Структуры и перечисления

Михаил Марков
C++ - Разработчик

Проверка связи

Поставьте “+”, если меня видно и слышно

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включен звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Михаил Марков

О спикере:
C++-разработчик, фрилансер

- Разработка алгоритма для релевантной выдачи объявлений.
- Разработка эмуляторов оборудования

Вспоминаем прошлое занятие

Вопрос: что такое IDE?

Вспоминаем прошлое занятие

Вопрос: что такое IDE?

Ответ: IDE - это сокращение от Integrated Development Environment, в переводе - интегрированная среда разработки

Вспоминаем прошлое занятие

Вопрос: что происходит перед стартом нового кода?

Вспоминаем прошлое занятие

Вопрос: что происходит перед стартом нового кода?

Ответ: перед стартом нового кода происходит процесс его сборки - компиляция

Вспоминаем прошлое занятие

Вопрос: как можно отладить программу?

Вспоминаем прошлое занятие

Вопрос: как можно отладить программу?

Ответ: с помощью точек останова - брейкпоинтов

Вспоминаем прошлое занятие

Вопрос: как попасть в директорию, в которой содержатся исходные файлы?

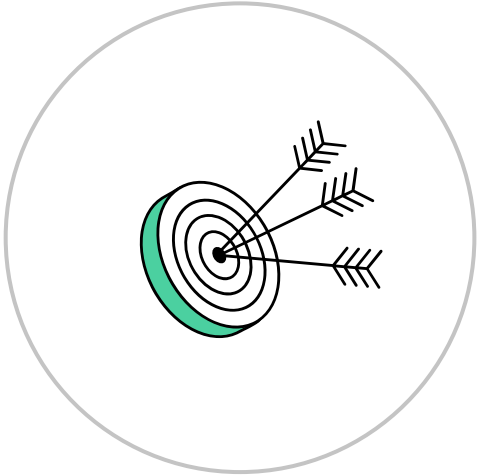
Вспоминаем прошлое занятие

Вопрос: как попасть в директорию, в которой содержатся исходные файлы?

Ответ: щелчок правой кнопкой мыши по значку проекта, в контекстном меню выбрать “Открыть папку в проводнике”

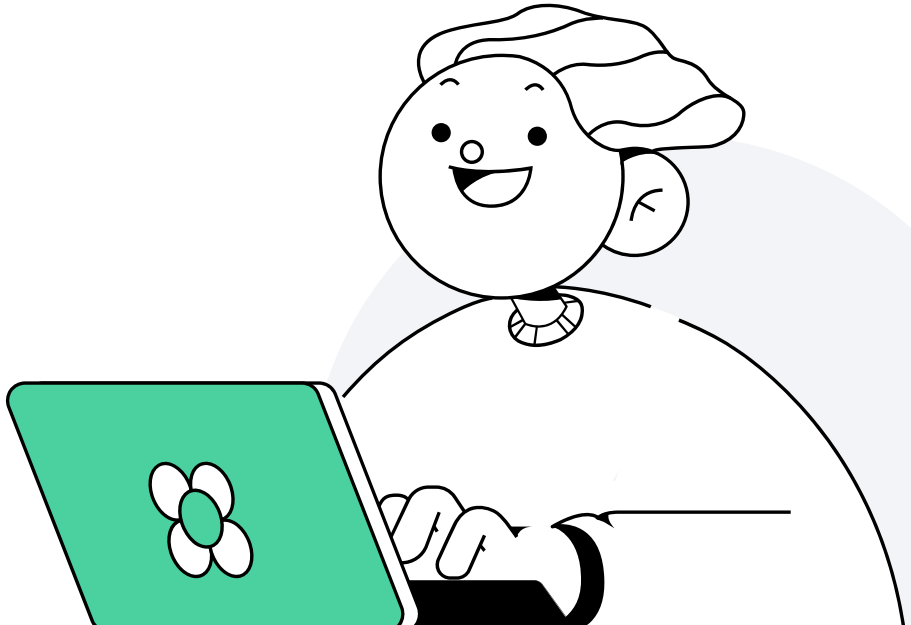
Цели занятия

- Разберёмся, что такое перечисления
- Познакомимся с использованием перечислений в программах
- Узнаем, что такое структуры
- Выясним, как работать со структурами



