Лекция VI

3 декабря 2016

Пользовательские типы данных

Производные типы данных

↙

Псевдонимы (aliases)

typedef

using (C++)

↘

Составные типы данных

struct

class

enum, **enum class** (C++)

Пользовательские типы данных. Псевдонимы

C/C++: оператор **typedef** - добавление псевдонимов для любого существующего типа данных

```
typedef <тип_данных> <псевдоним1>
                [, <пс2>, <пс3>, ...];
```

Пользовательские типы данных. Псевдонимы

C/C++: оператор **typedef**

```
1  typedef unsigned int uint_t;
2
3  uint_t val1 = 444;
4  unsigned int val2 = val1;
5
6  // Объявляем псевдонимы для типа double
7  // и указателя на double
8  typedef double velocity, *p_velocity;
9
10 velocity v1 = 45.5;
11 // Два варианта определения переменных-↩
       указателей
12 p_velocity pv1 = &v1;
13 velocity *pv2 = pv1;
```

Пользовательские типы данных. Псевдонимы

C/C++: оператор **typedef**: есть польза при использовании статических массивов - забыть о квадратных скобках при объявлении или передачи в функцию.

```
1 const size_t TEN = 10;
2
3 typedef unsigned int uint, uint_tenth[TEN];
4
5 void print_my_arra(uint_array uarr);
6 uint sum_of_elements(uint_array uarr);
7 /* Однако внутри функций нужно или ↩
     использовать
8    константу TEN, или явно передавать размер
9 */
```

Пользовательские типы данных. Псевдонимы

Современный C++: оператор **typedef**

Полезный совет

При написании программ на C++ не используйте **typedef**, если компилятор поддерживает стандарт C++11

Решение

Используйте **using**!

Пользовательские типы данных. Псевдонимы

C++11: оператор **using** - добавление псевдонимов, аналогичное **typedef**, возможно с более понятным синтаксисом

```
using <псевдоним> = <тип_данных>;
```

Пользовательские типы данных. Псевдонимы

C++11: оператор **using**

```
1  const size_t TEN = 10;
2
3  /*
4  typedef unsigned int uint,
5                       *p_uint,
6                       uint_tenth[TEN];
7  */
8
9  using uint       = unsigned int;
10 using p_uint     = unsigned int*;
11 using uint_tenth = unsigned int[TEN]
12
13 uint val1 = 555;
14 p_uint = &val1;
15 std::cout << *p_uint;
```

Составные типы данных. Структуры

Структура (в смысле языка C) - это составной тип данных, объединяющий множество *проименнованных* элементов. **Элемент структуры** называют **её полями** и под ним понимается любой, известные к моменту объявления структуры, тип данных. Общий синтаксис:

```
struct <название_структуры>
{
  <тип_1> <поле_1_1> [, <поле_1_2>, ...]
  <тип_2> <поле_2_1> [, <поле_2_2>, ...]
  ...
  <тип_n> <поле_n_1> [, <поле_n_2>, ...]
};
```

Использование структур: определение и объявление переменных

```
1  struct MaterialPoint
2  {
3     int x, y;
4     int z;
5     double weight;
6  };
7
8  MaterialPoint mp1;
9
10 // В языке C объявление:
11 // struct MaterialPoint mp1;
```

Использование структур: обращение к полям переменных через оператор «.»

```
1  struct MaterialPoint
2  {
3    int x, y;
4    int z;
5    double weight;
6  };
7
8  // начальные значения полей не ↩
     устанавливаются
9  MaterialPoint mp1;
10 mp1.x = mp1.y = 4;
11 mp.weight = 45.5;
```

Использование структур: определение и объявление переменных

```
1  struct MaterialPoint
2  {
3     int x, y;
4     int z;
5     double weight;
6  } g_mp1, g_mp2;
7
8  //  ...
9  g_mp1.weight = 45.3;
10 g_mp2.x = g_mp2.y = g.mp2.z = 1;
```

