WYDANIU

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8329-6
Cena: 199,00 zł
Addison-Wesley
Pearson Education
ALWAYS LEARNING
PEARSON