

# Лекция VII

8 декабря 2017

```
istream& in_stream.ignore(size_t count,  
                           char delim = EOF);
```

- Метод **ignore** пропускает заданное количество символов (байт) из файла и оставляет их необработанными (то есть не происходит сохранение или преобразование извлечённых символов). Пропуск прекращается или по достижении считывания **count** символов, или при встрече символа-разделителя **delim**.
- Оба параметра метода - **count** и **delim** имеют значения по умолчанию: **count** равен единице, а разделитель **delim** специальному символу (**EOF**), означающему конец файла
- Если пропуск символов прекращается при нахождении разделителя, то он тоже извлекается из потока и не участвует в дальнейших операциях чтения информации

Когда может быть полезен метод **ignore**?

Во многих программах для ввода начальных параметров более уместно использовать *конфигурационные файлы*, вместо ввода значений через консоль. Особенно это относится к вычислительным задачам: граничные условия при расчёте задач по вычислению различных интегралов или систем уравнений; количество частиц и параметры вроде температуры для задач термодинамики; размеры матриц в каких-нибудь вычислениях.

Конфигурационные файлы предпочтительней хотя бы тем, что при изменениях параметров быстрее и надёжнее поменять их в текстовом файле, чем каждый раз сосредотачиваться на консольном вводе.

Пример конфигурационного файла некой абстрактной вычислительной задачи:

Максимальное число итераций:	26
Количество строк матриц:	15
Количество столбцов матриц:	25
Количество слоёв:	5
Сила трения между слоями:	-7.8

Что можно выделить из описания файла на предыдущем слайде?

- Есть повторяющаяся структура: описание параметра - двоеточие - значение
- Программе нужны значения
- Комментарии нужны для человека

Для написания универсального разбора и пригодится метод `ignore`

```
1 const size_t PASS_COUNT = 500;
2 int      max_iter, rows, cols, layers_count;
3 double fric_force;
4
5 ifstream config_file{"config_file.dat"};
6 if ( config_file.is_open() ) {
7     // пропускаем символы до двоеточия
8     config_file.ignore(PASS_COUNT, ':');
9     // безопасно считываем первое значение
10    config_file >> max_iter;
11
12    config_file.ignore(PASS_COUNT, ':');
13    config_file >> rows;
14    // ... Остальные переменные – аналогично
15 }
```

Код с предыдущего слайда разбирает приведённую конфигурацию со следующими особенностями:

- Разумно предположить, что более 500 символов в качестве описания параметра человеку будет просто лень набирать
- Перед вводом *каждого* числового значения ищется символ двоеточия
- После нахождения - считываем числовое значение в нужную переменную

**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.

**Возвращаемое 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 для 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] = real_rnd_a_b(-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) начал
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) начал
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 }
```

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/rewrite_example.cpp)

[https://github.com/posgen/OmsuMaterials/blob/master/2course/Programming/examples/8\\_file\\_operations\\_example/save\\_and\\_get\\_structs.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)

**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.

Практически полностью совместимые с C. 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. **Только для C++**. Ключевое слово для объявления:

**enum class**

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**.

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;
```

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

Синтаксис определения (перечисления данного типа присутствуют только в C++):

```
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 19.



# 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 OneArgFunPtr = double (*)(double);
2
3 // Объявляем 2 указателя на функцию
4 // вида double fun_name(double);
5 OneArgFunPtr f1, f2;
6
7 f1 = sin;
8 f2 = log;
9
10 cout << f1(5.5 * M_PI) << '\n';
11 cout << f2(5.5 * M_PI) << '\n';
```

# 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 (left * right < 0) ? left : 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 (auto elem : arr1) {
15     cout << elem << ' ';
16 }
17 cout << '\n';
```

# 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) <<
17     << '\n';
18 cout << "10000 разбиений: " << integrate(0.0, 1.0, 10000, fun_x) <<
19     << '\n';
20 cout << "10000 разбиений: " << integrate(0.0, 1.0, 10000, exp) <<
21     << '\n';
```

# Препроцессор в C++



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

- 1 **Препроцессинг:** обработка исходного текста программы с раскрытием специальных "команд" в некоторый текст
- 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**, 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>`

- ❶ Определяет **идентификатор** для пустого макроса
- ❷ Определяет макрос замены **идентификатора** на **текст\_для\_замены**
- ❸ **Идентификатор** может получать параметры и использовать в подставляемом тексте. Синтаксис параметров аналогичен функциям, за исключением отсутствия каких-либо упоминаний об типах
- ❹ Отменяет любой ранее определённый **идентификатор**

# Директивы препроцессора

## 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!);
```

**Условные директивы**, используемые для задания логики при препроцессинге:

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

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
```

Встроенные макросы - проверить самостоятельно, что произойдёт

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



**Директивы**, используемые 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)** сначала 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.