Лаб. 7. Лямбда вирази.

Мета: Вивчити синтаксис та дослідити можливості лямбда виразів у мові С++ останніх редакцій.

Завдання: Переписати код лабораторних 1 (розв'язок квадратного рівняння) та 2 (розв'язок нелінійного рівняння методом поділу відрізку навпіл) з використанням, де це доцільно, лямбда виразів. Пояснити, як це вплине на роботу програми.

1. Що таке лямбда вирази?

За допомогою мов програмування реалізуються усе більш складні структури. Часто універсальний інструмент стає занадто складним та масивним для використання у простих випадках. Для більшої гнучкості мови у версії С++ 11 були введені лямбда вирази для заміни часто використовуваного типу класів:

Фактично, лямбда-вираз створює клас (тут клас типу структура) з перевантаженим методом круглі дужки "( )", тобто замість такого виразу

```
struct {
    void operator()(int x) const {
        std::cout << x << '\n';
    }
} mylambda;
```

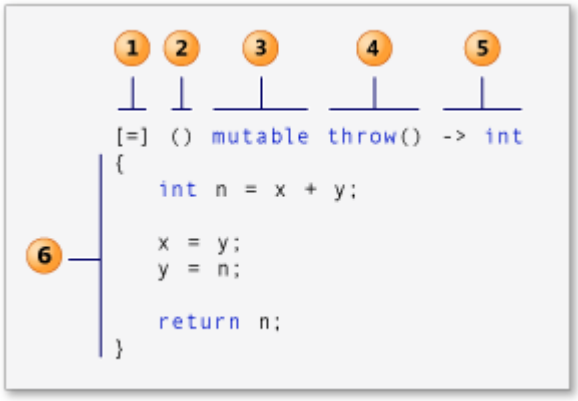
отримуємо такий запис

```
[](int x) {std::cout << x << '\n';}
```

Превагою такого представлення є те, що у такому вигляді можна вбудовувати лямбда-вираз локально у код, у тому числі і як параметри у різних методах. Компілятор створює тимчасову область пам'яті під час виклику скритого конструктора, де зберігає значення параметрів та зміни, які виконуються над ними, під час виходу з лямбда виразу виконується скритий деструктор, який звільняє цю область пам'яті.

2. Структура лямбда виразу

Розглянемо більш детально структуру лямбда виразу та методи виклику.



1) Список захоплення (у специфікации С ++ часто згадується як лямбда-интродьюсер)
2) Список параметрів. Не є обов'язковим. (часто називають – лямбда-декларатор)
3) Специфікація змін. Не є обов'язковою.
4) Специфікація виключення. Не є обов'язковою.
5) Тип значення, що повертається. Не є обов'язковим.
6) Тіло лямбда-виразу.

2. Захоплення змінних лямбда виразом

Як і функції, у лямбда-вирази змінні можна передавати по значенню `[x]` чи по посиланню `[&x]`. Також у лямбда вирази змінні можна передавати явно `[x]`, [&y] та неявно `[=]`, `[&]`.

Такий запис `[=]` означає, що усі змінні, які є у області видимості, передаються по значенню (копіюється), тобто зміни не вийдуть за межі тіла виразу.

Аналогічно запис `[&]` означає, що усі змінні області видимості передаються по посиланню тому, якщо змінна є об'єктом класу, можна використовувати усі неконстантні методи цього класу у тілі виразу.

Подивимось, як це працює на прикладі:

У наш лямбда-вираз ми явно передаємо дві змінні, одну за посиланням, другу за значенням. Для того, щоб захоплені змінні у лямбда-виразі можна було змінювати, використаємо ключове слово `mutable`

```
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1, y = 1;
  std::cout << x << " " << y << std::endl;
  auto lambda = [&x, y]() mutable {
    ++x; ++y;
    std::cout << x << " " << y << std::endl;
    };
  lambda();
  std::cout << x << " " << y << std::endl;
}
```

Ось вивід, який ми отримаємо:

```
Test lambda!
1 1
2 2
2 1
```

Обидві змінні були збільшені на одиницю у тілі виразу, але зміни залишились лише у тієї, яку передали за посиланням. Фактично, у тілі виразу ми працювали із змінною x та копією змінної y, яка буле знищена деструктором лямбда-виразу.

