C++

Podróż po języku dla zaawansowanych

Wydanie II

Bjarne Stroustrup

Helion

Tytuł oryginału: A Tour of C++ (2nd Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-5424-1

Helion SA
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/cppoz2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

1

Podstawy

Na początek zabijmy wszystkich językowych mądrali.
— Henryk VI, część II

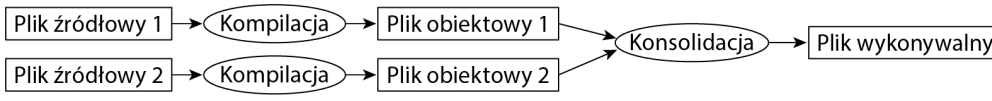
- Wprowadzenie
- Programy
  Witaj, świecie!
- Funkcje
- Typy, zmienne i arytmetyka
  Działania arytmetyczne. Inicjalizacja
- Zakres i cykl istnienia
- Stałe
- Wskaźniki, tablice i referencje
  Wskaźnik pusty
- Testy
- Mapowanie sprzętowe
  Przypisanie. Inicjalizacja
- Porady

1.1 Wprowadzenie

W tym rozdziale nieformalnie przedstawiam notację języka C++, zastosowany w nim model pamięci i obliczeniowy oraz podstawowe narzędzia organizacyjne programu. Wszystkie te elementy służą do programowania w stylu najbardziej znanym z języka C i czasami nazywanym **programowaniem proceduralnym**.

1.2 Programy

C++ to język wymagający kompilacji, co oznacza, że aby uruchomić program, jego kod źródłowy w postaci tekstu należy wprowadzić do kompilatora. Ten wygeneruje z nich pliki obiektowe, które następnie przekaże do konsolidatora w celu wygenerowania programu wykonywalnego. Typowy program w języku C++ składa się z wielu plików z kodem źródłowym (dla uproszczenia zwanych **plikami źródłowymi**).



Program wykonywalny można uruchamiać na określonej platformie sprzętowo-systemowej. Nie można na przykład uruchomić pliku z komputera Mac na komputerze z systemem Windows. Kiedy jest mowa o przenośności programów w języku C++, to najczęściej dotyczy ona kodu źródłowego, co znaczy, że kod ten można skompilować i uruchomić w różnych systemach.

W standardzie ISO elementy języka C++ dzielą się na dwie grupy:

- **Składniki rdzenne** — na przykład typy wbudowane (np. `char` i `int`) oraz pętle (np. instrukcje `for` i `while`).
- **Składniki biblioteki standardowej** — na przykład kontenery (`vector` i `map`) czy operacje wejścia i wyjścia (np. `<<` i `getline()`).

Komponenty biblioteki standardowej są napisane przy użyciu całkiem zwyczajnych konstrukcji języka C++, które są dostępne w każdej jego implementacji. Oznacza to, że bibliotekę C++ można zaimplementować w C++ i istotnie tak zrobiono (nie licząc bardzo drobnych wyjątków użycia kodu maszynowego w takich przypadkach jak przełączanie kontekstu wątków). Z tego wynika, że C++ to bardzo ekspresywny język, który doskonale sprawdza się nawet przy programowaniu najbardziej wymagających systemów.

C++ to język typizowany statycznie, to znaczy każdy element (np. obiekt, wartość, nazwa i wyrażenie) musi być znany kompilatorowi w chwili jego użycia. Typ obiektu określa zestaw operacji, jakie można na nim wykonywać.

1.2.1 Witaj, świecie!

Najmniejszy możliwy program w języku C++ wygląda tak:

```
int main() { } // najmniejszy możliwy program w języku C++
```

Jest to definicja funkcji o nazwie `main`, która nie przyjmuje żadnych argumentów i nic nie robi.

Klamra, {}, w języku C++ służy do grupowania. W tym przypadku została użyta do wyznaczenia początku i końca treści właściwej funkcji. Podwójny ukośnik, //, oznacza początek komentarza, który ciągnie się do końca wiersza. Komentarze są przeznaczone dla ludzi, a kompilator je ignoruje.

Każdy program musi zawierać dokładnie jedną globalną funkcję o nazwie `main()`. Wykonywanie programu zaczyna się od tej funkcji. Wartość całkowitoliczbowa typu `int` zwracana przez funkcję `main()`, jeśli w ogóle zostanie zwrócona, stanowi wartość zwrotną przekazywaną do „systemu”. Jeśli funkcja nie zwróci żadnej wartości, system otrzyma wartość wskazującą na bezbłędne zakończenie wykonywania. Wartość zwrotna funkcji `main()` różna od zera oznacza błąd. Nie każdy system operacyjny i nie każde środowisko wykonawcze wykorzystuje wartość zwrotną. Na przykład systemy linuksowe/uniksowe ją wykorzystują, natomiast środowiska typu Windows robią to rzadko.

Typowy program generuje jakiś wynik. Oto program drukujący napis *Witaj, świecie!*:

```
#include <iostream>

int main()
{
    std::cout << "Witaj, świecie!\n";
}
```

Wiersz `#include <iostream>` nakazuje kompilatorowi *dołączenie* deklaracji funkcji standardowego strumienia wejścia – wyjścia znajdujących się w `iostream`. Bez tych deklaracji wyrażenie

```
std::cout << "Witaj, świecie!\n";
```

nie miałoby sensu. Operator << („wstaw do”) drukuje swój drugi argument w swoim pierwszym argumencie. W tym przypadku drukuje literał łańcuchowy `"Witaj, świecie!\n"` w standardowym strumieniu wyjściowym `std::cout`. Literał łańcuchowy jest ciągiem znaków w podwójnym cudzysłowie prostym. W literale łańcuchowym wsteczny ukośnik, `\`, z następującym po nim znakiem reprezentuje jeden „specjalny znak”. W tym przypadku została użyta sekwencja specjalna `\n` oznaczająca przejście do nowego wiersza, dzięki czemu wszystko, co znajdzie się za napisem *Witaj, świecie!*, zostanie wydrukowane w nowym wierszu.

Przedrostek `std::` wskazuje, że nazwy `cout` należy szukać w przestrzeni nazw biblioteki standardowej (3.4). W opisach elementów standardowych zazwyczaj pomijam ten człon. W podrozdziale 3.4 można się dowiedzieć, jak sprawić, by nazwy z wybranej przestrzeni nazw były widoczne bez potrzeby stosowania bezpośredniego kwalifikatora.

Cały wykonywalny kod znajduje się w funkcjach i jest wywoływany pośrednio lub bezpośrednio z funkcji `main()`. Na przykład:

```
#include <iostream>       // dołącza („importuje”) deklaracje z biblioteki strumieni wejścia – wyjścia

using namespace std;      // sprawia, że nazw z przestrzeni std można używać bez członu std:: (3.4)

double square(double x)   // podnosi liczbę zmiennoprzecinkową o podwójnej precyzji do kwadratu
{


    return x*x;


}

void print_square(double x)
```

```
{
    cout << "Kwadrat liczby " << x << " wynosi " << square(x) << "\n";
}

int main()
{
    print_square(1.234); // drukuje: Kwadrat liczby 1,234 wynosi 1,52276
}
```

„Typ zwrotny” `void` oznacza, że funkcja nie zwraca żadnej wartości.

