top КОЛЛЕДЖ

Часть 2. Лямбда-функции

Анонимные функции в виде переменных

Предисловие

top КОЛЛЕДЖ

- Итак, представим, что в некоторую функцию нам необходимо передавать другую функцию. С обыкновенными функциями нам необходимо такую функцию объявить, определить, дать ей некоторое имя, а в вызове использовать указатель на эту функцию.
- Если передаваемая функция мала, то возникает несколько проблем:
  - Для нее нужно определить имя,
  - Она все равно занимает определенную область кода,
  - Если ее нужно поменять, или она перестает использоваться, то нужно искать ее определение для внесения изменения.

```
void foo(int i, int (*func)(int))
{
    int k = i;
    do
    {
        k = func(k);
        std::cout << k << "\n";
    } while (k != 1);
}

int bar(int i)
{
    return i % 2 == 0 ? i / 2 : 3 * i + 1;
}

int main()
{
    foo(i: 41, func: bar);

    return 0;
}
```

Лямбда-функции

top КОЛЛЕДЖ

- Чтобы не передавать в функцию указатель по имени, можно передать саму инструкцию — то есть тело функции в виде определенного объекта.
- Такой объект называется ***лямбда-функцией***. Его часто используют для передачи в виде функции в некоторый алгоритм — такое поведение называют ***функцией обратного вызова*** (англ. ***callback***).

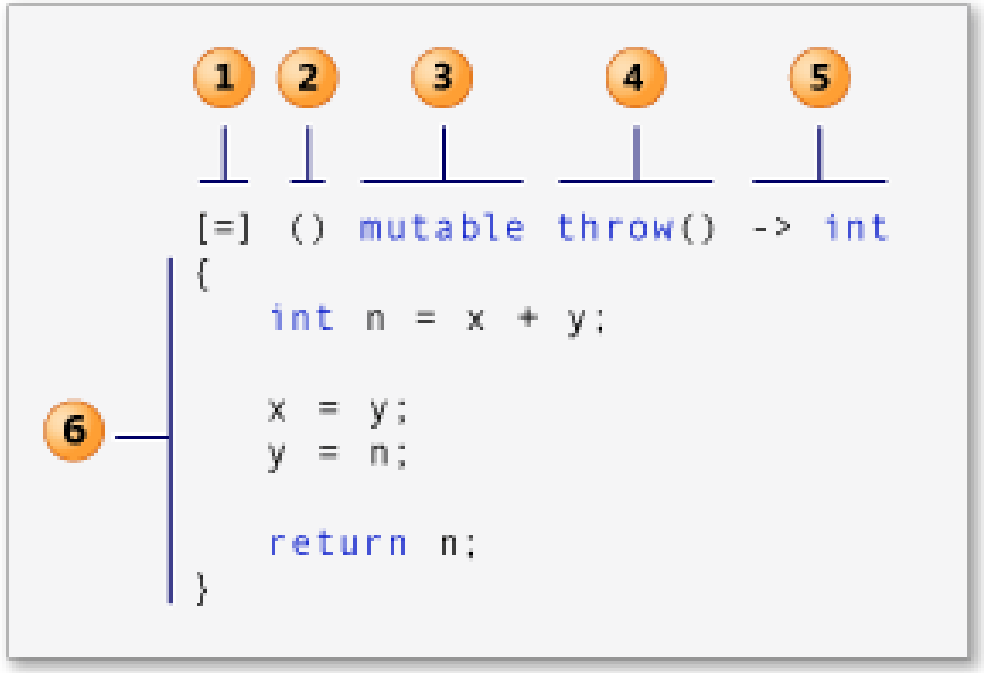
```
void foo(int i, int (*func)(int))
{
    int k = i;
    do
    {
        k = func(k);
        std::cout << k << "\n";
    } while (k != 1);
}

int main()
{
    foo(41, (int){
        // do something
    });
    return 0;
}
```

Разбор синтаксиса

top КОЛЛЕДЖ

- У лямбда-функций достаточно сложный синтаксис. Для определения лямбда функции используют следующую конструкцию:



1. Предложение захвата переменных
2. Аргументы лямбда-функции
3. *Ключевое слово, обозначающее изменяемость захваченных переменных*
4. *Спецификация исключений (устарела в стандарте C++17, кроме noexcept)*
5. *Возвращаемый тип*
6. Тело лямбда-функции

** Курсивом обозначены необязательные поля*

Использование лямбда-функций

top КОЛЛЕДЖ

- Как уже говорилось, лямбда функции можно использовать прямо в аргументах функции.
- Пока что опустим понятие захвата переменных, о нем будет дальше отдельный слайд.
- Заметьте, что таким способом не определяется лишняя функция, а также не используется имя — поэтому часто ***лямбда-функции*** также называют ***анонимными функциями***.

```
void foo(int i, int (*func)(int))
{
    int k = i;
    do
    {
        k = func(k);
        std::cout << k << "\n";
    } while (k != 1);
}

int main()
{
    foo(i: 41, func: [](int i) -> int {
        return i % 2 == 0 ? i / 2 : 3 * i + 1;
        });

    return 0;
}
```

