Podstawy programowania: Laboratorium nr 11
Wyrażenia lambda.

2017-2018

mgr inż. Przemysław Walkowiak
dr inż. Michał Ciesielczyk

Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. umieszczeniu wszystkich rozwiązań w jednym pliku, chyba, że w poleceniu napisano inaczej.
5. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie).

Wprowadzenie

Wyrażenia Lambda

W standardzie C++11 został wprowadzony nowy rodzaj funkcji anonimowej, wyrażenia lambda. Składnia wyrażenia lambda jest następująca:

```
[<capture>](<arguments>) -> <return_type> {
    <body>
}
```

gdzie:

<capture> jest to lista zmiennych z zasięgu lokalnego/globalnego jakie mają być przekazane (przechwycone) do lambdy

<arguments> lista argumentów wyrażenia lambdy (funkcji anonimowej), analogiczna jak przy klasycznych funkcjach.

<return_type> opcjonalnie zdefiniowany typ wartości zwracanej z wyrażenia (część `-> <return_type>` jest opcjonalna.

<body> sekwencja instrukcji jaka ma być wykonana podczas wywołania lambdy (analogicznie jak w funkcji).

Z racji tego, że wyrażenia lambda są funkcjami anonimowymi, to nie mają swoją nazwy. Jednakże można je przypisać do "zmiennej lokalnej" i za pomocą niej się wyrażenie lambda wywoływać. Innym podejściem jest przekazywanie wyrażenia lambdy zdefiniowanego "inline" jako parametr innej funkcji (np. któregoś algorytmu standardowego).

Ponadto, w odróżnieniu od funkcji klasycznych, mogą być zdefiniowane wewnątrz innej funkcji (jej widoczność jest wtedy ograniczona odpowiednim blokiem instrukcji (nawiasami).

Przykłady:

```
int x = 5;
int y = 10;

// definiujemy proste wyrażenie, które dodaje dwie liczby
// i przypisujemy do zmiennej 'add'
// ponieważ w czasie pisania programu nie jesteśmy w stanie
// określić typu funkcji anonimowej, należy skorzystać ze
// słow kluczowego 'auto'
auto add = [](int a1, int a2) { return a1 + a2; };

int z = add(x, y); // wywołuje się tak jak każdą inną funkcję.

// lista argumentów jest definiowana tak samo jak
// w klasycznych funkcjach.
auto inc = [](int &a) { a++; };
inc(x); // inkrementuje zmienną 'x'
        // (parametr przekazany przez referencję)

const int M = 5;

// do wyrażenia można przekazać również inne zmienne,
// obiekty z lokalnego zasięgu. Poniżej przekazuje
// stałą 'M' przez sekcję <capture>.
auto mulM = [M](int &x) { x *= M; };

mulM(y); // wartość zmiennej 'y' zostanie pomnożona
         // przez 'M', czyli przez 5
```

```
enum class SwordType { Bastard, Great, Short, Katana };
struct Sword {
    SwordType type;
    float length;
};

std::vector<Sword> swords;

auto is_bastard = [](const Sword &sword) {
                        return sword.type == SwordType::Bastard;
                    };
```

```
// zliczanie ile mieczy bastardowych jest zawartych w kontenerze
auto number_of_bastard_swords = std::count_if(swords.begin(), swords.end(),
                                              is_bastard);

// czy istnieje co najmniej jedna katana?
bool contain_katana = std::any_of(swords.begin(), swords.end(),
                                  [](const auto& sword) {
                                     sword.type == SwordType::Katana;
                                  });

const float m2ft = 3.28084; \\ 1 metr = 3.24084 stopy
auto metric_to_imperial = [m2im](Sword sword) {
                                 sword.length *= m2ft;
                                 return sword;
                              };

// konwersja długości mieczy z metrycznej na imperialną.
std::transform(swords.begin(), swords.end(), swords.begin(),
               metric_to_imperial);
```

Algorytmy

Przeglądanie obu kolekcji można realizować na dwa sposoby:

1. za pomocą iteratorów,

```
for (auto it = imiona.begin(); it != imiona.end(), ++it)
    std::cout << *it << std::endl;
```