Использование структур: анонимные структуры

```
1  struct
2  {
3     int x, y;
4     int z;
5     double weight;
6  } g_mp1, g_mp2;
7
8  //  ...
9  g_mp1.weight = 45.3;
10 g_mp2.x = g_mp2.y = g.mp2.z = 1;
```

Составные типы данных. Структуры

Использование структур: определение переменных (задание начальных значений полей при объявлении)

```
1  struct MaterialPoint
2  {
3    int x, y;
4    int z;
5    double weight;
6  };
7
8  // Используется список инициализаторов
9  MaterialPoint mp1 = { 3, 4, 5, 8.8 };
10 std::cout << mp1.weight << '\n';
11
12 MaterialPoint mp2 = { 5, 8 };
13 // mp2.z == 0, mp2.weigth == 0.0
14 std::cout << (mp2.z == 0);
```

Использование структур: указатели на переменную структуры и оператор «->»

```
1  struct MaterialPoint
2  {
3    int x, y;
4    int z;
5    double weight;
6  };
7
8  MaterialPoint mp1 = { 3, 4, 5, 8.8 };
9  MaterialPoint *p_mp = &mp1;
10
11 std::cout << (*p_mp).y << '\n';
12
13 // Получение значения поля по указателю
14 std::cout << p_mp->weight << '\n';
```

Использование структур: параметры функции

```
1  struct MaterialPoint
2  {
3    int x, y;
4    int z;
5    double weight;
6  };
7
8  double get_distance(MaterialPoint mp1,
9                      MaterialPoint mp2)
10 {
11   double dx = mp2.x - mp1.x;
12   ...
13   return std::sqrt( dx * dx + ... );
14 }
```

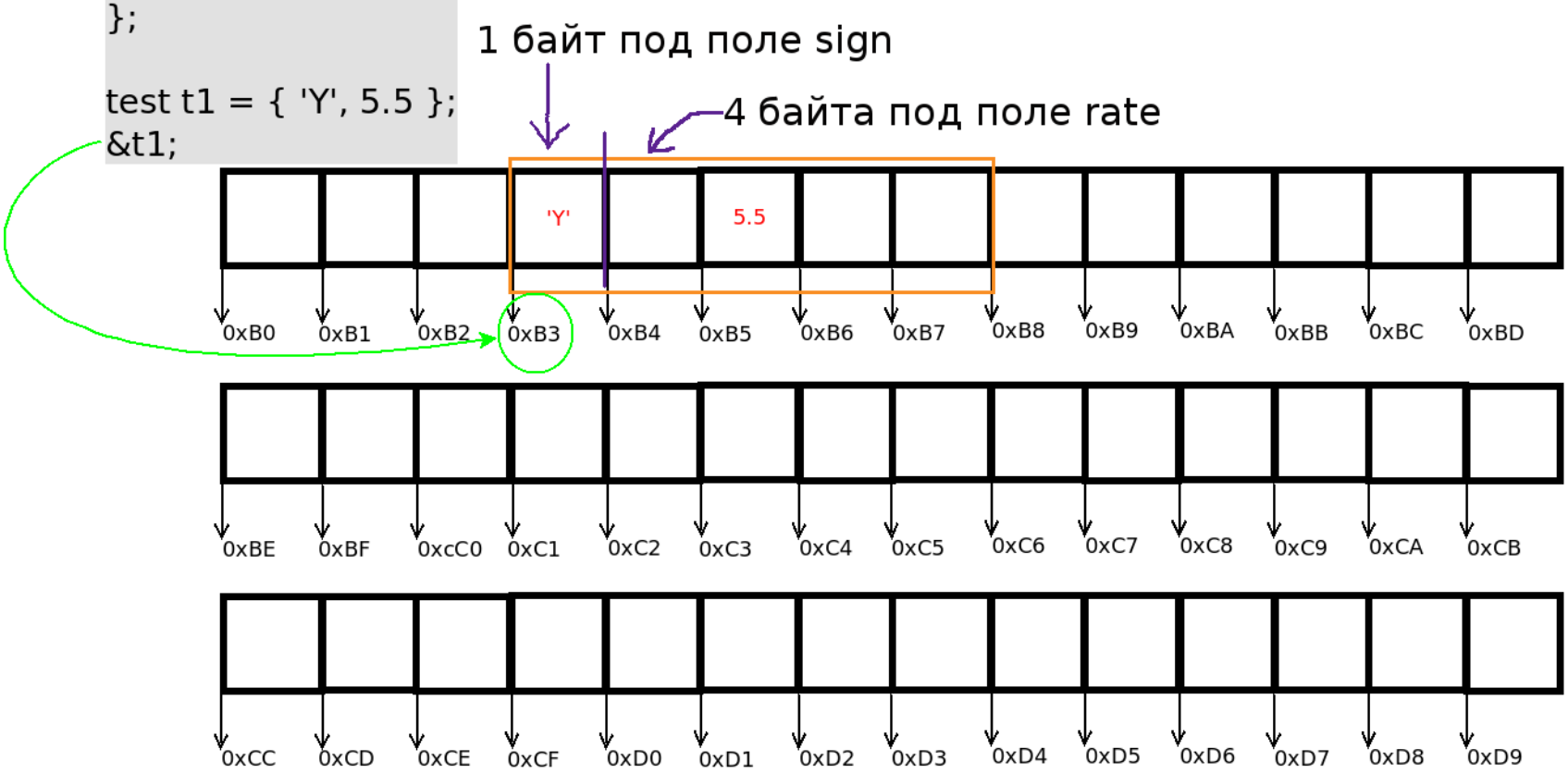
Использование структур: поле-указатель на саму себя

```
1 struct MaterialPoint
2 {
3   int x, y;
4   int z;
5   double weight;
6
7   // Указателей можно добавлять сколько ↩
        угодно
8   MaterialPoint *p_mp;
9 };
```

