Лекция IX - X

1, 15 марта 2019

Обобщённое повторение: все формы C++ для создания переменных

Создание переменных

В общем случае, язык C++ предоставляет пять форм для инициализации:

```
1 Type var1;
2 Type var2 = value;
3 Type var3(value, ...);
4 Type var4{value, ...};
5 Type var5 = { value, ... };
```

Это все способы C++ для создания переменных любого типа. Для *фундаментальных*, *специальных* (статические массивы, указатели) типов - данные формы работают по умолчанию. В **(3)** - **(5)** в скобках можно и не указывать ни одного значения. Для *составных, имеющих открытые поля, типов* по умолчанию не предоставляется форма **(3)**. Для *классов* (в целом для типов, определяемых пользователем), которые включают в себя только закрытые поля, форма **(5)** отсутствует, формы **(3)** и **(4)** - работают в ограниченном случае.

Создание переменных

На примере типа **int** (фундаментальный (базовый) тип)

```
1 int var1;
2 int var2 = 105;
3 int var3(-111);
4 int var4{var3 / 5};
5 int var5 = { var4 };
```

- Форма **(1)** создаёт переменную с неопределённым значением, формы **(2) - (5)** - сохраняют строго определённые целые числа.
- Для базовых типов формы **(3) - (5)** должны принимать только **одно значение**; перечисление нескольких чисел в данном примере недопустимо.

Создание переменных

На примере структуры (составной пользовательский тип)

```
1  struct Thing
2  {
3      string name;
4      size_t amount;
5  };
6
7  Thing var1;
8  Thing var2 = {"hammer", 3};
9  Thing var3{"cell phone", 10};
10 Thing var4{var3};
11 Thing var5 = { "unknown", 103 };
```

При замене в девятой строке фигурных скобок на круглые возникнет ошибка компиляции.

Создание переменных

На примере класса

```
1 class Date
2 {
3     int day, month, year;
4 };
5
6 Date var1;      Date var2 = var1;
7 Date var3{};    Date var4{var3};
8 Date var5 = {};
```

Замена фигурных скобок на круглые в восьмой строке приведёт к ошибке компиляции.

Что стоит понимать?

Для составных пользовательских типов язык C++ предоставляет средства для полного или частичного переопределения форм для инициализации переменных. Любая форма по желанию может быть исключена. Любая форма может быть расширена согласно требуемой логике поведения при создании объектов.

+1 специальный тип: типизированные ссылки

Ссылки в C++

В дополнении к **указателям**, C++ позволяет получать доступ к значениям переменных через механизм, называемый **ссылками** (англ. reference). Общий вид создания *переменных-ссылок* такой:

```
Type & ref_name = var;
```

Знак амперсанда (**&**) говорит о том, что создаётся именно ссылка на тип **Type**. Кроме того, здесь знак присваивания и некоторая существующая переменная **var** *того же самого типа* - обязателены. **Ссылка** в C++ всегда должна быть связан с какой-то переменной, у неё нет никакого «нулевого» значения.

Ссылки в C++

На примере использование ссылок выглядит так:

```
1  int i_num = 293;
2  double real = 55.88;
3
4  int & i_ref = i_num;
5  double & r_ref = real;
6  r_ref *= 2.0; // удавиваем значение переменной real
7  std::cout << "real = " << real << std::endl;
8
9  // Когда хочется создавать несколько ссылок в одной
10 // инструкции, надо не забывать добавлять "&" к
11 // каждой из переменных
12 int i1 = 10, i2 = 12, i3 = 14;
13 int &r1 = i1, &r2 = i2;
14 // Ссылка может быть константной (неизменяемой)
15 const int &r3 = i3;
16
17 std::cout << "i1 + i2 + i3 = " << r1 + r2 + r3 << "\n";
```

Ссылки в C++

Различия ссылок и указателей:

- у ссылок нет «нулевого» значения, ссылка всегда должна быть инициализированна какой-нибудь переменной;
- ссылки не имеют специальной операции доступа к тому значению, на которое они ссылаются (ср. разыменование для указателей). Для получения значения - просто используется переменная-ссылка.

Вспоминая былое

Ссылками уже пользовались в функциях для избегания копирования при передаче аргументов.