2. korzystając z pętli for-range.

```
for (auto &ocena : Oceny)
    std::cout << "Przedmiot" << ocena.first << " - ocena: "
            << ocena.second  << std::endl;
```

Większość algorytmów standardowych (`http://en.cppreference.com/w/cpp/algorithm`) jako pierwsze argumenty przyjmuje iteratory początkowe oraz końcowe zakresu, który chcemy wykorzystać. Kolejnym argumentem jest często jakiś predykat, który w zależności od przeznaczenia algorytmu jest predykatem warunkowym (zwraca wartość prawda-fałsz), albo przekształca element kontenera lub dokonuje innych obliczeń.

Przykładowo algorytm `std::any_of`:

```
template< class InputIt, class UnaryPredicate >
bool any_of( InputIt first, InputIt last, UnaryPredicate p );
```

przyjmuje dwa argumenty będące iteratorami oraz trzeci predykat jednoargumentowy zwracający wartość typu **bool**. Predykatem może być dowolna funkcja, funktor lub wyrażenie lambda spełniające wymagania. Np.:

```
        // sprawdzamy czy istnieje produkt tanszy niz 2 zł.
        std::any_of(cennik.begin(), cennik.end(),
                    [](auto &cena) { return cena.second < 2.0; })
```

Wykorzystanie algorytmów standardowych jest dużo bardziej ekspresyjne (wyrażające intencje), aniżeli pisanie bezpośrednio pętli **for**. Co z kolei wiąże się ze wzrostem czytelności kodu zarówno dla nas jak i dla innych programistów.

Porównaj:

```
std::string text = "...";
{
    int liczba_cyfr = 0;
    for (auto &znak : text) {
        if (std::isdigit(znak) {
            liczba_cyfr++;
        }
    }
}
{
    int liczba_cyfr = std::count_if(test.begin(), test.end(), std::isdigit);
}
```

```
std::vector<Oceny> oceny;
{
    bool czy_zaliczone = true;
    for (auto &ocena : oceny) {
        if (ocena == 2) {
            czy_zaliczone = false;
            break;
        }
    }
}
{
    bool czy_zaliczone = std::none_of(oceny.begin(), oceny.end(),
                                [](Ocena &ocena) { return ocena == 2; });
}
{ // lub
    bool czy_zaliczone = !std::find(oceny.begin(), oceny.end(), 2);
}
```

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0, 1);

const int N = 100;
```

```
{
    std::vector<float> numbers;
    for (int i = 0; i< N; ++i){
        numbers.push_back(dis(gen));
    }
}
{
    std::vector<float> numbers(N);
    std::generate(numbers.begin(), numbers.end(),
                  [&]() { return dis(gen); });
}
```

Zadania

Zadanie 1

Małgosia stwierdziła, że pora odwiedzić swoją babcię w głębokim lesie. W związku z tym, zaczęła pakować do koszyka różne owoce i warzywa by podarować je babci. Rozpoczęła tę operację od stworzenia nowej struktury reprezentującej odpowiednie rośliny:

```
enum class TypRosliny { Owoc, Warzywo };

struct Roslina {
    TypRosliny typ;
    std::string nazwa;
};
```

Później, przygotowała sobie koszyk jako implementację zbioru:

```
typedef std::vector<Roslina> Koszyk;
```

oraz utworzyła swój wymarzony kolorowy koszyk:

```
Koszyk koszyk;
```

i zaczęła do niego wkładać różne warzywa i owoce, po jednym z każdego rodzaju. Pomóż Małgosi umieścić warzywa i owoce w koszyku. Spróbuj tego dokonać na wiele sposobów.

Zadanie 2

Po zapakowaniu koszyka Małgosia postanowiła sprawdzić co takiego się w tym koszyku znajduje. Zaimplementuj operatory

```
std::ostream& operator<<(std::ostream& out, const Roslina& roslina) { }
std::ostream& operator<<(std::ostream& out, const Koszyk &koszyk) { }
```

by ułatwić Małgosi to sprawdzanie i zademonstruj działanie.

Zadanie 3

Marta spytała się Małgosi czy zapakowała ulubione przez babcię gruszki. Korzystając z algorytmu `std::find` zaimplementuj funkcję:

```
bool czy_jest_gruszka(const Koszyk &koszyk)  { }
```

oraz odpowiedz na pytanie Marty.

Zadanie 4

Nagle do Małgosi podchodzi jej mama, zerka do koszyka i pyta się „Czy naprawdę zanosisz Babci same owoce?". Małgosia zaprzeczyła i pokazała na to dowód.

Napisz funkcje, które sprawdzają czy w koszyku są: same owoce, same warzywa, co najmniej jeden owoc, co najmniej jedno warzywo, żadnego owocu, żadnego warzywa. Skorzystaj z algorytmów standardowych `std::any_of`, `std::none_of`, `std::all_of` (`http://en.cppreference.com/w/cpp/algorithm/all_any_none_of`).

Przykładowa funkcja:

```
bool czy_same_owoce(const Koszyk &koszyk) {
    return std::all_of(koszyk.begin(), koszyk.end(),
        [](const Roslina &roslina){ return roslina.typ == TypRosliny::Owoc; }
        );
}
```

Zadanie 5

Koszyk Małgosi wydaje się bardzo ciężki. Policz ile sztuk owoców oraz ile sztuk warzyw zostało do niego zapakowane. Zaimplementuj dwie funkcje: `zlicz_owoce()`, oraz `zlicz_warzywa()`. Skorzystaj z funkcji `std::count_if`.

```
int zlicz_rosliny_na_litere_m(const Koszyk &koszyk) {
    return std::count_if(koszyk.begin(), koszyk.end(),
            [](const Roslina& roslina) { return roslina.nazwa[0] == 'M'
                                          || roslina.nazwa[0] == 'm'; }
            );
}
```

Zadanie 6

Marta bardzo lubi wszystkie owoce na literę G. W związku z tym ukradkiem wyciągnęła wszystkie swoje ulubione smakołyki z koszyka. Korzystając z funkcji `erase` i `remove_if`, zaimplementuj funkcję `usun_jezeli()` usuwającą wszystkie elementy danego typu zaczynające się na podaną literę z koszyka.

Zadanie 7

Okazało się jednakże, że brakuje możliwości rozróżnienia i porządkowania poszczególnych roślin. Siostra Małgosi, Marta, podpowiedziała jej by dodatkowo zaimplementowała operator porównania.

```
bool operator<(const Roslina& r1, const Roslina& r2)  {  ... }
```

Pomóż go zaimplementować.

Implementacja tego operatora jest potrzebna przy korzystaniu z algorytmów z kolejnych zadań.

Zadanie 8

Marta stwierdziła, że również odwiedzi swoją kochaną Babcię i też zaczęła przygotowywać swój koszyk. Po zakończeniu przygotowań siostry zaczęły porównywać co takiego włożyły do koszyków. Zaczęły od sprawdzenia jakie owoce i warzywa znajdują się w obu koszykach.

Marta skorzystała w tym celu z algorytmu `std::set_intersection`:

```
Koszyk koszyk_wspolne;
set_intersection(koszyk.begin(), koszyk.end(),
                 koszyk2.begin(), koszyk2.end(),
                 std::back_inserter(koszyk_wspolne));

std::cout << koszyk_wspolne << std::endl;
```

Małgosia też chciała się pochwalić i pokazała czym koszyki się różnią (`std::set_difference`). Zaimplementuj odpowiednią funkcjonalność.

Uwaga! Jak można wyczytać w dokumentacji, wymagane jest by funkcje `std::set_intersection` i `set_difference` operowały na **posortowanym** zakresie. W tym celu należy wpierw posortować uporządkować oba koszyki. Można skorzystać z funkcji `std::sort`.

Zadanie 9

Mama, widząc jak długo zajmuje dzieciom zabawa w pakowanie koszyków, kazała zawartość obu koszyków umieścić w jednym wielkim.

Zaimplementuj funkcjonalność korzystając z algorytmu `std::set_union`.