# VI

# О чём материал?

В данной лекции будет продолжено изучение различных типов данных, которые предоставляет или позволяет определить язык C++. Если подойти неформально, любая вычислительная задача решается с помощью выбора набора типов, через переменные которых будут храниться данные, и последовательности преобразований данных, определяемые через функции (как собственные в программе, так и библиотечные). Поскольку предметных областей для вычислительных задач — море, то от выбора типов в первую очередь зависит понятность кода программы. Поэтому, для успешного освоения языка программирования необходимо, как минимум, иметь представление как о встроенных в язык типах, так и о возможностях определения **пользовательских** типов данных.

# Типы данных в C++

Все типы в C++ можно разделить на категории следующей диаграммой:

**Типы данных**

↙ ↓ ↘

Фундаментальные

**int**
**char**
**double**
и прочие...

Специальные

**массивы**
**указатели**
**ссылки**

Пользовательские

**псевдонимы**
**составные типы**

# Типы данных в C++

Ещё раз хотелось бы подчеркнуть, что выделение «специальной» группы типов сделано для обращения дополнительного внимания на них. Как демонстрировалось для массивов и указателей, некоторые операции с ними подчиняются отдельным правилам: сложение указателей с целыми числами; передача массивов в функции и т.п. И эти правила операций отличаются от общих для группы «фундаментальных» типов. При этом, «специальные» типы также являются встроенными в язык C++.
Конечно, в C++ ещё остаются функции. Однако, они не являются так называемыми **объектами первого порядка**. Это те сущности языка программирования, которые могут использоваться в качестве параметров других функций. Как разбиралось в четвёртой лекции (49 слайд), функции можно передавать как параметры других функций с помощью механизма указателей. Но это не то же самое, что использовать отдельную функцию как параметр. Поэтому в C++ функции не ассоциируют с типами данных (как правило).

# Типы данных в C++

В свою очередь, все возможности для создания пользовательских типов представляются диаграммой:

**Пользовательские типы данных**

Псевдонимы (aliases)

**using**
**typedef**

Составные типы

**struct**
**union**
**enum**
**enum class**
**class**

# Типы данных в C++

Для начала, рассмотрим последний из **специальных** типов — ссылки. Затем вернёмся к пользовательским типам.

# Ссылки в C++

В дополнении к **указателям**, C++ позволяет получать доступ к значениям *объектов* C++ («объект» — в смысле некоторого хранилища в памяти, в котором записано некоторое значение) через механизм, называемый **ссылками** (англ. reference). Как следует из названия, это дополнительный способ добраться до значения некоторого объекта через дополнительные переменные-ссылки.

В C++ есть два вида ссылок. В целях упрощения изложения матерала и терминологии, будем различать их как **ссылки на переменные** и **ссылки на временные объекты**. Если с переменными, хочется верить, примеров приводить не надо, то во втором случае можно напомнить: в языке может быть создан временный объект любого типа и для него всегда есть *литерал*, с помощью которого его значение выражается в тексте программы.

Для повторения, вот примеры временных объектов для различных типов:

```
1  4;    // литерал типа int
2  3.25; // литерал типа double
3
4  // Ниже всё, что в фигурных скобках,
5  // является литералом составного типа.
6  // В данном случае, этот тип — массив int'ов
7  // на 4 элемента.
8  int arr[] = {3, 4, 5, 6};
9
10 "abcde"; // литерал типа const char*
```

В строках **1, 2, 8, 10** присутствуют *временные объекты* (также употребляется термин «временные значения»)

# Ссылки в C++

Возращаясь к ссылкам, общий вид создания *переменных-ссылок* для обоих случаев следующий:

```
// ссылка на переменную
Type & ref_name = var;

// ссылка на временный объект
Type && ref_name = temp_var;
```

Знаки одного (**&**) и двух (**&&**) амперсандов говорят о том, что создаются ссылки каждого вида на тип **Type**. Кроме того, знак присваивания и некоторый объект являются обязательными. Под **var** понимается какая-то переменная соответствующего типа, под **temp_var** — временный объект. Ссылки в C++ не имеют никакого **нулевого значения**, они всегда должны быть связаны с каким-нибудь объектом программы. И, конечно, ссылки *типизированны*.

# Ссылки в C++

Для начала, пример с **ссылками на переменные**:

```
1  int i_num = 293;
2  double real = 55.88;
3
4  int &i_ref = i_num;
5  double & r_ref = real;
6  // Вычитаем 13 и сохраняем результат в i_num
7  i_ref -= 13;
8  printf("i_num = %d; i_ref = %d\n",
9         i_num, i_ref);
10 // удавиваем значение переменной real
11 r_ref *= 2.0;
12 printf("real = %.3f; r_ref = %.3f\n",
13        real, r_ref);
```

Ссылки создаются в строках **4, 5**. В строках **7, 11** через ссылки меняются значения исходных переменных. Если выполнить этот кусок кода, в консоли значения и переменной, и ссылки на неё будут совпадать.