Никогда не делать

Не возвращейте из **функций** ссылку/указатель на локальные данные этой функции!

И снова про классы и их возможности в C++

Стоит напомнить, что основы классов и связанная с ними терминология - были приведены в 6 лекции. Так что далее, не будет очень развёрнутых пояснений об терминах **конструктор/деструктор**, **метод/поле** класса, **объект**. Вместо этого, попробуем сосредоточиться на сути.

В основе написания собственных типов, в общем, и классов, в частности, лежит, как правило, идея удобства. А именно, если будет придуман и создан некоторый тип, упростит ли использование переменных этого типа некоторые действия, ранее выполняемые другим способом.
Для демонстрации реализуем новый тип: **одномерный динамический массив действительных чисел с положительной и отрицательной индексацией**

Положительная и отрицательная индексация означает следующее. Пусть есть переменная разрабатываемого типа:

```
1 DynArray1D reals;
2 ...
3 cout << reals[0];    // Вывод 1-го элемента в терминал
4 cout << reals[-1];   // Вывод 1-го с конца элемента
```

Прежде, чем переходить к технической реализации, конкретизируем *действия*, которые хотим выполнять с объектами разрабатываемого типа:

- создание массива: нулевой массив; массив заданной длины с неопределёнными значениями элементов; массив заданной длины с конкретным значением каждому элементу
- доступ к элементу по индексу: как положительному, так и отрицательному
- добавление элементов в уже созданный массив

Для начала, действий достаточно.

Итоговый результат: определение класса **DynArray1D**

```
1  class DynArray1D
2  {
3  public:
4      // Конструкторы и деструктор
5      DynArray1D() = default;
6      DynArray1D(size_t array_size);
7      DynArray1D(size_t array_size, double value);
8      DynArray1D(const DynArray1D& other);
9      ~DynArray1D();
10     // Методы класса
11     size_t length() const;
12     size_t capacity() const;
13     // Перегруженные операторы
14     double& operator[](int index);
15     DynArray1D& operator<<(double value);
16     void operator=(const DynArray1D& other);
17
18 private:
19     // Поля класса
20     double *_arr = nullptr;
21     size_t _length = 0;
22     size_t _capacity = 0;
23 };
```

Начинаем разбор. Основа любого составного типа - это его поля. Они отвечают за хранение конкретных данных, над которыми мы и определяем нужные нам действия. В данном случае, определено три **закрытых** поля (со значениями по умолчанию):

```
1 class DynArray1D
2 {
3 private:
4     // Поля класса
5     double *_arr = nullptr;
6     size_t _length = 0;
7     size_t _capacity = 0;
8 };
```

Указатель **_arr** используется для хранения адреса динамической памяти, в которой будут располагаться элементы массива. Поле **_length** отвечает за хранение количества элементов массива в конкретный момент времени.

Поле **_capacity** хранит в себе реальный размер блока динамической памяти, выделенной под массив. Это поле несёт в себе больше техническую особенность реализации контейнерных структур.
А именно, при добавлении нового элемента возможна ситуация, что динамическая память уже заполнена. В этом случае создаваемый массив должен выделить новый блок большего размера, скопировать туда текущие элементы и добавить новый элемент. Как правило, для уменьшения обращений к оператору выделения памяти, динамический блок никогда не расширяют на один элемент, а выделяют с некоторым запасом (есть нет существенных ограничений на объём используемой динамической памяти).
В нашем случае определим правило: если нужно расширение динамической памяти при добавлении нового значения в массив, будет выделять новый блок **в два раза больше** текущего.

Далее вспомним про методы класса. Пока в приведённом примере их - два:

```
1 class DynArray1D
2 {
3 public:
4     // Методы класса
5     size_t length() const;
6     size_t capacity() const;
7 };
```

И суть методов заключается в том, что мы можем их применить для *объектов* класса:

```
1 DynArray1D errors, values;
2 // Допускаем, что массивы как-то заполнены значениями.
3 cout << "Длина errors: " << errors.length() << "\n";
```

И методы, в первую очередь, и определяют те действия, что мы захотели делать с объектами класса.

```
1 class DynArray1D
2 {
3 public:
4     // Методы класса
5     size_t length() const;
6     size_t capacity() const;
7 };
```

Данные методы отвечают за следующие действия:

- **length** - получить количество элементов в массиве;
- **capacity** - получить размер выделенного блока памяти под массив.