Использование структур: что происходит в памяти

```
struct test
{
   char sign;
   int rate;
};

test t1 = { 'Y', 5.5 };
&t1;
```



Включаем воображение: есть сосуд с неким газом, есть образец, который вам захотелось исследовать. Помещаем образец в сосуд и включаем программу, которая контролирует давление газа, стравливая лишнее по необходимости, и температуру, охлаждая/подогревая сосуд.
Внутри идёт какая-то реакция, вы наблюдаете за этим. Для написания управляющей программы, кроме всего прочего, нужно помнить об уравнении состоянии:

$$PV \sim T$$

которое указывает на связь основных параметров.
В данном случае, объём - величина постоянная (сосуд не расширяется), а давление и температура зависят от реакции образца внутри и от окружающей среды (подогреваем или охлаждаем).

Как можно описать программу с использованием структур?

```
1  struct GasContainer
2  { // полей может быть гораздо больше
3    double pressure, temperature;
4    unsigned volume;
5  };
6
7  GasConteiner vol1 = { 25, 0, 20.5, 8 };
8
9  // Заполняем газом
10 vol1.pressure = 15;
11 vol1.temperature = 22.8;
12
13 // Следим за давлением
14 if ( vol1.pressure > CRIT_PRESSURE ) {
15   release_gas_from( vol1 );
16 }
```

Как можно создать проблему кодом?

```
1  struct GasContainer
2  { // полей может быть гораздо больше
3    double pressure, temperature;
4    unsigned volume;
5  };
6
7  GasConteiner vol1 = { 25, 0, 20.5, 8 };
8
9  // Заполняем газом
10 vol1.pressure = 15;
11 vol1.temperature = 22.8;
12
13 vol1.preasure = CRIT_PRESSURE - 10; // тут!
14 if ( vol1.pressure > CRIT_PRESSURE ) {
15   release_gas_from( vol1 );
16 }
```