1.3 Funkcje

Aby w programie C++ wykonać jakąś czynność, najlepiej jest wywołać funkcję. Definicja funkcji określa, w jaki sposób ma zostać wykonana pewna operacja. Wywołać można tylko funkcję, która wcześniej została zadeklarowana.

Deklaracja funkcji zawiera nazwę, typ zwracanej wartości (jeśli jest) oraz liczbę i typy argumentów, które należy przekazać w wywołaniu. Na przykład:

```
Elem* next_elem();     // brak argumentów; zwraca wskaźnik do Elem (Elem*)
void exit(int);        // argument int; nic nie zwraca
double sqrt(double); // argument typu double; zwraca wartość typu double
```

W deklaracji funkcji typ zwrotny występuje przed nazwą funkcji, a typy argumentów występują za nią i znajdują się w nawiasie.

Semantyka operacji przekazywania argumentów jest identyczna z semantyką inicjalizacji (3.6.1). To znaczy, że typy argumentów są sprawdzane i w razie potrzeby przeprowadzana jest niejawna konwersja ich typów (1.4). Na przykład:

```
double s2 = sqrt(2);       // wywołanie funkcji sqrt() z argumentem double{2}
double s3 = sqrt("trzy"); // błąd: funkcja sqrt() wymaga argumentu typu double
```

Nie należy bagatelizować znaczenia tego sprawdzania i konwersji typów podczas kompilacji.

Argumenty w deklaracji funkcji mogą mieć nazwy. Są one pomocne dla osoby czytającej kod programu, ale jeśli deklaracja nie jest jednocześnie definicją funkcji, kompilator po prostu ignoruje te nazwy. Na przykład:

```
double sqrt(double d); // zwraca pierwiastek kwadratowy z d
double square(double); // zwraca pierwiastek kwadratowy z argumentu
```

Na typ funkcji składają się jej typ zwrotny i typy jej argumentów. Na przykład:

```
double get(const vector<double>& vec, int index); // typ: double(const vector<double>&,int)
```

Funkcja może być składową klasy (2.3, 4.2.1) i wówczas nazwa tej klasy także wchodzi w skład typu tej **funkcji składowej**. Na przykład:

```
char& String::operator[](int index); // typ: char& String::(int)
```

Kod źródłowy powinien być zrozumiały, ponieważ od tego zależy łatwość jego utrzymania. Podstawą czytelności jest podział zadań obliczeniowych na logiczne fragmenty (w postaci

funkcji i klas) oraz nadanie im nazw. Tak zdefiniowane funkcje stanowią wówczas podstawowy słownik obliczeniowy, tak jak typy (wbudowane i zdefiniowane przez użytkownika) stanowią podstawowy słownik danych. Na początek najlepiej jest używać standardowych algorytmów C++ (np. `find`, `sort` i `iota` — rozdział 12.). Potem z funkcji reprezentujących ogólne lub specjalistyczne działania można budować większe obliczenia.

Liczba błędów w kodzie silnie koreluje z ilością kodu i jego poziomem złożoności. Oba te problemy można rozwiązać poprzez tworzenie większej liczby krótszych funkcji. Rozwiązując konkretne zadania za pomocą funkcji, można uniknąć konieczności pisania określonego kodu w środku innego kodu. Utworzenie funkcji wymusza nadanie nazwy i opisanie jej zależności.

Jeśli zostaną zdefiniowane dwie funkcje o takiej samej nazwie, ale różnych typach argumentów, w każdym wywołaniu kompilator wybierze najlepiej pasującą. Na przykład:

```
void print(int);      // przyjmuje argument całkowitoliczbowy
void print(double); // przyjmuje jako argument liczbę zmiennoprzecinkową
void print(string); // przyjmuje łańcuch jako argument

void user()
{
    print(42);            // wywołuje print(int)
    print(9.65);          // wywołuje print(double)
    print("Barcelona"); // wywołuje print(string)
}
```

Jeśli dopuszczalne jest wywołanie dwóch funkcji, ale żadna nie jest lepsza od drugiej, wywołanie jest niejednoznaczne i kompilator zgłasza błąd. Na przykład:

```
void print(int,double);
void print(double,int);

void user2()
{
    print(0,0); // błąd: niejednoznaczne
}
```

Definiowanie wielu funkcji o tej samej nazwie nazywa się **przeciążaniem funkcji** i stanowi jeden z fundamentów programowania generycznego (7.2). Kiedy funkcja jest przeciążona, każda jej wersja powinna charakteryzować się taką samą semantyką. Przykładem zastosowania tej techniki jest rodzina funkcji `print()` — każda funkcja o tej nazwie drukuje swój argument.

1.4 Typy, zmienne i arytmetyka

Każda nazwa i każde wyrażenie ma typ określający zbiór operacji, jakie można na nich wykonać. Na przykład deklaracja

```
int inch;
```

stwierdza, że `inch` jest typu `int`, czyli zmienną typu całkowitoliczbowego.

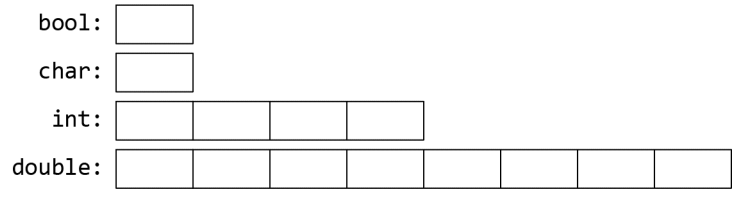
Deklaracja jest instrukcją wprowadzającą do programu pewien element. Określa jego typ:

- **Typ** definiuje zbiór możliwych wartości i operacji (w przypadku obiektu).
- **Obiekt** to fragment pamięci, w którym jest przechowywana wartość pewnego typu.
- **Wartość** to zbiór bitów, które są interpretowane zgodnie z typem.
- **Zmienna** to nazwany obiekt.

