C++17 i STL

Obiekty funkcyjne i lambdy

Funkcje jako argumenty algorytmów

- Niektóre algorytmy umożliwiają przekazywanie funkcji pomocniczych zdefiniowanych przez użytkownika – funkcje te są następnie wewnętrznie wywoływane przez te algorytmy.
- Najpopularniejszym przykładem jest algorytm for_each(), wywołujący funkcję zdefiniowaną przez użytkownika wobec każdego elementu z podanego zakresu.
- Przykład:

```
// funkcja wypisująca przekazany argument
void print (int elem) {
    cout << elem << ' ';
}
int main() {
    vector<int> coll;
    // wstaw elementy od 1 do 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    // wypisz wszystkie elementy
    for_each(coll.cbegin(), coll.cend(), print);
    cout << endl;
}
```

Funkcje jako argumenty algorytmów

- Wykorzystanie argumentów funkcyjnych:
  - kryterium wyszukiwania,
  - kryterium sortowania,
  - definicja operacji, która ma być wykonana na elementach kolekcji.
- Przykład:

```
int square (int value) {
    return value*value;
}
int main() {
    set<int> coll1;
    vector<int> coll2;
    // wstaw do kolekcji coll1 elementy od 1 do 9
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }
    //PRINT_ELEMENTS(coll1,"wartości początkowe: ");
    // transformuj każdy element z kolekcji coll1 do coll2
    transform (coll1.cbegin(),coll1.cend(),
        back_inserter(coll2),
        square);
    //PRINT_ELEMENTS(coll2,"podniesione do kwadratu: ");
}
```

Predykaty

- Predykat (ang. *predicate*) jest funkcją zwracającą wartość boolowską.
- Predykaty są często używane do określenia kryterium sortowania lub wyszukiwania.
- W zależności od przeznaczenia predykaty mogą być jedno- lub dwuargumentowe.
- Biblioteka *STL* wymaga, aby predykaty były bezstanowe, to znaczy dla tej samej wartości zawsze zwracały ten sam wynik – wyklucza to obiekty funkcyjne, które w trakcie wywołania modyfikują swój wewnętrzny stan.

Predykaty jednoargumentowe

- Predykaty jednoargumentowe sprawdzają określoną właściwość pojedynczego argumentu.
- Typowy przykład stanowi funkcja wykorzystywana jako kryterium wyszukiwania do znalezienia pierwszej liczby pierwszej:

```
bool isPrime (int number) {
    // liczby 0, 1 i ujemne nie są pierwsze
    if (number <= 1)
        return false;
    // znajdź podzielnik
    for (int div = 2; div * div <= number; div++)
        if (number % div == 0)
            return false; // jest podzielnik
    return true; // nie ma podzielnika
}
int main() {
    list<int> coll;
    ...
    auto pos = find_if (coll.cbegin(), coll.cend(),
        isPrime);
    if (pos != coll.end()) // znaleziono
        cout << *pos << " to liczba pierwsza" << endl;
    else // nie znaleziono
        cout << "nie znaleziono liczby pierwszej" << endl;
}
```

Predykaty dwuargumentowe

- Predykaty dwuargumentowe porównują określoną właściwość dwóch argumentów.
- Aby na przykład posortować elementy zgodnie z własnym kryterium sortowania, możemy podać własną funkcję predykatową:

```
class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

bool sortCriterion (const Person& p1, const Person& p2) {
    // osoba jest 'mniejsza' niż druga osoba jeśli
    // - nazwisko jest 'mniejsze'
    // - nazwisko jest 'równe' oraz imię jest 'mniejsze'
    return p1.lastname() < p2.lastname() ||
        (p1.lastname() == p2.lastname() &&
        p1.firstname() < p2.firstname());
}
int main() {
    deque<Person> coll;
    ...
    sort(coll.cbegin(),coll.cend(), // zakres
        sortCriterion); // kryterium sortowania
}
```

Obiekty funkcyjne

- Argumenty funkcyjne algorytmów nie muszą być funkcjami – mogą to być również obiekty, które zachowują się jak funkcje.
- Taki obiekt nazywamy **obiektem funkcyjnym** (ang. *function object*) lub inaczej **funktorem**.
- Możemy zdefiniować obiekt funkcyjny jako obiekt klasy udostępniającej operator wywołania funkcji `operator()`.
- Było to możliwe również przed C++11.

