Неделя 2

Функции и контейнеры

2.1. Функции

2.1.1. Объявление функции. Возвращаемое значение.

Прежде код программы записывался внутри функции `main`. В данном уроке будет рассмотренно, как определять функции в C++. Для начала, рассмотрим преимущества разбиения кода на функции:

- Программу, код которой разбит на функции, проще понять.
- Правильно подобранное название функции помогает понять ее назначение без необходимости читать ее код.
- Выделение кода в функцию позволяет его повторное использование, что ускоряет написание программ.
- Функции — это единственный способ реализовать рекурсивные алгоритмы.

Теперь можно приступить к написанию первой функции. Объявление функции содержит:

- Тип возвращаемого функцией значения.
- Имя функции.
- Параметры функции. Перечисляются через запятую в круглых скобках после имени функции. Для каждого параметра нужно указать не только его имя, но и тип.
- Тело функции. Расположено в фигурных скобках. Может содержать любые команды языка C++. Для возврата значения из функции используется оператор `return`, который также завершает выполнение функции.

Например, так выглядит функция, которая возвращает сумму двух своих аргументов:

```
int Sum(int x, int y) {
  return x + y;
}
```

Чтобы воспользоваться функцией, достаточно вызвать ее, передав требуемые параметры:

```
int x, y;
cin >> x >> y;
cout << Sum(x, y);
```

Данный код считает два значения из консоли и выведет их сумму.

Важной особенностью оператора `return` является то, что он завершает выполнение функции. Рассмотрим функцию, проверяющую, входит ли слово в некоторый набор слов:

```
bool Contains(vector <string> words, string w) {
  for (auto s : words) {
    if (s == w) {
      return true;
    }
  }
  return false;
}
```

Функция принимает набор строк и строку `w`, для которой надо проверить, входит ли она в заданный набор. Возвращаемое значение — логическое значение (входит или не входит), имеет тип `bool`.

Результат работы функции для нескольких случаев:

```
cout << Contains({"air", "water", "fire"}, "fire");  // 1
cout << Contains({"air", "water", "fire"}, "milk");  // 0
cout << Contains({"air", "water", "fire"}, "water"); // 1
```

Функция работает так, как ожидалось. Стоит отметить, что в C++ значения логического типа выводятся как `0` и `1`. Чтобы лучше понять, как выполнялась функция, запустим отладчик.

Далее представлен код программы с указанием номеров строк и результат пошагового исполнения в виде таблицы. Первый столбец — номер шага, второй — строка, которая исполняется на данном шаге, а третий — значение переменной `s`, определенной внутри функции.

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

using namespace std;

bool Contains(vector <string> words, string w)
 ↪  {
  for (auto s : words) {
    if (s == w) {
      return true;
    }
  }
  return false;
}

int main() {
  cout << Contains({"air", "water", "fire"},
   ↪  "water") << endl; // 1
  return 0;
}
```

№	LN	string s
0	16	-
1	17	-
2	7	-
3	8	air
4	9	air
5	8	water
6	9	water
7	10	water
8	17	-
9	18	-

Видно, что программа в цикле успевает перебрать только первые два значения из вектора, после чего возвращает `true` и выполнение функции прекращается. Этот пример демонстрирует то, что оператор `return` завершает выполнение функции.

Ключевое слово void

Рассмотрим функцию, которая выводит на экран некоторый набор слов, который был передан в качестве параметра.

```cpp
??? PrintWords(vector<string> words) {
  for (auto w : words) {
    cout << w << " ";
  }
} /* */
```

Остается вопрос: что следует написать в качестве типа возвращаемого значения. Функция по своей сути ничего не возвращает и не понятно, какой тип она должна возвращать.

В случаях, когда функция не возвращает никакого значения, в качестве возвращаемого типа используется ключевое слово `void` (*англ.* пустой). Таким образом, код функции будет следующим:

```
void PrintWords(vector<string> words) {
  for (auto w : words) {
    cout << w << " ";
  }
}
```

После того, как функция была определена, ее можно вызвать:

```
PrintWords({"air", "water", "fire"})
```

2.1.2. Передача параметров по значению

Рассмотрим следующую функцию, которая была написана, чтобы устанавливать значение 42 передаваемому ей аргументу:

```
void ChangeInt(int x) {
  x = 42;
}
```

В функции `main` эта функция вызывается с переменной `a` в качестве параметра, значение которой до этого было равно 5.