W języku C++ dostępny jest szeroki wybór typów podstawowych, ale ponieważ nie jestem systematykiem, nie przedstawiam ich tu wszystkich w postaci uporządkowanej listy. W razie potrzeby można je znaleźć w innych podręcznikach (np. [Stroustrup, 2003] lub [Cppreference]) w internecie. Oto kilka przykładów:

```
Bool     // typ logiczny, możliwe wartości to true i false
char     // znak, np. 'a', 'z', i '9'
int      // liczba całkowita, np. -273, 42 i 1066
double   // liczba zmiennoprzecinkowa o podwójnej precyzji, np. -273.15, 3.14 i 6.626e-34
unsigned // nieujemna liczba całkowita, np. 0, 1 i 999 (używany w bitowych operacjach logicznych)
```

Każdy typ podstawowy wprost koresponduje z odpowiednimi elementami sprzętowymi i ma stały rozmiar determinujący zakres dopuszczalnych wartości, jakie można w nim przechowywać:



Zmienna typu `char` ma rozmiar odpowiedni do przechowywania jednego znaku na danej platformie sprzętowej (najczęściej 8-bitowy bajt), a rozmiary pozostałych typów są wielokrotnością rozmiaru typu `char`. Rozmiar typu zależy od implementacji (tzn. może się różnić w zależności od maszyny) i można go sprawdzić za pomocą operatora `sizeof`. Na przykład `sizeof(char)` wynosi `1`, a `sizeof(int)` to często `4`.

Liczby mogą być zmiennoprzecinkowe lub całkowite:

- Liczby zmiennoprzecinkowe wyróżniają się obecnością kropki dziesiętnej, np. `3.14`, lub wykładnika, np. `3e-2`.
- Literały całkowitoliczbowe domyślnie mają format dziesiętny (np. `42` oznacza czterdzieści dwa). Przedrostek `0b` oznacza literał binarny (liczbę o podstawie 2), np. `0b10101010`. Przedrostek `0x` oznacza literał szesnastkowy (liczbę o podstawie 16), np. `0xBAD1234`. Natomiast przedrostek `0` oznacza literał ósemkowy (liczbę o podstawie 8), `0334`.

Dla zwiększenia czytelności literałów cyfry w długich liczbach można grupować za pomocą apostrofów. Na przykład π wynosi około `3.14159'26535'89793'23846'26433'83279'50288`, a jeśli ktoś woli format szesnastkowy — `0x3.243F'6A88'85A3'08D3`.

1.4.1 Działania arytmetyczne

Za pomocą operatorów arytmetycznych można wykonywać działania na wartościach typów podstawowych:

```
x+y    // dodawanie
+x     // plus jednoargumentowy
x-y    // odejmowanie
-x     // minus jednoargumentowy
x*y    // mnożenie
x/y    // dzielenie
x%y    // reszta (modulo) z dzielenia liczb całkowitych
```

Podobnie można używać operatorów porównywania:

```
x==y   // równość
x!=y   // brak równości
x<y    // mniejszy niż
x>y    // większy niż
x<=y   // mniejszy lub równy
x>=y   // większy lub równy
```

Dostępne są też operatory logiczne:

```
x&y    // bitowe „i”
x|y    // bitowe „lub”
x^y    // bitowe „lub” wykluczające
~x     // dopełnienie bitowe
x&&y   // logiczne „i”
x||y   // logiczne „lub”
!x     // logiczne „nie” (negacja)
```

Bitowy operator logiczny wykonuje operację na każdym bicie przekazanego argumentu i zwraca wartość takiego samego typu, jakiego jest ten argument. Operatory logiczne `&&` i `||` zwracają tylko wartość `true` lub `false` zależnie od wartości argumentów.

W operacjach przypisania i arytmetycznych C++ automatycznie przeprowadza wszelkie niepozbawione sensu konwersje typów podstawowych, dzięki czemu dowolnie można je mieszać:

```
void some_function() // funkcja, która nie zwraca wartości
{
    double d = 2.2;  // inicjalizacja liczby zmiennoprzecinkowej
    int i = 7;       // inicjalizacja zmiennej całkowitoliczbowej
    d = d+i;         // przypisanie sumy do d
    i = d*i;         // przypisanie iloczynu do i; uwaga: skrócenie liczby double d*i do typu int
}
```

Konwersje stosowane w wyrażeniach nazywają się **zwykłymi konwersjami arytmetycznymi** i są wykonywane na najwyższym poziomie precyzji w odniesieniu do ich argumentów. Na przykład dodawanie liczby typu `double` do liczby typu `int` zostanie wykonane przy użyciu arytmetyki liczb zmiennoprzecinkowych o podwójnej precyzji.

Należy zwrócić szczególną uwagę, że `=` jest operatorem przypisania, a `==` to operator pozwalający sprawdzić, czy dwa argumenty są sobie równe.

Oprócz zwykłych operatorów arytmetycznych i logicznych język C++ zawiera jeszcze zestaw innych operacji modyfikowania wartości zmiennych:

```
x+=y   // x = x+y
++x    // inkrementacja: x = x+1
x−=y   // x = x-y
−−x    // dekrementacja: x = x-1
x*=y   // skalowanie: x = x*y
x/=y   // skalowanie: x = x/y
x%=y   // x = x%y
```

Operatory te są zwięzłe, wygodne w użyciu i dlatego bardzo często stosowane.

Prawie wszystkie wyrażenia są wykonywane od lewej strony. Wyjątkiem są operacje przypisania, które są wykonywane od prawej. Niestety kolejność obliczania argumentów funkcji jest nieokreślona.

1.4.2 Inicjalizacja

Aby można było użyć obiektu, najpierw należy mu nadać wartość. W języku C++ inicjalizację można wyrazić na kilka sposobów, na przykład za pomocą pokazywanego już wcześniej znaku = lub przy użyciu uniwersalnej formy w postaci listy inicjalizacyjnej w klamrze:

```
double d1 = 2.3;              // inicjalizacja d1 liczbą 2,3
double d2 {2.3};              // inicjalizacja d2 liczbą 2,3
double d3 = {2.3};            // inicjalizacja d3 liczbą 2,3 (w notacji z klamrą znak = jest opcjonalny)
complex<double> z = 1;        // liczba zespolona złożona ze skalarów zmiennoprzecinkowych o podwójnej
                              // precyzji
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2}; // w notacji z klamrą znak = jest opcjonalny
vector<int> v {1,2,3,4,5,6};  // wektor liczb typu int
```

Notacja ze znakiem = to tradycyjna forma wywodząca się jeszcze z języka C, ale w razie wątpliwości należy posługiwać się ogólną notacją w formie listy w klamrze. Pozwala ona uniknąć przynajmniej konwersji powodujących utratę informacji:

```
int i1 = 7.8;   // i1 będzie mieć wartość 7 (niespodzianka?)
int i2 {7.8};   // błąd: konwersja liczby zmiennoprzecinkowej na całkowitą
```

Niestety konwersje powodujące utratę informacji, **konwersje zawężające**, na przykład `double` do `int` i `int` do `char`, są dozwolone i stosowane niejawnie. Problemy związane z automatycznym wykonywaniem konwersji zawężających są ceną za zgodność z językiem C (16.3).