В объявлениях методов добавлено ключевое слово **const** после списка параметров. Оно говорит о том, что при вызове этих методов для конкретных объектов класса не произойдёт изменения полей объекта. Причём, за тем, чтобы поля не менялись, следит компилятор.

И на примере метода **length** вспомним, что существует два способа определить метод: непосредственно внутри определения класса и вне его. **Первый способ**:

```
1 class DynArray1D
2 {
3 public:
4     // Методы класса
5     size_t length() const
6     {
7         return _length;
8     }
9 };
```

Сам метод внутри себя просто возвращает значение поля **_length** для конкретного объекта класса.

Второй способ:

```
1  class DynArray1D
2  {
3  public:
4      // Методы класса
5      size_t length() const;
6  };
7
8  size_t DynArray1D::length() const
9  {
10     return _length;
11 }
```

Всё тоже самое, только определение метода переехало из определения класса в отдельное место.

Переходим к конструкторам: по сути это специальные методы, которые отвечают за действия, происходящие при создании переменных-класса (объектов в принятой терминологии). И конечно же, прежде, чем реализовывать конструкторы в коде, надо для себя решить, что должно происходить при следующем коде:

```
1 DynArray1D example1;
2 DynArray1D example2{10}, example3{12, 1.8};
```

Создаётся три массива. Первый является массиво нулевой длины: нет ни элементов, ни динамической памяти под них, ни ёмкости. Второй и третий по идеи создают массивы заданной размерности (10 и 12, соответственно). А всем элементам третьего массива ещё и значение **1.8** присваивается.
Таким образом, определены как минимум три способа для создания объектов.

Вспоминая былое

Ради повторения отметим, что в примере на предыдущем слайде **для каждой** из трёх переменных (**example1**, **example2**, **example3**) создаются поля **_arr**, **_length** и **_capacity**.

Возращаемся к конструкторам. В типе **DynArray1D** определены 4 вида:

```
1 class DynArray1D
2 {
3 public:
4     DynArray1D() = default;
5     DynArray1D(size_t array_size);
6     DynArray1D(size_t array_size, double value);
7     DynArray1D(const DynArray1D& other);
8 };
```

1. конструктор без параметров (**example1** в предыдущем примере);
2. конструктор с одним параметром (**example2**);
3. конструктор с двумя параметрами (**example3**);
4. конструктор копий (или копирующий конструктор).

1. Конструктор без параметров: позволяет создавать объекты класса, не передавая никаких параметров

```
1 class DynArray1D
2 {
3 public:
4     DynArray1D() = default;
5 };
```

В данном случае строка 4 говорит о том, что будет использоваться *конструктор по умолчанию*. В шестой лекции говорилось о том, что такой метод просто создаёт составной объект с нужными полями. Если у полей нет значений по умолчанию, то они остаются **неинициализированными**. Иначе - присваиваются указанные значения. Как только мы объявляем хотя бы один собственный конструктор в классе, *конструктор по умолчанию* не добавляется неявно в тип. Но с помощью конструкции выше его можно вернуть.

Нужен конструктор без параметров или нет - вопрос логики и того, какую сущность пытаемся определить с помощью класса. Для динамического массива можно определить *массив нулевой длины*: просто создаём контейнер, который будет заполняться позднее. Поэтому конструктор без параметров присутствует в проектируемом классе. Значения по умолчанию для полей обсуждались ранее. Таким образом, создание объектов без параметров работает:

```
1 DynArray1D first, second, third;
```

2. Конструктор с одним параметром: позволяет создавать массив определённой длины, при этом ни один из элементов не инициализирован (не присвоено конкретное значение)

```
1  class DynArray1D
2  {
3  public:
4      // Объявление
5      DynArray1D(size_t array_size);
6  };
7
8  // Определение
9  DynArray1D::DynArray1D(size_t array_size)
10 {
11     _arr = new double[array_size];
12     _length = _capacity = array_size;
13 }
```

Теперь появилась возможность создавать объекты класса следующим образом:

```
1 DynArray1D values{10}, errors{5};
```

Вместо фигурных скобок могут быть использованы круглые.