Последствия

Локальный взрыв в отдельно взятой лабаратории

Выводы

А ведь связанные величины не должны меняться произвольно

Пример из другой области - операции с банковскими счетами

```
1  struct BankAccount
2  {
3    unsigned client_number;
4    unsigned number;
5    int balance;
6  };
7
8  void get_accounts_report(BankAccount arr[],
       size_t count)
9  {
10   for (size_t i = 0; i < count; ++i) {
11     print_account(arr[i]);
12   }
13 }
```

Но человеческий фактор не дремлет!

```
1  struct BankAccount
2  {
3    unsigned client_number;   ...
4  };
5
6  void get_accounts_report(BankAccount arr[],
       size_t count)
7  {
8    for (size_t i = 0; i < count; ++i) {
9      if (i == count / 2) {
10       arr[i].clinet_number = arr[0].
             client_number;
11     }
12     print_account(arr[i]);
13   }
14 }
```

Последствия

Сами додумайте

Выводы

- Иногда данные надо защищать от произвольного доступа
- Изменение значения поля может требовать гораздо более сложной логики, чем операция присвоения

Так, основоположники современных компьютерных наук пришли к тому, что неплохо бы в программах со сложной логикой иметь объекты - более продвинутые переменные, чем мы знали прежде.
Во многих языках тип данных для подобных объектов назвали **классом**, в том числе и в C++.

Неформальное определение класса

Составной тип данных, аналогичный структурам (**в смысле языка C**), в котором:
а) полями могут быть как другие типы данных, так и функции;
б) доступ к полям может быть ограничен для кода, использующего объекты класса.

Общий вид объявления класса

```
class <название_класса>
{
private:
   <тип_n> <закрытое_поле_n_1>
        [, <закрытое_поле_n_2>, ...];
   <тип> <закрытая_функция>(<аргументы>)
   { <код_функции> };

public:
   <тип_m> <открытое_поле_m_1>
        [, <открытое_поле_m_2>, ...];
   <тип> <открытая_функция>(<аргументы>)
   { <код_функции> };
};
```

Использование классов: объявление - по умолчанию все поля закрыты

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure;
5    int volume;
6  };
7
8  // Так ок, но бесполезно
9  GasContainer vol1;
10
11 //Две строки ниже запрещены
12 //v1.pressure = 10.5;
13 //v1.volume = 12;
```

Использование классов: объявление - полный аналог предыдущего слайда

```
1  class GasContainer
2  {
3  private: // необязательная метка
4    double pressure, temperature;
5    double crit_pressure;
6    int volume;
7  };
8
9  // Так ок, но бесполезно
10 GasContainer vol1;
11
12 //Две строки ниже запрещены
13 //v1.pressure = 10.5;
14 //v1.volume = 12;
```

Использование классов: объявление - добавляем функцию в качестве поля

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure;
5    int volume;
6
7  public:
8    void init(int vol, double temp)
9    { temperature = temp; volume = vol;
10     pressure = temperature / volume; }
11 };
12
13 GasContainer vol1;
14 vol1.init(7, 25.5);
```

Использование классов: терминология

- переменную класса - называем **объект**
- функцию в качестве поля - называем **методом класса**
- поля других типов - называем **полями-данными**
- совокупность полей-данных конкретного объекта - образует его **внутреннее состояние**

Использование классов: объявление -даём возможность посмотреть значения скрытых полей

```
1  class GasContainer
2  { // описание полей – слайды выше
3
4  public:
5    void init(int vol, double temp)
6    { ... } // тело – предыдущие слайды
7
8    double get_pressure()
9    { return pressure; }
10 };
11
12 GasContainer vol1;
13 vol1.init(7, 25.5);
14
15 std::cout << vol1.get_pressure();
```

Использование классов: объект - пост-установка значений

```
1  class GasContainer
2  { // описание полей – слайды выше
3  public:
4    // определение init и get_pressure
5    void set_temp(double new_temp)
6    {
7      if (new_temp < 0) { return; }
8
9      double new_press = new_temp / volume;
10     if (new_press > crit_pressure) {
11       send_alert_msg(new_temp);
12     } else {
13       // установить новую температуру
14     }
15   }
16 };
```