Stała (1.6) nie może pozostać niezainicjalizowana, a zmienną można pozostawić bez inicjalizacji tylko w nielicznych, bardzo rzadkich przypadkach. Nie należy wprowadzać do programu nazwy, jeśli nie ma jeszcze dla niej wartości. Typy zdefiniowane przez użytkownika (np. `string`, `vector`, `Matrix`, `Motor_controller` czy `Orc_warrior`) mogą być inicjalizowane niejawnie (4.2.1).

Jeśli typ zmiennej można wywnioskować na podstawie inicjalizatora, można go pominąć w definicji:

```
auto b = true;     // wartość logiczna
auto ch = 'x';     // znak
```

```
auto i = 123;       // liczba całkowita
auto d = 1.2;       // typ double
auto z = sqrt(y); // z będzie mieć typ zwrócony przez sqr t(y)
auto bb {true};     // bb jest typu logicznego
```

Ze słowem kluczowym `auto` zwykle używamy notacji `=`, ponieważ nie istnieje ryzyko wystąpienia kłopotliwych konwersji typów. Jeśli jednak dla spójności ktoś woli stosować notację z klamrą, to nic nie stoi na przeszkodzie.

Słowa kluczowego `auto` używa się w przypadkach, gdy nie ma konkretnego powodu, aby określić typ jawnie. „Konkretne powody" to:

- Definicja znajduje się w szerokim zakresie, w którym typ powinien być wyraźnie widoczny dla osoby czytającej kod źródłowy.
- Chcemy wprost określić zakres lub precyzję zmiennej (np. `double`, a nie `float`).

Użycie słowa kluczowego `auto` pozwala uniknąć redundancji i pisania długich nazw typów. Szczególne znaczenie ma to w programowaniu generycznym, w którym czasami trudno dokładnie określić typ obiektu, a nazwy typów bywają bardzo długie (12.2).

1.5 Zakres i cykl istnienia

Deklaracja wprowadza nazwę do zakresu:

- **Zakres lokalny**: nazwa zadeklarowana w funkcji (1.3) lub lambdzie (6.3.2) jest **nazwą lokalną**.

  Jej zakres dostępności obejmuje kod od miejsca deklaracji do końca bloku, w którym deklaracja ta się znajduje. Granice **bloku** wyznaczają znaki { i }. Nazwy argumentów funkcji także są nazwami lokalnymi.
- **Zakres klasowy**: nazwa zdefiniowana w klasie (2.2, 2.3, rozdział 4.), nie we wnętrzu jakiejkolwiek funkcji (1.3), lambdy (6.3.2) lub klasy wyliczeniowej (2.5) to **nazwa składowa** (lub **nazwa składowej klasy**). Zakres jej dostępności obejmuje kod od znaku { otwierającego zawierającą ją deklarację do końca tej deklaracji.
- **Zakres przestrzeni nazw**: nazwa znajdująca się w przestrzeni nazw (3.4), nie we wnętrzu jakiejkolwiek funkcji, lambdy (6.3.2), klasy (2.2, 2.3, rozdział 4.) lub klasy wyliczeniowej (2.5) to **nazwa składowa przestrzeni nazw**. Jej zakres rozciąga się od miejsca deklaracji do końca przestrzeni nazw, do której należy.

Nazwa niezadeklarowana w żadnej innej konstrukcji to **nazwa globalna**, która znajduje się w tzw. **globalnej przestrzeni nazw**.

Ponadto można tworzyć obiekty pozbawione nazw, takie jak obiekty tymczasowe i obiekty tworzone za pomocą operatora `new` (4.2.2). Na przykład:

```
vector<int> vec; // vec to nazwa globalna (globalny wektor liczb całkowitych)

struct Record {
    string name; // name jest składową struktury Record (składowa łańcuchowa)
    // ...
};
```

```
void fct(int arg) // fct jest nazwą globalną (funkcja globalna)
                  // arg jest nazwą lokalną (argument całkowitoliczbowy)
{
    string motto {"Do odważnych świat należy"}; // nazwa motto jest lokalna
    auto p = new Record{"Hume"};                // p wskazuje obiekt typu Record bez nazwy
(utworzony przez operator new)
    // ...
}
```

Przed użyciem obiekt musi zostać utworzony (zainicjalizowany), a na końcu jego zakresu dostępności następuje jego zniszczenie. W przypadku obiektu należącego do przestrzeni nazw miejscem destrukcji jest koniec programu. Punkt destrukcji składowej jest tożsamy z punktem destrukcji obiektu, do którego ona należy. Obiekt utworzony za pomocą operatora `new` „żyje", aż zostanie skasowany przez operator `delete` (4.2.2).

1.6 Stałe

W języku C++ występują dwa rodzaje niezmienności:

- `const` — oznacza mniej więcej „obiecuję, że nie zmienię tej wartości". Stałych tego rodzaju najczęściej używa się w definicjach interfejsów, aby można było przekazywać dane do funkcji za pomocą wskaźników i referencji bez obaw o ich modyfikację. Kompilator pilnuje, czy obietnica złożona poprzez `const` została dotrzymana. Wartość `const` może być obliczona w czasie działania programu.
- `constexpr` — oznacza mniej więcej „wartość zostanie obliczona w czasie kompilacji". Tego rodzaju konstrukcji używa się głównie do definiowania stałych, przechowywania danych w pamięci tylko do odczytu (gdzie istnieje niewielkie ryzyko zniszczenia) oraz w celu optymalizacji wydajności. Wartość `constexpr` musi zostać obliczona przez kompilator.

Na przykład:

```
constexpr int dmv = 17;        // dmv jest stałą mającą nazwę
int var = 17;                  // var nie jest stałą
const double sqv = sqrt(var); // sqv to stała mająca nazwę, prawdopodobnie o wartości obliczanej podczas
                               // działania programu

double sum(const vector<double>&); // sum nie zmieni wartości swojego argumentu (1.7)

vector<double> v {1.2, 3.4, 4.5};  // v nie jest stałą
const double s1 = sum(v);          // Ok.— wartość sum(v) jest obliczana w czasie działania programu
constexpr double s2 = sum(v);      // błąd: sum(v) nie jest wyrażeniem stałym
```

Aby funkcji można było użyć w **wyrażeniu stałym**, to znaczy wyrażeniu, którego wartość zostanie obliczona przez kompilator, w jej definicji musi stać słowo kluczowe `constexpr`. Na przykład:

```
constexpr double square(double x) { return x*x; }
constexpr double max1 = 1.4*square(17);  // OK — 1.4*square(17) to wyrażenie stałe
constexpr double max2 = 1.4*square(var); // błąd : var nie jest wyrażeniem stałym
const double max3 = 1.4*square(var);     // OK, wartość może zostać obliczona w czasie działania programu
```

Funkcji `constexpr` można używać z argumentami, które nie są stałymi, ale wówczas wynik nie jest wyrażeniem stałym. Dozwolone jest wywoływanie funkcji `constexpr` z argumentami niebędącymi wyrażeniami stałymi w kontekstach, w których nie są wymagane wyrażenia stałe. Dzięki temu nie ma konieczności definiowania zasadniczo takiej samej funkcji dwa razy: raz dla wyrażenia stałego i raz dla zmiennych.