Obiekty funkcyjne

- Obiekty funkcyjne stanowią kolejny przykład możliwości programowania ogólnego i koncepcji czystej abstrakcji – można powiedzieć, że funkcją jest wszystko to, co zachowuje się jak funkcja (jeśli więc zdefiniujemy obiekt zachowujący się jak funkcja, będzie on mógł zostać użyty jako funkcja).
- Zachowanie funkcyjne jest czymś, co można wywołać z wykorzystaniem nawiasów zwykłych i przekazaniem argumentów.
- Wszystko, co musimy zrobić, to zdefiniować operator() z odpowiednimi typami parametrów.
- Standard C++ używa pojęcia obiektu funkcyjnego w odniesieniu do wszystkich obiektów, których da się użyć jako funkcji – pojęcie to obejmuje więc wskaźniki do funkcji, obiekty klas przeciążających operator (), obiekty klas definiujących konwersję na wskaźnik do funkcji oraz lambdy.

Obiekty funkcyjne

- Obiekty funkcyjne są jednak czymś więcej niż funkcje:
  - Obiekty funkcyjne są jakby funkcjami ze stanem.
    - Obiekty funkcyjne mogą zawierać inne funkcje składowe oraz atrybuty. Oznacza to, że obiekty funkcyjne posiadają stan.
    - Inną zaletę obiektów funkcyjnych stanowi możliwość ich inicjalizacji podczas wykonywania przed ich użyciem (wywołaniem).
  - Każdy obiekt funkcyjny posiada swój własny typ.
    - Zwykłe funkcje posiadają różne typy tylko wtedy, gdy różne są ich sygnatury. Obiekty funkcyjne natomiast mogą posiadać różne typy nawet w przypadku, gdy ich sygnatury są takie same. Fakt ten stanowi istotne usprawnienie z punktu widzenia programowania ogólnego wykorzystującego szablony, ponieważ pozwala na przekazanie zachowania funkcyjnego jako parametru szablonu.
    - Istnieje również możliwość zaprojektowania hierarchii obiektów funkcyjnych tak, żeby na przykład utworzyć różne specjalne warianty jednego ogólnego kryterium.
  - Obiekty funkcyjne działają zwykle szybciej od zwykłych funkcji.
    - Koncepcja szablonów pozwala zwykle na lepszą optymalizację, ponieważ na etapie kompilacji zdefiniowanych może być więcej szczegółów. Przekazywanie obiektów funkcyjnych zamiast zwykłych funkcji daje więc często lepszą wydajność.

Obiekty funkcyjne – przykłady

- Załóżmy, że do wszystkich elementów kolekcji chcemy dodać określoną wartość. Jeśli na etapie kompilacji znamy wartość, którą chcemy dodać, możemy użyć zwykłej funkcji:

```
void add10 (int& elem) {
    elem += 10;
}
...
for_each (coll.begin(), coll.end(), add10);
```

Obiekty funkcyjne – przykłady

- Jeśli potrzebujemy różnych wartości znanych na etapie kompilacji, w zamian możemy użyć szablonu:

```
template <int theValue>
void add (int& elem) {
    elem += theValue;
}
for_each (coll.begin(), coll.end(), add<36>);
```

Obiekty funkcyjne – przykłady

- Ponieważ obiekt może posiadać stan, można go zainicjalizować prawidłową wartością:

```
class AddValue {
private:
    int theValue; // wartość do dodania
public:
    // konstruktor
    AddValue(int v = 0) : theValue(v) { }
    // operator wywołania funkcji
    void operator() (int& elem) const {
        elem += theValue;
    }
};
...
for_each (coll.begin(), coll.end(),
    AddValue(36));
```

