

WSKAZÓWKI I ELEMENTY JĘZYKA C++

Andrzej Görlich

7 października 2021

I Wskazówki

Rozszerzenie nazw programów

Programy wykonywalne powinny mieć *rozszerzenie* .x. Ułatwia to pisanie reguł kompilacji w pliku Makefile.

***Bardzo ważne:** O ile wyraźnie nie wskazano w zestawie inaczej, wszystkie programy **muszą** czytać dane wejściowe ze standardowego wejścia i zapisywać dane wyjściowe do standardowego wyjścia. Pod żadnym pozorem, nie wolno czytać/zapisywać z/do plików, których nazwy wpisano na sztywno w kodzie źródłowym.*

Potoki, strumienie i przekierowania

Programy **powinny** **muszą** czytać dane ze standardowego wejścia stdin (np. przy użyciu scanf lub std::cin) oraz zapisać wynik do standardowego wyjścia stdout (np. przy użyciu printf lub std::cout). Takie rozwiązanie, oprócz swojej prostoty i wydajności, jest bardzo uniwersalne. Mechanizm przekierowania strumieni pozwala na łatwe wczytanie danych z pliku oraz zapis danych do pliku z poziomu linii komend (terminala/konsoli):

```
# Odczyt danych ze standardowego wejścia i zapis na standardowe wyjście
./program.x
# Zapis danych wyjściowych do pliku "output.txt"
./program.x > output.txt
# Odczyt danych wejściowych z pliku "input.txt"
./program.x < input.txt
# Odczyt danych z pliku "input.txt" i zapis do "output.txt"
./program.x < input.txt > output.txt
# Dopisanie danych wyjściowych do pliku
./program.x >> append.txt
# Zapis standardowego wyjścia błędów do pliku
./program.x 2> error.txt
# Wyjście pierwszego programu staje się wejściem drugiego
./program1.x | ./program2.x
# Przekierowuje stdout i stderr na wejście drugiego programu
./program1.x |& ./program2.x
```

Przykład:

```
echo {1..10} | tr " " "\n" | sort -r > out.txt
cat - < out.txt # Efekt taki sam jak "cat out.txt"
```

Wczytywanie danych

Aby przyspieszyć korzystanie ze standardowego wejścia w C++ można wyłączyć synchronizację strumieni wejścia/wyjścia (cin, cout) z stdio. W tym celu należy na samym początku programu użyć `ios_base::sync_with_stdio(false)`.

Wypełnienie kontenera `std::vector` danymi ze standardowego wejścia:

```
int x;
std::vector<int> v;

while(std::cin >> x)
    v.push_back(x);
```

Wersja w jednej linii

```
vector<int> v(std::istream_iterator<int>(std::cin), std::istream_iterator<int>());
```

Pomiar czasu wykonania

Do pomiaru czasu wykonania kodu można użyć funkcji `std::chrono::high_resolution_clock` oraz komendy `time`. Schemat użycia klasy `std::chrono`:

```
#include <chrono>
...
{
    auto start = std::chrono::high_resolution_clock::now();
    // fragment kodu, którego czas wykonania mierzymy
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cerr << "Elapsed time[s] = " << elapsed.count() << endl;
}
```

Poprawa wydajności

Aby wykorzystać optymalizację kodu przez kompilator g++ pod daną architekturę warto użyć flag `-O3` (pełna optymalizacja czasowa) oraz `-march=native` (użycie instrukcji dostępnych na danym procesorze).

Proszę zapoznać się z następującymi funkcjami w standardowej bibliotece C++11, przydatnymi w implementacji algorytmów sortowania, które operują m.in. na strukturach danych `std::vector` i `std::array`:

```
swap, iter_swap, min_element, lower_bound, rotate
```

Ponieważ są one zoptymalizowane pod daną architekturę, np. używają instrukcji `AVX`, często są wydajniejsze od ręcznie napisanego kodu.

Strażnik

Zazwyczaj w językach C i C++ chcemy unikać wielokrotnego dołączania tego samego pliku nagłówkowego. W tym celu stosuje się idiom *strażnika*:

```
#ifndef FILENAME_H_
#define FILENAME_H_
...
#endif /* FILENAME_H_ */
```

Można również użyć dyrektywy, która pozwala na optymalizację odczytu plików:

```
#pragma once
```

Choć nie opisana w standardzie, to jest wspierana przez główne kompilatory (*gcc*, *clang*, *icc*, *MSVC*). Wbrew pozorom, kwestia rozstrzygnięcia przez kompilator/preprocesor czy dwa pliki są identyczne nie jest trywialna.