Funkcja typu `constexpr` musi być dość prosta, aby nie miała żadnych skutków ubocznych, oraz wykorzystywać tylko informacje przekazywane do niej jako argumenty. W szczególności nie może modyfikować zmiennych innych niż lokalne, ale może zawierać pętle i używać własnych zmiennych lokalnych. Na przykład:

```
constexpr double nth(double x, int n) // założenie 0<=n
{
    double res = 1;
    int i = 0;
    while (i<n) {                     // pętla while: wykonywana dopóki warunek jest spełniony (1.7.1)
        res*=x;
        ++i;
    }
    return res;
}
```

W pewnych nielicznych przypadkach wyrażenia stałe są wymagane przez reguły języka (np. do określania granic tablic (1.7), w etykietach `case` (1.8), argumentach wartości szablonów (6.2) oraz stałych ze słowem `constexpr` w deklaracji). W innych przypadkach obliczanie wartości w czasie kompilacji jest ważne ze względów wydajnościowych. Niezależnie od takich spraw jak wydajność pojęcie niezmienności (obiekt o niezmiennych stanie) jest ważną kwestią projektową.

1.7 Wskaźniki, tablice i referencje

Najprostszym zbiorem danych jest alokowany w sposób ciągły szereg elementów tego samego typu zwany **tablicą**. Taki sposób przechowywania danych zapewnia platforma sprzętowa. Tablicę elementów typu `char` można zadeklarować tak:

```
char v[6]; // tablica 6 znaków
```

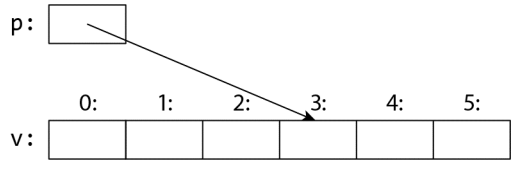
Natomiast wskaźnik można zadeklarować następująco:

```
char* p;    // wskaźnik do znaku
```

W deklaracjach konstrukcja `[ ]` oznacza tablicę elementów, z kolei znak `*` symbolizuje wskaźnik do czegoś. Dolna granica numeracji elementów w tablicy to zawsze `0`, a więc `v` zawiera sześć elementów: od `v[0]` do `v[5]`. Rozmiar tablicy musi być określony wyrażeniem stałym (1.6). Zmienna wskaźnikowa przechowuje adres obiektu odpowiedniego typu:

```
char* p = &v[3]; // p wskazuje czwarty element tablicy v
char x = *p;     // *p jest obiektem wskazywanym przez p
```

W wyrażeniu przedrostkowy, jednoargumentowy operator * oznacza „zawartość czegoś”, natomiast przedrostkowy, jednoargumentowy operator & znaczy „adres czegoś”. Wynik przytoczonej definicji można przedstawić graficznie w następujący sposób:



Weźmy na przykład operację kopiowania dziesięciu elementów z jednej tablicy do innej:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                  // będzie kopią v1

    for (auto i=0; i!=10; ++i) // kopiowanie elementów
        v2[i]=v1[i];
    // ...
}
```

Powyższą instrukcję `for` można przeczytać tak: „ustaw `i` na zero; dopóki `i` nie ma wartości 10, kopiuj `i`-ty element i zwiększaj o 1 wartość `i`”. Operator inkrementacji, `++`, dodaje 1 do liczby całkowitej lub zmiennoprzecinkowej, która jest jego argumentem. W języku C++ dostępna też jest prostsza wersja instrukcji `for` zwana zakresową instrukcją `for`. Służy ona do przeglądania sekwencji elementów w najprostszy możliwy sposób:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v) // dla każdego x w v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

Pierwszą zakresową instrukcję `for` można przeczytać tak: „dla każdego elementu tablicy `v`, od pierwszego do ostatniego, zapisz kopię w `x` i wydrukuj go”. Zauważ, że nie ma potrzeby określania granicy tablicy inicjalizowanej za pomocą listy. Zakresowej instrukcji `for` można używać z dowolną sekwencją elementów (12.1).

Gdybyśmy nie chcieli skopiować wartości z `v` do zmiennej `x`, tylko w `x` odnieść się do jednego z elementów, to moglibyśmy napisać taki kod:

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
```

```
    for (auto& x : v) // dodaje 1 do każdego x w v
        ++x;
    // ...
}
```

W deklaracji jednoargumentowy operator & w postaci przyrostka oznacza „referencję do". Referencja jest podobna do wskaźnika, z tym że nie ma potrzeby stosowania przedrostka *, aby odwołać się do wskazywanej przez nią wartości. Ponadto zainicjalizowanej referencji nie można przestawić, aby wskazywała inny obiekt.

Referencje są wyjątkowo przydatne przy przekazywaniu argumentów do funkcji. Na przykład:

```
void sort(vector<double>& v); // sortuje v (v jest wektorem liczb typu double)
```

Stosując referencję, zapewniamy sobie, że wywołanie `sort(my_vec)` nie spowoduje skopiowania `my_vec` oraz że posortowany zostanie właśnie wektor `my_vec`, a nie jego kopia.

Jeśli modyfikacja argumentu nie jest planowana, ale pożądane jest uniknięcie kosztu związanego z kopiowaniem, można użyć referencji stałej (1.6). Na przykład:

```
double sum(const vector<double>&)
```

Funkcje przyjmujące referencje spotyka się bardzo często.

W deklaracjach operatory (np. &, * czy [ ]) nazywają się **operatorami deklaracji**:

```
T a[n] // T[n]: a jest tablicą n obiektów typu T
T* p   // T*: p jest wskaźnikiem do T
T& r   // T&: r jest referencją do T
T f(A) // T(A): f jest funkcją przyjmującą argument typu A i zwracającą wynik typu T
```

1.7.1 Wskaźnik pusty

Zawsze należy sprawdzić, czy wskaźnik wskazuje jakiś obiekt, aby mieć pewność, że operacja jego dereferencji zakończy się powodzeniem. Kiedy nie ma obiektu, który można byłoby wskazać, lub potrzeba zaznaczyć „brak dostępnego obiektu" (np. na końcu listy), należy wskaźnikowi nadać wartość `nullptr` (wskaźnik pusty). Istnieje tylko jedna wartość `nullptr` wspólna dla wszystkich typów wskaźnikowych:

```
double* pd = nullptr;
Link<Record>* lst = nullptr; // wskaźnik do Link do Record
int x = nullptr;             // błąd: nullptr jest wskaźnikiem, nie liczbą całkowitą
```

Dobrym zwyczajem jest sprawdzanie, czy argument wskaźnikowy cokolwiek wskazuje:

```
int count_x(const char* p, char x)
    // liczy wystąpienia x w p[]
    // p ma wskazywać zakończoną zerem tablicę znaków (lub nic)
{
    if (p==nullptr)
        return 0;
    int count = 0;
    for (;*p!=0; ++p)
        if (*p==x)
```

```
            ++count;
    return count;
}
```

Zwróć uwagę na sposób przesuwania wskaźnika na następny element tablicy za pomocą operatora ++ oraz na to, że inicjalizator w instrukcji `for`, jeśli jest niepotrzebny, można opuścić.