# Ссылки в C++

Для создания нескольких ссылок в одном выражении, нужно указывать знак амперсанда у каждой переменной. Кроме того, ссылка может быть константной: можно получать значение переменной, на которую ссылаемся, но нельзя изменить:

```
1 int i1 = 10, i2 = 12, i3 = 14;
2
3 int &r1 = i1, &r2 = i2;
4 const int &r3 = i3;
5 // r3 *= 3; // Не получится!
6
7 printf("i1 + i2 + i3 = %d\n", r1 + r2 + r3);
```

Плюс пример демонстрирует, что ссылка может участвовать в любом выражении. Будет использовано значение переменной, на которую и происходит ссылка.

# Ссылки в C++

По смыслу, сама по себе **переменная-ссылка** аналогична **переменной-указателю**: это некоторое хранилище, в котором сохранён адрес переменной, на которую ссылаемся. Но изменить сохранённый адрес в случае ссылок **нельзя**. Более того, операция взятия адреса для переменной-ссылки всегда будет возвращать адрес исходного объекта. Для демонстрации:

```
1 int i4 = 125, i5 = 555;
2 int &ref4 = i4;
3
4 printf("addr of i4   = %p\n", &i4);
5 printf("addr of ref4 = %p\n", &ref4);
6
7 ref4 = i5;
8 printf("addr of i5   = %p\n", &i5);
9 printf("addr of ref4 = %p\n", &ref4);
```

В **седьмой строке** переменной **i4** было присвоено значение переменной **i5**.

# Ссылки в C++

Второй вид ссылок: **ссылки на временные объекты**. По сути, всё аналогично **ссылкам на переменные**, только в качестве объектов нужно **всегда использовать** временные значения.

```
1 //double &&ref; // Не компилируется!
2 double&& rv_ref = -555.444;
3 rv_ref += 1.33;
4 printf("value of rv_ref: %.3f, addr of rv_ref: %p\n",
5        rv_ref, &rv_ref);
6
7 const int &&rv_i_ref = 333;
8 //rv_i_ref *= 10; // Не получится!
9 printf("rv_i_ref = %d\n", rv_i_ref);
```

Также можно создавать константные ссылки на временные значения.

# Ссылки в C++

И существенное отличие между двумя видами ссылок: константные (неизменяемые) **ссылки на переменные** могут ссылаться на временные значения, но константные **ссылки на временные объекты** не могут ссылаться на переменные.

```
1 const int &ref1 = 777; // Всё ок
2
3 int i_val = 101;
4 //const int &&ref2 = i_val; // Не сработает
```

Данная особенность неявно упоминалась при обсуждении передачи параметров по константным ссылкам в функции. Далее следует резюме по ссылка в C++.

# Ссылки в C++: резюме

- у ссылок нет **нулевого** значения, ссылка всегда должна быть инициализирована каким-нибудь объектом;
- для получения значения используется сама переменная-ссылка, без дополнительных операторов (в противоположность разыменованию для указателей);
- при взятии адреса ссылки возвращается адрес объекта, на который она ссылается;
- ссылками уже пользовались при передаче параметров в функции и рассмотрении *диапазонного* цикла **for**;
- **никогда** не возвращайте из **функций** ссылки на локальные переменные/временные значения. Ещё раз — просто **никогда**;
- ссылки играют важную роль при реализации объектно-ориентированного функционала в C++, так что познакомиться с ними заранее не бесполезно.

# Ссылки в C++: пропустить этот слайд

Если есть желающие посмотреть на формализацию видов ссылок: в C++ объекты (или даже обобщение — *выражения*) делятся на различные категории. На данный момент есть пять категорий, основными являются, так называемые, **lvalue** и **rvalue**. В их определении есть нюансы, но на практике можно следовать упрощённому правилу:

- **lvalue** — это те объекты/выражения, которые могут быть **левым операндом** оператора присваивания и имеют адрес. Так вот, *ссылки на lvalue* выше обозначены как **ссылки на переменные**;
- **rvalue** — значения, которые не имеют адреса (временные значения) и используются либо в качестве **правого операнда** оператора присваивания, либо в качестве *литералов типов*. *Ссылки на rvalue* — это **ссылки на временные объекты**.

Основы категорий можно подчерпнуть тут:
https://en.cppreference.com/w/cpp/language/value_category

Переходим к **пользовательским** типам (англ. *user-defined types*). Поскольку немного слайдов было посвящено ссылкам, вновь вернёмся к диаграмме способов определения собственных типов:

**Пользовательские типы данных**

↙ ↘

Псевдонимы (aliases)

**using**
**typedef**

Составные типы

**struct**
**union**
**enum**
**enum class**
**class**

# Типы данных в C++

Итак, пользовательские типы определяются через два механизма C++:

1. Задание псевдонимов: возможность ввести новое название для любого известного программе типа. Под «известным программе» следует понимать любой встроенный тип языка и любой тип из уже подключённых в программу библиотек.
2. Определение собственных *составных* типов. Как следует из названия, данные типы объединяют какие-то сущности (и операции над ними) языка программирования. Что за сущности и как объединяют — вскоре разберёмся.