Predefiniowane obiekty funkcyjne

- Predefiniowane obiekty funkcyjne są zdefiniowane w pliku nagłówkowym `<functional>`:
  - obiekty funkcyjne arytmetyczne (`negate<>()`, `plus<>()`, …);
  - obiekty funkcyjne relacyjne (`less<>()`, `greater<>()` …);
  - obiekty funkcyjne logiczne (`logical_not<>()`, `logical_and<>()`, `logical_or<>()`);
  - obiekty funkcyjne bitowe (`bit_and<>()`, `bit_or<>()`, `bit_xor<>()`).

Adaptatory i obiekty wiązania wywołań

- Adaptator funkcji jest to obiekt funkcyjny, który umożliwia składanie obiektów funkcyjnych ze sobą nawzajem, z określonymi wartościami lub ze specjalnymi funkcjami.
- Adaptator `bind()` wiąże argumenty wywołania dla obiektów wywoływalnych; jeśli funkcja, funkcja składowa, obiekt funkcyjny albo lambda wymaga wywołania z argumentami, można związać z obiektem wiążącym konkretne wartości argumentów, a funkcji wywoływanej udostępnić także argumenty wywołania obiektu wiążącego.
- Argumenty przekazane do wywołania obiektu wiążącego są w wyrażeniu wiążącym widoczne jako symbole zastępcze `std::placeholders::_1`, `std::placeholders::_2` i tak dalej.
- Typowym zastosowaniem obiektów wiążących argumenty wywołania jest parametryzowanie działania predefiniowanych obiektów funkcyjnych z biblioteki standardowej.

Adaptatory i obiekty wiązania wywołań

- Przykłady:
  - auto plus10 =
    std::bind(std::plus<int>(), std::placeholders::_1, 10);
    std::cout << "(_1) + 10: " << plus10(7) << std::endl;
  - auto minusxy =
    std::bind(std::minus<int>(), std::placeholders::_1, std::placeholders::_2);
    std::cout << "(1)-(2): " << minusxy(8, 5) << std::endl;

Czym są lambdy?

- **Lambdy** są anonimowymi obiektami funkcyjnymi i definiują sposób dookreślania zachowania wewnątrz wyrażenia albo instrukcji – skutkiem tego można definiować obiekty reprezentujące zachowanie funkcyjne i przekazywać te obiekty jako tworzone w miejscu wywołania algorytmów predykaty albo inne dookreślenia.
- Na przykład w poniższej instrukcji:

```
// przekształć wszystkie wartości na sześciany
transform (coll.begin(), coll.end(), // źródło
    coll.begin(), // przeznaczenie
    // lambda jako obiekt funkcyjny
    [] (double d) { return d * d * d; }
);
```

- Lambdy nie posiadają ani konstruktora domyślnego, ani operatora przypisania.

Po co używać lambd?

- **Lambda** jest uproszczoną notacją do definiowania i używania anonimowych obiektów funkcyjnych.
- Zamiast definiować nazwaną klasę z funkcją `operator()` i później tworzyć jej obiekt a następnie go wywoływać, można pójść na skróty – zdefiniować lambdę.
- Lambdy są szczególnie przydatne, gdy trzeba przekazać operację jako argument do algorytmu.

Definicja lambdy

- Wyrażenie lambda składa się z kilku części:
  - `[ ]` – lista zmiennych lokalnych (ang. capture list) określająca, które nazwy ze środowiska definicji mogą być używane wewnątrz wyrażenia lambda oraz czy dostępne są przez skopiowanie czy przez referencję;
  - `( )` – lista parametrów, określająca argumenty lambdy;
  - opcjonalny specyfikator `mutable` oznaczający, że w treści wyrażenia lambda można modyfikować stan lambdy (zmieniać wartość zmiennych pobranych przez skopiowanie);
  - opcjonalnego specyfikator `noexcept` oznaczający, że w treści wyrażenia lambda nie będą zgłaszane wyjątki;
  - opcjonalnej deklaracji typu zwrotnego lambdy w postaci `-> T;` (gdzie T to nazwa typu);
  - `{}` – treść lambdy (kod do wykonania).
- Najprostsza lambda to: `[ ]( ){}`