II Elementy standardu C++11

C++ jest *potężnym i wydajnym* językiem programowania. Jednak, aby efektywnie z niego korzystać, należy **rozumieć** co się dzieje *pod maską*.

Standard C++11 wprowadza między innymi następujące nowe elementy:

- r-wartości i referencje do r-wartości (ang. *r-value references*, `&&`),
- semantyka przenoszenia (ang. *move semantics*) - `std::move`, Alex Allain, *Move semantics and rvalue references in C++11*,
- uniwersalne odwołania (ang. *universal references*),
- doskonałe przekazywanie (ang. *perfect forwarding*) - `std::forward`, Scott Meyers, *Universal References in C++11*).

Proszę się z nimi zapoznać.

Szablony

Deklaracje

```
int add(int, int); // Deklaracja funkcji, domyślnie extern
```

Specyfikatory klasy pamięci (ang. *storage class specifiers*)

Specyfikatory `extern`, `static`, `inline` są dyrektywami wykorzystywanymi **jedynie** przez konsolidator (ang. *linker*), a nie kompilator.

- `static` - zmienna lub funkcja nie jest widoczna w pozostałych jednostkach kompilacji (modułach), nie może być przez nie użyta (chyba, że pośrednie przez wskaźnik) lub modyfikowana.
- `extern` - definicja zmiennej lub funkcja może znajdować się w innej jednostce kompilacji. Zadaniem linker jest znalezienie adresu symboli zadeklarowanych jako `extern`. Funkcje są domyślnie deklarowane jako `extern`.
- `inline` - Funkcje może być zdefiniowana w więcej niż jednej jednostce kompilacji. Linker nie będzie traktował tego jako błąd i zapewni, że wszędzie zostanie użyta ta sama instancja funkcji. Zazwyczaj występuje w połączeniu ze `static`.

***Uwaga:** Specyfikator `inline` bardziej przypomina `static` lub `extern` niż dyrektywę, która mówi aby inlinować funkcję. Specyfikator `inline` jest jedynie wskazówką dla kompilatora. Obecnie, kompilator ma pełną swobodę wstawienia kodu funkcji (jeżeli ma dostęp do definicji) lub nie wstawienia.*

Powyższe słowa kluczowe powinny się znajdować na początku deklaracji, a więc

```
static const int * const ptr;  
// const static int * const ptr;
```

Deklaracje zazwyczaj umieszczamy w pliku nagłówkowym.

Definicje

```
auto add(int a, int b) -> int {  
    return a + b;  
}
```

Definicje zazwyczaj umieszczamy w module (.c, .cpp). Ale, **definicje szablonów funkcji** i **szablonów klas** muszą być dostępne dla wszystkich modułów, które z nich korzystają, czyli muszą się znajdować w pliku nagłówkowym.

- Dlaczego?
- Jaka jest różnica pomiędzy szablonami klas w C++ a klasami generycznymi w Javie?
- Jakie są wady takiego rozwiązania?
- Jakie są zalety takiego rozwiązania?

Referencje do r-wartości T&& (r-value reference)

R-wartości (ang. *r-values*) wskazują obiekty podatne na operacje przenoszenia, na ogół odpowiadają tymczasowym obiektom np. zwracanym przez funkcje. Nazwa pochodzi od tego, że wartości te **na ogół** mogą się znaleźć tylko po prawej (ang. *right*) stronie znaku równości (przypisania). Referencje do r-wartości deklarujemy tak samo jak referencje do l-wartości tylko, że zamiast jednego symbolu & używamy dwóch.

Przykłady r-wartości:

```
int i = 5;           // Liczba 5 jest pr-wartością  
// 5 = i            // Liczba 5 nie może znaleźć się po lewej stronie  
// 3 * x = 6         // Wyrażenie 3 * x jest pr-wartością  
                    // i nie może znaleźć się po lewej stronie  
f();                // Wartość zwracana przez funkcję jest r-wartością  
f(std::string("Hello")); // Obiekt tymczasowy jest r-wartością  
const int ci = 1;    // Zmienna ci jest niemodyfikowalną l-wartością,  
// ci = 2;           // ale może znaleźć się tylko po prawej stronie  
// int& k{5};         // Błąd, 5 nie jest l-wartością  
const int& k{5};     // OK, stała referencja może się odwoływać do stałej  
int&& k{5};          // k jest referencją do r-wartości
```

Referencja (zwykła referencja do l-wartości) nie może Nie możemy stworzyć zwykłej referencji do r-warto

Wyjątkiem jest stała referencja , która ... do r-wartości i wartości tymczasowych wydłużając ich czas ...