3. Конструктор с двумя параметрами: позволяет создавать массив определённой длины, при этом каждому из элементов будет присвоено конкретное число

```
1 class DynArray1D
2 {
3 public:
4     // Объявление
5     DynArray1D(size_t array_size, double value);
6 };
```

Ссылки на определение конструктора и создание объектов с его помощью будут приведены позже.

4. Конструктор копий: позволяет создавать новый объект и копировать в него элементы из другого уже существующего объекта

```
1 class DynArray1D
2 {
3 public:
4     // Объявление
5     DynArray1D(const DynArray1D& other);
6 };
```

Эта перегрузка конструктора позволяет делать так:

```
1 DynArray1D rates;
2 // Как-нибудь заполняем массив rates
3 // Создаём копию имеющегося массива
4 DynArray1D copy_of_rates{rates};
```

Суть в следующем: по умолчанию **любому** составному типу предоставляется конструктор копий по умолчанию. Но всё, что он делает - это копирование полей существующего объекта в создаваемый. В данном примере у класса **DynArray1D** три поля, одно из них - **_arr** - указатель на динамическую память, используемую для хранения элементов массива. И при *копировании по умолчанию*, реально происходит просто копирование адреса из одного указателя в другой. И получается, что **два разных объекта** класса, ссылаются на один и тот же блок динамической памяти. Что приводит к таким проблемам, как изменение элемента в одном массиве, влияет на, вроде бы, другой массив. Это нежелательное поведение, поэтому возникает необходимость реализовать собственный конструктор копий.

Реализация конструктора копий выглядит так:

```
1  DynArray1D::DynArray1D(const DynArray1D& other)
2  {
3    _length = other._length;
4    _capacity = other._capacity;
5    _arr = new double[_capacity];
6
7    for (size_t i = 0; i < _length; i++) {
8       _arr[i] = other._arr[i];
9    }
10 }
```

Здесь **other** - это уже существующий объект класса. Смысл копирования сохранён: узнаём длину массива, выделяем аналогичный блок памяти, поэлементно копируем значения. Тем не менее, в достаточно тривиальном куске кода скрыта пара существенных нюансов.

Конструктор - это специальный вид метода; следующие особенности полностью применимы и к обычным методам класса.

- В строках **(3) - (5)** происходит обращение к полям **создаваемого** объекта. К ним внутри любого метода всегда есть доступ по имени.
- В конструктор по ссылке передаётся по ссылке **другой** объект того же типа. И в строках **(3) - (5), (8)** происходит обращение к **закрытым** полям объекта **other**. Это общее правила для методов класса: если в метод передаётся объект того же типа, то внутри метода можно обратиться к любому полю объекта, независимо от открытости/закрытости.

О **деструкторе**. Деструктор класса - может быть только один и предназначен он для освобождения каких-нибудь ресурсов (динамическая память, файлы, дескрипторы) при выходе переменной из области видимости. В описываемом типе в конструкторах используется выделение динамической памяти через **new[]**, соответственно в деструкторе требуется вернуть эту память в ОС через **delete[]**

```
1  class DynArray1D
2  {
3  public:
4      ~DynArray1D();
5  };
6
7  DynArray1D::~DynArray1D()
8  {
9      delete[] _arr;
10 }
```

Для примера,

```
1 if ( secret_condition ) {
2     DynArray1D reals{20};
3     reals[1] = 5.5;
4     reals[10] = 8.0;
5 }
6 // Здесь уже конструктор был вызван, и
7 // динамическая память под массив *reals*
8 // была возвращена операционной системе
```

Перегрузка операторов.
В C++ используется множество операторов - арифметические операции, сдвиги вправо/влево, квадратные и круглые скобки, операции сравнения - для них используются специальные символьные обозначения. И практически все операторы язык позволяет переопределять для **пользовательских** типов.
Конено же, переопределение операторов для собственного типа - это не самоцель и пользоваться этим следует в случаях, когда есть уверенность, что такое переопределение не запутает работу с объектами класса.