Преобразование к указателю

top КОЛЛЕДЖ

- Однако, если одну и ту же функцию необходимо использовать несколько раз, то для экономии места кода и удобства рефакторинга можно преобразовать лямбда-функцию к обыкновенному указателю на функцию.
- Для этого также можно использовать и автоматическое выведение типов с использованием ключевого слова auto.

```
int main()
{
    int (*callback)(int) = [](int i) -> int {
        return i % 2 == 0 ? i / 2 : 3 * i + 1;
        };

    foo(i: 41, func: callback);
    bar(i: 0, func: callback);

    return 0;
}
```

```
int main()
{
    auto lambda [](int i)->int callback = [](int i) -> int {
        return i % 2 == 0 ? i / 2 : 3 * i + 1;
        };

    foo(i: 41, func: callback);
    bar(i: 0, func: callback);

    return 0;
}
```

Замыкания

top КОЛЛЕДЖ

- ***Замыканиями*** (англ. ***closure***) называют лямбда-функции в которых применяются правила захвата переменных (пункт 1 в разборе синтаксиса).
- Захват переменных означает использование в лямбда-функциях «захваченных» локальных переменных без указания их в качестве аргументов. Сравните функции с рисунка.
- Захват переменных по умолчанию не осуществляется.

```
int main()
{
    int local = 2;

    auto lambda [](int i, int m)->int callback = [](int i, int m) -> int {
        return i % m == 0 ? i / m : 3 * i + 1;
        };

    auto lambda [](int i)->int callback2 = [local](int i) -> int {
        return i % local == 0 ? i / local : 3 * i + 1;
        };


    std::cout << callback(i: 10, m: local) << "\n";
    std::cout << callback2(i: 10) << "\n";

    return 0;
}
```

```
5
5
```

Замыкания

top КОЛЛЕДЖ

- Как вы уже поняли, для захвата переменных достаточно в квадратных скобках указать ее имя.
- Однако, захват переменных можно осуществлять несколькими способами:
  - `[]` — не захватывать локальные переменные (*аналогично обычным функциям*),
  - `[local]` — захват локальной переменной `local` по значению,
  - `[&local]` — захват локальной переменной `local` по ссылке,
  - `[&]`/`[=]` — захват всех локальных переменных по ссылке/значению,
  - `[&, local]` — захват всех локальных переменных по ссылке, кроме `local`, для которой захват осуществляется по значению.
  - `[=, &local]` — захват всех локальных переменных по значению, кроме `local`, для которой захват осуществляется по ссылке.

Вопрос

top КОЛЛЕДЖ

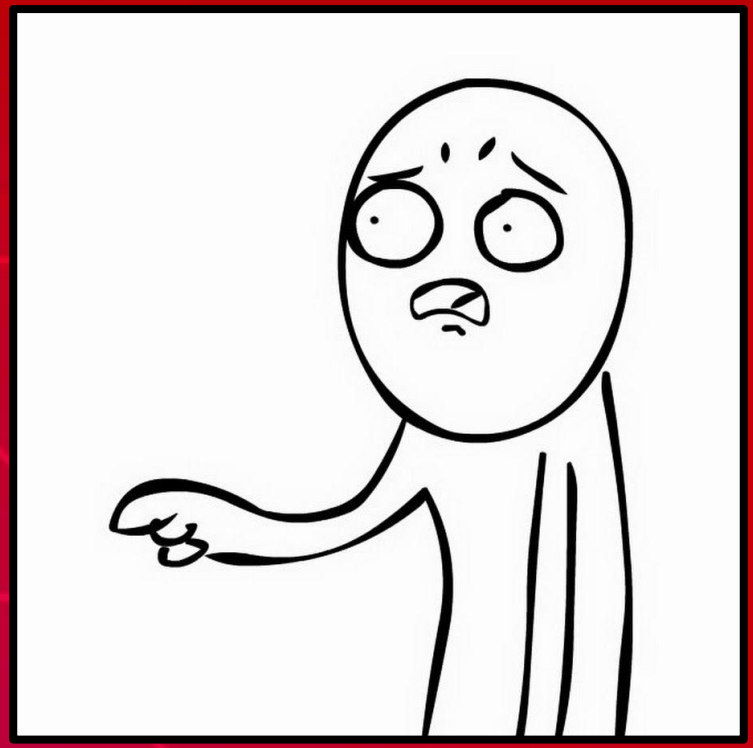
Можно ли замыкание преобразовать к указателю на функцию?

Если да, то как, если нет, то почему?

Преобразование к указателю

top КОЛЛЕДЖ

- На самом деле, не существует преобразования из замыкания к указателю на функцию, потому что в данном случае это разные объекты.
- Функция никак не связана с локальными переменными, то есть свободна от контекста выполнения, а в свою очередь замыкание очень сильно связано с контекстом.
- Однако, в стандартной библиотеке C++ существует способ преобразования как функции к некоторому типу, так и замыкания к некоторому типу — это можно использовать.



std::function

top КОЛЛЕДЖ

- В заголовочном файле `<functional>` можно найти тип `std::function` — это оберточный тип, который позволяет хранить как функции, так и замыкания, и даже ***функторы*** — объекты, которые можно использовать как функции.
- Для использования данной обертки требуется следующее объявление:

```cpp
#include <functional>

int foo(int i, std::function<int(int)> func)
{
    func( _Args: i);
}

int main()
{
    int local = 2;

    auto lambda [](int i)->int callback = [&local](int i) -> int {
        return i % local == 0 ? i / local : 3 * i + 1;
        };

    foo( i: 10, func: callback); // OK

    return 0;
}
```

```
std::function<{возвращаемый_тип}({аргументы}...)>
```