Semantyka przenoszenia

Semantyka przenoszenia umożliwia zastąpienie kosztownych operacji kopiowania przez mniej kosztowne operacje przenoszenia. Funkcja `std::move` wskazuje, że obiekt może zostać *przeniesiony*. W przypadku przeniesienia, zasoby źródła są usuwane (zerowane), ale sam obiekt wciąż istnieje,

```
std::string s = "abc";  
std::string t = std::move(s); // t == "abc", s == ""
```

***Uwaga:** referencja do r-wartości jest l-wartością (każda referencja jest l-wartością), ale funkcja `std::move(expr)` zwraca `expr` jako r-wartość.*

Uniwersalne odwołania

Referencje do l-wartości, r-wartości i odniesienia uniwersalne

Referencje do r-wartości są deklarowane przy użyciu `&&`, ale nie zawsze `&&` oznacza referencję do r-wartości. Symbol `&&` może oznaczać referencję do *r-wartości* lub *odwołanie uniwersalne*. Referencje deklarowane z użyciem `&&` do typu dedukowanego (auto) lub parametryzowanego (template) nie są *odwołaniami do r-wartości*, tylko są *odwołaniami uniwersalnymi*. Referencje uniwersalne mogą się odwoływać zarówno do *l-wartości*, jak i *r-wartości*.

Odwołania uniwersalne

- Jeżeli wyrażenie inicjujące *odwołanie uniwersalne* jest *l-wartością*, to *odwołanie uniwersalne* staje się *referencją do l-wartości*
- Jeżeli wyrażenie inicjujące *odwołanie uniwersalne* jest *r-wartością*, to *odwołanie uniwersalne* staje się *referencją do r-wartości*

```
void f(Foo&& x);           // && oznacza referencję do r-wartości, brak dedukcji typu  
Foo&& var1 = foo();        // referencja do r-wartości, brak dedukcji typu  
Foo&& var1 = someFoo();    // referencja do r-wartości, brak dedukcji typu  
auto&& var2 = var1;        // && oznacza uniwersalne odwołanie  
  
template<typename T>  
void f(std::vector<T>&& param); // && oznacza referencję do r-wartości  
  
template<typename T>  
void f(T&& param);          // && oznacza uniwersalne odwołanie
```

Doskonałe przekazywanie

Doskonałe przekazywanie umożliwia szablonom funkcji przekazywanie argumentów z zachowaniem wartościowości (*r-wartość* czy *l-wartość*). Szablony funkcji `std::move` i `std::forward` służą tylko do odpowiednio bezwarunkowego i warunkowego rzutowania. Funkcja `std::move` niczego nie przenosi, funkcja `std::forward` niczego nie przekazuje. Funkcje te nie generują żadnego kodu wykonywalnego.

Użyć powyższych technik np. w funkcji `push()`.

Przykład:

```

template<typename T>
class Stack {
    ...
    template<typename U>
    void push(U&& obj) {
        // Tutaj U&& oznacza uniwersalne odwołanie, nie r-wartość
        // Dla l-wartości, U = Typ&, U&& jest referencją do l-wartości (Typ&)
        // Dla r-wartości, U = Typ, U&& jest referencją do r-wartości (Typ&&)
        head = new node(std::forward<U>(obj), nullptr, head);
        // std::forward<U> zapewnia doskonałe przekazanie
        ...
    }
    ...
};

```

Uwagi:

- Odwołanie (referencja) do *r-wartości* jest *l-wartością*. Argument funkcji zawsze jest *l-wartością*. Funkcja `std::move` zwraca odwołanie do *r-wartości*, ale wartości zwracane przez funkcję są *r-wartościami*!
- To czy wyrażenie jest *l-wartością* czy *r-wartością* jest niezależne od typu.
- *R-wartości* pozwalają na semantykę przenoszenia oraz tworzenie konstruktorów przenoszących (`Foo(Foo&&)`) i przenoszących operatorów przypisania (`Foo& operator=(Foo&&)`).
- Operacje przenoszenia nie zawsze są tańsze od kopiowania i nie są tak tanie, jak często oczekujemy.
- Semantyka przenoszenia pozwala także tworzyć typy, które mogą być tylko przenoszone (np. `std::unique_ptr`, `std::future` i `std::thread`).
- *R-wartość* z cechą `const` nie może zostać przeniesiona, zostanie przekopiowana. Rzutowanie `std::move` nie gwarantuje, że obiekt będzie mógł być przeniesiony.
- Symbol `&&` może oznaczać *odwołanie do r-wartości* **lub** *odwołanie uniwersalne*.