Środowisko pracy lambdy

- Wewnątrz nawiasów klamrowych możemy zawrzeć elementy, które lambda ma przechwycić z zakresu, w którym jest tworzona.
  - [] pusta lista zmiennych lokalnych.
  - [lista-zmiennych] bezpośrednie określenie listy nazw zmiennych do przechwycenia (do zapisania w obiekcie funkcyjnym) przez referencję lub wartość; zmienne, których nazwy zostały poprzedzone znakiem &, są przechwytywane przez referencję, pozostałe są przechwytywane przez wartość; na liście zmiennych może też znajdować się słowo `this` oraz nazwy z operatorem.
  - [=] niejawne przechwytywanie przez wartość – można używać wszystkich nazw lokalnych i odnoszą się one do kopii zmiennych lokalnych pobranych w miejscu definicji wyrażenia lambda.
    - Wewnątrz lambdy tworzone są stałe pola zainicjalizowane wartościami zmiennych zewnętrznych takimi, jakie były w momencie tworzenia lambdy (domknięcie).
  - [=, lista-zmiennych] niejawne przechwycenie przez wartość wszystkich zmiennych lokalnych, których nazwy nie zostały wymienione na liście; lista zmiennych nie może zawierać słowa kluczowego `this`; przed nazwami na liście musi być znak & (zmienne wymienione na liście są przechwytywane przez referencję).
  - [&] niejawne przechwytywanie przez referencję - można używać wszystkich nazw lokalnych i są one dostępne przez referencję.
    - Lambda ma dostęp do odczytu i zapisu zmiennych z zakresu, w którym została utworzona.
  - [&, lista-zmiennych] niejawne przechwycenie przez referencję wszystkich zmiennych lokalnych, których nazwy nie zostały wymienione na liście; lista zmiennych może zawierać słowo kluczowe `this`; przed nazwami na liście nie może pojawić się znak & (zmienne wymienione na liście są przechwytywane przez wartość).

Stosowanie lambd – zalety

- Zamierzamy wyszukać w kolekcji pierwszy element o wartości z zakresu od x do y:

```
deque<int> coll { 1, 3, 19, 5, 13, 7, 11, 2, 17 };
int x = 5;
int y = 12;
auto pos = find_if (
    coll.cbegin(), coll.cend(), // zakres
    [=] (int i) { return i >= x && i =< y; } // kryterium
);
cout << "pierwszy element >5 i <12: " << *pos << endl;
```

- Określenie wciąganych symboli zewnętrznych poprzez [=] to dla kompilatora informacja, że w ciele lambdy mają być widoczne wszystkie symbole bieżącego zasięgu i że będą tam dostępne przez wartość (wartość pobrana w monecie tworzenia obiektu funkcyjnego lambdy).
- Użycie [&] oznaczałoby, że symbole bieżącego zasięgu będą dostępne przez referencję, co pozwalałoby na ich modyfikowanie z wnętrza lambdy.

Stosowanie lambd – zalety

- W starym C++ zamiast predykatowej lambdy można było stosować funkcje:

```
bool pred (int i) {
    return i > x && i < y;
}
...
pos = find_if (coll.begin(), coll.end(),
    pred);
```

- W nowym C++11 eliminujemy problem dookreślania zachowania algorytmu poza miejscem użycia algorytmu.

Stosowanie lambd – zalety

- W starym C++ zamiast predykatowej lambdy można było stosować obiekty funkcyjne:

```
class Pred
{
    int x, y;
public:
    Pred (int xx, int yy) : x(xx), y(yy) { }
    bool operator() (int i) const {
        return i > x && i < y;
    }
};
...
pos = find_if (coll.begin(), coll.end(),
    Pred(x, y)
);
```

- W starym C++ dostęp do parametrów sterujących zachowaniem (x i y) jest doprawdy uciążliwy i nieelegancki.

