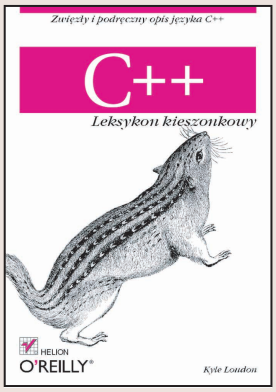
C++. Leksykon kieszonkowy



Autor: Kyle Loudon
Tłumaczenie: Przemysław Steć
ISBN: 83-7361-242-4
Tytuł oryginału: C++ Pocket Reference
Format: B5, stron: 164

C++ jest złożonym językiem o wielu subtelnych aspektach. Książka „C++. Leksykon kieszonkowy" umożliwia programistom C++ szybkie sprawdzenie sposobu użycia oraz składni najczęściej używanych konstrukcji tego języka. Na jej stronach umieszczono możliwie jak najwięcej informacji, a jej niewielki rozmiar pozwala mieć ją zawsze przy sobie. „C++. Leksykon kieszonkowy" opisuje następujące zagadnienia:

- Instrukcje języka C++ oraz dyrektywy preprocesora C++
- Przestrzenie nazw oraz zakres w C++
- Programowanie szablonowe oraz obsługa wyjątków
- Klasy oraz dziedziczenie
- Typy języka C++ oraz ich konwersje, w tym rzutowanie typów w C++

Poza spełnianiem funkcji podręcznego leksykonu dla programistów C++, książka „C++. Leksykon kieszonkowy" będzie także pomocna dla programistów języków Java oraz C, którzy przesiadają się na C++ lub tych, którzy w C++ programują od czasu do czasu. Te trzy języki są często mylące podobne. Książka ta umożliwia programistom zaznajomionym z językiem C lub Java szybkie orientowanie się w sposobie implementacji w C++.
„C++. Leksykon kieszonkowy" jest jednym z najbardziej zwięzłych i podręcznych opisów języka C++ dostępnych na rynku.

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

Spis treści

C++
Leksykon kieszonkowy

Wprowadzenie

„C++. Leksykon kieszonkowy" stanowi skrócony opis języka programowania C++ w postaci zdefiniowanej przez międzynarodowy standard INCITS/ISO/IEC 14882-1998. Książka składa się z krótkich rozdziałów, z których każdy podzielony jest na tematy. Wiele zagadnień zilustrowano precyzyjnymi, kanonicznymi przykładami.

C++ jest bardzo obszernym językiem, który trudno opisać w formie leksykonu kieszonkowego. W rezultacie książka ta poświęcona jest niemal wyłącznie prezentacji języka. Dostępne są inne pozycje z serii O'Reilly, które opisują Bibliotekę Standardową C++, będącą samą w sobie obszernym tematem. Biblioteka Standardowa C++ zawiera wszystkie funkcje i możliwości Biblioteki Standardowej C, a także wiele nowych, takich jak Standardowa Biblioteka Szablonów (STL) czy strumienie I/O.

Książka napisana została dla czytelników o zróżnicowanym stopniu zaawansowania i doświadczenia w programowaniu w C++. Dla doświadczonych programistów C++ będzie wyjątkowo skondensowanym leksykonem najczęściej wykorzystywanych możliwości języka. Nowicjusze powinni najpierw zapoznać się z jakimś wprowadzeniem do języka, a później wrócić do tego leksykonu, aby poznać określone zagadnienia.

Konwencje typograficzne

W książce stosowane są następujące konwencje typograficzne:

`Czcionka o stałej szerokości`
: używana do oznaczania przykładowych fragmentów kodu, poleceń, słów kluczowych oraz nazw typów, zmiennych, funkcji i klas.

`Kursywa o stałej szerokości`
: używana do oznaczenia zastępowalnych parametrów.

Kursywa
: używana do nazw plików oraz elementów wyróżnianych w tekście.

Podziękowania

Podziękowania należą się przede wszystkim Jonathanowi Gennickowi, redaktorowi w wydawnictwie O'Reilly, za wsparcie oraz wskazówki dotyczące tej książki. Dziękuję również Uwe Schnitkerowi, Danny'emu Kalevowi oraz Ronowi Passeriniemu za to, że poświęcili czas na lekturę i poczynili komentarze do wczesnego szkicu książki.