Перегрузка операторов.

Что можно придумать для одномерного динамического массива?

В первую очередь, это конечно же обращение к элементам массива через *квадратные скобки* и индекс внутри них. Так работают статические массивы в языке, логично, что так будет привычно будет использовать собственный класс:

```
1 DynArray1D my_arr{15};
2
3 my_arr[2] = 13.0 / 78;
4 my_arr[-3] = -45.123;
5 cout << "Третий элемент с конца: " << my_arr[-3];
```

В примере отражено, что договорились использовать как положительные, так и отрицательные индексы.

Перегрузка операторов.

В дополнении можно провести аналогию для добавления элементов в массив с выводом значений в терминал/файл: попробуем для этого использовать оператор $<<$.

```
1 DynArray1D nw_arr;
2
3 nw_arr << 5.5 << 3.4 << 7.8;
4 for (size_t i = 0; i < nw_arr.length(); i++) {
5   cout << nw_arr[i] << ' ';
6 }
7 cout << endl;
```

Здесь добавили три элемента в объект **nw_arr** и вывели их значения в терминал.

Перегрузка операторов.
Технически в C++ два способа добавить перегрузку операторов к собственному типу:

1. задать *функцию* со специальным именем, в неё передать объект(ы) класса, выполнить нужные действия;
2. определить *метод* со специальным именем внутри класса.

Независимо от способа, идентификатор определяемой сущности должен начинаться со слова **operator**, за которым сразу следует символьное обозначение оператора, а уже затем - список параметров для него. Для примера:

```
1 double operator[](int index);
2 void operator<<(double value);
3 void operator+=(double value);
```

Для закрепления, по сути перегруженные операторы - это или функции, или методы со специальным названием.

Для класса **DynArray1D** на 14 слайде объявлены три перегруженных оператора:

```
1 class DynArray1D
2 {
3 public:
4     double& operator[](int index);
5     DynArray1D& operator<<(double value);
6     void operator=(const DynArray1D& other);
7 };
```

1. обращение по индексу к элементу массива. Индекс - целое число;
2. добавление элемента в конец массива;
3. оператор присваивания: нужен для тех же целей, что и конструктор копий: избежать ситуации, когда два разных объекта ссылаются на один динамический блок памяти.

Оператор **обращения по индексу** возвращает ссылку на конкретный элемент массива. Здесь стоит заметить, что в отличии от *функций*, возврат ссылок на внутренние поля объекта из методов вполне допустим. Это следует из того, что внутри метода поля не являются локальными переменными по отношению к нему. Возврат ссылки на элемент массива в данном случае позволяет осуществлять привоение конкретных значений элементам, индекс которого передаётся в оператор (см. 36 слайд).

Оператор для **добавления элементов** в конец массива возвращает ссылку на тот объект, для которого он был вызван. Это позволяет организовывать цепочки вызово аналогичные примеру на слайде 37, строка 3.

Оператор **присваивания** для составных объектов предоставляется по умолчанию. Но, аналогично конструктору копий, всё, что он делает, это копирует поля объекта, стоящего справа от оператора =, в поля объекта, стоящего слева.

```
1 DynArray1D first{15}, second{150};
2 //...
3 second = first;
```

При работе оператора присваивания по умолчанию, в данном примере был бы потерян блок динамической памяти на 150 элементов типа **double**. Для того, чтобы избежать этого нежелательного поведения, приходиться переопределять этот оператор.
Ссылки на реализацию всех операторов будут приведены далее.

Указатель this.

В C++ в каждом методе любого класса неявно доступен специальный указатель, обозначаемый ключевым словом **this**. Этот указатель типизирован, его типом является тот класс, внутри которого он используется. Через него также можно образаться к полям, например:

```
1 size_t DynArray1D::length() const
2 {
3     // Синтаксис '->' – см. лекция 6, район 16 слайда
4     return this->_length;
5 }
```

Так можно было бы реализовать метод **length**.

Указатель **this** используется в реализациях перегруженных операторов присваивания и добавления элемента в массив.

