

Структурирование программы и данных

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осенний семестр 2015 г.

Определение функции

Тип возвращаемого значения.

Имя функции.

double area (
 double width,
 double height)

Параметры и их типы.

- Тип указывается каждому!

Возврат значения
и выход из функции.

тело
функции

{
}
}


return width * height;

double S = area (4, 5); // S == 20

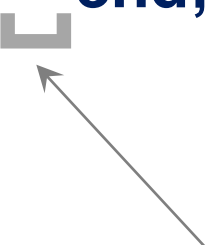
area (3, 2); // 6 (игнорируется)

Пример функции на Pascal

```
function Find(  
    Where: array of String; What: String): Integer;  
begin  
    Result := Length(Where) – 1;  
    while (Result >= 0) and (Where[Result] <> What) do  
        Dec(Result);  
end;
```

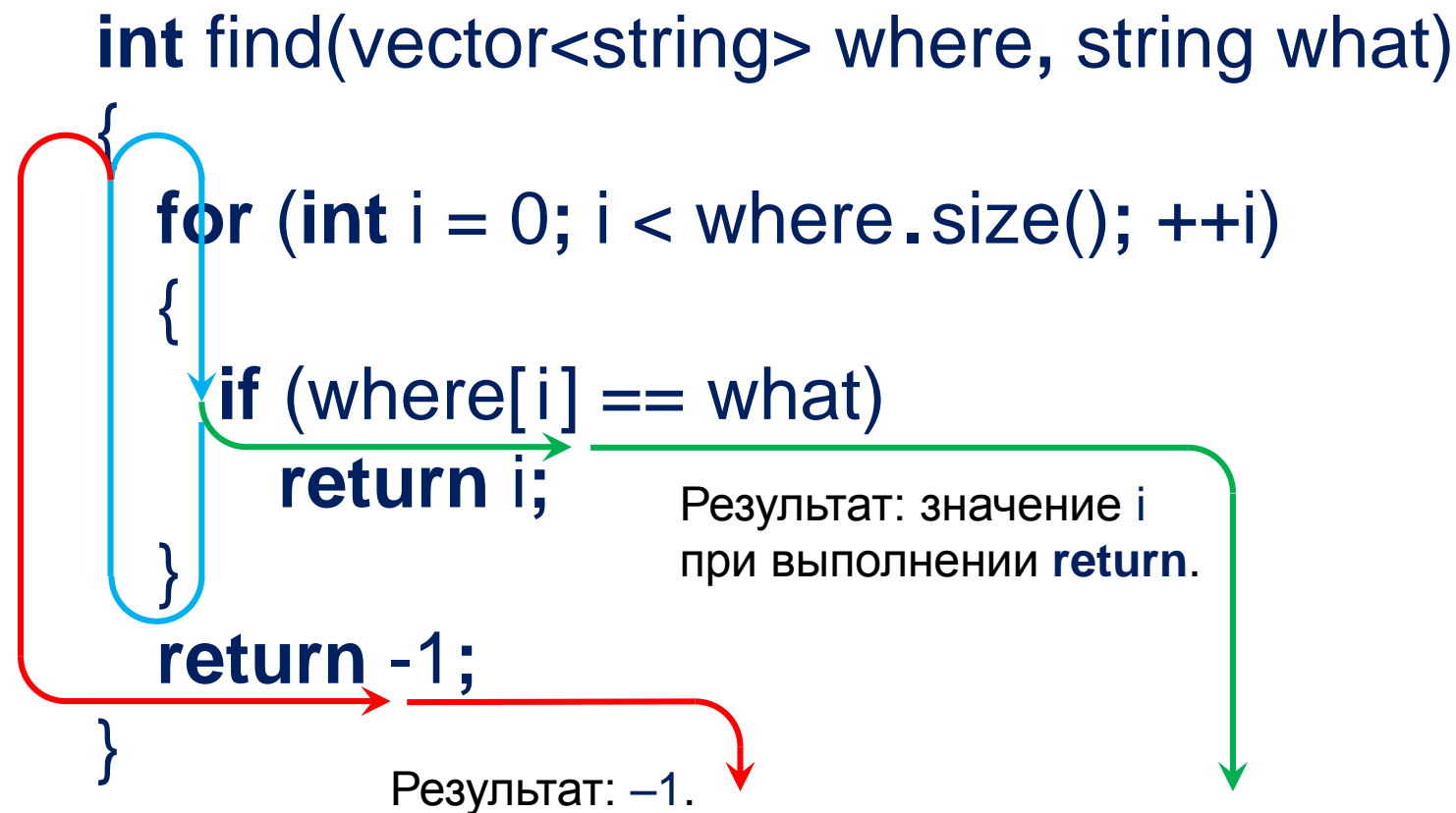


Если это условие стало ложным,
прекратить можно не только ~~лишь~~ цикл,
но и работу функции.



Возвращаемое значение — то, чему в этой точке
равна встроенная переменная **Result**.

Пример функции на C++



Оператор **return**

- Оператор **return** *X*:
 - указывает, что возвращаемое значение — *X*;
 - производит выход из функции.
- Аналог в Pascal:
`Result := X;`
`Exit;`
- Процедура в Pascal = **void**-функция в C++
 - «возвращает» специальный тип **void**;
 - возвращаемого значения нет:
`return; // выход из void-функции`
- Не-**void** функции обязаны вернуть значение.