Zgodność z językiem C

Z pewnymi drobnymi wyjątkami język C++ stworzony został jako rozszerzenie języka C. Oznacza to, że właściwie napisane programy w języku C na ogół skompilują się i będą działać jako programy C++ (większość niezgodności bierze się ze ściślejszej kontroli typów realizowanej w języku C++). Programy w języku C++ wyglądają zazwyczaj podobnie pod względem składniowym do programów w języku C i wykorzystują wiele pierwotnych możliwości i funkcji języka C.

Nie należy jednak dać się zwieść podobieństwom pomiędzy C oraz C++ i uznać, że C++ jest jedynie banalną pochodną języka C. W rzeczywistości bowiem jest to rozbudowany język, który wzbogaca C o bardzo istotne elementy, a mianowicie o:

- programowanie obiektowe,
- programowanie generyczne wykorzystujące szablony,
- przestrzenie nazw,
- funkcje typu `inline` (czyli wstawiane w miejscu wywołania),
- przeciążanie operatorów i funkcji,
- lepsze mechanizmy zarządzania pamięcią,
- referencje,
- bezpieczniejsze formy rzutowania,
- informację o typie czasu wykonywania,
- obsługę wyjątków,
- rozszerzoną Bibliotekę Standardową.

Struktura programu

Na najwyższym poziomie program w języku C++ składa się z jednego lub wielu *plików źródłowych* zawierających kod źródłowy C++. We wszystkich tych plikach łącznie zdefiniowane jest dokładnie jedno miejsce rozpoczęcia wykonywania programu i być może wiele miejsc jego zakończenia.

W plikach źródłowych C++ często importowany jest, czyli *dołączany* (ang. included), dodatkowy kod źródłowy znajdujący się w tzw. *plikach nagłówkowych* (ang. header files). Za dołączenie kodu z tych plików przed kompilacją każdego pliku źródłowego

odpowiedzialny jest preprocesor języka C++. Jednocześnie preprocesor, poprzez zastosowanie tzw. *dyrektyw preprocesora*, może wykonać także inne operacje. Plik źródłowy po przetworzeniu przez preprocesor zwany jest *jednostką translacji* (ang. translation unit).

Miejsce rozpoczęcia wykonywania

Oznaczonym początkiem programu C++, który programista musi zdefiniować, jest funkcja `main`. W standardowej postaci funkcja ta może nie przyjmować żadnych lub przyjmować dwa argumenty podawane przez system operacyjny przy uruchomieniu programu, chociaż wiele implementacji języka C++ dopuszcza także inne, dodatkowe parametry. Typem zwracanym funkcji `main` jest `int`. Na przykład:

```
int main()
int main(int argc, char *argv[])
```

Parametr `argc` określa liczbę argumentów podanych w wierszu polecenia, a `argv` jest tablicą łańcuchów zakończonych znakiem pustym (`\0`) — format języka C — zawierającą argumenty w kolejności ich występowania. Nazwa pliku wykonywalnego zapisana jest jako `argv[0]` i może być, lecz nie musi, poprzedzona pełną ścieżką. Wartość elementu `argv[argc]` wynosi 0.

Poniższy listing demonstruje kod funkcji `main` prostego programu, który zachęca użytkownika do przeprowadzenia pewnych działań na koncie bankowym:

```
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

#include "Account.h"
```

```
int main(int argc, char *argv[])
{
   Account        account(0.0);
   char           action;
   double         amount;

   if (argc > 1)
      account.deposit(atof(argv[1]));

   while (true)
   {
      cout << "Balance is "
           << account.getBalance()
           << endl;

      cout << "Enter d, w, or q: ";
      cin  >> action;

      switch (action)
      {
         case 'd':
            cout << "Enter deposit: ";
            cin  >> amount;
            account.deposit(amount);
            break;

         case 'w':
            cout << "Enter withdrawal: ";
            cin  >> amount;
            account.withdraw(amount);
            break;

         case 'q':
            exit(0);

         default:
            cout << "Bad command" << endl;
      }
```

```
    }

    return 0;
}
```

Klasa reprezentująca konto bankowe zdefiniowana jest w późniejszym przykładzie. Na konto wpłacona zostaje kwota podana w wierszu polecenia przy uruchomieniu programu. Do konwersji argumentu wiersza polecenia z łańcucha na typ `double` służy funkcja `atof` (z Biblioteki Standardowej C++).

