Лекция VII

14 декабря 2018

Файловый ввод/вывод

Неформатированный ввод/вывод - предназначен для записи/чтения строго определённого количества байт. Для потоковых объектов доступны следующие методы:

- Поток вывода: записать байты в файл

```
ostream& stream_var.write(const char *ptr,
                          size_t count)
```

- Поток ввода: прочитать байты из файла:

```
istream& stream_var.read(char* ptr,
                         size_t count);
```

, где **ptr** - указатель на начало блока памяти (либо куда записываются байты, либо откуда берутся для вывода в файл); **count** - количество байт (размер данных для ввода/вывода), **stream_var** - переменная соответствующего потока.

Возращаемое значение: ссылка на самого себя

Неформатированный ввод/вывод

Задача: одной программой записать в файл заданное количество массивов целых чисел из 10 элементов. Второй программой - определить количество записанных массивов (сколько штук) и загрузить один из них по выбору

Для решения подобной задачи используется двоичный режим открытия файла дял чтения/записи.

Неформатированный ввод/вывод: запись в файл.

```
1  const size_t SZ = 10;
2  ofstream out_arrays{"arrays.bin", ios_base::binary};
3
4  if (out_arrays.is_open()) {
5    int *arr = new int[SZ];
6    size_t how_many;
7
8    cout << "Введите количество массивов: ";
9    cin >> how_many;
10
11   for (size_t att = 1; att <= how_many; ++att) {
12     for (size_t i = 0; i < SZ; ++i) {
13       arr[i] = rand_a_b_incl(-5, 7);
14     }
15     const char *start_ptr = static_cast<char*>(arr);
16     out_arrays.write(start_ptr, SZ * sizeof(int));
17     if ( !out_arrays ) { break; }
18   }
19   delete[] arr;
20 }
```

Неформатированный ввод/вывод

Объект потока ввода/вывода имеет поле, сохраняющее его **позицию** в файле: на каком байте от начала файла находится поток (смещение происходит в результате операций ввода/вывода).

Узнать текущую позицию потока:

```
streampos out_stream.tellp(); //для потоков вывода

streampos in_stream.tellg();  //для потоков ввода
```

В случае ошибки - методы вернут значение **-1**, сам поток переходит в состояние *ошибки*. В случае успеха - количество байт от начала файла.

streampos - специальный тип данных, совместимый с знаковым целым типом (без проблем преобразуется в него неявно), достаточный для хранения файлов максимального размера в ОС

Неформатированный ввод/вывод

Изменить позицию потока:

```
ostream& out_stream.seekp(streampos pos);
ostream& out_stream.seekp(streamoff offset, way);

ifstream& in_stream.seekg(streampos pos);
ifstream& in_stream.seekg(streamoff offset, way);
```

pos - позиция в конкретном файле.
offset - отступ от некоторой позиции в файле, заданной аргументом **way**. В качестве последнего используются три константы: **ios_base::beg** (начало файла), **ios_base::cur** (текущая позиция), **ios_base::end** (конец файла).
streamoff - как правило, *псевдоним* одного из знаковых целочисленных типов данных.

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.

Что нужно для второй программы?

1. Узнать количество массивов в файле
2. Запросить номер загружаемого массива
3. Считать нужный массив из файла

Неформатированный ввод/вывод: чтение из файла.

```
1  const size_t SZ = 10, ARR_BYTES = sizeof(int) * SZ;
2  ifstream in_obj{"arrays.bin", ios_base::binary};
3
4  if (in_obj.is_open()) {
5    in_obj.seekg(0, ios_base::end); // Шаг (1) начат
6    long how_many = in_arrays.tellg(in_stream) / ↩
         ARR_BYTES;
7    if ( !in_obj || how_many == 0) {
8      cerr << "Нет массивов в файле"; exit(1);
9    }
10   in_obj.seekg(0, ios_base::beg); // Шаг (1) выполнен
11
12   size_t arr_num = 0;  // Шаг (2) начат
13   do {
14     cout << "Введите номер (всего - " << how_many
15          << "): ";
16     cin >> arr_num;
17   } while (arr_num < 1 || arr_num > how_many);
18                        // Шаг (2) выполнен
19   // Продолжение — ниже
```