 - Иначе — не ошибка, но *опасное* предупреждение!

Выходные переменные

Функция ничего
не возвращает
(как процедура).



```
void solve_quadric_equation(  
    double a, double b, double c,  
    double& x1, double& x2)  
{  
    double const D = b*b - 4*a*c;  
    x1 = (-b + sqrt(D)) / (2*a);  
    x2 = (-b - sqrt(D)) / (2*a);  
}
```

& — амперсанд

```
double x1, x2;  
solve_quadric_equation(1, 3, 2, x1, x2);  
// x1 == -1, x2 == -2
```

Передача по ссылке

- Удобна для возврата нескольких значений.
- Аналог **var** в Pascal.
- Проблема — читаемость:

```
double a = 1, b = 3, c = 2, x1 = 0, x2 = 0;  
solve_quadric_equation(a, b, c, x1, x2);  
// Какие переменные изменились?
```

Параметр-ссылка

- Проблема — обязательность всех аргументов:

```
void get_statistics(  
    vector<double> samples,  
    double & mean, double & variance)  
{  
    // Расчет мат. ожидания и дисперсии.  
}
```

```
vector<double> data { 1, 2, 3, 4, 5 };  
double mean;  
double variance;  
get_statistics(data, mean, variance);
```

Не нужна!



Параметр-указатель

```
void get_statistics(  
    vector<double> samples,  
    double* mean, double* variance)  
{  
    double mx = ...;  
    if (mean)  
        *mean = mx;  
    if (variance) {  
        // Расчет дисперсии.  
    }  
}
```