Zakończenie wykonywania

Wykonywanie programu w języku C++ kończy się w momencie opuszczenia funkcji `main` na skutek wykonania instrukcji `return`. Wartość, która zostaje zwrócona, przekazywana jest z powrotem do systemu operacyjnego i staje się wartością zwróconą danego pliku wykonywalnego. Jeśli w treści funkcji `main` nie występuje instrukcja `return`, po wykonaniu wszystkich instrukcji funkcji `main` zwrócona zostaje niejawnie wartość 0. Wykonywanie programu zakończyć można również przez wywołanie funkcji `exit` (z Biblioteki Standardowej C++), która jako argument przyjmuje wartość zwracaną pliku wykonywalnego.

Pliki nagłówkowe

Pliki nagłówkowe zawierają kod źródłowy, który ma zostać dołączony do wielu różnych plików. Posiadają zwykle rozszerzenie *.h*. W pliku nagłówkowym umieszczamy każdy kod, który ma zostać dołączony w wielu miejscach. Plik nagłówkowy nie powinien nigdy zawierać:

- definicji zmiennych oraz statycznych danych składowych (różnica pomiędzy deklaracjami a definicjami wyjaśniona jest w części „Deklaracje"),

- definicji funkcji, za wyjątkiem tych zdefiniowanych jako funkcje szablonowe lub funkcje typu `inline`,
- anonimowych przestrzeni nazw.

UWAGA

Pliki nagłówkowe w Bibliotece Standardowej C++ nie posiadają rozszerzenia *.h* — nie mają w ogóle rozszerzenia.

Często dla każdej ważniejszej klasy, którą definiujemy, tworzymy jeden plik nagłówkowy. Klasa `Account` na przykład zdefiniowana jest w pliku nagłówkowym *Account.h*, którego zawartość przedstawiono poniżej. Oczywiście pliki nagłówkowe służą także do innych celów, a ponadto nie wszystkie definicje klas muszą zostać umieszczone w pliku nagłówkowym (np. klasy pomocnicze definiowane są po prostu wewnątrz pliku źródłowego, w którym są wykorzystywane).

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
                    Account(double b);

   void             deposit(double amt);
   void             withdraw(double amt);
   double           getBalance() const;

private:
   double           balance;
};

#endif
```

Implementacja tej klasy znajduje się w pliku *Account.cpp*. Plik nagłówkowy dołączyć można wewnątrz innego pliku za pomocą dyrektywy preprocesora `#include` (patrz punkt: „Dyrektywy preprocesora").

Ponieważ pliki nagłówkowe dołączane są często przez inne pliki nagłówkowe, trzeba uważać, aby nie dołączyć kilka razy tego samego pliku nagłówkowego, co spowodować może błędy kompilacji. Aby uniknąć takiej sytuacji, treść plików nagłówkowych zwyczajowo umieszcza się wewnątrz dyrektyw preprocesora `#ifndef`, `#define` oraz `#endif`, jak zrobiono to w powyższym przykładzie.

Taka metoda „opakowania" pliku nagłówkowego wymusza na preprocesorze konieczność przetestowania identyfikatora. Jeśli identyfikator nie jest zdefiniowany, preprocesor definiuje go i przetwarza zawartość danego pliku. Biorąc pod uwagę omawiany przykład, zawartość pliku *Account.h* przetwarzana jest tylko wtedy, gdy nie jest zdefiniowany identyfikator `ACCOUNT_H`, a pierwszą operacją wykonywaną podczas tego przetwarzania jest zdefiniowanie identyfikatora `ACCOUNT_H` w celu zapewnienia, aby dany plik nagłówkowy nie został dołączony po raz drugi. W celu zagwarantowania niepowtarzalności jako identyfikator używany jest zwykle symbol *X_H*, gdzie *X* jest nazwą pliku nagłówkowego bez rozszerzenia.

Pliki źródłowe

Pliki źródłowe C++ posiadają zwykle rozszerzenie *.cpp* i zawierają kod źródłowy języka C++. Podczas kompilacji kompilator tłumaczy normalnie pliki źródłowe na *pliki wynikowe* (ang. object files), które często posiadają rozszerzenie *.obj* lub *.o*. Konsolidator łączy następnie pliki wynikowe w końcowy plik wykonywalny lub bibliotekę.

Często, choć oczywiście nie zawsze, dla każdej ważniejszej klasy, którą implementujemy, tworzymy jeden plik źródłowy. Na przykład implementacja klasy `Account` umieszczona jest w pliku nagłówkowym *Account.cpp*, którego zawartość przedstawiono poniżej. Pliki źródłowe zawierają często więcej kodu niż tylko implementację pojedynczej klasy.