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.

```
19    // Начало – выше
20    arr_num--; // Для вычисления смещения. Шаг (3) начат
21    int *arr = new int[SZ];
22    in_obj.seekg(arr_num * ARR_BYTES, ios_base::beg);
23    char *start_ptr = static_cast<char*>(arr);
24    in_obj.read(start_ptr, ARR_BYTES);
25    // Узнать количество реально считанных байт
26    size_t read = in_obj.gcount()   // Шаг (3) выполнен
27
28    if ( read != SZ ) {
29      cerr << "Количество элементов меньше 10";
30      exit(1);
31    }
32    cout << "Прочитанный массив:\n  ";
33    for (size_t i = 0; i < SZ; ++i) {
34      cout << arr[i] << ' ';
35    }
36  }
```

Файловый ввод/вывод

Ещё примеры на неформатированный ввод/вывод и методы **tellg(p)/seekg(p)**

https://github.com/posgen/OmsuMaterials/blob/master/2course/Programming/examples/8_file_operations_example/rewrite_example.cpp

https://github.com/posgen/OmsuMaterials/blob/master/2course/Programming/examples/8_file_operations_example/save_and_get_structs.cpp

Общая справка по файловому вводу-выводу C++ также доступна здесь: https://github.com/posgen/OmsuMaterials/wiki/File-input-output

Перечисления (Enumerations)

Перечисления

Переисления - это пользовательский тип данных, состоящий из *ограниченного* набора констант **целого типа**. По умолчанию, типом каждой константы является **int**. В современном C++ перечисления делятся на

Открытые (unscoped) - каждая константа становится доступной глобально по имени и допускается неявное приведение значений констант к числовым типам данных. Ключевое слово для объявления:

enum

Закрытые (scoped) - каждая константа доступна только через название перечисления с использованием оператора **::** и своего имени. Не допускаются неявные преобразования в числовые типы данных. Ключевое слово для объявления:

enum class

Открытые перечисления

Синтаксис определения перечисления:

```
enum <название_перечисления>
{
  <константа_1> [= <значение>],
  [<константа_2>, <константа_3>, ...]
};
```

1. По умолчанию значение первой константы перечисления равно **нулю**.
2. Каждая константа, кроме первой, получает **на единицу большее значение**, чем предшествующая.
3. Каждой константе может быть присвоено **произвольное значение целого типа**.
4. Как только константе присваивается значение, то все следующие за ней меняются по **второму пункту**.
5. Разные константы могут иметь **одинаковые значения**.

Открытые перечисления

Пример простого перечисления

```
1  enum ComputingState
2  {
3    NOT_STARTED, // значение — 0
4    STARTED,     // 1
5    COMPLETED    // 2
6  };
7
8  // Значения неявно приводятся к типу int
9  // и печатаются как числа
10 cout << NOT_STARTED << '\n';
11 cout << STARTED << '\n';
12 cout << ComputingState::COMPLETED;
```

Открытые перечисления

Пример: использование переменных

```
1  enum ComputingState
2  {
3    NOT_STARTED = 7,    // 7
4    STARTED,            // 8
5    COMPLETED   = 11    // 11
6  };
7
8  ComputingState bound_task;
9  bound_task = STARTED;
10 cout << bound_task << '\n';
11
12 // Поля перечислений могут участвовать
13 // в числовых операциях
14 int value = (COMPLETED * 2) & STARTED;
15 bool equals = (value == STARTED);
```

Открытые перечисления

Пример: возращение значений из функции

```
1  enum ComputingState
2  {  NOT_STARTED, STARTED, COMPLETED };
3
4  ComputingState solve_smth(int steps, double &result)
5  {
6    ComputingState status;
7
8    if ( steps < 10 ) {
9      result = 10.0; status = NOT_STARTED;
10   } else if ( steps >= 10 && steps <= 20 ) {
11     result = 55.873; status = STARTED;
12   } else {
13     result = 99.99; status = COMPLETED;
14   }
15
16   return status;
17 }
18
19 double result;
20 ComputingState calc_state = solve_smth(25, result);
```