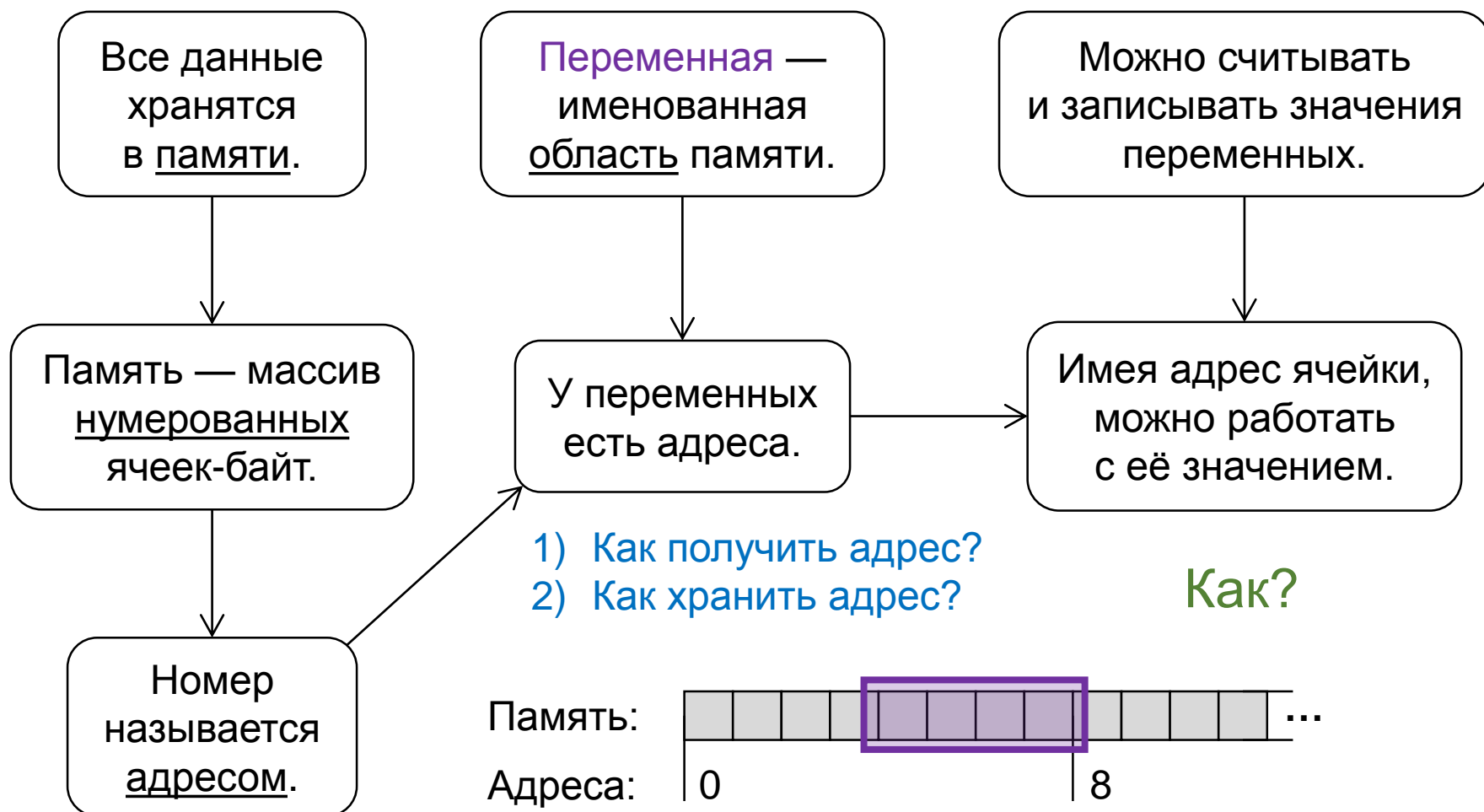
Проверка, передана ли переменная или **nullptr**.

Вместо ненужной выходной переменной.

```
vector<double> data { ... };  
double mean;  
get_statistics(data, &mean, nullptr);
```

Видно, что **mean** может измениться.

Что такое указатель?



Указатели

- Это переменные, содержащие адрес памяти.
- Указатель **pointer**:
 - **pointer** — адрес;
 - ***pointer** — значение по адресу (разыменование).
- Переменная **mean** (не указатель):
 - **mean** — значение;
 - **&mean** — адрес (взятие адреса).
- **nullptr == NULL == 0**:
 - нулевой указатель;
 - значение по нему получить нельзя.

Тип данных «указатель»

Звездочка перед переменной.

double x = 42;

double * unknown;

double* pointer = &x;

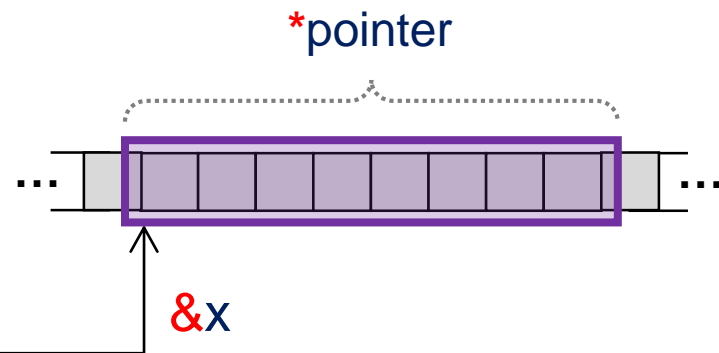


До присваивания значения указывает неизвестно, куда.