# Пользовательские типы. Псевдонимы

Псевдонимы означают именно то, что содержится в смысловом определении слова «псевдоним»: определяется новое название для существующего типа, при этом никаких дополнительных операций или свойств у типа не появляется. Для их создания могут быть использованы два ключевых слова — **using** или **typedef**. Первое из них появилось со стандарта C++11 (конец 2011 года) и на данный момент является предпочтительным способом для определения псевдонимов, второе — пришло из языка C, долгое время использовалось и в C++, и теперь уже осталось для совместимости с ранее написанным кодом.

# Пользовательские типы. Псевдонимы

В актуальный C++ добавление псевдонимов осуществляется с помощью ключевого слова **using**. Общий синтаксис следующий:

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

Основными причинами использования псевдонимов являются:

1. упрощение длинного, сложного или «некрасивого»* названия типа;
2. использование терминологии из предметной области задачи.

* Про «некрасивые» названия саркастическая шутка (или жалоба, как посмотреть): в программировании есть две сложные проблемы. И одна из них — как выбирать имена переменных, типов или функций.

# Пользовательские типы. Псевдонимы I

Рассмотрим тип **unsigned long long**. Вот есть необходимость его использовать. Набирать его для переменных, параметров функций, возвращаемых значений, массивов — достаточно быстро утомляет. Тут и может помочь объявление псевдонима:

```
1 using ull      = unsigned long long;
2 using ull_ptr = unsigned long long*;
3
4 ull_int     val1 = 5555555555;
5 ull_int_ptr ptr1 = &val1;
6
7 printf("ptr pointed to %Lu\n", *ptr1);
```

- в 1-ой и 2-ой строках объявили два псевдонима: для типа **unsigned long long** (1-ая строка) и для указателя на **unsigned long long** (2-ая);

# Пользовательские типы. Псевдонимы II

- в 4-ой и 5-ой строках объявили соответствующие переменные и инициализировали их. Обратите внимание, для указателя не пришлось ставить знак «*»;
- в 7-ой строке использовали указатель как обычно: разыменовали и показали на консоли значение, на которое ссылается указатель в этом примере.

**Полная взаимозаменяемость**

Переменные, создаваемые с помощью псевдонимов, полностью взаимозаменяемы с переменными исходных типов. Для примера, если реализуем функцию один из параметров которого будет иметь переопределённый выше тип **ull**, то любая переменная типа **unsigned long long** может выступать в качестве аргумента для этого параметра.

# Пользовательские типы. Псевдонимы

К двум предыдущим псевдонимам вокруг типа **unsigned long long** добавим ещё один: псевдоним для типа *статический массив на 10 беззнаковых целых*.

```
1 using tenth_ulls = unsigned long long[10];
2
3 tenth_ulls counters = {0};
4 // unsigned long long counters[10] = {0};
5
6 counters[2] = 88;
7 printf("Third elem: %Lu\n", counters[2]);
```

- в 1-ой строке задали псевдоним;
- в 3-ей создали переменную **counters**, тип которой и есть статический массив на 10 беззнаковых целых; 4-ая строка демонстрирует объявление такой же переменной без псевдонима;
- 6-ая, 7-ая строки — работаем с переменной как со статическим массивом.

# Пользовательские типы. Псевдонимы I

Ещё более удобным псевдоним может быть для многомерных статических массивов. Например, нужно преобразовывать каким-нибудь образом матрицы размера **4x4**. Чтобы не таскать за собой константу-размер, воспользуемся псевдонимом:

```
1  const size_t FOUR = 4;
2  using matrix4x4 = double[FOUR][FOUR];
3
4  void fill_matrix_by_rand(matrix4x4& matr,
5                           double a, double b)
6  {
7    for (size_t i = 0; i < FOUR; i++) {
8      for (size_t j = 0; j < FOUR; j++) {
9        matr[i][j] = rand_a_b(a, b);
10     }
11   }
12 }
```

# Пользовательские типы. Псевдонимы II

```
13
14 matrix4x4 matr1;
15 fill_matrix_by_rand(matr1, -5, 5);
16 printf("Elem[%d][%d] = %.4f", 2, 3, matr1[2][3]);
```

Задали константу для размера, задали псевдоним (**строка 2**) для двумерного массива с числом строк и столбцов равным четырём, определили функцию для заполнения произвольного массива случайными числами в заданном диапазоне. Ни для параметра функции, ни при объявлении переменной нужного типа не была засвечена константа **FOUR**.

Для сравнения, без псевдонима объявление функции выглядело бы так:

```
1 void fill_matrix_by_rand(
2                          double (&matr)[FOUR][FOUR],
3                          double a, double b);
```

Псевдоним позволил уменьшить синтаксический шум.

# Пользовательские типы. Псевдонимы

Использование псевдонимов полезно для осуществления предметно-ориентированной разработки: когда по названиям типов переменных или по объявлениям функций можно однозначно сказать, какой смысл несёт значение переменной/возвращаемое значение. Предположим, программа проверяет набор устройств и с каждого периодически надо получать значение тока и напряжения. Очевидно, что для идентификации устройств подойдёт целое положительное значение, для значений силы тока/напряжения — числа с плавающей точкой.