Открытые перечисления

Пример: форматированный вывод значения перечисления на экран (или файл)

```
1  #include <ostream>
2
3  enum ConsoleColor
4  { RED, GREEN, YELLOW, PURPLE };
5
6  // Демонстрация перегрузки оператора вывода
7  // для пользовательского типа данных
8  std::ostream& operator<<(std::ostream& os, ConsoleColor c)
9  {
10   switch (c)
11   {
12     case RED     :  os << "{красный}";     break;
13     case GREEN   :  os << "{зелёный}";     break;
14     case YELLOW  :  os << "{жёлтый}";      break;
15     case PURPLE  :  os << "{фиолетовый}"; break;
16     default      :  os << "{нет никакого цвета}";
17   }
18   return os;
19 }
20
21 ConsoleColor color = YELLOW;
22 cout << color << endl;
```

Закрытые перечисления

Синтаксис определения **закрытого перечисления**:

```
enum class <название_перечисления>
{
  <константа_1> [= <значение>],
  [<константа_2>, <константа_3>, ...]
};
```

При определении все параметры задаются в точности также, как и для *открытых* перечислений на слайде 13.

Закрытые перечисления

```
1  enum class Output {  CONSOLE_TEXT, FILE_TEXT = 20,
2    FILE_BINARY, FILE_HTML, FILE_XML };
3
4  Output choise;
5
6  // Допустимая операция
7  choise = Output::FILE_TEXT;
8  // Допустимая операция: явное приведение к int
9  int status = int(choise) * 2;
10 cout << int(choise) << '\n';
11
12 // Недопустимая: нет названия перечисления
13 // choise = FILE_XML
14
15 // Недопустимая: нет перегрузки оператора вывода
16 // cout << choise << std::endl;
17
18 // Недопустимые: нет неявного приведения к int
19 // int some_num = choise + 2;
20 // bool equals_to_zero = (choise == 0);
```

Указатель на функцию (function pointer)

Указатель на функцию

Указатель на функцию - указатели специального типа, позволяющие использовать функции языка как переменные. Их основные характеристики:

- позволяют передавать функции как аргументы в другие функции;
- позволяют объявлять массивы функций, одинаковых по типу возвращаемого значения и со совпадающим списком аргументов;
- позволяют делать отложенный вызов функций;
- не требуют разыменования;
- не требуют явного присвоения адреса существующей функции.

Общий синтаксис:

```
<тип_возвращаемого_значения>
          (*<имя_указателя>) (<типы_аргументов>);
```

Указатель на функцию

Пример использования

```
1  char up_character(char symbol)
2  {
3    if (symbol < 'a' || symbol > 'z')
4      return symbol;
5
6    return symbol - 32;
7  }
8
9  char (*p_func)(char);
10 // Ниже символ & можно не указывать
11 p_func = up_character;
12
13 char str[] = "dhs3%#@Js@Edhwh82h2e3*hIk";
14 for (char sym : str) {
15   cout << p_func(sym);
16 }
```

Указатель на функцию

Пример использования совместно с псевдонимами

```
1  using func_x_ptr = double (*)(double);
2
3  // Объявляем 2 указателя на функцию
4  // вида double func_name(double);
5  func_x_ptr f1, f2;
6
7  f1 = sin;
8  f2 = log;
9
10 cout << f1(5.5 * M_PI) << endl;
11 cout << f2(5.5 * M_PI) << endl;
```

Указатель на функцию

Уже было: передача функции сравнения в функцию сортировки

```
1  #include <algorithm>
2
3  bool my_compare(int left, int right)
4  {
5
6    return abs(left) > abs(right);
7  }
8
9  int arr1[] = { 3, 1, 5, 4, 3, 2,
10                1, 8, 4, 76, 4, 67 };
11 sort(arr1, arr1 + 12, my_compare);
12
13 cout << "После сортировки: ";
14 for (int elem : arr1) {
15   cout << elem << ' ';
16 }
17 cout << endl;
```