Указатель на **double**:

- может хранить адрес **double**
 - и только **double**;
- разыменование дает **double**.

x = *pointer;



Неизменяемость и указатели

Что неизменяемо?

- всё изменяемо:..... **int * p;**
 ***pointer = 0;**
 pointer = nullptr;
- адрес:..... **int * const cp;**
 ***const_pointer = 0;**
 ~~**const_pointer = nullptr;**~~
- значение:..... **const int * pc;**
 ~~***pointer_to_const = 0;**~~
 pointer_to_const = nullptr;
- и то, и другое:..... **const int * const cpc;**
 ~~***const_pointer_to_const = 0;**~~
 ~~**const_pointer_to_const = nullptr;**~~

Тип данных «ссылка»

- Ссылка — новое имя ячейки памяти.

```
double x = 1;
```

```
double y = 3;
```

```
double& z = x;
```

```
z = 2;
```

```
// x == 2, y == 3, z == 2
```

```
z = y;
```

```
// x == 3, y == 3, z == 3
```

```
y = 4;
```

```
// x == 3, y == 4, z == 3
```

Амперсанд перед именем переменной.

Инициализация:

- «привязка» к значению (переменной);
- обязательна
 - иначе — «новое имя» для чего?

Действия над ссылкой
равнозначны действиям
над привязанной переменной.

Привязку изменить нельзя.

Применение ссылок

- Сокращение кода:
 - **double&** middle = data[data.size() / 2];
middle = 42;
// data[data.size() / 2] == 42
 - **double&** x = change_a_or_b ? a : b;
x += 2;
- Неизменяемые ссылки:
 - **const double&** middle = data[data.size() / 2];
~~middle = 42;~~
 - Неизменяемость всегда относится к значению.

Неизменяемые параметры

Будет создана копия значения **a** и помещена в **x**.

↓

```
void f(int x) {  
    x = 42;  
    // x == 42  
}
```

← Копия разрушается.

```
void f(const int x) {  
    // ...  
}
```

↑ Копию нельзя изменить
• и обычно не нужно.

```
int a = 0;  
f(a);  
// a == 0
```

Действия
над копией
не влияют
на аргумент.

- А если **x** — вектор или строка?
 - Большого размера?
- Зачем вообще копия?
 - Нужна независимость **x** и **a**.
 - Обычно нужна неизменяемость.

Передача без копирования

- Передача по ссылке:

```
void function ( vector<int>& data ) { ... }
```

- Нет копирования.
- Аргумент и data связаны.

- Передача по неизменяемой ссылке:

```
void function ( const vector<int>& data ) { ... }
```

- Копирования нет.
- Случайно изменить data нельзя.
 - Изменять параметры — плохая практика!
- Имеет смысл использовать по умолчанию.
 - Кроме **int**, **double**, ... (пользы нет, вреда — тоже).

Рекурсия

- Вызов функцией самой себя.
- Для случаев, когда



путь(от Новокосино до Авиамоторной) =

«Новокосино — Новогиреево» + путь(от Новогиреево до Авиамоторной)

Рекурсивный вызов

`power(2, 3); // 23 = 8`

$$f(a, n) = a^n = \begin{cases} a \cdot a^{n-1}, & n > 0 \\ 1, & n = 0 \end{cases} =$$

$$= \begin{cases} a \cdot f(a, n-1), & n > 0 \\ 1, & n = 0 \end{cases}$$

условие окончания

```
double power(2, 3) {  
    if (3 == 0)  
        return 1;  
    return 2 * power(2, 3 - 1);  
}
```

```
double power(2, 2) {  
    if (2 == 0)  
        return 1;  
    return 2 * power(2, 2 - 1);  
}
```

```
double power(2, 1) {  
    if (1 == 0)  
        return 1;  
    return 2 * power(2, 1 - 1);  
}
```

```
double power(double a, int n) {  
    if (n == 0)  
        return 1;  
    return a * power(a, n - 1);  
}
```

```
double power(2, 0) {  
    if (0 == 0)  
        return 1;  
    return 2 * power(2, 0 - 1);  
}
```

Рекурсия (продолжение)

- ❑ Вызов функции расходует часть ограниченной области памяти — стека.
 - Этот расход возвращается по выходе из функции.
 - Глубокая рекурсия сильно расходует стек.
 - Бесконечная рекурсия невозможна.
 - Ошибка: «Stack overflow» («переполнение стека»).