```
#include "Account.h"

Account::Account(double b)
{
   balance = b;
}

void Account::deposit(double amt)
{
   balance += amt;
}

void Account::withdraw(double amt)
{
   balance -= amt;
}

double Account::getBalance() const
{
   return balance;
}
```

Dyrektywy preprocesora

Preprocesor języka C++ wykorzystać można do przeprowadzenia szeregu pożytecznych operacji sterowanych przez kilka dyrektyw. Każda dyrektywa rozpoczyna się od znaku # jako pierwszego znaku w wierszu, który nie jest znakiem odstępu. Pojedynczą dyrektywę można zapisać w wielu wierszach, wstawiając lewy ukośnik (\) na końcu wierszy pośrednich.

#define

Dyrektywa `#define` powoduje zastąpienie danego identyfikatora tekstem, który został po nim określony, we wszystkich miejscach występowania tego identyfikatora w pliku źródłowym. Na przykład:

```
#define          BUFFER_SIZE 80;
char             buffer[BUFFER_SIZE};
```

Jeśli po identyfikatorze nie podamy żadnego tekstu, wówczas preprocesor zdefiniuje ten identyfikator w taki sposób, aby każde sprawdzenie istnienia jego definicji dało w wyniku wartość `true`, a wszystkie wystąpienia identyfikatora w kodzie źródłowym zostały zastąpione tekstem pustym (czyli zostały usunięte). Z takim przypadkiem mieliśmy do czynienia wcześniej przy definiowaniu identyfikatora `ACCOUNT_H`.

UWAGA

W języku C++, zamiast stosowania dyrektywy `#define` bardziej wskazane jest definiowanie danych stałych jako wyliczeń oraz zmiennych i danych składowych deklarowanych z zastosowaniem słów kluczowych `const` lub `static const`.

Dyrektywa `#define` może również przyjmować argumenty w celu zastąpienia makroinstrukcji w tekście. Na przykład:

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))

int              x = 5, y = 10, z;

z = MIN(x, y);   // Ta instrukcja przypisuje zmiennej
                    ↳z wartość 5
```

W celu uniknięcia nieoczekiwanych problemów związanych z pierwszeństwem operatorów, parametry występujące w tekście umieszczać należy w nawiasach, jak pokazano powyżej.

UWAGA

W języku C++ bardziej wskazane jest stosowanie szablonów oraz funkcji typu `inline` zamiast makroinstrukcji. Użycie szablonów i funkcji typu `inline` eliminuje niespodziewane efekty powodowane przez makra, jak np. dwukrotna inkrementacja zmiennej `x` przez makro `MIN(x++,y)`, w przypadku gdy parametr `a` jest mniejszy od `b` (przy zastępowaniu makra jako pierwszy parametr traktowane jest wyrażenie `x++`, a nie wynik operacji `x++`).

#undef

Dyrektywa `#undef` usuwa definicję identyfikatora tak, aby sprawdzenie jego istnienia dawało w wyniku wartość `false`. Na przykład:

```
#undef LOGGING_ENABLED
```

#ifdef, #ifndef, #else, #endif

Dyrektywy `#ifdef`, `#ifndef`, `#else`, `#endif` stosowane są łącznie. Dyrektywa `#ifdef` powoduje dołączenie przez preprocesor innego kodu w zależności od istnienia lub braku definicji danego identyfikatora. Na przykład:

```
#ifdef LOGGING_ENABLED
cout << "Logging is enabled" << endl;
#else
cout << "Logging is disabled" << endl;
#endif
```

Użycie dyrektywy `#else` jest opcjonalne. Dyrektywa `#ifndef` działa podobnie, lecz powoduje dołączenie występującego po niej kodu tylko wtedy, gdy brakuje definicji danego identyfikatora.