Работа с контейнерами в C++: как использовать стандартные возможности и библиотеку для собственного типа

Итераторы в C++

Оставновимся на двух особенностях работы со статическими массивами:

```
1  double stat_array[5] = {1.1, 2.2, 6.6, -1.3, 5.3};
2
3  for (const double& elem : stat_array) {
4    cout << elem << " ";
5  }
6  cout << endl;
7
8  sort(stat_array, stat_array + 5);
9
10 for (const double& elem : stat_array) {
11   cout << elem << " ";
12 }
13 cout << endl;
```

Создаём статический массив, выводим на экран, сортируем, снова печатаем элементы. Используется специальная форма цикла **for**.

Итераторы в C++

Для класса **DynArray1D** хотелось бы получить такое же поведение (в первую очередь, использование функции сортировки из стандартной библиотеки).
Чтобы синтаксис работы с произвольными контейнерами был одинаков, в C++ стали использовать концепцию **итераторов**.
Это специальный объект (далее будет обозначаться - **it**), который позволяет пройти по всем элементам некоторого контейнера и получать доступ к ним по необходимости.
Итератор обязан поддерживать, как минимум, следующие операции:

- **++it**, **it++** - инкремент, переход к следующему элементу контейнера;
- ***it** - получение доступа к текущему элементу с помощью оператора разыменования;
- **it1 == it2, it1 != it3** - операторы проверки на равенство и неравенство.

Итераторы в C++

И все операции с предыдущего слайда прям очень похожи на работу с указателями в C++. В этом заключается одна из идей использования концепции итераторов в C++: как был не была сложна структура контейнера, работа с перебором элементов сводится к инкрементированию указатель-подобного объекта, а доступ к значению - через оператор-разыменования.
Итераторы реализуются как специальные классы внутри библиотек. В C++ они ещё и по типу различаются: есть одно-направленные, двух-направленные, итераторы произвольного доступа и другие.
Но для демонстрации итератора для класса **DynArray1D** попробуем обойтись указателями, которые уже присутствуют в нём через поле **_arr**.

Итераторы в C++

Для работы со стандартной библиотекой алгоритмов, контейнер как минимум должен предоставлять два итератора:

1. итератор на **первый** элемент контейнера;
2. итератор на специальное значение, означающее окончание перебора элементов.

Для универсального синтаксиса, было принято соглашение, что первый итератор будет получен с помощью функции или метода **begin**, второй - с помощью **end**. Хотя выбор кажется аналогичным перегрузке операторов (что использовать - функции или методы), стандартная библиотека C++ придерживается соглашений, что будут реализованы как функции **begin** и **end**, так и аналогиные методы класса.

Итераторы в C++

Для **DynArray1D** приходим к тому, что добавляются следующие конструкции:

```
1  class DynArray1D
2  {
3  public:
4    // Методы для предоставления итераторов
5    double* begin();
6    const double* begin() const;
7
8    double* end();
9    const double* end() const;
10 };
11
12 double* begin(DynArray1D& obj);
13 double* end(DynArray1D& obj);
```

Методы **begin** и **end** продублированы для того, чтобы специальная форма цикла **for** могла работать и с обычными объектами, и с константными.

В качестве типа итератора - выбран *указатель на double*.

Возможности классов в C++

Готовый код доступен тут:
https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/examples/2018_2019/

Интерес представляют файлы **lecture_9_10_1903_01to15_1st.cpp**, **lecture_9_10_1903_01to15_2nd.cpp** и **lecture_9_10_dyn_array_final.cpp**.

Первые два - это эволюция, третий - итоговый результат. Вопросы по реализации - приветствуются.

Шаблонное (обобщённое) программирование в C++ или как заставить компилятор писать код вместо себя

Шаблоны (templates) - механизм языка C++, позволяющий переложить конкретную реализацию функций / классов для разичных типов данных на компилятор.

И не только реализацию функций/классов, но и провести часть вычислений, проверок типов на различные условия, и получение, по необходимости, информации, исходя из переданных в момент компиляции объектов.

