


Лекция VIII

25 февраля 2017

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 std::cout << NOT_STARTED << '\n';
11 std::cout << STARTED << '\n';
12 std::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 std::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);
```

Отличия от использования перечислений в С.

- Объявление переменных включает в себя ключевое слово **enum**

```
1 enum ComputingState bound_task;  
2 bound_task = NOT_STARTED;
```

- В С переменной типа *перечисление* может быть присвоено любое значение типа **int**

```
1 // В C++ современные компиляторы  
2 // не позволяют так делать  
3 bound_task = 18;
```

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;
10        status = NOT_STARTED;
11    } else if ( steps >= 10 && steps <= 20 ) {
12        result = 55.873;
13        status = STARTED;
14    } else {
15        result = 99.99;
16        status = COMPLETED;
17    }
18
19    return status;
20 }
21
22 double result;
23 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 std::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 4.

Закрытые перечисления

```
1 enum class Output
2 {
3     CONSOLE_TEXT,
4     FILE_TEXT      = 20,
5     FILE_BINARY, FILE_HTML, FILE_XML
6 };
7
8 Output choise;
9
10 // Допустимая операция
11 choise = Output::FILE_TEXT;
12
13 // Недопустимая: нет названия перечисления
14 // choise = FILE_XML
15
16 // Недопустимая: нет перегрузки оператора вывода
17 // std::cout << choise << std::endl;
18
19 // Недопустимые: нет неявного приведения к int
20 // int some_num = choise + 2;
21 // bool equals_to_zero = (choise == 0);
22
23 // Допустимая операция: явное приведение к int
24 int status = int(choise) * 2;
```

Указатель на функцию (function pointer)

Указатель на функцию - специальный тип данных, позволяющий использовать функции аналогично переменным:

- делать отложенный вызов функций;
- передавать функцию в качестве аргумента в другие функции;
- создавать массивы функций, одинаковых по сигнатуре.

Общий синтаксис:

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

Указатель на функцию

Пример использования

```
1 char up_character(char symbol)
2 {
3     if (symbol < 97 || symbol > 122)
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     std::cout << p_func(sym);
16 }
```

Указатель на функцию

Пример: передача функции сравнения в функцию сортировки

```
1 void buble_sort(int *arr, size_t arr_size,  
2               bool (*cmp)(const int&, const int&))  
3 {  
4     for (size_t i = 0; i < arr_size - 1; ++i) {  
5         for (size_t j = 1; j < arr_size; ++j) {  
6             if ( cmp(arr[j - 1], arr[j]) ) {  
7                 int tmp = arr[j - 1]; arr[j - 1] = arr[j];  
8                 arr[j] = tmp;  
9             }  
10        }  
11    }  
12 }  
13  
14 bool sort_asc(const int& left, const int& right)  
15 { return left > right; }  
16  
17 bool sort_desc(const int& left, const int& right)  
18 { return left < right; }  
19  
20 int arr1[12] = { 3, 1, 5, 4, 3, 2, 1, 8, 4, 76, 4, 67};  
21 buble_sort(arr2, 12, sort_desc);
```

Указатель на функцию

Пример: вычисление одномерного интеграла методом прямоугольников

```
1 double integral(double left, double right, unsigned 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 std::cout << "100 splits: " << integral(0.0, 1.0, 100, fun_x) <<
17     << std::endl;
18 std::cout << "10000 splits: " << integral(0.0, 1.0, 10000, fun_x) <<
19     << std::endl;
20 std::cout << "10000 splits: " << integral(0.0, 1.0, 10000, &fun_x) <<
21     << std::endl;
```


Классы



Задача: написать класс для работы с двумерным целочисленным массивом, который использует непрерывный динамический блок памяти

Придумываем поля: число строк (**cols**), число столбцов (**rows**),
указатель на данные (**data**)

Придумываем поля: число строк (**cols**), число столбцов (**rows**), указатель на данные (**data**)

```
1 class IntArray2D
2 {
3     private:
4         unsigned cols, rows;
5         int *data;
6 };
```

Как инициализировать объекты класса: планируем нужные конструкторы

Как инициализировать объекты класса: планируем нужные конструкторы

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6
7 private:
8     unsigned cols, rows;
9     int *data;
10 };
```

Минимум нужных методов: доступ к размерностям, конкретному элементу(?), скрытый метод для выделения памяти

Минимум нужных методов: доступ к размерностям, конкретному элементу(?), скрытый метод для выделения памяти

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6
7     int      at(unsigned i, unsigned j);
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10
11 private:
12     void allocate();
13
14     unsigned cols, rows;
15     int *data;
16 };
```


Чего достигли: сделали **объявление класса**

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6
7     int      at(unsigned i, unsigned j);
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10
11 private:
12     void allocate();
13
14     unsigned cols, rows;
15     int *data;
16 };
```

Что нового: определение методов класса (в том числе конструкторов) может происходить вне его описания

