Семинар #5: Вывод типов и другие возможности C++.

Часть 1: Ключевое слово `auto`

Ключевое слово `auto` используется для автоматического вывода типа.

```
#include <string>
int main() {
    auto a = 123;   // a будет иметь тип int
    auto b = 4.1;   // b будет иметь тип double
    auto b = 4.1f;  // b будет иметь тип float

    auto s1 = "Hello";               // s1 будет иметь тип const char*
    auto s2 = std::string("Hello");  // s2 будет иметь тип std::string
}
```

Задачи:

- В примере ниже создан вектор строк и напечатано его содержимое. Тип итератора имеет очень длинное название (и название будет ещё больше если контейтер будет хранить не просто строки, а что-нибудь посложнее). Используйте `auto`, чтобы упростить код.

  ```
  #include <iostream>
  #include <vector>
  #include <string>

  int main() {
      std::vector<std::string> v {"Cat", "Dog", "Elephant"};
      for (std::vector<std::string>::iterator it = v.begin(); it != v.end(); ++it) {
          std::cout << *it << std::endl;
      }
  }
  ```

- Протестируйте, можно ли использовать `auto` вместо возвращаемого типа функции. Напишите функцию, которая принимает на вход вектор строк и возвращает строку, которая является результатом конкатенации всех строк. Вместо возвращаемого типа используйте `auto`.

- Протестируйте, можно ли создать функцию, которая будет принимать целое число и, в зависимости от этого числа, возвращать значения разных типов. (Если вместо возвращаемого типа используется `auto`).

- Протестируйте, можно ли использовать о указатель с помощью `auto`. Пусть есть такой участок кода:

  ```
  int a = 123;
  auto p = &a;
  auto* q = &a;
  ```

  Какой тип будет у `p` и `q`?

- Функция вычисления факториала, написанная ниже с использованием `auto` не работает.

  ```
  auto factorial(int n) {
      if (n > 0)
          return n * factorial(n - 1);
      return 1;
  }
  ```

  Почему? Исправьте эту функцию, не убирая `auto`.

Часть 2: Пользовательские литералы

Существует возможность перегрузить суффикс литерала, используя оператор `operator""`. Вот как это работает:

```
#include <iostream>
unsigned long long operator"" _k(unsigned long long a) {
    return 1000 * a;
}
void operator"" _print(long double a) {
    std::cout << a << '\n';
}

int main() {
    std::cout << 97_k << std::endl;     // Вызывает _k(97)
    123.5_print;                        // Вызывает _print(123.5)
}
```

Есть множество ограничений для такой перегрузки. Во-первых она работает только с литералами и суффикс нельзя применить, например, к переменной. Во-вторых, типы аргументов, которые может принимать `operator""` сильно ограничены. Ему можно передать такие аргументы:

- `T operator _x(unsigned long long)` – для литералов положительный целых чисел
- `T operator _x(long double)` – для литералов вещественных чисел
- `T operator _x(unsigned char)` – для литералов символов
- `T operator _x(const char*, size_t)` – для строковых литералов

Желательно не злоупотреблять этими операторами и использовать их только в редких случаях, если нужно

Задача

- Напишите перегруженный оператор суффикса `_deg`, так чтобы можно было удобнее работать с градусами. Нужно чтобы работал следующий код:

  ```
  auto angle = 1.0 + 45.0_deg; // В angle запишется 1.7854, так 45 градусов = 0.7854 радиан
  ```

- Напишите перегруженный оператор суффикса `_b` для строкового литерала так, чтобы можно было удобнее работать с двоичными числами:

  ```
  auto a = 20 + "11010"_b; // В a запишется 46
  ```

Литералы `std::string`

Возможность создания пользовательских литералов используется стандартной библиотекой, для создания литералов типа `std::string`. В пространстве имён `std::string_literals` содержится перегруженный оператор, который позволяет проще создавать такие литералы:

```
std::string operator"" s(const char* p, size_t n) {
    return std::string(p, n);
}
```

Вот так их можно создавать в программе:

```
#include <iostream>
#include <string>
using namespace std::string_literals;

int main() {
    auto str = "Cat"s;    // str имеет тип std::string
    std::cout << str + "Dog"s << std::endl;
}
```