Указатель на функцию

Пример: вычисление одномерного интеграла методом прямоугольников

```
1  double integrate(double left, double right, size_t split_num,
2                   double (*f)(double))
3  {
4    if (split_num == 0) { split_num = 5; }
5
6    double h = (right - left) / split_num, result = 0;
7    for (unsigned i = 1; i <= split_num; ++i) {
8      result += h * f(left + i * h);
9    }
10
11   return result;
12 }
13
14 double fun_x(double x) { return x; }
15
16 cout << "100 разбиений: " << integrate(0.0, 1.0, 100, fun_x) ↩
       << '\n';
17 cout << "10000 разбиений: " << integrate(0.0, 1.0, 10000, ↩
       fun_x) << '\n';
18 cout << "10000 разбиений: " << integrate(0.0, 1.0, 10000, exp)↩
        << '\n';
```

Препроцессор в C++

Сборка программ

Схематично, создание исполняемого или библиотечного файла состоит из трёх шагов, выполняемых компилятором:

1. **Препроцессинг**: обработка исходного текста программы с раскрытием специальных "команд"
2. **Компиляция**: преобразование расширенного исходного файла(-ов) в объектный(-ые), содержащий представление на языке *ассемблера* (создание объектного файла)
3. **Связывание** (linking): преобразование объектного файла программы в двоичный файл (исполняемый или библиотечный) для данной операционной системы

Директивы препроцессора

Директивы, использующиеся для замены одного текста другим (определение макросов):

```
(1) #define <идентификатор>
(2) #define <идентификатор> [текст_для_замены]
(3) #define <идентификатор>(<параметры>) <текст>
(4) #undef  <идентификатор>
```

1. Определяет **идентификатор** для пустого макроса
2. Определяет макрос замены **идентификатора** на **текст_для_замены**
3. **Идентификатор** может получать параметры и использовать в подстовляемом тексте. Синтаксис параметров аналогичен функциям, за исключением отсутствия каких-либо упоминаний об типах
4. Отменяет любой ранее опредлённый **идентификатор**

Примеры **макросов**

```
1  #define ROWS 10
2  #define COLS 15
3
4  #define AUTHOR "Это я"
5
6  #define MAX(x, y) (x > y) ? x : y
7  ...
8
9  double matrix[ROWS][COLS];
10 /* После работы препроцессора, в исходном
11    файле появляется строка:
12      double matrix[10][15];
13 */
14
15 cout << AUTHOR;
16 // cout << "Это я";
17
18 int val = MAX(15, -8);
19 // int val = (15 < -8) ? 15 : -8;
```

Примеры **макросов**

```
1  #define FUNCTION(name, a) int fun_##name() { return a;}
2
3  FUNCTION(first, 12)
4  FUNCTION(second, 2)
5  FUNCTION(third, 23)
6
7  #undef FUNCTION
8  #define FUNCTION 34
9  #define OUTPUT(a) cout << #a "\n";
10
11
12 cout << "first: " << fun_first() << '\n';
13 cout << "first: " << fun_second() << '\n';
14 cout << "first: " << fun_third() << '\n';
15
16 cout << "Значение FUNCTION: " << FUNCTION << '\n';
17
18 OUTPUT(Русский текст без кавычек и переносов!);
```

Директивы препроцессора

Условные директивы, использующиемся для задание логики при препроцессинге:

```
(1) #if <выражение>
(2) #ifdef <выражение>
(3) #ifndef <выражение>
(4) #elif <выражение>
(5) #else
(6) #endif
```

Директивы препроцессора

Пример использования **условных директив**

```
1 #if defined(WINDOWS_H)
2   #error Не буду компилироваться в ОС Windows
3 #endif
```