```
int a = 5;
ChangeInt(a);
cout << a;
```

После вызова функции `ChangeInt` значение переменной `a` выводится на экран. Вопрос: что будет выведено на экран, 5 или 42?

Верный способ проверить — запустить программу и посмотреть. Запустив ее, можно убедиться, что программа выводит «5», то есть значение переменной `a` внутри `main` не поменялось в результате вызова функции `ChangeInt`.

Этот пример призван продемонстрировать то, что параметры функции передаются по значению. Другими словами, в функцию передаются копии значений, переданные ей во время вызова.

Посмотрим на то, как это происходит с помощью пошагового выполнения.

```
1  #include <iostream>
2  using namespace std;
3
4  void ChangeInt(int x) {
5    x = 42;
6  }
7
8  int main() {
9    int a = 5;
10   ChangeInt(a);
11   cout << a;
12   return 0;
13 }
```

№	LN	int a	int x
0	9	-	-
1	10	5	-
2	4	5	5
3	5	5	42
4	11	5	-

Видно, что меняется значение переменной внутри функции `ChangeInt`, а значение переменной в `main` остается тем же.

2.1.3. Передача параметров по ссылке

Поскольку параметры функции передаются по значению, изменение локальных формальных параметров функции не приводит к изменению фактических параметров. Объекты, которые были переданы функции на месте ее вызова, останутся неизменными. Но что делать в случае, если функция по смыслу должна поменять объекты, которые в нее передали.

Допустим, нужно написать функцию, которая обменивает значения двух переменных. Проверим, подходит ли такая функция для этого:

```
void Swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Определим две переменные, a и b:

```
int a = 1;
int b = 2;
```

От правильно работающей функции ожидается, что значения переменных поменяются местами, а именно переменная `a` будет равна двум, а `b` — одному. Применим функцию `Swap`.

```
Swap(a, b);
```

Выведем на экран значения переменных:

```
cout << "a == " << a << '\n'; // 1
cout << "b == " << b << '\n'; // 2
```

Мы видим, что значения переменных не изменились. Действительно, поскольку параметры функции при вызове были скопированы, изменение переменных `x` и `y` никак не привело к изменению переменных внутри функции `main`.

Чтобы реализовать функцию Swap правильно, параметры `x` и `y` нужно передавать по ссылке. Это соответствует тому, что в качестве типа параметров нужно указывать не `int`, а `int&`. После исправления функция принимает вид:

```
void Swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Запустив программу снова, можно убедиться, что функция отработала так, как и ожидалось.

Таким образом, для модификации передаваемых в качестве параметров объектов, их нужно передавать не по значению, а по ссылке. Ссылка — это особый тип языка C++, является синонимом переданного в качестве параметра объекта. Ссылка оформляется с помощью знака & после типа передаваемой переменной.

Можно привести еще один пример, в котором оказывается полезным передача параметров функции по ссылке. Уже говорилось, что в библиотеке algorithm существует функция сортировки. Например, отсортировать вектор из целых чисел можно так:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};
sort(begin(nums), end(nums));
```

Чтобы проверить, что все работает, также выведем элементы вектора на экран:

```
for (auto x : nums) {
  cout <<  x << " ";
}
```

Запускаем программу. Программа выводит: «0 1 2 2 3 6», то есть вектор отсортировался.

Однако, у данного способа есть недостаток: при вызове sort дважды указывается имя вектора, что увеличивает вероятность ошибки из-за невнимательности при написании кода. В результате опечатки программа может не скомпилироваться или, что гораздо хуже, работать неправильно. Поэтому хотелось бы написать такую функцию сортировки, при вызове которой имя вектора нужно указывать лишь раз.

Без использования ссылок такая функция выглядела бы примерно так:

```
vector<int> Sort(vector<int> v) {
  sort(begin(v), end(v));
  return v;
}
```

Она принимает в качестве параметра вектор из целых чисел и возвращает также вектор целых чисел, а внутри выполняет вызов функции sort. Запустим программу:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};
nums = Sort(nums);
for (auto x : nums) {
  cout <<  x << " ";
}
```

Убеждаемся, что программа дает тот же результат. Но мы не избавились от дублирования: мало того, что в месте вызова мы также указываем имя вектора дважды, так и в определении функции Sort тип вектора указывается также два раза.

Перепишем функцию, используя передачу параметра по ссылке:

```
void Sort(vector<int>& v) {
  sort(begin(v), end(v));
}
```

Такая функция уже ничего не возвращает, а изменяет переданный ей в качестве параметра объект. Поэтому при ее вызове указывать имя вектора нужно один раз:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};
Sort(nums);
for (auto x : nums) {
  cout <<  x << " ";
}
```

Именно это и хотелось получить.