# Пользовательские типы. Псевдонимы

Для упрощения чтения такой условной программы можно воспользоваться псевдонимами:

```
1 using amperage_t  = double; // Амперы
2 using voltage_t   = double; // Вольты
3 using device_id_t = size_t; // Идентификатор
4
5 amperage_t current_now(device_id_t id);
6 voltage_t  voltage_now(device_id_t id);
```

В примере по объявлениям функций уже понятно, какие значения по смыслу они вернут. Опять же, без псевдонимов аналогичная функциональность могла иметь такой вид:

```
1 double current_now(size_t device_id);
2 double voltage_now(size_t device_id);
```

, где о том, что за числа нужно передавать в функцию можно догадаться только по названию параметров; а в каких единицах (вольты, киловольты и прочие) функции возвращают значения — нужно читать документацию.

# Пользовательские типы. Псевдонимы

Предметно-ориентированность: тригонометричекие функции, принимающие значения в градусах, а не радианах.

```
1  using angle_t = double;
2
3  double sin_at(angle_t angle)
4  {  return sin(angle * M_PI / 180); }
5
6  double cos_at(angle_t angle)
7  { return cos(angle * M_PI / 180); }
8
9  double tg(angle_t angle)
10 { return tan(angle * M_PI / 180); }
11
12 printf("tg(45 degree) = %.3f\n", tg(45.0));
```

Опять же, уже по сигнатуре функции становится понятнее, как будет интерпретировать переданное в неё значение.

# Пользовательские типы. Псевдонимы

Для совместимости с предыдущими версиями стандарта C++ псевдонимы могут быть определены с помощью ключевого слова **typedef**. Кому интересно, в языке C это единственный способ их определения. Синтаксис:

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

Определение общее, далее продемонстрируем на примерах.

# Пользовательские типы. Псевдонимы

Оператор **typedef**

```
1 typedef unsigned long long ull, *ull_ptr;
2 typedef double matrix4x4[FOUR][FOUR];
```

В первой строке аналоги 21 слайда, во второй — определение псевдонима для двумерного массива типа **double** размера 4 на 4.

**Полезный совет**

При написании программ не используйте **typedef**, если компилятор поддерживает стандарт C++11 и новее: синтаксис **using** всё-таки очевиднее при просмотре программы (разделены псевдоним и сам тип. Опять же, сравните пример текущего слайда против 21-го и 24-го).

27 слайдов спустя после упоминания

# Добираемся до составных типов

# Составные типы I

На 17 слайде были перечислены 5 ключевых слов, предоставляемых языком C++ для создания составных типов. Из них,

1. **Структуры** **struct** и **классы** **class** служат для определения новых типов, используя уже известные в качестве «строительных кирпичей». Буквально, с их помощью определяется «прототип» составного хранилища, в которое может входить произвольное количество элементов. Элементом может быть любой известный к моменту определения тип. Технически, создаваемые этими ключевыми словами типы являются идентичными (с точностью до малозначащих в данный момент нюансов). Но есть историческое смысловое различие между **struct** и **class**. Так, классы являются главными составляющими для реализации

# Составные типы II

объектно-ориентированного подхода в C++. Структуры будут расмотрены далее, знакомство с классами отложим на некоторое время.

2. **Объединения** **union** позволяют хранить значения различных типов в одном хранилище. При этом, в один момент времени переменная-объединения может хранить только единственное значение из набора типов, задаваемых при определении типа.
3. Открытые **enum** и закрытые **enum class** **перечисления** служат для объединения набора целочисленных именованных констант под одним типом.

# Составные типы. Структуры

Начнём со структур. Само ключевое слово **struct** также пришло в C++ из языка C. Суть структур заключается в объединении набора значений, связанных по смыслу, в отдельном типе. Объединяемые значения могут быть произвольных типов, никак ограничений нет. Переменные структурного типа являются составным хранилищем, которое состоит из *подхранилищ* под каждое значение. Далее мы рассмотрим структуры только в этом качестве: как проектирование составных хранилищ. Назовём это дело **структурами в смысле языка C**.

В хранении нескольких значений структуры похожи на статические массивы. Различие в том, что структуры могут хранить значения разных типов. И тут уже не построишь доступ к конкретному элементу за счёт байтовых сдвигов в оперативной памяти. Поэтому, каждое *подхранилище* структуры должно быть проименованно. Используя идентификатор элемента структуры, компилятор осуществляет запись/чтение соответствующих значений.