Директивы препроцессора

Пример использования **условных директив**

```
1  #define MACROS1 2
2
3  #ifdef MACROS1
4    printf("1: определён\n");
5  #else
6    printf("1: не определён\n");
7  #endif
8
9  #ifndef MACROS1
10   printf("2: не определён\n");
11 #elif MACROS1 == 2
12   printf("2: определён\n");
13 #else
14   printf("2: не определён\n");
15 #endif
16
17 #if !defined(DCBA) && (MACROS1 < 2*4-3)
18   printf("3: выражение истинно\n");
19 #endif
```

Директивы препроцессора

Встроенные макросы - проверить самостоятельно, что произойдёт

```
1 cout << __DATE__ " " __TIME__ "\n";
2 cout << __FILE__ "\n";
```

Директивы препроцессора

Директивы, использующиеся для включения других исходных файлов

```
(1) #include <file_name>
(2) #include "file_name"
```

Вообще говоря - две эквивалентные формы включения стандартных или внешних **библиотек** (файлов, которые предоставляют некоторый набор констант, переменных, функций, структур и т.п. для решения каких-либо задач). Разница только в том, что форма **(2)** сначало ищет указанный файл **filename** в той же директории, что и файл, который хотим скомпилировать. Если не найден - делается попытка поиска в *стандартных путях поиска*. Форма **(1)** - производит поиск только в стандартных путях.
Стандартные пути поиска библиотек зависят от способа, как компилятор языка был установлен в ОС, а также могут быть добавлены с помощью дополнительных опций компилятора.

Пространства имён (**namespaces**)

Пространства имён

В C++ любое **объявление** и **определение** переменной, функции, псевдонимов, пользовательских типов данных (структуры, классы, перечисления, объединения) может быть помещено в **пространство имён**.

Технически, **пространство имён** - это **лексическая** область видимости для группы *идентификаторов*.

По смыслу, **пространство имён** - это именованное множество, название которого необходимо для точного указания некоторой переменной, функции или типа данных.

Пространства имён в C++ - **открыты для расширения**: в любое из них (хоть из стандратной библиотеки, хоть из собственной, хоть из внешней) каждая программа может добавить свой набор констант, функций, типов, объектов и прочего.

Пространства имён

Пространство имён создаётся с помощью ключевого слова **namespace** и выбора названия. Например,

```
1  namespace ff_omsu
2  {
3  const size_t players = 7;
4
5  enum class StudyState { READY, SOMETIMES_LATER, NEVER };
6
7  bool ready_to_pass(StudyState status)
8  {
9    return status == StudyState::READY;
10 }
11
12 struct Participant
13 {
14   std::string name;
15   StudyState status;
16 };
17 }
```

Имя пространства имён - **ff_omsu**, внутри него содержатся - константа, перечисление, функция, структура - каждой сущности по одной штуке.

Пространства имён

Само пространство имён ограничено блоком из фигурных скобок (строки 2 и 17 с предыдущего слайда).
Ко всему содержимому пространства имён можно обратиться с помощью его имени и **двойного двоеточия**. В виде кода:

```
18
19 cout << "Константа равна: " << ff_omsu::players << '\n';
20
21 ff_omsu::Participant p1 = {"Nikitin",
22                             ff_omsu::StudyState::READY};
23
24 if ( ff_omsu::ready_to_pass(p1.status) ) {
25   cout << "Студент " << p1.name
26        << ", вероятно, проги сдаст.\n";
27 }
```

Всё как обычно, разве что для всех сущностей появилась постоянная приставка «**ff_omsu::**». Как можно заметить из кода выше, при частом использовании идентификаторов из пространства имён, каждый раз добавлять его имя - утомительно.

Пространства имён

Для упрощения доступа к идентификаторам используется оператор **using**. И часть из вас его видело и не раз:

```
1 using namespace std;
```

- данное использование **using** делает **все идентификаторы** из пространства имён **std** доступными в текущей **лексической области видимости** (другими словами, происходит *импорт названий*).