2.1.4. Передача параметров по константной ссылке

Раньше было показано, как в C++ можно создавать свои типы данных. А именно была определена структура Person:

```
struct Person {
  string name;
  string surname;
  int age;
};
```

Допустим, что была проведена перепись Москвы и вектор из Person, который содержит в себе данные про всех жителей Москвы, можно получить с помощью функции GetMoscowPopulation:

```
vector<Person> GetMoscowPopulation();
```

Здесь специально не приводится тело этой функции, которое может быть устроено очень сложно, отправлять запросы к базам данных и так далее. Вызвать эту функцию можно так:

```
vector<Person> moscow_population = GetMoscowPopulation();
```

Требуется написать функцию, которая выводит на экран количество людей, живущих в Москве. Эта функция ничего не возвращает, принимает в качестве параметра вектор людей и выводит красивое сообщение:

```
void PrintPopulationSize(vector<Person> p) {
  cout << "There are " << p.size() <<
    " people in Moscow" << endl;
}
```

Воспользуемся этой функцией:

```
vector<Person> moscow_population = GetMoscowPopulation();
PrintPopulationSize(moscow_population);
```

Программа вывела, что в Москве 12500000 людей: «There are 12500000 people in Moscow».

Замерим время выполнение функции GetMoscowPopulation и функции PrintPopulationSize. Подключим специальную библиотеку для работы с промежутками времени, которая называется chrono:

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

using namespace std;
using namespace std::chrono;
```

После этого до и после места вызова каждой из интересующих функций получим текущее значение времени, а затем выведем на экран разницу:

```
auto start = steady_clock::now();
vector<Person> moscow_population = GetMoscowPopulation();
auto finish = steady_clock::now();
cout << "GetMoscowPopulation "
  << duration_cast<milliseconds>(finish - start).count()
  << " ms" << endl;

start = steady_clock::now();
PrintPopulationSize(moscow_population);
finish = steady_clock::now();
cout << "PrintPopulationSize "
  << duration_cast<milliseconds>(finish - start).count()
  << " ms" << endl;
```

В результате получаем:

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 1034 ms
```

Получается, что функция, которая возвращает вектор из 12 миллионов строк, работает быстрее функции, которая всего-то печатает размер этого вектора. Функция PrintPopulationSize ничего больше не делает, но работает дольше.

Но мы уже говорили, что при передаче параметров в функции происходит полное глубокое копирование передаваемых переменных, в данном случае — вектора из 12 500 000 элементов. Фактически, чтобы вывести на экран размер вектора, мы тратим целую секунду на его полное копирование. С этим нужно как-то бороться.

Избежать копирования можно с помощью передачи параметров по ссылке:

```
void PrintPopulationSize(vector<Person>& p) {
  cout << "There are " << p.size() <<
    " people in Moscow" << endl;
}
```

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 0 ms
```

Теперь все работает хорошо, но у данного способа есть несколько недостатков:

- Передача параметра по ссылке — способ изменить переданный объект. Но в данном случае функция не меняет объект, а просто печатает его размер. Объявление этой функции

  ```
  void PrintPopulationSize(vector<Person>& p)
  ```

  может сбивать с толку. Может создаться впечатление, что функция как-то меняет свой аргумент.

- В случае, если промежуточная переменная не создается:

  ```
  PrintPopulationSize(GetMoscowPopulation());)
  ```

  программа даже не скомпилируется. Дело в том, что в C++ результат вызова функции не может быть передан по ссылке в другую функцию (почему это так будет сказано позже в курсе).

Получается, что при передаче по значению, мы вынуждены мириться с глубоким копированием всего вектора при каждом вызове функции печати размера, а при передаче по ссылке — мириться с вышеназванными двумя проблемами. Существует ли идеальное решение без всех этих недостатков?

Выход заключается в использовании передачи параметров по так называемой константной ссылке. Это делается с помощью ключевого слова `const`, которое добавляется слева от типа параметра. Символ & остается на месте и указывает, что происходит передача по ссылке.

Определение функции принимает вид:

```
void PrintPopulationSize(const vector<Person>& p) {
  cout << "There are " << p.size() <<
    " people in Moscow" << endl;
}
```

В результате PrintPopulationSize выполняется за 0 мс, а также работает передача результата вызова функции в качестве параметра другой функции по константной ссылке:

```
PrintPopulationSize(GetMoscowPopulation());)
```

Также мы не вводим в заблуждение пользователей нашей функции и явно указываем, что параметр не будет изменен, так как он передается по константной ссылке.

Такой подход также защищает от случайного изменения фактических параметров функций. Допустим, по ошибке в функцию печати количества

людей в Москве попал код, добавляющий туда одного жителя Санкт-Петербурга.