Definicja `count_x()` zakłada, że `char*` jest **łańcuchem w stylu języka C**, czyli że wskaźnik ten wskazuje zakończoną zerem tablicę elementów typu `char`. Znaki w literale łańcuchowym są niezmienne, dlatego aby było możliwe wykonanie operacji `count_x("Witaj!")`, w `count_x()` zadeklarowałem argument `const char*`.

W starszych programach zamiast `nullptr` najczęściej można spotkać `0` lub `NULL`. Zaletą wartości `nullptr` jest brak ryzyka pomyłki i wzięcia liczby całkowitej (np. `0` lub `NULL`) za wskaźnik (taki jak `nullptr`).

W przykładzie funkcji `count_x()` został opuszczony inicjalizotor instrukcji `for`, w związku z czym można użyć prostszej instrukcji `while`:

```
int count_x(const char* p, char x)
    // liczby wystąpienia x w p[]
    // p ma wskazywać zakończoną zerem tablicę znaków (lub nic)
{
    if (p==nullptr)
        return 0;
    int count = 0;
    while (*p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

Instrukcja `while` działa, dopóki spełniony jest jej warunek.

Test wartości liczbowej (np. `while (*p)` w `count_x()`) jest równoznaczny z porównywaniem z wartością `0` (np. `while (*p!=0)`). Test wartości wskaźnikowej (np. `if (p)`) jest równoważny z porównywaniem wartości z `nullptr` (np. `if (p!=nullptr)`).

Nie istnieje natomiast „referencja zerowa”, ponieważ referencja musi odnosić się do prawidłowego obiektu (a implementacje zakładają, że tak jest). Istnieją wątpliwe, sprytne sztuczki pozwalające złamać tę regułę, ale nie należy z nich korzystać.

1.8 Testy

W języku C++ dostępny jest zestaw instrukcji pozwalających na łatwe wyrażanie procesów wyboru i powtarzania. Należą do nich na przykład instrukcje `if`, `switch` i `while` oraz pętle `for`. Poniżej na przykład znajduje się prosta funkcja, która drukuje pytanie do użytkownika i zwraca wartość logiczną oznaczającą odpowiedź:

```
bool accept()
{
    cout << "Czy chcesz kontynuować (t lub n)?\n"; // drukuje pytanie
```

```
    char answer = 0;                                // inicjalizacja wartością, która nie pojawi się
                                                    // na wejściu
    cin >> answer;                                  // wczytanie odpowiedzi

    if (answer == 't')
        return true;
    return false;
}
```

Odpowiednikiem operatora wyjściowego << jest operator wejściowy >> służący do odbierania danych z wejścia. Z kolei `cin` to standardowy strumień wejściowy (rozdział 10.). Typ argumentu z prawej strony operatora >> określa rodzaj przyjmowanych danych i w tym argumencie właśnie zostaną one zapisane. Znak `\n` na końcu łańcucha wyjściowego reprezentuje nowy wiersz (1.2.1).

Definicja zmiennej `answer` znajduje się tam, gdzie jest potrzebna (nie wyżej). Deklaracja może pojawić się wszędzie tam, gdzie instrukcja.

Powyższy przykład można ulepszyć, uwzględniając także możliwość otrzymania odpowiedzi `n` (oznaczającej „nie”):

```
bool accept2()
{
    cout << "Czy chcesz kontynuować (t lub n)?\n"; // drukuje pytanie
    char answer = 0;                                // inicjalizacja wartością, która nie pojawi się
                                                    // na wejściu
    cin >> answer;                                  // wczytanie odpowiedzi

    switch (answer) {
    case 't':
        return true;
    case 'n':
        return false;
    default:
        cout << "Uznam to za odmowę.\n";
        return false;
    }
}
```

Instrukcja `switch` porównuje wartość z zestawem stałych. Te tak zwane etykiety `case` muszą różnić się między sobą i jeśli sprawdzana wartość nie pasuje do żadnej z nich, zostaje wybrana klauzula `default`. Jeśli wartość nie pasuje do żadnej etykiety `case` i nie ma klauzuli domyślnej `default`, nie zostaje wykonana żadna operacja.

Z klauzuli `case` nie trzeba wychodzić poprzez zwrócenie wartości przez funkcję zawierającą całą instrukcję `switch`. Często programista chce tylko coś sprawdzić w instrukcji `switch`, a potem dalej wykonywać kod funkcji. W takich przypadkach używa się instrukcji `break`. W ramach przykładu przyjrzymy się bardzo sprytnemu, ale prymitywnemu parserowi z prostej gry komputerowej:

```
void action()
{
    while (true) {
        cout << "Napisz, co chcesz zrobić:\n"; // prośba o określenie czynności
```

```
        string act;
        cin >> act;          // wczytanie znaków do łańcucha
        Point delta {0,0}; // Point zawiera parę {x,y}

        for (char ch : act) {
            switch (ch) {
            case 'g': // w górę
            case 'p': // północ
                ++delta.y;
                break;
            case 'p': // w prawo
            case 'w': // wschód
                ++delta.x;
                break;
            // …inne czynności…
            default:
                cout << "Stoję!\n";
            }
            move(current+delta*scale);
            update_display();
        }
    }
}
```

Podobnie jak `for` (1.7), instrukcja `if` może wprowadzać zmienną i ją sprawdzać. Na przykład:

```
void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0) {
        // …program dojdzie tu, jeśli if n!=0…
    }
    // …
}
```

W tym przypadku w instrukcji `if` została zdefiniowana zmienna całkowitoliczbowa `n`, która została zainicjalizowana wyrażeniem `v.size()` i od razu sprawdzona za pomocą warunku `n!=0` za średnikiem. Nazwa zadeklarowana w warunku należy do zakresu obu gałęzi instrukcji `if`.

Podobnie jak w przypadku instrukcji `for`, nazwę w warunku instrukcji `if` deklaruje się po to, aby ograniczyć zakres jej dostępności, dzięki czemu program jest czytelniejszy i łatwiej uniknąć błędów.

Najczęściej zmienną porównuje się z `0` (lub `nullptr`). W takim przypadku można po prostu opuścić warunek. Na przykład:

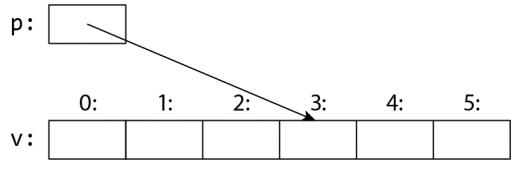
```
void do_something(vector<int>& v)
{
    if (auto n = v.size()) {
        // …ten kod zostanie wykonany, jeśli n!=0…
    }
    // …
}
```

Lepiej w miarę możliwości zawsze używać tej prostszej i zwięźlejszej formy.

1.9 Mapowanie sprzętowe

Język C++ zapewnia bezpośrednie mapowanie sprzętowe. Kiedy programista używa jednej z operacji podstawowych, to korzysta z tego, co zapewnia platforma sprzętowa — najczęściej jest to pojedyncza operacja maszynowa. Dodawanie dwóch liczb typu `int` na przykład powoduje wykonanie maszynowej instrukcji dodawania całkowitoliczbowego.

Dla implementacji C++ pamięć maszyny jest ciągiem miejsc, w których można umieszczać (typizowane) obiekty i adresować je za pomocą wskaźników:



W pamięci wskaźnik jest reprezentowany jako adres maszynowy, dlatego wartość numeryczna `p` na tym rysunku wynosi 3. Jeśli komuś przypomina to tablicę (1.7), to dlatego, że tablica w języku C++ jest najprostszą abstrakcją „nieprzerwanego ciągu obiektów w pamięci".

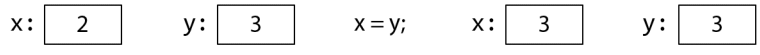
Proste mapowanie podstawowych konstrukcji języka na elementy platformy sprzętowej jest kluczowe dla dużej niskopoziomowej wydajności, z której języki C i C++ słyną od dziesięcioleci. Model maszynowy tych języków opiera się na sprzęcie komputerowym, nie na jakichkolwiek abstrakcjach matematycznych.

1.9.1 Przypisanie

Przypisanie typu wbudowanego jest prostą operacją kopiowania maszynowego. Weźmy taki przykład:

```
int x = 2;
int y = 3;
x = y; // x zmienia wartość na 3
// uwaga: x==y
```

To oczywiste. Graficznie można to przedstawić tak:



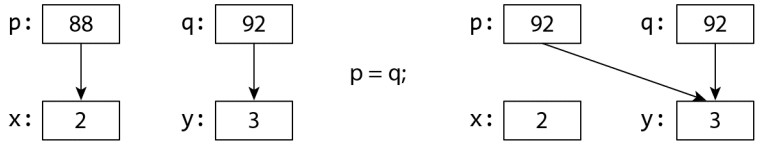
Każdy z tych dwóch obiektów jest niezależny od drugiego. Wartość `y` można zmienić bez jakiegokolwiek wpływu na wartość `x`. Na przykład operacja `x=99` nie spowoduje zmiany wartości zmiennej `y`. W odróżnieniu od Javy, C# i innych języków, nie licząc C, dotyczy to wszystkich typów, nie tylko `int`.

Jeśli różne obiekty mają odnosić się do tej samej (wspólnej) wartości, trzeba to wyrazić wprost. Służą do tego wskaźniki:

```
int x = 2;
int y = 3;
```

```
int* p = &x;
int* q = &y; // teraz p!=q i *p!=*q
p = q;       // p przyjmuje wartość &y; teraz p==q, a więc (oczywiście)*p == *q
```

Na poniższym rysunku zostało to przedstawione w formie schematu:

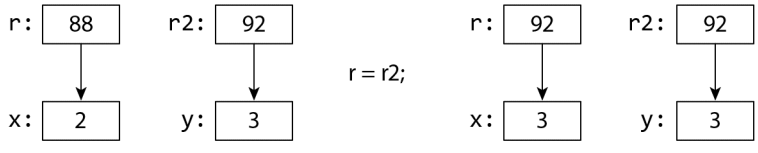


Adresy obiektów 88 i 92 wybrałem losowo. Jak widać, obiekt będący celem przypisania otrzymuje wartość od obiektu przypisywanego, w wyniku czego powstają dwa niezależne obiekty (tutaj wskaźniki) o takiej samej wartości, to znaczy `p=q` daje `p==q`. Po operacji `p=q` oba wskaźniki wskazują na `y`.

Zarówno referencja, jak i wskaźnik odnoszą się do obiektu (wskazują go) i w pamięci są reprezentowane jako adres maszynowy. Różnią się natomiast sposobami użycia. Przypisanie do referencji nie zmienia tego, do czego się ona odnosi, tylko zapisuje wartość we wskazywanym przez nią obiekcie:

```
int x = 2;
int y = 3;
int& r = x;  // r odnosi się do x
int& r2 = y; // teraz r2 odnosi się do y
r = r2;      // odczyt przez r2, zapis przez r: x przyjmuje wartość 3
```

Na poniższym rysunku przedstawiono to w ujęciu graficznym:



Aby odnieść się do wartości wskazywanej przez wskaźnik, należy użyć operatora *. W przypadku referencji odbywa się to automatycznie (niejawnie).

Dla każdego typu wbudowanego i dobrze zaprojektowanego typu użytkownika (rozdział 2.) udostępniającego operacje = (przypisania) i == (sprawdzania równości) po operacji `x=y` spełniony jest warunek `x==y`.

1.9.2 Inicjalizacja

Inicjalizacja różni się od przypisania. Ogólnie rzecz biorąc, aby operację przypisania dało się prawidłowo przeprowadzić, obiekt będący jej celem musi mieć wartość. Natomiast inicjalizacja polega na zamianie niezainicjalizowanego fragmentu pamięci w prawidłowy obiekt. Skutek odczytu lub zapisu niezainicjalizowanej zmiennej w przypadku prawie wszystkich typów

jest niezdefiniowany. Jeśli chodzi o typy wbudowane, najbardziej oczywiste jest to w odniesieniu do referencji:

```
int x = 7;
int& r {x}; // wiąże r z x (r odnosi się do x)
r = 7;      // przypisanie wartości do tego, do czego odnosi się r
int& r2;    // błąd: niezainicjalizowana referencja
r2 = 99;    // przypisanie do tego, do czego odnosi się r2
```

Na szczęście referencja nie może pozostać niezainicjalizowana. Gdyby tak było, to operacja `r2=99` przypisałaby 99 do jakiegoś bliżej nie określonego miejsca w pamięci, w efekcie czego program zwracałby nieprawidłowe wyniki albo uległby awarii.

Za pomocą operatora = można zainicjalizować referencję, ale nie należy dać się zmylić. Na przykład:

```
int& r = x; // wiąże r z x (r odnosi się do x)
```

To nadal jest inicjalizacja wiążąca `r` z `x`, a nie jakakolwiek forma kopiowania wartości.

Różnica między inicjalizacją i przypisaniem jest kluczowa także w przypadku wielu typów zdefiniowanych przez użytkownika, takich jak `string` i `vector`, gdy obiekt będący celem przypisania ma zasoby, które kiedyś będą musiały zostać zwolnione (5.3).

Przekazywaniem argumentów do funkcji i zwracaniem wartości przez funkcje rządzi semantyka inicjalizacji (3.6). Tak właśnie uzyskujemy przekazywanie przez referencję.

1.10 Porady

Poniższe porady stanowią wyciąg z *C++ Core Guidelines* [Stroustrup, 2015]. Odwołania do wytycznych mają postać [CG: ES.23], co oznacza 23. regułę w sekcji *Expressions and Statement*. W wytycznych z reguły można znaleźć dodatkowe objaśnienia i przykłady.

[1] Nie panikuj! Za jakiś czas wszystko stanie się jasne (1.1) [CG: In.0].

[2] Nie używaj wbudowanych składników wyłącznie. W istocie podstawowych (wbudowanych) elementów języka najlepiej jest używać pośrednio przez biblioteki, na przykład bibliotekę standardową ISO C++ (rozdziały 8. – 15.) [CG: P.10].

[3] Nie trzeba znać C++ w najdrobniejszych szczegółach, aby pisać dobre programy.

[4] Skoncentruj się na technikach programowania, a nie na elementach języka.

[5] W razie wątpliwości w odniesieniu do definicji językowych rozstrzygającym źródłem informacji jest standard ISO C++ (16.1.3) [CG:P.2].

[6] Konkretne operacje definiuj w postaci funkcji o dobrze dobranych nazwach (1.3) [CG: F.1].

[7] Funkcja powinna wykonywać jedną logiczną operację (1.3) [CG: F.2].

[8] Pisz krótkie funkcje (1.3) [CG: F.3].

[9] Jeśli kilka funkcji realizuje tę samą koncepcyjnie operację na różnych typach danych, wykorzystaj przeciążanie (1.3).

[10] Jeśli wartość funkcji można obliczyć w czasie kompilacji, zadeklaruj ją jako `constexpr` (1.6) [CG: F.4].

[11] Poznaj sposób mapowania typów podstawowych na sprzęt (1.4, 1.7, 1.9, 2.3, 4.2.2, 4.4).
[12] Długie literały liczbowe przedzielaj separatorami dla zwiększenia czytelności (1.4) [CG: NL.11].
[13] Unikaj skomplikowanych wyrażeń [CG: ES.40].
[14] Unikaj konwersji zawężających (1.4.2) [CG: ES.46].
[15] Maksymalnie ograniczaj zakres dostępności zmiennych (1.5).
[16] Unikaj „magicznych stałych”, używaj stałych symbolicznych (1.6) [CG: ES.45].
[17] Stawiaj na dane niezmienne (1.6) [CG: P.10].
[18] W każdej deklaracji deklaruj (tylko) jedną nazwę [CG: ES.10].
[19] Często używane nazwy lokalne powinny być krótkie, a rzadziej używane i nielokalne — dłuższe [CG: ES.7].
[20] Nie twórz wielu podobnych nazw [CG: ES.8].
[21] Unikaj tworzenia nazw pisanych samymi `WIELKIMI_LITERAMI` [CG: ES.9].
[22] W deklaracjach zawierających typ mający nazwę wybieraj składnię inicjalizacji z {} (1.4) [CG: ES.23].
[23] Aby nie musieć powtarzać nazw typów, używaj `auto` (1.4.2) [CG: ES.11].
[24] Unikaj pozostawiania niezainicjalizowanych zmiennych (1.4) [CG: ES.20].
[25] Utrzymuj małe zakresy dostępności (1.5) [CG: ES.5].
[26] Deklarując zmienną w warunku instrukcji `if`, wybieraj wersję z niejawnym porównaniem z `0` (1.8).
[27] Typów bez znaku (`unsigned`) używaj tylko w operacjach na bitach (1.4) [CG: ES.101] [CG: ES.106].
[28] Wskaźników używaj w prostych konstrukcjach (1.7) [CG: ES.42].
[29] Używaj `nullptr` zamiast `0` i `NULL` (1.7) [CG: ES.47].
[30] Zmienną deklaruj dopiero wtedy, kiedy masz wartość do jej inicjalizacji (1.7, 1.8) [CG: ES.21].
[31] Nie pisz w komentarzach tego, co w sposób oczywisty wynika z kodu [CG: NL.1].
[32] W komentarzach opisuj swoje zamiary [CG: NL.2].
[33] Stosuj spójny schemat wcinania kodu źródłowego [CG: NL.4].

Skorowidz

G

H

I

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Na pewno znasz C++. Ale czy sprawnie się nim posługujesz?

Żadnego programisty nie trzeba przekonywać o zaletach C++. To język dojrzały, wszechstronny, pozwalający uzyskiwać maksymalną wydajność kodu. Do tego wciąż konsekwentnie rozwijany — współczesny C++ wygląda zupełnie inaczej niż dwadzieścia lat temu. Oznacza to, że profesjonalny programista, który chce w pełni wykorzystać zalety nowoczesnego C++, musi bardzo dobrze orientować się w nowościach przynoszonych przez kolejne specyfikacje języka. Warto również wiedzieć, jak zmieniają się dostępne narzędzia językowe i biblioteki, a także które paradygmaty programowania są przez nie wspierane i w jakim zakresie.

Ta książka jest idealnym wyborem dla programisty C lub C++, który chce lepiej zapoznać się z nowościami w języku C++. Jest to zwięzły i dokładny przewodnik po najważniejszych elementach języka i komponentach biblioteki standardowej z uwzględnieniem niedawno wprowadzonych udoskonaleń i udogodnień. Znalazły się tu również liczne przykłady i praktyczne wskazówki, które szczególnie istotne okażą się w kontekście paradygmatów programowania, takich jak programowanie generyczne i zorientowane obiektowo. Poza dość podstawowymi zagadnieniami omówiono tu tematykę semantyki przenoszenia, jednolitej inicjalizacji, wyrażeń lambda, kontenerów i współbieżności. Książka kończy się interesującym opisem projektu i ewolucji języka C++.

Najciekawsze zagadnienia:

- podstawy działania kodu C++
- operacje standardowe oraz operacje wejścia-wyjścia
- szablony, w tym szablony funkcji i szablony zmienne
- algorytmy w C++ i narzędzia pomocnicze

Dr Bjarne Stroustrup może mówić o sobie, że jest ojcem C++: zaprojektował go i jako pierwszy zaimplementował. Jest dyrektorem działu technologicznego banku Morgan Stanley w Nowym Jorku i profesorem wizytującym w Columbia University. Wcześniej pracował w Bell Labs, AT&T Labs oraz Texas A&M University. Otrzymał wiele wyróżnień, w tym przyznawaną przez National Academy of Engineering Nagrodę Charlesa Starka Drapera. Jest członkiem National Academy of Engineering, Institute of Electrical and Electronics Engineers (IEEE) oraz Association for Computing Machinery (ACM).

ISBN 978-83-283-5424-1