❑ Прямая рекурсия



В каком порядке
описывать функции?

Косвенная рекурсия

```
bool is_even ( unsigned int n ) {  
    return n == 0 || is_odd ( n - 1 );  
}  
bool is_odd ( unsigned int n ) {  
    return n != 0 && is_even ( n - 1 );  
}
```

Объявление и определение

double get_mean (**const** vector<**double**>& xs); ← **Объявление** функции (прототип).

```
int main() {  
    vector<double> data { 1, 2, 3, 4, 5 };  
    cout << "Mean is " << get_mean(data);  
}
```

← Благодаря объявлению, компилятор уже «знает», что такая функция есть.

```
double get_mean ( const vector<double>& xs ) {  
    double mean = 0;  
    for ( const double& x : xs ) {  
        mean += x;  
    }  
    return mean / xs.size();  
}
```

Определение функции.

← Копия значения в векторе не нужна, менять его не нужно.

Какими должны быть функции?

```
int square(int x)
{
    return x * x;
}
```

- ✓ Одна задача;
- ✓ ничего лишнего;
- ✓ полезна широко.

```
int square(int& x, int& count)
{
    cout << "Enter element #" << count << ": ";
    cin >> x;
    count++;
    return x * x;
}
```

Задачи:

- 1) Ввод и вывод,
 - а если не нужны?
- 2) подсчет;
 - зачем?
- 3) возведение в квадрат.

1) Повторно используемыми (reusable).

- Решать одну задачу.
- Не иметь побочных эффектов:
 - зависеть только от входных данных (не от ввода, времени и т. п.);
 - выдавать результат только возвращаемым и выходными значениями.

Какими должны быть функции?

2) Могут обозначать логику работы программы.

«Как съесть слона? — По кусочкам!»

Расчет корреляции \vec{x} и \vec{y} :

1) ввести N , \vec{x} , \vec{y} ;

```
vector<double> input(  
    unsigned int how_many)  
{ return { }; }
```

2) вычислить m_x и m_y ;

```
double get_mean(  
    const vector<double> & data)
```

3) вычислить s_x и s_y ;

```
{ return 0; }
```

4) $S = \sum_{i=0}^{N-1} (x_i - m_x)(y_i - m_y)$;

```
double get_stdev(  
    const vector<double> & data,  
    double mean)
```

5) $cov(x, y) = S / (N-1)$;

Цикл?

6) $r_{xy} = \frac{cov(x,y)}{s_x s_y}$.

```
{ return 0; }
```

Декомпозиция

unsigned int N;

- 1) **cin** >> N;
vector < **double** > x = input (N);
vector < **double** > y = input (N);
- 2) **double** m_x = get_mean (x);
double m_y = get_mean (y);
- 3) **double** s_x = get_stdev (x, m_x);
double s_y = get_stdev (y, m_y);
- 4) **double** sum = 0;
for (**unsigned int** i = 0; i < N; ++i) {
 sum += (x [i] - m_x) * (y [i] - m_y);
}
- 5) **double** covariance = sum / (N - 1);
- 6) **double** correlation = covariance / (s_x * s_y);

Расчет корреляции \vec{x} и \vec{y} :

- 1) ввести N , \vec{x} , \vec{y} ;
- 2) вычислить m_x и m_y ;
- 3) вычислить s_x и s_y ;
- 4) $S = \sum_{i=0}^{N-1} (x_i - m_x)(y_i - m_y)$;
- 5) $cov(x, y) = S / (N - 1)$;
- 6) $r_{xy} = \frac{cov(x, y)}{s_x s_y}$.