```
void PrintPopulationSize(const vector<Person>& p) {
  cout << "There are " << p.size() <<
    " people in Moscow" << endl;
  p.push_back({"Vladimir", "Petrov", 40});
}
```

В случае передачи по ссылке такая ошибка могла бы остаться незамеченной, но при передаче по константной ссылке такая программа даже не скомпилируется:

```
main.cpp: In function 'void PrintPopulationSize(const std::vector<
   Person>&)':
main.cpp:20:41: error: passing 'const std::vector<Person>' as 'this
   ' argument discards qualifiers [-fpermissive]
  p.push_back({"Vladimir", "Petrov", 40});
                                        ^
```

Компилятор в таком случае выдает ошибку, так как нельзя изменять принятые по константной ссылке фактические параметры.

2.2. Модификатор const как защита от случайных изменений

На самом деле **const** — специальный модификатор типа переменной, запрещающий изменение данных, содержащихся в ней.

Например, рассмотрим следующий код:

```
int x = 5;
x = 6;
x += 4;
cout << x;
```

В этом коде переменная **x** изменяется в двух местах. Как несложно убедиться, в результате будет выведено «10». Добавим ключевое слово const, не меняя ничего более:

```
const int x = 5;
x = 6;
x += 4;
cout << x;
```

При попытке скомпилировать этот код, компилятор выдает следующие сообщения об ошибках:

```
main.cpp: In function 'int main()':
main.cpp:9:7: error: assignment of read-only variable 'x'
   x = 6;
       ^
main.cpp:10:8: error: assignment of read-only variable 'x'
   x += 4;
        ^
```

Обе строчки, в которых переменная подвергается изменениям, приводят к ошибкам. Закомментируем их:

```
const int x = 5;
//x = 6;
//x += 4;
cout << x;
```

Теперь программа компилируется как надо и выводит «5». Чтение переменной является немодифицирующей операцией и не вызывает ошибок при компиляции.

Рассмотрим пример со строковой переменной `s`:

```
string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s << endl;
```

Здесь представлены операции: получение длины строки, добавление текста в конец строки, инициализация другой строки значением `s+'!'`, вывод значения строки в консоль. Запускаем программу:

```
5
hello, world
hello, world!
```

Теперь добавляем модификатор `const`.

```
const string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s;
```

При компиляции только в одном месте выводится ошибка:

```
basic_string.h:1131:7: note:   in call to 'std::__cxx11::
   basic_string<_CharT, _Traits, _Alloc>& std::__cxx11::
   basic_string<_CharT, _Traits, _Alloc>::operator+=(const _CharT*)
    [with _CharT = char; _Traits = std::char_traits<char>; _Alloc =
    std::allocator<char>]'
       operator+=(const _CharT* __s)
       ^~~~~~~~
```

Вывод длины строки, использование строки при инициализации другой строки и вывод строки в консоль — немодифицирующие операции и ошибок не вызывают. А вот добавление в конец строки еще как-либо текста — уже нет. Закомментируем соответствующую строку:

```
const string s = "hello";
cout << s.size() << endl;
//s += ", world";
string t = s + "!";
cout << s;
```

```
5
hello
hello!
```

Более сложный пример: рассмотрим вектор строк и попытаемся изменить первую букву первого слова этого вектора с прописной на заглавную:

```
vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

Программа успешно компилируется и выводит «Hello» как и ожидалось. Установим модификатор `const`.

```
const vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

В итоге компиляция завершается ошибкой:

```
main.cpp:9:13: error: assignment of read-only location '(& w.std::
   vector<std::__cxx11::basic_string<char> >::operator[](0))->std::
   __cxx11::basic_string<char>::operator[](0)'
   w[0][0] = 'H';
```

Здесь важно отметить следующее: мы не меняем вектор непосредственно (не добавляем элементы, не меняем его размер), а модифицируем только его элемент. Но в C++ модификатор const распространяется и на элементы контейнеров, что и демонстрируется в данном примере.

Зачем вообще в C++ нужен модификатор const?

Главное предназначение модификатора const — помочь программисту не допускать ошибок, связанных с ненамеренными модификациями переменных. Мы можем пометить ключевым словом const те переменные, которые не хотим изменять, и компилятор выдаст ошибку в том месте, где происходит ее изменение. Это позволяет экономить время при написании кода, так как избавляет от мучительных часов отладки.