Неделя 3

Переменные в C++. Пользовательские типы данных

3.1. Алгоритмы. Лямбда-выражения

3.1.1. Вычисление минимума и максимума

Напишем функцию Min, которая будет принимать два числа, вычислять минимальное и возвращать его:

```
int Min(int a, int b){
  if (a < b) {
    return a;
  }
  return b;
}
```

Аналогично реализуем функцию, которая будет возвращать максимум из двух чисел:

```
int Max(int a, int b){
  if (a > b) {
    return a;
  }
  return b;
}
```

Проверим, как эти функции работают:

```
cout << Min(2, 3) << endl; // 2
cout << Max(2, 3) << endl; // 3
```

Чтобы реализовать функцию нахождения минимума и масимума других типов, их пришлось бы определять дополнительно.

Но в стандартной библиотеке C++ существуют встроенные функции вычисления минимума и максимума, которые могут работать с переменными различных типов, которые могут сравниваться друг с другом.

Для работы со стандартными алгоритмами нужно подключить заголовочный файл:

```
#include <algorithm>
```

Теперь остается изменить первую букву в вызовах функции с большой на маленькую, чтобы использовать встроенные функции:

```
cout << min(2, 3) << endl;
cout << max(2, 3) << endl;
```

Использование встроенных функций позволяет избежать ошибок, связанных с повторной реализацией их функциональности.

Точно также можно искать минимум и максимум двух строк:

```
string s1 = "abc";
string s2 = "bca";
cout << min(s1, s2) << endl;
cout << max(s1, s2) << endl;
```

Точно так же можно искать минимум и максимум всех типов, которые можно сравнивать между собой, то есть для которых определен оператор <.

3.1.2. Сортировка

Пусть необходимо отсортировать вектор целых чисел:

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

Для удобства определим функцию, выводящую значения вектора в консоль:

```
void Print(const vector<int>& v, const string& title){
  cout << title << ": ";
  for (auto i : v) {
    cout << i << ' ';
  }
}
```

Вторым параметром передается строка `title`, которая будет выводиться перед выводом вектора.

Распечатаем вектор до сортировки с «заголовком» `"init"`:

```
Print(v, "init");
```

После этого воспользуемся функцией сортировки. Чтобы это сделать, ей нужно передать начало и конец интервала, который нужно отсортировать. Взять начало и конец интервала можно с помощью встроенных функций `begin` (возвращает начало вектора) и `end` (возвращает конец вектора):

```
sort(begin(v), end(v));
```

После этого распечатаем вектор с меткой «sort»:

```
cout << endl;
Print(v, "sort");
```

Результат работы программы:

```
init: 1 3 2 5 4
sort: 1 2 3 4 5
```

Программа работает так, как и ожидалось.

3.1.3. Подсчет количества вхождений конкретного элемента

Допустим, необходимо подсчитать сколько раз конкретное значение встречается в контейнере.

Например, необходимо подсчитать количество элементов «2» в векторе из целых чисел. Для этого можно воспользоваться циклом range-based for:

```
vector<int> v = {
  1, 3, 2, 5, 4
};
int cnt = 0;
for (auto i : v) {
  if (i == 2) {
    ++cnt;
  }
}
cout << cnt;
```

Несмотря на то, что этот код работает, не следует подсчитывать число вхождений таким образом, поскольку в стандартной библиотеке есть специальная функция.

Функция count принимает начало и конец интервала, на котором она работает. Третим аргументом она принимает элемент, количество вхождений которого надо подсчитать.

```
vector<int> v = {
  1, 3, 2, 5, 4
};
cout << count(begin(v), end(v), 2);
```

3.1.4. Подсчет количества элементов, которые удовлетворяют некоторому условием

Подсчитать количество элементов, которые обладают некоторым свойством, можно с помощью функции count_if. В качестве третьего аргумента в этом случае нужно передать функцию, которая принимает в качестве аргумента элемент и возвращает true (если условие выполнено) или false (если нет). Чтобы подсчитать количество элементов, которые больше 2, определим внешнюю функцию:

```
bool Gt2(int x) {
  if (x > 2) {
    return true;
  }
  return false;
}
```

Теперь эту функцию можно передать в count_if:

```
vector<int> v = {
  1, 3, 2, 5, 4
};
cout << count_if(begin(v), end(v), Gt2);
```

По аналогии можно определить функцию «меньше двух»:

```
bool Lt2(int x) {
  if (x < 2) {
    return true;
  }
  return false;
}
```

Которую также можно использовать в count_if:

```
cout << count_if(begin(v), end(v), Lt2);
```

Недостаток такого подхода заключается в следующем: функция Gt2 является достаточно специализированной функцией, и вряд ли она будет повторно использоваться. Также определение функции расположено далеко от места ее использования.

3.1.5. Лямбда-выражения

Лямбда-выражения позволяют определять функции на лету — сразу в месте ее использования. Синтаксис следующий: сначала идут квадратные скобки, после которых — аргументы в круглых скобках и тело функции.

```
cout << count_if(begin(v), end(v), [](int x) {
  if (x > 2) {
    return true;
  }
  return false;
});
```

В этом примере лямбда-выражение принимает на вход целое число и возвращает true, если переданное число больше 2.

Пусть необходимо сделать так, чтобы число, с которым происходит сравнение, например, приходило из консоли.

```
int thr;
cin >> thr;
```

Если попытаться воспользоваться этой переменной в лямбда-выражении:

```
cout << count_if(begin(v), end(v), [](int x) {
  if (x > thr) {
    return true;
  }
  return false;
});
```

компилятор выдаст ошибку «'thr' is not captured». Непостредственно использовать в лямбда-выражении переменные из контекста нельзя. Чтобы сообщить, что переменную следует взять из контекста как раз используются квадратные скобки.

```
cout << count_if(begin(v), end(v), [thr](int x) {
  if (x > thr) {
    return true;
  }
  return false;
});
```

3.1.6. Mutable range-based for

Допустим, необходимо увеличить все значения в некотором массиве на 1.

```
vector<int> v = {
  1, 3, 2, 5, 4
};
Print(v, "init");
```

Вывод вектора на экран производится в функции Print:

```
void Print(const vector<int>& v, const string& title){
  cout << title << ": ";
  for (auto i : v) {
    cout << i << ' ';
  }
}
```

Для этого можно воспользоваться обычным циклом for:

```
for (int i = 0; i < v.size(); ++i) {
  ++v[i];
}
cout << endl;
Print(v, "inc");
```

Такая программа отлично работает и выдает ожидаемый от нее результат. Но все же хочется использовать цикл range-based for, так как в этом случае значительно меньше вероятность внести ошибку.

```
for (auto i : v) {
  ++i;
}
```

Но такой код не работает: значения вектора не изменяются, так как по умолчанию на каждой итерации берется копия объекта из контейнера. Получить доступ к объекту в цикле range-based for можно добавив после ключевого слова auto символ &, обозначающий ссылку.

```
for (auto& i : v) {
  ++i;
}
```