Часть 3: Вывод шаблонных аргументов класса

Начиная со стандарта C++17 появилась возможность автоматического вывода шаблонных аргументов классов. Например, в примере ниже больше не нужно указывать шаблонный тип вектора. Компилятор догадается о нём сам.

```
#include <iostream>
#include <vector>

int main() {
    std::vector v {4, 8, 15, 16};
    for (auto it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << std::endl;
    }
}
```

Задачи:

- Создайте вектор, содержащий несколько элементов типа `double` и напечатайте его.
- Предположим, что мы создали вектор строк и попытались добавить к концу каждой строки букву `s` вот так:

  ```
  std::vector v {"Mouse", "Cat", "Dog"};
  for (auto it = v.begin(); it != v.end(); ++it) {
      (*it) += "s";
  }
  ```

  Данный код выдаст ошибку. Почему и как её исправить?
- Создайте контейнер `std::set`, содержащий строки `Mouse`, `Dog`, `Cat`. Напечатайте всё содержимое этого контейнера, оно должно напечататься в алфавитном порядке.
- Создайте вектор, содержащий пары(`std::pair`) целых чисел и напечатайте его. Используйте вывод шаблонных аргументов класса.

Часть 4: Range-based циклы

Циклы, основанные на диапазоне, предоставляют более простой способ обхода контейнера:

```
#include <iostream>
#include <vector>

int main() {
    std::vector v {6, 1, 7, 4};
    for (int num : v) {
        std::cout << num << std::endl;
    }
}
```

Для изменения элементов контейнера при обходе нужно использовать ссылки:

```
for (int& num : v) {
    num += 1;
}
```

Задачи:

- Проверьте, можно ли использовать ключевое слово `auto` внутри таких циклов.
- Пусть у нас есть вектор строк:

  ```
  vector<string> v {"Cat", "Axolotl", "Bear", "Elephant"};
  ```

  - Напишите range-based цикл, который будет печатать все элементы вектора
  - Напишите range-based цикл, который будет добавлять в конец каждой строки символ `s`.
  - Напишите range-based цикл, который будет обращать каждую строку. Используйте стандартную функцию `reverse`.
- Проверьте, можно ли использовать range-based циклы если контейнер является:
  - `std::list`
  - `std::set`
  - `std::map`
  - `std::pair`
  - Обычным массивом
  - `std::string`
  - Строкой в стиле `C`
- Для печати массива целых чисел была написана следующая функция:

  ```
  void print(int array[]) {
      for (int num : array) {
          std::cout << num << std::endl;
      }
  }
  ```

  Оказывается, что она не работает. В чём заключается ошибка?

Часть 5: Direct и copy-list-initialization

По сравнению с C и с ранними стандартами C++ добавились новые способы инициализации объектов. Пусть есть простая структура или класс Book:

```
struct Book {
    string title;
    int pages;
    float price;
};
```

```
class Book {
private:
    string title;
    int pages;
    float price;
public:
    Book(string a, int b, float c)
     : title(a), pages(b), price(c) {};
};
```

Direct-list-initialization

Структуры инициалируются поэлементно, а для объектов класса вызовется конструктор. Работает для всех(?) типов: базов типов, структур, объектов с конструкторами и даже для динамического выделения памяти с new. Поддерживает вложенность.

```
Book a {"Doctor Zhivago", 592, 800};
```

Copy-list-initialization, когда применяется:

- По идее должен создасться временный объект, а потом вызваться конструктор копирования. Но на практике временный объект оптимизируется и эта инициализация аналогична предыдущей.

  ```
  Book a = {"War and Peace", 1225, 800};
  ```

- Правая часть оператора присваивания. Сначала создастся временный объект, а потом вызовется оператор присваивания.

  ```
  a = {"War and Peace", 1225, 800};
  ```

- Аргумент функции. Сначала создастся временный объект, а потом вызывается конструктор копирования, чтобы скопировать объект внутрь функции. На практике же это всё оптимизируется (смотрите Copy elision)

  ```
  void print(Book b) {
      cout << a.title << endl;
  }
  ...
  print({"War and Peace", 1225, 800});
  ```

- Возвращаемое значение функции. Сначала создастся временный объект, а потом вызывается конструктор копирования, чтобы скопировать объект из функции. На практике же это всё оптимизируется (смотрите Return value optimization).

  ```
  Book get_war_and_peace() {
      return {"War and Peace", 1225, 800};
  }
  ```

Более подробный пример и задания – в файле 0copy_list_initialization.cpp.