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6
7     int      at(unsigned i, unsigned j);
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10
11 private:
12     void allocate();
13
14     unsigned cols, rows;
15     int *data;
16 };
17
18 IntArray2D::IntArray2D(unsigned cols_) :
19     cols{cols_}, rows{cols_}
20 {
21     allocate();
22 }
```

Что нового: определение методов класса (в том числе конструкторов) может происходить вне его описания

```
1 class IntArray2D
2 {
3 public:
4     ...
5 private:
6     ...
7 };
8 ...
9
10 IntArray2D::IntArray2D(unsigned cols_, unsigned rows_) :
11     cols{cols_}, rows{rows_}
12 {
13     allocate();
14 }
15
16 void IntArray2D::allocate()
17 {
18     data = new int[cols * rows];
19 }
```

Что хотелось бы: память то под матрицу мы выделяем при использовании конструктора. Кто её высвободить будет?

Что хотелось бы: память то под матрицу мы выделяем при использовании конструктора. Кто её высвободить будет?

Деструктор! (строка номер 6 ниже)

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     int      at(unsigned i, unsigned j);
9     unsigned get_cols_size();
10    unsigned get_rows_size();
11
12 private:
13     void allocate();
14
15     unsigned cols, rows;
16     int *data;
17 };
```

Что нового: определение деструктора

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     int      at(unsigned i, unsigned j);
9     unsigned get_cols_size();
10    unsigned get_rows_size();
11
12 private:
13     void allocate();
14
15     unsigned cols, rows;
16     int *data;
17 };
18 ...
19 IntArray2D::~IntArray2D()
20 {
21     std::cout << "Вызов деструктора: возврат памяти в ОС\n";
22     delete [] data;
23 }
```

Дописываем оставшиеся методы

```
1 class IntArray2D
2 {
3 public:
4     ...
5 private:
6     ...
7 };
8 ...
9
10 int IntArray2D::at(unsigned i, unsigned j)
11 {
12     if ( (i > cols) || (j > rows) ) { return 0; }
13
14     return data[i * rows + j];
15 }
16
17 unsigned IntArray2D::get_cols_size()
18 { return cols; }
19
20 unsigned IntArray2D::get_rows_size()
21 { return rows; }
```

Что было забыто: установка значения конкретного элемента

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     int      at(unsigned i, unsigned j);
9     unsigned get_cols_size();
10    unsigned get_rows_size();
11    void      set(unsigned i, unsigned j, int value);
12 private:
13     void allocate();
14
15     unsigned cols, rows;
16     int *data;
17 };
18
19 void IntArray2D::set(unsigned i, unsigned j, int value)
20 {
21     if ( (i > cols) || (j > rows) ) { return; }
22     data[i * rows + j] = value;
23 }
```


Промежуточный итог

```
1 class IntArray2D
2 {
3 public:
4     ...
5     int      at(unsigned i, unsigned j);
6     unsigned get_cols_size();
7     unsigned get_rows_size();
8     void      set(unsigned i, unsigned j, int value);
9 private:
10    ...
11 };
12
13 IntArray2D sq_arr{6};           // Массив 6x6
14 { IntArray2D rectangle{150, 250}; } // 150x250
15 // Здесь память для rectangle уже возвращена в ОС
16
17 for (unsigned i = 0; i < sq_arr.get_cols_size(); ++i) {
18     for (unsigned j = 0; j < sq_arr.get_rows_size(); ++j) {
19         sq_arr.set(i, j, i * j);
20     }
21 }
22
23 std::cout << sq_arr.at(3, 5) << '\n';
```

Можно ли доступ к элементам сделать элегантнее?

Можно ли доступ к элементам сделать элегантнее?

Пробуем перегрузить оператор!

Можно ли доступ к элементам сделать элегантнее?

Попробуем перегрузить оператор!

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10
11     int      operator()(unsigned i, unsigned j);
12     void     operator()(unsigned i, unsigned j, int value);
13 private:
14     void allocate();
15
16     unsigned cols, rows;
17     int *data;
18 };
```

Перегрузка оператора **operator()**

```
1 class IntArray2D
2 {
3 public:
4     ...
5
6     int  operator()(unsigned i, unsigned j);
7     void operator()(unsigned i, unsigned j, int value);
8 private:
9     ...
10 };
11
12 int IntArray2D::operator()(unsigned i, unsigned j)
13 {
14     if ( i > cols || j > rows) { return 0; }
15     return data[i * rows + j];
16 }
17
18 void IntArray2D::operator()(unsigned i, unsigned j, int value)
19 {
20     if ( i > cols || j > rows) { return; }
21     data[i * rows + j] = value;
22 }
```

Теперь использование объектов упростилось

```
1 class IntArray2D
2 {
3 public:
4     ...
5 private:
6     ...
7 };
8
9 IntArray2D sq_arr{6};
10
11 for (unsigned i = 0; i < sq_arr.get_cols_size(); ++i) {
12     for (unsigned j = 0; j < sq_arr.get_rows_size(); ++j) {
13         sq_arr(i, j, i * j);
14     }
15 }
16
17 std::cout << sq_arr(3, 5) << '\n';
```

Пробуем оставить только одну версию для **operator()**

Попробуем оставить только одну версию для **operator()**

Попытка не пытка