Pytania

1. Czym się różni *odwołanie uniwersalne* od *odwołania do r-wartości*?

Konstruktory i operatory przypisania

Klasy w języku C++ mogą posiadać między innymi następujące konstruktory i operatory

- **Domyślny konstruktor** (`T::T()`) - nie posiada argumentów. Jest wykorzystywany przez agregaty. Gdy nie istnieje, jest generowany niejawnie przez kompilator (ang. *implicitly-declared*).
- **Konstruktor kopiujący** (`T::T(const T&)`) - tworzy kopię obiektu podanego jako argument. Zazwyczaj kopiuje również zasoby. Jeżeli nie został zdefiniowany przez użytkownika, jest generowany niejawnie przez kompilator i kopiuje wszystkie pola.
- **Kopiujący operator przypisania** (`T& operator=(const T&)`) - kopiuje zasoby obiektu. Jeżeli nie został zdefiniowany przez użytkownika, podczas definicji obiektu wykorzystywany jest konstruktor kopiujący.

Gdy nie istnieją, są one generowane niejawnie przez kompilator. Można temu zapobiec używając konstrukcji `= delete`.

```

class T {
    T(); // Default constructor
    T(int); // Parametrised constructor
    T(const T&); // Copy constructor. Could be T&, volatile T&, const volatile T&
    T& operator=(const T&); // Copy assignment operator

    T(T&&); // Move constructor
    T& operator=(T&&); // Move assignment operator

    ~T(); // Destructor
}

```

Referencje do r-wartości i semantyka przenoszenia pozwalają na definicję dodatkowych metod:

- **Konstruktor przenoszący** ($T(T\&\&)$) - konstruowany obiekt przejmuje zasoby obiektu podanego jako argument (sam obiekt wciąż będzie istniał).
- **Przenoszący operator przypisania** ($T\& \text{operator}=(T\&\&)$) - przenosi zasoby obiektu (sam obiekt wciąż będzie istniał).

Efektywny kod w C++

Proszę stworzyć klasę `Info`, która śledzi liczbę utworzonych instancji i wykorzystanych zasobów oraz liczbę kopiowań i przeniesień zasobów. W naszym przypadku, obiekty klasy `Info` są *lekkie* i zawierają tylko dwa pola typu `int`. Załóżmy jednak, że są to ciężkie obiekty i zasobem są skany stron książek telefonicznych ¹.

```

class Info {
private:
    static int licznik_obiektow;
    static int licznik_zasobow;
    static int stworzone_obiekty;
    static int wykorzystane_zasoby;
    static int licznik_kopiowan;
    static int licznik_przeniesien;

    int numer; // Unikalny numer obiektu
    int zasob; // Dodatnia liczba naturalna

public:

    Info(); // Default constructor
    Info(int); // Parametrised constructor
    Info(const Info&); // Copy constructor
    Info& operator=(const Info&); // Copy assignment operator
    Info(Info&&); // Move constructor
    Info& operator=(Info&&); // Move assignment operator
    ~Info(); // Destructor

    info(); // Wypisuje numer i zasób
}

```

¹Książka telefoniczna to zbiór papierowych kartek z listą numerów telefonów i danymi wszystkich abonentów. Stosowana w zamierzonych czasach.

```
static podsumowanie(); // Wypisuje liczniki  
};
```

Wykorzystując powyższą klasę proszę sprawdzić ile obiektów i zasobów jest tworzonych przez poniższy krótki kod oraz ile razy dane są kopiowane lub przenoszone.

```
Info f(Info o) {  
    o.info();  
    return o;  
}  
int main() {  
    Info o;  
    o = Info{2};  
    f(o);  
    Info::podsumowanie();  
}
```

Pytania

- Jak zminimalizować liczbę tworzonych obiektów i zasobów?
- Jak zminimalizować liczbę kopiowań zasobów?
- Jak wykorzystać semantykę przenoszenia?
- Jak wykorzystać uniwersalne referencje i doskonałe przekazywanie?
- Sprawdzić, czy poziom optymalizacji wpływa na wyniki (-O0 vs -O3).

C++ Standard Library versus Standard Template Library

Chociaż STL (ang. *Standard Template Library*) miała istotny wpływ na rozwój standardowej biblioteki C++, to obie biblioteki są od siebie zupełnie niezależne i nie należy ich utożsamiać. *STL* nie jest częścią *standardu języka C++* (ISO 14882 C++ standard). Więcej informacji na ten temat można znaleźć tutaj.

Więcej informacji

- Universal References in C++11—Scott Meyers
- Thomas Becker's overview
- C++ and Beyond 2012: Scott Meyers - Universal References in C++11
- Rvalue references
- Custom containers in C++11
- Alex Allain, *Move semantics and rvalue references in C++11*