Лексическая область видимости определяется просто - это или блок внутри пары *фигурных* скобок (и любые вложенные в него подблоки), или весь файл с исходным кодом, начиная со следующей строки. Так, в примере кода выше, **using namespace** используется вне любых скобок, то всё из пространства имён **std** доступно по прямым именам, начиная со второй строки.

Пространства имён

Для собственного пространтсва имён **ff_omsu** правила аналогичны. Для примера ограничения лексической области видимости, рассмотрим код

```
1  int main()
2  {
3    // здесь нужно обращаться с добавлением приставки
4    ff_omsu::Participant person;
5
6    { // создаёт дополнительную область видимости
7      std::cout << "Введите имя: ";
8      std::cin >> person.name;
9
10     using namespace ff_omsu;
11     // далее всё из пространства имён доступно без приставки
12
13     if (person.find("С") == 0 || person.find("Д") == 0) {
14       person.status = StudyState::READY;
15     } else {
16       person.status = StudyState::SOMETIMES_LATER;
17     }
18   }
19   // тут снова имена из *ff_omsu* доступны только с приставкой
20   std::cout << "Студент " << person.name
21   if ( ff_omsu::ready_to_pass(person.status) ) {
22     std::cout << ", наиболее вероятно, проги сдаст!\n";
23   } else {
24     std::cout << " в течении двух лет проги сдаст!\n"
25   }
26 }
```

Пространства имён

С помощью **using** можно делать доступным только часть идентификаторов из пространства имён

```
1  using std::cout;
2  using std::endl;
3
4  using ff_omsu::ready_to_pass;
5  using ff_omsu::Participant;
6
7  Participant p1 = {
8    "Nekto", ff_omsu::StudyState::SOMETIMES_LATER};
9
10 if ( !ready_to_pass(p1.status) ) {
11   cout << "не смог, так не смог" << endl;
12 }
```

В примере после действия операторов **using** название функии (**ready_to_pass**) и структуры (**Participant**) можно использовать напрямую, а вот имя перечисления (**StudyState**) - только с явным указанием пространства имён. **Минус данного подхода** только один - нужно явно каждый идентификатор добавить с оператором **using**.

Пространства имён

Правило хорошего тона

В современном C++ рекомендуется по возможности использовать полный импорт идентификаторов (**using namespace ff_omsu** и ему подобные) только внутри отдельных блоков кода, ограниченных парой *фигурных скобок* (функции, классы, другие пространства имён).

Зачем вообще нужны?

Пространства имён позволяют использовать разные библиотеки, содержащие одинаковые по именованию сущности. Например, если есть класс **Vector** в библиотеках **lib1** и **lib2**, то в своей программе можно использовать обе реализации, если внутри библиотек используются разные пространства имён. И у компилятора не будет никаких претензий.

Пространства имён

К слову, про претензии компилятора: следующий код не скомпилируется:

```
1  namespace sp1
2  {
3  struct MyRecord {};
4  }
5
6  namespace sp2
7  {
8  struct MyRecord {};
9  }
10
11 int main()
12 {
13   using namespace sp1;
14   using namespace sp2;
15   // не компилируется
16   MyRecord mr;
17 }
```

Пространства имён

Для исправления ситуации отлично подходят **псевдонимы**

```
1  namespace sp1
2  {
3  struct MyRecord {};
4  }
5
6  namespace sp2
7  {
8  struct MyRecord {};
9  }
10
11
12 int main()
13 {
14   using MyRecordV1 = sp1::MyRecord;
15   using MyRecordV2 = sp2::MyRecord;
16   // Всё работает
17   MyRecordV2 mr;
18 }
```

Пространства имён

Стандартная библиотека C++

В настоящий момент все типы данных, константы, перечисления и объекты стандартной библиотеки помещаются в пространство имён **std**.

Расширение пространтсва имён

В завершении слайдов о **namespaces**, пример на расширение:

```
1 namespace ff_omsu
2 {
3   const size_t real_players = 5;
4 }
```

такой код работает без проблем, с учётом того, что пространство имён **ff_omsu** уже определено.