Использование классов: объект - пост-установка значений

```
1  class GasContainer
2  { // описание полей — слайды выше
3  public:
4    void init(int vol, double temp) { ... }
5
6    double get_pressure() { ... }
7
8    void set_temp(double new_temp) { ... }
9  };
10
11 GasContainer vol1;
12 vol1.init(7, 25.5);
13
14 vol1.set_temp(32.8);
```

Использование классов: сравнение со структурой

```
1  struct GasContainerOld
2  { ... };
3
4  class GasContainer
5  { // описание полей – слайды выше
6  public:
7    void init(int vol, double temp) { ... }
8
9    // ...
10 };
11
12 GasContainerOld str_vol = { 1, 2.3 };
13
14 GasContainer vol1;
15 vol1.init(7, 235.5);
16 vol1.init(8, 345.5);
```

Концепция **конструктора для класса**: специальная функция, которая позволяет устанавливать состояние объекта в момент его объявления (то есть, при добавлении переменной появляется возможность передать значения закрытым полям).

В C++ **конструктором** являются специальные методы класса, имя которых **совпадает с названием класса** и которые **не возвращают никакого значения**.

Для любого объекта конструктор может быть вызван только **единожды**.

Использование классов: добавление конструктора

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure; int volume;
5  public:
6    GasContainer(int vol, double temp) : ↩
       temperature{temp}, volume{vol}
7    { pressure = temperature / volume; }
8
9    // ...
10 };
11
12 GasContainer vol1{7, 235.7};
13 // GasContainer vol2; -> не компилируется
```

Использование классов: почему **vol2** не компилируется?

- В каждый класс, в котором не объявлено ни одного конструктора неявно добавляется **конструктор по умолчанию**
- **Конструктор по умолчанию** позволяет объявлять переменные класса, не передавая им никаких начальных значений
- **Конструктор по умолчанию** - не принимает никаких аргументов и единственное, что он делает - выделяет нужную память под все поля класса
- Если определён вручную хотя бы один конструктор, неявный конструктор по умолчанию не добавляется в класс
- Для его возращения нужно определить перегруженную функцию конструктора без аргументов

Использование классов: добавление конструктора

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure; int volume;
5  public:
6    GasContainer(int vol, double temp) : ↩
       temperature{temp}, volume{vol}
7    { pressure = temperature / volume; }
8
9    GasContainer() {}
10   // ...
11 };
12
13 GasContainer vol1{7, 235.7};
14 GasContainer vol2; // Всё ок
```

Использование классов: перегрузка конструкторов

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure; int volume;
5  public:
6    GasContainer(int vol, double temp) : ↩
       temperature{temp}, volume{vol}
7    { pressure = temperature / volume; }
8
9    GasContainer(int vol) : volume{vol}
10   { temperature = pressure = 0; }
11   // ...
12 };
13
14 GasContainer vol1{5, 180.5}, vol2{10};
```

Страшная правда о структурах в C++

```
1  class GasContainer
2  {
3    double pressure, temperature;
4    double crit_pressure; int volume;
5  };
6
7  // класс выше — тоже самое, что и:
8
9  struct GasContainer
10 {
11 private:
12   double pressure, temperature;
13   double crit_pressure; int volume;
14 };
```

Составные типы данных. Общие советы

- Используйте **структуры** исключительно в смысле языка C - как группировку значений других типов данных
- Если есть хоть одна причина добавить функции в тип данных - используйте **классы**
- При программировании на C++ передачу любых (своих и библиотечных) структур или классов осуществляйте **по ссылке** (избегайте ненужного копирования)
- При написании классов не определяйте конструктор без параметров, если на то нет веских оснований
- Не экономьте на названиях полей структур/классов (предпочитайте понятные названия кратким сокращениям)