Також можуть зустрітись такі варіанти передачі змінних:

`[=, &x]` – усі змінні захоплюються по значенню, окрім змінної x.

`[&, x]` – усі змінні захоплюються по посиланню, окрім змінної x.

Варіант `[&, =]` є неможливим, бо тоді компілятор не знає, як саме передавати змінні.

3. Тип поверненого значення з лямбда виразу

Тип замикання для лямбда-вирази без захоплення має відкриту невіртуальну неявну функцію перетворення константи в покажчик на функцію, що має той же параметр і типи, що і оператор виклику функції типу замикання.

Фактично це означає, що тип значення, що повертає лямбда вираз автоматично перетворюється залежно від типів значень, що повертаються з тіла виразу. Подивимось, як це працює:

Модифікуємо наш код так, щоб наш вираз повертав різні значення залежно віт змінних

```
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1;
  double y = 1.1;
  std::cout << x << " " << y << std::endl;
  auto lambda = [x, y]()  {
    if (x>0) {
    return y;
    }
    else
    {
    return x;
    };
    };
  std::cout << lambda() << std::endl;
}
```

При спробі виконати цей код отримаємо помилку типів, бо `x` та `y` мають різні типі. Тобто вираз спрацює лише тоді, коли усі значення, що повертаються мають однаковий тип, наприклад, так:

```cpp
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1;
  double y = 1.1;
  std::cout << x << " " << y << std::endl;
  auto lambda = [x, y]()  {
    if (x>0) {
    return y;
    }
    else
    {
    return double(x);
    };
    };
  std::cout << lambda() << std::endl;
}
```

Отримаємо наступне на екрані

```
Test lambda!
1 1.1
1.1
```

Наступний код дасть ті самі результати

```cpp
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1;
  double y = 1.1;
  std::cout << x << " " << y << std::endl;
  auto lambda = [](int a, double b)  {
    if (a>0) {
    return b;
    }
    else
    {
    return double(a);
    };
    };
  std::cout << lambda(x, y) << std::endl;
}
```

Але якщо замінити `int a => double a` явного перетворення значення, що повертається буде не потрібно

```
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1;
  double y = 1.1;
  std::cout << x << " " << y << std::endl;
  auto lambda = [](double a, double b)  {
    if (a>0) {
    return b;
    }
    else
    {
    return a;
    };
    };
  std::cout << lambda(x, y) << std::endl;
}
```

Також лямбда-вирази підтримують автоматичні типи

```
auto lambda = [](auto a, auto b)  {
```

Але у нашому випадку ми знов зіткнемось із неспівпадінням типів.
Такі вираз зручно використовувати як шаблони незалежні від типу

```
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  int x = 1;
  double y = 1.1;
  std::cout << x << " " << y << std::endl;
  auto lambda_sum = [](auto a, auto b)  {
    return a+b;
    };
  std::cout << lambda_sum(x, y) << std::endl;
}
```

У цьому випадку проблем із типами не виникне, бо повертається значення лише один раз.

Також можна явно вказати тип значення, що повертається

```cpp
#include <iostream>

int main() {
  std::cout << "Test lambda!\n";
  double x = 1.0;
  double y = 1.1;
  auto lambda_sum = [](auto x, auto y)-> double
  {
    return x+y;
  };
  std::cout << x << " " << y << std::endl;
  std::cout << lambda_sum(x,y) << std::endl;
}
```

У цьому випадку завжди повертатиметься дійсне число подвійної точності.

Додаткове завдання:

Немає;)