# Составные типы. Структуры

Формальное определение структуры:
**Структура** (в смысле языка C) - это составной тип данных, объединяющий множество *проименнованных* типизированных элементов. **Элементы структуры** называют **её полями**. Тип поля может быть любой, известный к моменту определения структуры.
Общий синтаксис определения структурного типа:

```
struct [<название_нового_типа>]
{
  <тип_1> <поле_1> [, <поле_2>, ...];
  <тип_2> <поле_1> [, <поле_2>, ...];
  ...
  <тип_n> <поле_1> [, <поле_2>, ...];
} [переменная1, переменная2, ...];
```

# Составные типы. Структуры

Для демонстрации практических особенностей определения и использования структур в первую очередь нужен какой-нибудь математический объект, в котором объединение разных значений выглядело бы естественно.
Обратимся к школьной физике и такому понятию как *материальная точка*. Это некоторый объект, обладающий массой и координатами в трехмерном пространстве.
Материальная точка появлялась в задачах, в которых формой и размером физического тела можно было пренебречь.
Уверен, вы помните: «тело брошено под углом к горизонту ... сопротивлением среды пренебречь» и подобные задачки.
С точки зрения языка программирования, материальная точка представляет собой составной объект, состоящий из трёх значений для координат и одного значения для массы. Для упрощения (а также ради демонстрации комбинации различных типов) сделаем координаты дискретными: выразим их только целыми числами.

# Составные типы. Структуры

И первый практический пример определения структуры под *материальную точку*:

```
1 struct MaterialPoint
2 {
3   int x, y, z;
4   double mass;
5 };
```

- **В первой строке** определяется название пользовательского типа — **MaterialPoint**, которое следует за ключевым словом **struct**.
- **В третьей строке** объявляем три поля под декартовы координаты. Каждое поле имеет тип **int**. Перечисление полей через запятую по смыслу аналогично объявлению нескольких переменных в одной инструкции.
- **В четвёртой строке** определяем поле для хранения массы материальной точки.
- Точка с запятой **в пятой строке** — обязательна.

# Составные типы. Структуры

Определение структуры на предыдущем слайде создало новый тип в рамках условной программы. При этом, само по себе определение не добавило ни одного объекта в программу. Это, если хотите, некоторая техническая спецификация составного типа, которая декларирует его (типа) название и составляющие (то есть — поля).
Кроме того, задание полей абсолютно произвольно. Нет никаких требований на то, чтобы поля одного типа перечислялись обязательно через запятую. Например,

```
1 struct MaterialPoint
2 {
3   int x;
4   int y;
5   int z;
6   double mass;
7 };
```

является полным аналогом рассматриваемой структуры.

# Составные типы. Структуры

Раз есть новый тип, объявим его переменные. Продолжая пример с 37-го слайда, создание трёх переменных типа **MaterialPoint** делается так:

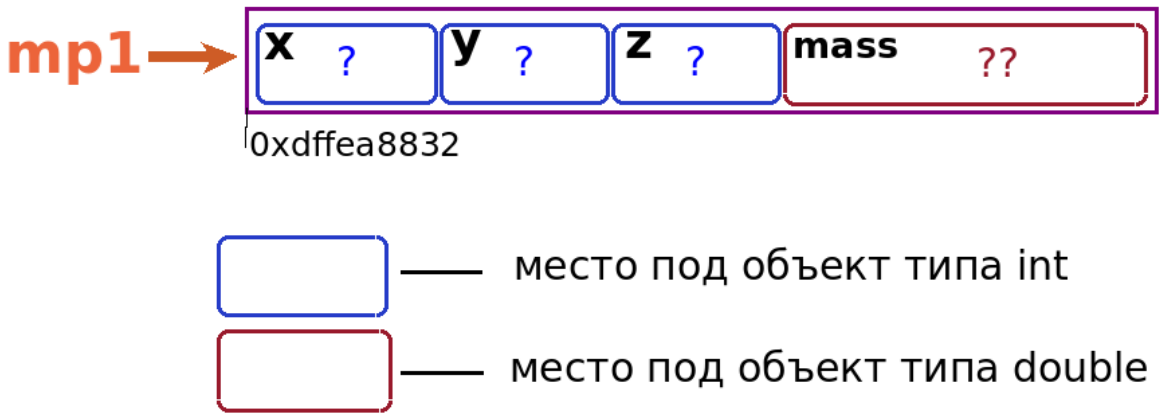
```
7 MaterialPoint mp1, mp2, mp3;
```

Вот уже здесь, при объявлении переменных происходит создание объектов (в смысле C++): выделение хранилища (в памяти) и инициализация.

- **Выделение хранилища** под структурный объект: для каждой переменной выделяется хранилище, которое представляет собой **непрерывный** блок в памяти, состоящий из «подблоков» под каждое поле.
- **Инициализация**: в приведённом виде, к каждому полю конкретной переменной применяется **инициализация по умолчанию** (лекция 4, 26 слайд). Причём, применение идёт последовательно, от первого поля к четвёртому.

На примере переменной **mp1** картина в памяти будет следующей:

**Схематичное представление переменной структурного типа**




Общая рамка говорит о том, что это непрерывный блок памяти. Внутри подблоки под каждое поле располагаются **в порядке объявления полей** в структурном типе. Знаки вопросов подходят к примеру предыдущего слайда, где переменным не предоставлено никаких начальных значений.