На первое - *шаблонные функции*

Шаблонные функции

Для начала надо понять проблематику, а именно, зачем появилась необходимость перекладывать реализацию конкретных функций на компилятор. Рассмотрим обмен значениями двух переменных для типов **int**, **double** и **string**:

```
1  void my_swap(int& lhs, int& rhs)
2  {
3    int tmp = lhs;
4    lhs = rhs;
5    rhs = tmp;
6  }
7
8  void my_swap(double& lhs, double& rhs)
9  {
10   double tmp = lhs;
11   lhs = rhs;
12   rhs = tmp;
13 }
14
15 void my_swap(std::string& lhs, std::string& rhs)
16 {
17   std::string tmp = lhs;
18   lhs = rhs;
19   rhs = tmp;
20 }
```

Шаблонные функции

Что видно из кода на предыдущем слайде:

- объявлены три функции; они перегружены для соответствующих типов;
- все они выполняют одни и теже *действия* (в примере - три присваивания), для переменных **разных типов**;
- все они принимают два параметра нужного типа данных;
- строки «**3, 4, 5**», «**10, 11, 12**» и «**17, 18, 19**» - ничем не отличаются друг от друга за исключением **типа временной переменной**

В итоге, три функции можно обобщить псевдокодом:

```
1 void my_swap(Type& lhs, Type& rhs)
2 {
3   Type tmp = lhs;
4   lhs = rhs;
5   rhs = tmp;
6 }
```

и найти того, кто будет создавать функции для нужных нам типов.

Шаблонные функции

Как следует из заглавного слайда данной темы, создавать функции нам будет компилятор.

Суть шаблонного программирования в C++

Мы определяем общие **действия**, которые должны быть сделаны для некоторых объектов, а вот **могут ли эти действия быть сделаны** для объектов конкретных типов - проверяет уже компилятор.

Общий синтаксис объявления шаблонной функции (псевдокод):

```
1 template <typename Type1, [typename Type2, ...]>
2 return_value func_name( arguments )
3 {
4   func_body;
5 }
```

- функция начинается с ключевого слова **template** и **блока** в треугольных скобках;
- в **блоке** указываются **типы как параметры**, для этого используется слово **typename** и псевдоним для типа;

Шаблонные функции

- далее следует обычное определение функии. Только теперь, в аргументах и теле функции можно создавать переменные перечисленных в **блоке** типов;
- количество типов для шаблона - можно считать неограниченым (определяется задачей). Квадратные скобки в первой строке говорят о том, что второй и последующие параметры шаблонной функции - **опциональны**;
- ранее вместо ключевого слова **typename** использовалось слово **class**. Его и сейчас можно использовать, но всё-таки в современном C++ рекомендуется выбирать только **typename**.

Шаблонные функции

И теперь реализуем шаблонную функцию обмена значениями двух переменных:

```
1 template <typename Type>
2 void my_swap(Type& lhs, Type& rhs)
3 {
4   Type tmp = lhs;
5   lhs = rhs;
6   rhs = tmp;
7 }
```

Одна шаблонная функция, которой, для превращения в реальную функцию в программе, достаточно знать два аспекта:

1. **Тип** передаваемых аргументов (аргументы должны быть одинакого типа).
2. Возможность выполнения **операции присваивания** для объектов этого типа.

Второй пункт - ключевой: если действия (вызов операторов и/или методов), описанные в теле шаблонной функции, не определены для объектов - произойдёт ошибка компиляции.

Шаблонные функции

Использование шаблонной функции:

```
1  int i1 = 5, i2 = 10;
2  my_swap(i1, i2);
3  cout << i1 << ", " << i1 << '\n';
4
5  double r1 = 1.5, r2 = 8.8;
6  my_swap(r1, r2);
7  cout << r1 << ", " << r1 << '\n';
8
9  std::string s1 = "str1", s2 = "2str";
10 my_swap(s1, s2);
11 cout << s1 << ", " << s1 << '\n';
```

Кратко: компилятор видит вызовы шаблонной функции, видит аргументы и их типы, и создаёт реализацию конкретной функции, если действия в теле функции подходят для аргументов.
Тип можно указать явно с помощью следующего синтаксиса:

```
12
13 char sym1 = 'e', sym2 = 'w';
14 my_swap<char>(sym1, sym2);
15 cout << sym1 << ", " << sym1 << '\n';
```

Шаблонные функции

Использованная простая реализация функции обмена будет работать и для пользовательских классов

```
1 class Point
2 {
3 public:
4   double x, y, z;
5 };
6
7 Point p1 = {3.4, 5.5, 1.2}, p1 = {-1.1, -2.2, -3.3}
8 my_swap(p1, p2);
9 cout << p1.x << ", " << p2.y << '\n';
```

Всё работает из-за того, что классу предоставляется **оператор присваивания** по умолчанию.