```
1 class IntArray2D
2 {
3 public:
4     ...
5     int& operator()(unsigned i, unsigned j)
6     {
7         if ( i > cols || j > rows) { std::exit(1); }
8         return data[i * rows + j];
9     }
10 private:
11     ...
12 };
13
14 IntArray2D sq_arr{6};
15 for (unsigned i = 0; i < sq_arr.get_cols_size(); ++i) {
16     for (unsigned j = 0; j < sq_arr.get_rows_size(); ++j) {
17         sq_arr(i, j) = i * j;
18     }
19 }
20 std::cout << sq_arr(3, 5) << '\n';
```


Второй промежуточный итог

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10
11     int&      operator()(unsigned i, unsigned j);
12
13 private:
14     void allocate();
15
16     unsigned cols, rows;
17     int *data;
18 };
```

А что происходит при присвоении разных объектов класса?

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     ~IntArray2D();
7
8     unsigned get_cols_size();
9     unsigned get_rows_size();
10    int&      operator()(unsigned i, unsigned j);
11 private:
12     void allocate();
13
14     unsigned cols, rows;
15     int *data;
16 };
17
18 IntArray2D sq_arr{6};
19 ...
20 IntArray2D sq_arr2 = sq_arr; // <=== Очень плохо!
21 // поля data у обоих объектов
22 // указывают на один блок памяти
```

Решение: определение специального конструктора для копирования

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     IntArray2D(const IntArray2D& other); // <=== Вот он
7     ...
8 private:
9     ...
10 };
11
12 IntArray2D::IntArray2D(const IntArray2D& other) :
13     cols{other.cols}, rows{other.rows}
14 {
15     data = new int[cols * rows];
16     for (unsigned i = 0; i < cols; ++i) {
17         for (unsigned j = 0; j < rows; ++j) {
18             data[i * rows + j] = other.data[i * rows + j];
19         }
20     }
21 }
```

Теперь присваивание разных переменных не будет проблемой

```
1 class IntArray2D
2 {
3 public:
4     IntArray2D(unsigned cols_);
5     IntArray2D(unsigned cols_, unsigned rows_);
6     IntArray2D(const IntArray2D& other);
7     ...
8 private:
9     ...
10 };
11
12 IntArray2D sq_arr{6};
13 ...
14 IntArray2D sq_arr2 = sq_arr; // <=== Всё хорошо
15 // поля data у объектов указывают
16 // на разные блоки памяти
```

Препроцессор и сборка программ

Получение исполняемого или библиотечного файла состоит из трёх фаз, выполняемых компилятором:

- 1 Фаза **препроцессинга**: Обработка **препроцессорных** директив
- 2 Фаза **компиляции**: преобразование исходного файла(-ов) в объектный(-ые), содержащий представление на языке *ассемблера*
- 3 Фаза **связывания** (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 каких-либо упоминаний об типах
- ❹ Отменяет любой ранее определённый **идентификатор**

Директивы препроцессора

Примеры макросов

```
1 #include <iostream>
2 #include <locale>
3
4 #define FUNCTION(name, a) int fun_##name() { return a;}
5
6 FUNCTION(first, 12)
7 FUNCTION(second, 2)
8 FUNCTION(third, 23)
9
10 #undef FUNCTION
11 #define FUNCTION 34
12 #define OUTPUT(a) std::cout << #a << '\n'
13
14 int main()
15 {
16     std::setlocale(LC_ALL, "RUS");
17     std::cout << "first: " << fun_first() << '\n';
18     std::cout << "second: " << fun_second() << '\n';
19     std::cout << "third: " << fun_third() << '\n';
20
21     std::cout << FUNCTION << '\n';
22     OUTPUT(Русский текст без кавычек и переносов!);
23 }
```


Условные директивы, используемые для задания 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 #define MACROS1 2
2 #include <iostream>
3 #include <locale>
4
5 int main()
6 {
7     std::setlocale(LC_ALL, "RUS");
8     #ifdef MACROS1
9         std::cout << "1: определён\n";
10    #else
11        std::cout << "1: не определён\n";
12    #endif
13
14    #ifndef MACROS1
15        std::cout << "2: не определён\n";
16    #elif MACROS1 == 2
17        std::cout << "2: определён\n";
18    #else
19        std::cout << "2: не определён\n";
20    #endif
21
22    #if !defined(DCBA) && (MACROS1 < 2*4-3)
23        std::cout << "3: выражение истинно\n";
```

Директивы, используемые для включения других исходных файлов

(1) `#include <filename>`

(2) `#include "filename"`

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 C++ зависят от того, куда компилятор языка был установлен в ОС, а также могут быть добавлены с помощью дополнительных опций компилятора.