Поскольку переменные структурного типа, как и статические массивы, являются составными объектами, для задания начальных значений применяется *инициализация составных типов* (list initialization). Её суть заключалась в следующем: в паре фигурных скобок через запятую перечислятся значения, которые записываются в элементы объекта. Если вспоминать про инициализацию, то это пример — *direct initialization*. Значения подставляются последовательно: в массиве с первого элемента и далее по порядку, в структурах — с первого объявленного поля и далее по порядку. Если указанных в фигурных скобках значений меньше, чем элементов составного объекта, то к составляющим, которым значений не хватило, применяется *инициализация значением по умолчанию* (за подробностями терминов — в четвёртую лекцию).

Сформулированное выше выражается в коде следующим образом:

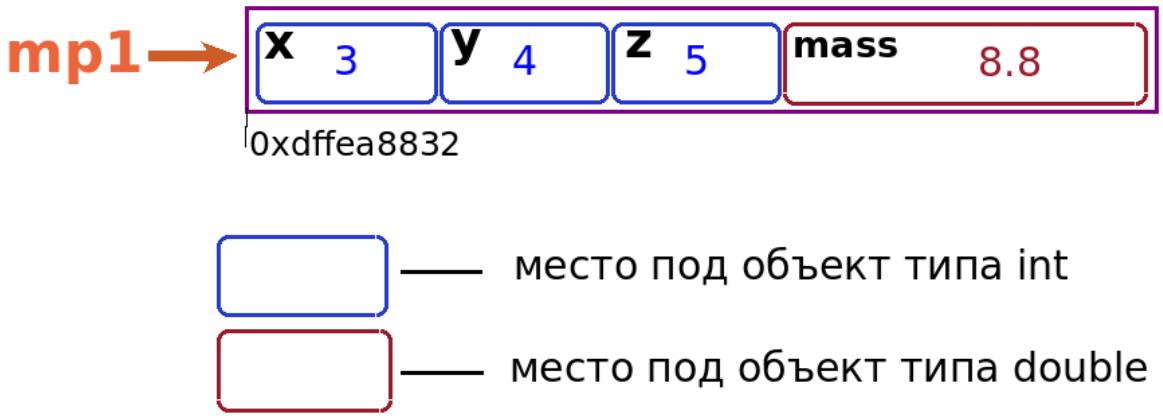
```
1 MaterialPoint mp1 {3, 4, 5, 8.8};
2 MaterialPoint mp2 = {5, 8};
```

- Полям **x, y, z, mass** переменной **mp1** последовательно присвоены значения **3, 4, 5** и **8.8**. Строго в порядке их (полей) объявления.
- В переменной **mp2** поля **x** и **y** получили значения **5** и **8**, а поля **z** и **mass** — **0** и **0.0**.
- Стоит упомянуть, что значением структурной переменной является весь совокупный набор значений её полей.

Обе формы *list initialization* срабатывают одинаково.

При задании начальных значений, представление в памяти переменной структурного типа будет чуть иным:

**Схематичное представление переменной структурного типа**




Как видно, принципиально ничего не поменялось, кроме уверенности в том, что за набор начальных значений хранится в структурной переменной.

Есть способ предотвратить создание структурных переменных с неопределёнными значениями полей — задать **значения по умолчанию** для полей.

```
1  struct Vector2D
2  {
3    int x = 1;
4    int y = 1;
5  };
6
7  Vector2D v1, v2 = {4};
8  printf("\t v1 --- v2\n");
9  printf("x: \t %d --- %d\n", v1.x, v2.x);
10 printf("y: \t %d --- %d\n", v1.y, v2.y);
```

Полям переменной **v1** будут присвоены единицы, полю **x** переменной **v2** — значение равное четырём.

# Составные типы. Структуры

Перейдём к общим операциям, которые по умолчанию работают для различных структурных типов. В первую очередь, это получение значения конкретного поля у структурной переменной. Делается это с помощью оператора, представленного в C++ символом точки: «.» (в литературе известен как оператор-точка).

```
1 MaterialPoint mp1 {3, 4, 5, 8.8};
2 MaterialPoint mp2 = {5, 8};
3
4 printf("Mass of mp1 is %.2f\n", mp1.mass);
5 mp2.z = 11;
6 printf("z-coord of mp2 is %d\n", mp2.z);
```

Как показывает пример, оператор «.» применяется к идентификатору переменной и ожидает название нужного поля. После этого с выражением «**var.field_name**» работаем как с объектом, представленным типом поля (читаем/записываем значение).

В отличие от статических массивов (как примера составного типа), структурным типам предоставляется операция присваивания. В этом случае все значения полей из правого операнда оператора «=» копируются в соответствующие поля левого операнда.

```
1 MaterialPoint mp3 {1, 1, 1, 0.5};
2 MaterialPoint mp4 {7, 1, 4, 1.5};
3
4 printf("Mass of mp3 is %.2f\n", mp3.mass);
5 mp3 = mp4;
6 printf("Mass of mp3 is %.2f\n", mp3.mass);
```

После копирования в 5-ой строке, в поле **mass** переменной **mp3** будет записано значение **1.5**. Соответствующее подтверждение выведется на консоль при запуске примера.