Copy-elision и return value optimization

Это полезные оптимизации, которые отбрасывают ненужные копирования при передачи объекта в функцию или из функции. Смотрите примеры в файлах 1copy_elision.cpp, 2copy_elision.cpp, 3copy_elision.cpp.

Часть 6: `std::initializer_list`

Применим полученные знания по инициализации для класса `std::vector`. Известно, что если написать так:

```
std::vector<int> v {1, 2, 3, 4, 5};
```

то создастся вектор размера 5, с соответствующими элементами. Причём количество элементов в инициализации может быть произвольным. Это означает, что у вектора должен был вызваться конструктор, который может принимать произвольное количество аргументов.

Такой конструктор у вектора на самом деле есть, но он принимает на вход не произвольное число элементов, а один специальный объект, который называется `std::initializer_list`. Это шаблонный контейнер константных элементов. Он прямо встроен в язык (для него не нужно подключать библиотеки). Его особеность в том, что он автоматически создаётся в некоторых ситуациях при инициализации с помощью фигурных скобочек.

Вот пример функции, которая тоже принимает на вход такой контейнер:

```
#include <iostream>
void print(std::initializer_list<int> elems) {
    for (int el : elems) {
        std::cout << el << " ";
    }
    std::cout << std::endl;
}
int main() {
    print({1, 2, 3});
    print({9, 8, 7, 6, 5});
}
```

`std::initializer_list` следует отличать от инициализации фигурными скобками (это лишь частный случай такой инициализации). Например, в пример предыдущей части он ни разу не использовался.

Задачи:

- Что будут содержать следующие векторы и почему?

  ```
  std::vector<int> v1 {3, 1};
  std::vector<int> v2 = {3, 1};
  std::vector<int> v3(3, 1);
  ```

- Что если создать переменную `a` следующим образом? Какой тип будет у `a`?

  ```
  auto a = {1, 2, 3, 4};
  ```

  Что если в фигурных скобках будут объекты разных типов?

- Будет ли работать такой код?

  ```
  for (auto x : {4, 8, 15, 16}) {
      cout << x << " ";
  }
  ```

- Сделайте функцию `print` из примера выше шаблонной так, чтобы работали следующие вызовы:

  ```
  print({1, 2, 3});
  print({"Cat", "Dog", "Mouse"});
  ```

- Создайте класс `SumInfo` у которого будет конструктор и метод `add` принимающие `std::initializer_list`

  ```
  SumInfo a {1, 2, 3, 9};
  cout << a.getCount() << " " << a.getSum() << endl;  // Напечатает 4 15
  a.add({7, 3, 1});
  cout << a.getCount() << " " << a.getSum() << endl;  // Напечатает 7 26
  ```

Часть 7: Structure binding (структурное связывание)

В стандарте C++17 был добавлен новый вид объявления и инициализации нескольких переменных. В коде ниже мы объявляем переменные a и b одной строкой с помощью структурного связывания.

```
#include <iostream>
#include <utility>

int main() {
    std::pair p {5, 1};
    auto [a, b] = p;

    std::cout << a << " " << b << std::endl;
}
```

Структурное связывание работает только в том случае, если размер контейнера справа известен на стадии компиляции. Например, пары, кортежи(std::tuple), статические массивы, std::array, простые структуры.

Задачи:

- Пусть у нас есть пара:

  ```
  std::pair p {std::string{"Moscow"}, 1147};
  ```

  - Создайте две переменные name и age и присвойте их соответствующим элементам пары.
  - Создайте две ссылки name и age и инициализируйте их соответствующими элементами пары. Убедитесь, что при изменении переменной name меняется и пара p.

- Метод insert контейнера std::set пытается вставить элемент в множество. Если же такой элемент в множестве уже существует, то он ничего с множеством не делает. Но этот метод возвращает пару из итератора на соответствующий элемент и переменной типа bool, которая устанавливается в true если новый элемент был добавлен и в false, если такой элемент уже существовал. Вот пример программы, которая пытается вставить элемент в множество и печатает соответствующее сообщение. В любом случае программа печатает все элементы, меньшие вставляемого.

  ```
  #include <iostream>
  #include <utility>
  #include <set>
  using std::cout;
  using std::endl;
  int main() {
      std::set<int> s {1, 2, 4, 5, 9};

      std::pair<std::set<int>::iterator, bool> result = s.insert(5);
      if (result.second == true) {
          cout << "Element added successfully" << endl;
      }
      else {
          cout << "Element already existed" << endl;
      }

      for (std::set<int>::iterator it = s.begin(); it != result.first; ++it) {
          cout << *it << " ";
      }
  }
  ```

  Упростите эту программу, используя ключевое слово auto и структурное связывание.