Шаблонные функции

Но оператор присваивания можно и запретить для класса. Демо:

```
1  class Point
2  {
3  public:
4    double x, y, z;
5
6    Point& operator=(const Point&) = delete;
7  };
8
9  // Пример не скомпилируется
10 Point p1 = {3.4, 5.5, 1.2}, p1 = {-1.1, -2.2, -3.3}
11 my_swap(p1, p2);
12 cout << p1.x << ", " << p2.y << '\n';
```

Будет ошибка компиляции с сообщением об невозможности превратить *шаблонную функцию* в *конкретную* для типа, у которого отсутствует *оператор присваивания*.

Шаблонные функции

Кроме того, компилятору можно дать команду на создание конкретной версии шаблонной функции. Это называется **явным инстанцированием**. Пример:

```
1 template<>
2 void my_swap(size_t&, size_t&);
```

Функция обмена для типа **size_t** будет создана, хотя ни разу не вызвана для конкретных переменных.

Шаблонные функции

Параметры шаблона могут быть не только **псевдонимами** для типа, но и значениями конкретных типов. Единственное условие на значения - они должны быть известны на этапе компиляции:

```
1  template<typename Type, size_t how_many>
2  void repeat_to_cout(const Type& obj)
3  {
4    for (size_t i = 0; i < how_many; ++i) {
5      std::cout << obj << "\n";
6    }
7  }
8
9  int i = 18;
10 repeat_to_cout<int, 20>(i); // Всё хорошо
11 // repeat_to_cout<int, i>(i); // Не получится
```

Пример не особо полезен, но показывает использование нетипового параметра шаблона.

Шаблонные функции

Более интересный пример - автоматический вывод размера *массива в стиле C* при передаче в функцию. Напишем функцию, которая будет складывать все элементы массива и возвращать сумму.

```
1  template<typename Type, size_t N>
2  Type reduce_sum(Type (&array)[N])
3  {
4    Type sum{};
5    for (size_t i = 0; i < N; ++i) {
6      sum += array[i];
7    }
8    return sum;
9  }
10
11 int arr1[] = {1, 4, 5, 6, 7, 8, 9};
12 double arr2[] = {5.5, 4.4, 3.3, 2.2, 1.1, 0.999};
13
14 cout << "sum of arr1 is " << reduce_sum(arr1) << "\n";
15 cout << "sum of arr2 is " << reduce_sum(arr2) << "\n";
```

Хинт: работает за счёт передачи массива в функцию по ссылке.

Шаблонные функции

Или функцию, которая возращает размер статитеческого *массива в стиле C*.

```
1  template<typename Type, size_t N>
2  size_t array_size(Type (&arr)[N])
3  {
4    return N;
5  }
6
7  int arr1[] = {1, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17};
8  double arr2[] = {5.5, 4.4, 3.3, 2.2, 1.1, 0.999};
9
10 cout << "size of arr1 is " << array_size(arr1) << "\n";
11 cout << "size of arr2 is " << array_size(arr2) << "\n";
```

Шаблонные классы

И, конечно же, C++ позволяет создавать шаблонные классы.
Принцип - тот же, что и с функциями: мы можем написать «прототип» класса, который будет делать одинаковые **действия** для **объектов** разных типов.
Для начала, класс, который хранит два значения **разных** типов.

```
1  template<typename T1, typename T2>
2  class Pair
3  {
4  public:
5    T1 first;
6    T2 second;
7  }
8
9  Pair<int, double> p1 = {5, 789.123};
10 Pair<char, std::string> p2 = {'f', "текст"};
11
12 cout << p2.second << '\n';
```

Стоит заметить, что подобный класс есть в стандартной библиотеке C++; называется **pair** и обитает в заголовочном файле **<utility>**.