# Составные типы. Структуры

Все поля рассмотренных ранее типов копируются очевидным образом, за исключением **статических массивов**. В этом случае будет выполнено **поэлементное** копирование значений одного массива в другой! Просто для напоминания, для массивов фиксированного размера не предоставляется оператор присвоения. Действительно, попытка скопировать массивы, пусть и одинакового размера, вызовет ошибку компиляции:

```
1 char st1[50] {"some smart expression"};
2 char st2[50] {"other sentence"};
3
4 printf("<<%s>>\n<<%s>>\n\n", st1, st2);
5 st1 = st2; // Тут будет ошибка копиляции!
6 printf("<<%s>>\n<<%s>>\n\n", st1, st2);
```

# Составные типы. Структуры

А теперь сделаем обёртку строки фиксированного размера в структурном типе:

```
1  struct StrWrapper
2  {
3    char str[50];
4  };
5
6  StrWrapper st1{"some smart expression"};
7  StrWrapper st2{"other sentence"};
8
9  printf("<<%s>>\n<<%s>>\n\n", st1.str, st2.str);
10 st1 = st2; // It works!
11 printf("<<%s>>\n<<%s>>\n\n", st1.str, st2.str);
```

И копирование работает без определения каких-либо дополнительных функций.

# Составные типы. Структуры

Данный приём работает для статических массивов любых типов:

```
1  const size_t SZ = 20;
2  struct DemoType
3  {
4    char indicator;
5    int arr[SZ];
6  };
7
8  DemoType var1 {'e', {2, 3, 4, 5, 6, 7, 8, 9}};
9  DemoType var2 {'U', {-1, -2, -3, -4, -5}};
10 for (size_t i = 0; i < SZ; i++)
11 { printf("%d ", var2.arr[i]); }
12 printf("\n");
13
14 var2 = var1;
15 for (size_t i = 0; i < SZ; i++)
16 { printf("%d ", var2.arr[i]); }
```

На этом операции по умолчанию закончились. Резюмируя, их всего две — получение доступа к значению поля и присвоение. Для произвольного структурного типа C++ по умолчанию не определеяет никаких операторов сравнения:

```
1 MaterialPoint mp3 {1, 1, 1, 0.5};
2 MaterialPoint mp4 {7, 1, 4, 1.5};
3
4 // mp3 < mp4; // Не компилируется
5 // mp3 == mp4; // Не компилируется
6 // mp3 >= mp4; // Не компилируется
```

Это практично: поскольку смысл вводимого в программу структурного типа компилятор никак узнать не может, какие-либо предположения об отношении между значениями этого типа априори нежелательны.

# Составные типы. Структуры

Поскольку операция присваивания (фактически — копирование) работает для структурных типов по умолчанию, эти типы могут использоваться как в качестве параметров функций, так и в качестве возвращаемого значения. Тут работают все правила, аналогичные *фундаментальным типам*: параметр структурного типа может передаваться как **по значению** (и при передаче аргумента происходит копирование), так и **по ссылке** (исключаем копирование). Для примера, проверка материальных точек на равенство:

```
1 bool is_equal(const MaterialPoint& p1,
2                const MaterialPoint& p2)
3 {
4   return (p1.x    == p2.x) &&
5          (p1.y    == p2.y) &&
6          (p1.z    == p2.z) &&
7          (p1.mass == p2.mass);
8 }
```

Функция **is_equal** принимает два параметра типа **MaterialPoint**, которые передаются по константной ссылке. Для упрощения проигнорированны особенности сравнения чисел с плавающей точкой. Использование тривиально:

```
1  MaterialPoint mp5 {1, 1, 1, 0.5};
2  MaterialPoint mp6 {7, 1, 4, 1.5};
3  MaterialPoint mp7 {7, 1, 4, 1.5};
4
5  if (is_equal(mp5, mp6)) {
6    printf("mp5 and mp6 has the same values for all ↩
         fields\n");
7  }
8  if (is_equal(mp6, mp7)) {
9    printf("mp6 and mp7 has the same values for all ↩
         fields\n");
10 }
```

Первый вызов функции **is_equal** вернёт **false**, соответственно ничего не напечатается в консоли. Второй же вернёт **true** и напечатает соответствующую строчку.

# Составные типы. Структуры

Возвращаемым типом структура тоже может быть:

```
1 MaterialPoint zero_point()
2 {
3   return {0, 0, 0, 0.0};
4 }
5
6
7 const MaterialPoint zero = zero_point();
8 printf("zero.x = %d\n", zero.x);
```

Причём использование структурного типа является второй возможностью (после передачи параметров по ссылке) для возврата из функции более одного значения. Константы структурного типа используются без каких-либо скрытых особенностей.