Структурное связывание можно использовать и в цикле.

```
#include <iostream>
#include <utility>
#include <vector>

int main() {
    std::vector<std::pair<std::string, int>> v {{"Moscow", 1147}, {"Berlin", 1237},
                                                {"Rome", -753},  {"Bogota ", 1538}};
    for (auto [city, year] : v) {
        std::cout << city << " " << year << std::endl;
    }
}
```

Задачи:

- В файле `books.cpp` лежит заготовка кода. В ней содержится инициализированный массив из структур. Сделайте следующее:
  - Напечатайте массив `books`, используя range-based цикл. Нужно напечатать все поля через запятую.
  - Напечатайте массив `books`, используя range-based цикл со структурным связыванием.
  - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл.
  - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл со структурным связыванием.
- Ниже есть пример программы – решение задачи с предыдущего семинара. Она считывает слова и печатает количества всех введённых до этого слов.

```
#include <iostream>
#include <map>
#include <utility>
#include <string>
using std::cout;
using std::endl;

int main() {
    std::map<std::string, int> word_count;

    while (true) {
        std::string word;
        std::cin >> word;
        std::pair<std::string, int> wc {word, 1};
        std::pair<std::map<std::string, int>::iterator, bool> p = word_count.insert(wc);
        if (p.second == false) {
            word_count[word] += 1;
        }

        cout << "Dictionary:" << endl;
        for (std::map<std::string, int>::iterator it = word_count.begin();
                                it != word_count.end(); ++it) {
            std::cout << (*it).first << ": " << (*it).second << endl;
        }
        cout << endl;
    }
}
```

Упростите код этой программы, используя `auto` и структурное связывание.

Часть 8: Введение в лямбда-функции

Лямбда-функции – это анонимные функции, с которыми можно работать как с обычными переменными. Тип лямбда функции разный для каждой лямбда функции и мы его не можем узнать (это **не** указатели на функцию). Но это не проблема, переменную лямбда-функцию всё равно можно создать, используя ключевое слово `auto`.

```
#include <iostream>
int main() {
    auto f = [](int x) {std::cout << x << std::endl;};
    f(123);
}
```

Самый частый случай применения лямбда-функций – это их передача аргументом в функцию. В частности, многие стандартные алгоритмы STL могут принимать на вход лямбда-функции. Например, функция `std::sort` третьим аргументом может принять лямбда-функцию-компаратор. А функция `std::for_each` вызывает лямбда-функцию поочерёдно от каждого элемента.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector v {18, 51, 2, 25, 14, 97, 73};
    std::sort(v.begin(), v.end(), [](int a, int b) {return a > b;});
    std::for_each(v.begin(), v.end(), [](int a) {std::cout << a << " ";});
}
```

- Отсортируйте массив чисел по возрастанию последней цифры числа, используя лямбда функции.
- Предположим, что у нас есть вектор строк:

  ```
  std::vector {"Elephant"s, "Cat"s, "Zebra"s, "Dog"s, "Hippopotamus"s, "Mouse"s, "Tiger"s};
  ```

  - Отсортируйте его лексиграфически по возрастанию
  - Отсортируйте его лексиграфически по убыванию
  - Отсортируйте его по длине каждой строки по возрастанию
- Используйте функцию `std::transform` лямбда функцию и `std::back_inserter`, чтобы по вектору `v` создать новый вектор, содержащий последние цифры чисел.
- В файле `movies.cpp` содержится заготовка кода. Отсортируйте массив `movies`, используя лямбда-функции:
  - по рейтингу
  - по названию
  - по дате
- Измените массив `movies`, с помощью `std::transform` и лямбда функций, так, чтобы
  - рейтинг каждого фильма уменьшился на 1
  - название каждого фильма было переведено в верхний регистр
- Создайте новый вектор, который будет содержать только фильмы с рейтингом `8` и выше. Используйте функцию `copy_if`, лямбда выражение и `std::back_inserter`.
- Удалите все фильмы из массива, который вышли в 90-е годы. Используйте функцию `remove_if` и `erase` и лямбда выражение.