Stosowanie lambd – zalety

- W starym C++ zamiast predykatowej lambdy można było stosować obiekty wiążące:

```
pos = find_if (coll.begin(), coll.end(),
    bind(logical_and<bool>(),
        bind(greater<int>(),_1,x),
        bind(less<int>(),_1,y)
    )
);
```

- W starym C++ taki zapis zachowania jest zdecydowanie nieczytelny.

Stosowanie lambd – kryteria sortowania

- W ramach kolejnego przykładu użyjemy wyrażenia lambda do określenia kryterium sortowania kontenera elementów typu Person:

```
class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};
...
deque<Person> coll;
...
// posortuj kolekcję coll według nazwiska i imienia:
sort(coll.begin(),coll.end(), // zakres przeszukiwany
    [] (const Person &p1, const Person &p2) {
        return p1.lastname() < p2.lastname() ||
            (p1.lastname() == p2.lastname() &&
            p1.firstname() < p2.firstname());
    }
);
```

Stosowanie lambd – ograniczenia

- Weźmy za przykład lambdę określającą kryterium sortowania dla kontenerów asocjacyjnych:

```
auto cmp = [] (const Person& p1, const Person& p2)
{
    return p1.lastname()<p2.lastname() ||
        (p1.lastname()==p2.lastname()&&
        p1.firstname()<p2.firstname());
};
...
set<Person, decltype(cmp)> coll(cmp);
```

- Ponieważ w deklaracji kontenera `set` trzeba podać kryterium sortowania, musimy uciec się do użycia konstrukcji `decltype`, zwracającego typ obiektu lambda; obiekt lambdy musi też być przekazany do konstruktora kontenera `coll`.
- Inny problem z lambdą polega na braku możliwości posiadania wewnętrznego stanu, zachowywanego pomiędzy wywołaniami lambdy. Jeśli taki stan jest potrzebny, trzeba przechowywać go w zmiennej zewnętrznej, zadeklarowanej w zasięgu, w którym definiowana jest lambda, i wciągniętej do lambdy przez referencję.

Dostęp lambdy do składowych obiektu

- `[this]` oznacza, że składowe w lambdzie będą dostępne przez referencję, a nie skopiowane.

```
class Request {
    // operacja
    function<map<string,string>(const map<string,string>&)> oper;
    map<string,string> values; // argumenty
    map<string,string> results; // cele
public:
    Request(const string& s); // parsuje i zapisuje żądanie
    void execute()
    {
        // wywołanie oper na values
        [this]() { results=oper(values); }
                                                    (); //
w celu otrzymania wyników
    }
};
```

- `[*this]` oznacza, że składowe w lambdzie będą dostępne przez skopiowanie.

Lambdy zmienne

- Domyślnie stan obiektu funkcyjnego nie powinien być zmieniany i standardowo nie ma takiej możliwości – funkcja `operator()()` dla wygenerowanego obiektu funkcyjnego jest funkcją składową `const`.
- Jeśli jednak należy zmodyfikować stan takiego obiektu funkcyjnego, to trzeba go zadeklarować jako `mutable`.
- Przykład:

```
int count = 0;
auto counter = [count] () mutable -> int {
    return ++count;
};
```

Lambdy rekurencyjne

- Aby zdefiniować lambdę rekurencyjną jej nazwa musi być znana kompilatorowi – należy ją wcześniej zadeklarować za pomocą `function<>`. Parametrem `function<>` są funkcje albo obiekty wywoływalne.
- Przykład:

```
std::function<int(int)> factorial;
factorial = [&factorial] (int n) {
    return n < 2 ? 1 : n * factorial(n - 1);
};
const std::function<int(int)> fact =
[&fact] (int n) {
    return n > 1 ? n * fact(n - 1) : 1;
};
```