# Составные типы. Структуры

Также для составных типов применим весь механизм указателей C++. Можно объявить указатель на структурный тип и добраться до значения структуры через операцию разыменования:

```
1 MaterialPoint point = {1, 2, 1, 2.1};
2 MaterialPoint *ptr_to_point = &point;
3
4 (*ptr_to_point).z = 8;
5 printf("{x = %d, y = %d, z = %d}\n",
6         (*ptr_to_point).x, (*ptr_to_point).y,
7         (*ptr_to_point).z);
```

Каждое разыменование обёрнуто в скобки, потому что приоритет оператора-точки выше, чем у оператора-разыменования, и он пытается выполниться первым. А у типа **указатель на MaterialPoint** никаких полей не существует. Таблица приоритетов операторов: github.com/posgen/OmsuMaterials/wiki/Operators-priority

# Составные типы. Структуры

Поскольку ставить скобки вокруг разыменования показалось утомительным ещё создателям языка C, то в него добавили оператор «->» (а затем оператор без изменений перешёл и в C++). В литературе встречается под названием *оператор-стрелка* (сложно предположить почему). Этот оператор осуществляет два действия: разыменовывает указатель и обращается к конкретному полю объекта, на который ссылается указатель. Предыдущий пример преобразуется в:

```
1 MaterialPoint point = {1, 2, 1, 2.1};
2 MaterialPoint *ptr_to_point = &point;
3
4 ptr_to_point->z = 8;
5 printf("{x = %d, y = %d, z = %d}\n",
6        ptr_to_point->x, ptr_to_point->y,
7        ptr_to_point->z);
```

Ничего не поменялось, а синтаксического шума меньше.

# Составные типы. Структуры

С помощью указателей осуществляется работа с динамическими объектами структурных типов. Для воспоминаний, динамический объект — это объект конкретного типа, временем жизни которого управляем вручную с помощью операторов **new**/**delete**.

```
1  MaterialPoint *dyn_point =
2                      new MaterialPoint{1, 0, 1, 2.5};
3
4  printf("{x = %d, y = %d, z = %d}\n",
5         dyn_point->x, dyn_point->y, dyn_point->z);
6
7  dyn_point->y = -5;
8
9  printf("{x = %d, y = %d, z = %d}\n",
10        dyn_point->x, dyn_point->y, dyn_point->z);
11
12 delete dyn_point;
```

Создали динамический объект, поработали с ним через указатели, вернули память в ОС.

И ещё несколько технических особенностей определения структурного типа — без подробных пояснений.

1. После определения структурного типа можно сразу создавать его переменные в одном выражении:

```
1 struct Course
2 {
3   size_t days;
4   size_t lessons;
5   char name[100];
6 } progi{90, 20, "C++ language"};
7
8 printf("course name is %s\n", progi.name);
```

2. Структурный тип может быть объявлен без названия: случай так называемых **анонимных структур**.

```
1 struct
2 {
3   size_t attempts;
4   double score;
5 } game_one{10, 4.5};
6
7 printf("%zu --- %.3f\n", game_one.attempts,
8        game_one.score);
```

3. Определение структурного типа может быть помещено в любой блок кода: в том числе, в функции и другие определения структур.

```
1  int just_show(int number)
2  {
3    struct LocalStruct
4    {
5      int f1;
6      int f2;
7    };
8
9    LocalStruct ls {10, 5};
10
11   return number * ls.f1 - ls.f2;
12 }
```

4. **Обновлённая информация**: До стандарта **C++17** была возможность *встраивания* анонимных структур в другие. Такой код можете где-нибудь встретить:

```
1  struct Outer
2  {
3    int attempts;
4
5    struct { int field1; double field2; };
6  };
7
8  Outer test = {10};
9  test.field1 = 8;
10 printf("%d, %.2f\n", test.field1,
11         test.field2);
```

Начиная со стандарта **C++17**, встраивание анонимных структур **запрещено правилами языка**.

5. И ещё случай вложенных структур: если вложенная структура не является анонимной, то этот тип может быть получен с помощью оператора разрешения области видимости «::»:

```
1  struct Outer
2  {
3    struct Inner {
4      int x;
5      int y;
6    };
7
8    double field;
9  };
10
11 Outer::Inner obj = {4, -7};
12 printf("%d, %d\n", obj.x, obj.y);
```

6. Поле структуры может являться **указателем на собственный тип**:

```
1 struct ListNode
2 {
3   double vip_data = 0.0;
4   ListNode *next = nullptr;
5 };
6
7 ListNode node;
8 printf("node.next == nullptr %d\n",
9        node.next == nullptr);
```

Здесь суть в том, что для создания указателя (или ссылок) на какой-нибудь тип, языку C++ достаточно знать название этого типа, даже если его определение было отложено. С отложенным определением для структурных типов познакомимся когда-нибудь в другой раз.

Резюме по структурам **в смысле языка C**:

- структуры полезны, когда есть данные, которые описывают одну логическую сущность, но сами по себе могут меняться независимо;
- полями могут быть любые известные к моменту объявления структуры типы данных;
- для задания начальных значений применяется *инициализация составных типов*;
- полям можно задавать значения по умолчанию;
- по умолчанию отсутствуют операторы сравнения;
- дополнительные возможности при использовании статических массивов в качестве полей.

# Составные типы

Слайдов уже многовато, **объединения** и **перечисления** будут рассмотрены в следующей лекции.