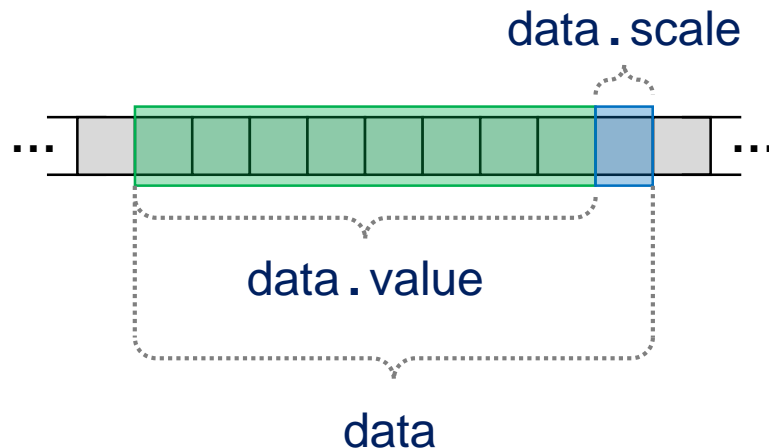
Структуры

- В первом приближении — записи Pascal (**record**).
- Хранят вместе несколько именованных значений разных типов.

- **struct** Temperature

```
{  
    double value;  
    char scale;  
};
```

- Temperature data;
data.value = 273.15;
data.scale = 'K';
cin >> data.value >> data.scale;
cout << data.value - 273.15 << 'C';



Перечисления

```
enum Scale
{
    Kelvin,
    Celsius,
    Fahrenheit
};
```

```
Temperature data;
data.value = 100;
data.scale = Kelvin;
```

~~data.scale = 'Q';~~

✓ Так и нужно.

```
enum class Scale
{
    Kelvin,
    Celsius,
    Fahrenheit
};
```

```
Temperature data;
data.value = 100;
data.scale = Scale :: Kelvin;
```

```
char symbol = data.scale;
cin >> data.scale;
```

× Возможность утрачена.

```
struct Temperature
{
    double value;
    Scale scale;
};
```

```
cout << data.scale; // 0
```

× Не этого мы хотели!

Преобразование перечислений

Из символа в шкалу

```
Scale to_scale (char symbol)
{
    switch (symbol) {
        case 'C': return Celsius;
        // ...
    }
}
```

```
char input;
cin >> input;
Scale value = to_scale(input);
```

Из шкалы в символ

```
char from_scale (Scale value)
{
    switch (value) {
        case Celsius: return 'C';
        // ...
    }
}
```

```
Scale value = Fahrenheit;
cout << from_scale (value);
```

Приведение типов (type cast)

- ✓ Значения для Kelvin, Celsius и т. д. разные,
- ✗ но неизвестные.

Сделаем их известными и удобными.

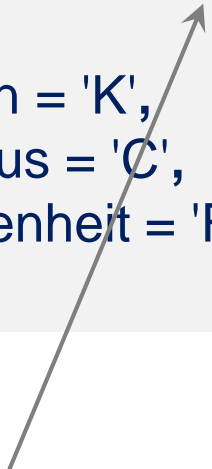
- Любые значения хранятся в памяти;
 - для `Scale` там коды символов `char`.
- На время вывода значения типа `Scale` можно и нужно рассмотреть как `char`.

```
Scale value = Celsius;  
cout << (char) value;
```

```
cin >> (char &) value;
```

- При вводе рассмотреть переменную типа `Scale` как переменную-`char`.

```
enum Scale : char  
{  
    Kelvin = 'K',  
    Celsius = 'C',  
    Fahrenheit = 'F'  
};
```



Нижележащий тип
(underlying type):
элементы `Scale` будут
константами типа `char`.

Литература к лекции

- *Programming Principles and Practices Using C++:*
 - глава 4, раздел 4.5 — функции;
 - глава 6, раздел 6.5 — декомпозиция;
 - глава 8 (пункт 8.5.8 — опционально);
 - пункт 9.4.1 — структуры, раздел 9.5 — перечисления;
 - упражнения к главам 4 и 8.
- *C++ Primer:*
 - глава 2, раздел 2.3 — указатели и ссылки;
 - глава 6 — функции;
 - раздел 19.3 — перечисления;
 - упражнения.