#if, #elif, #else, #endif

Dyrektywy `#if`, `#elif`, `#else`, `#endif`, podobnie jak dyrektywy `#ifdef`, stosowane są łącznie. Powodują one dołączenie lub wykluczenie przez preprocesor kodu w zależności od prawdziwości danego wyrażenia. Na przykład:

```
#if (LOGGING_LEVEL == LOGGING_MIN && \
    LOGGING_FLAG)
cout << "Logging is minimal" << endl;
#elif (LOGGING_LEVEL == LOGGING_MAX && \
    LOGGING_FLAG)
#elif LOGGING_FLAG
cout << "Logging is standard" << endl;
#endif
```

Dyrektywa `#elif` (else-if) służy do łańcuchowego łączenia szeregu testów, jak pokazano powyżej.

#include

Dyrektywa `#include` powoduje dołączenie przez preprocesor innego pliku, zazwyczaj pliku nagłówkowego. Nazwy standardowych plików nagłówkowych ujmujemy w nawiasy ostre, a plików nagłówkowych zdefiniowanych przez użytkownika — w znaki cudzysłowu. Na przykład:

```
#include <iostream>
#include "Account.h"
```

W zależności od sposobu podania nazwy pliku nagłówkowego preprocesor przeszukiwał będzie inne ścieżki. To, które ścieżki zostaną przeszukane, zależy od systemu.

#error

Dyrektywa `#error` powoduje przerwanie kompilacji i wyświetlenie określonego tekstu. Na przykład:

```
#ifdef LOGGING_ENABLED
#error Logging should not be enabled
#endif
```

#line

Dyrektywa `#line` sprawia, że preprocesor modyfikuje bieżący numer wiersza zapisywany wewnętrznie przez kompilator podczas kompilacji w makrodefinicji `__LINE__`. Na przykład:

```
#line 100
```

Po numerze wiersza można opcjonalnie podać nazwę pliku ujętą w znaki cudzysłowu. Powoduje to zmianę nazwy pliku wewnętrznie zapisywanej przez kompilator w makrodefinicji `__FILE__`. Na przykład:

```
#line 100 "NewName.cpp"
```

#pragma

Niektóre operacje, które preprocesor może wykonać, są specyficzne dla implementacji. Dyrektywa `#pragma` umożliwia sterowanie tymi operacjami przez podanie dyrektywy wraz z dowolnymi parametrami w postaci wymaganej przez tę dyrektywę. Na przykład:

```
#ifdef LOGGING_ENABLED
#pragma message("Logging enabled")
#endif
```

W przypadku kompilatora Microsoft Visual C++ 6.0 dyrektywa `message` nakazuje preprocesorowi wyświetlenie komunikatu podczas kompilacji w momencie napotkania zawierającego ją wiersza. Dyrektywa ta wymaga jednego parametru — komunikatu, który ma zostać wyświetlony. Jego tekst, ujęty w znaki cudzysłowu, umieszczany jest w nawiasach.

Makra preprocesora

Preprocesor języka C++ definiuje kilka makrodefinicji służących do wstawiania informacji do pliku źródłowego podczas kompilacji. Każdy identyfikator makra rozpoczyna się i kończy dwoma znakami podkreślenia, za wyjątkiem makra `__cplusplus`, które nie posiada kończących znaków podkreślenia.

`__LINE__`
: Rozwija się do bieżącego numeru wiersza kompilowanego pliku źródłowego.

`__FILE__`
: Rozwija się do nazwy kompilowanego pliku źródłowego.

`__DATE__`
: Rozwija się do daty kompilacji.

`__TIME__`
: Rozwija się do godziny kompilacji.

`__TIMESTAMP__`
: Rozwija się do daty i godziny kompilacji.

`__STDC__`
: Będzie zdefiniowane, jeśli kompilator jest w pełni zgodny ze standardem ANSI C.

`__cplusplus`
: Będzie zdefiniowane, jeśli kompilowany program jest programem w języku C++. Sposób, w jaki kompilator ustala, czy dany program jest programem C++, zależy od kompilatora. Może istnieć konieczność ustawienia odpowiedniej opcji kompilatora lub kompilator może brać pod uwagę rozszerzenie pliku źródłowego.