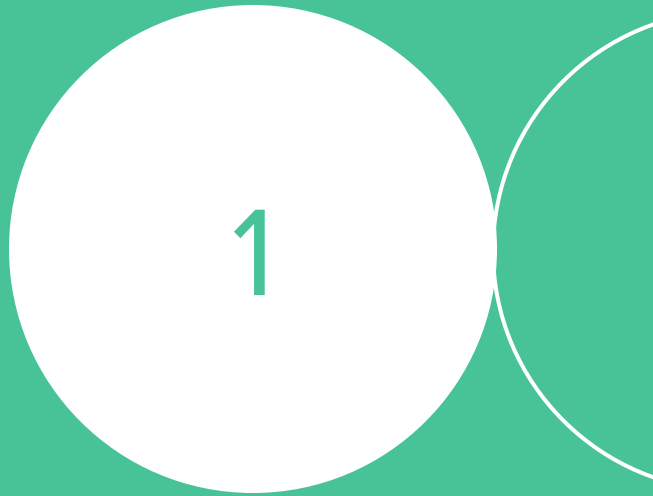
План занятия

1. Перечисления
2. Структуры
3. Как работать со структурами
4. Итоги
5. Домашнее задание



*Нажми на нужный раздел для перехода

Перечисления



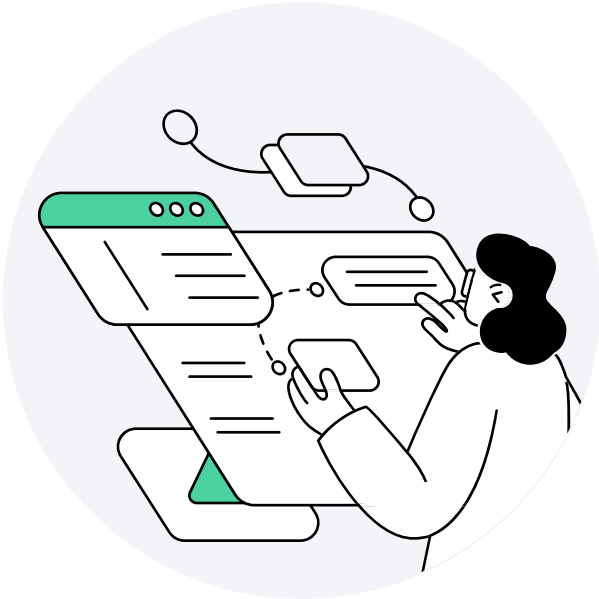
В чём задача?

Иногда в программах мы сталкиваемся с данными с ограниченным набором значений. Например, времена года - их всего 4, ни больше, ни меньше.

Или месяцы - их всего 12

Именно для таких данных были придуманы перечисления

Перечисление - это **связанный набор именованных констант**



Синтаксис

Перечисление объявляется с помощью ключевого слова **enum**. Синтаксис объявления выглядит следующим образом:

```
enum <название>
{
    значение1,
    значение2,
    ...
    значениеN
};
```

Название - это уникальный идентификатор перечисления.
Значения - это те значения, которые может принимать переменная, имеющая тип этого перечисления

Синтаксис. Пример

Перечисления нужно объявлять за пределами функций. Для того, чтобы использовать значение перечисления, нужно использовать одно из значений. Пример:

```
enum seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = spring;
}
```

Синтаксис. Квалификатор перечисления

Также можно (и нужно!) использовать указание типа перечисления при использовании значения. Это делается с помощью символов **::**
Пример:

```
enum seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = seasons::spring;
}
```

Синтаксис. Квалификатор class

Тип перечисления всегда лучше указывать, так как это значительно повышает читаемость кода. Для этого даже ввели квалификатор **class** - он указывается перед названием перечисления. При его использовании не указывать тип перечисления у значений **нельзя**! Пример:

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = seasons::spring;
}
```

Что лежит внутри?

Давайте посмотрим, что лежит внутри переменной перечисления. Как вы думаете, как нам это сделать?

Что лежит внутри?

Правильно! Нужно вывести значение на консоль

```
enum seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = seasons::spring;
    std::cout << season << std::endl; // 0
}
```

Что лежит внутри?

Вот это да! Давайте выведем все значения:

```
enum seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    std::cout << seasons::spring << std::endl; // 0
    std::cout << seasons::summer << std::endl; // 1
    std::cout << seasons::autumn << std::endl; // 2
    std::cout << seasons::winter << std::endl; // 3
}
```

Давайте разбираться

Дело в том, что основой перечисления является тип int. Каждое значение в перечислении - это какое-то целое число (поэтому оно и называется **перечисление**).

По умолчанию первому значению в перечислении соответствует 0, каждое следующее значение увеличивается на единицу (что мы и видели)

Давайте разбираться

Тем не менее, тип перечисления и тип int - это не одно и то же

Для **обычного** перечисления (без квалификатора class) доступно неявное преобразование в тип int - поэтому значение перечисления выводится как число

Но давайте попробуем вывести на консоль перечисление с квалификатором class

Пытаемся вывести enum class

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = seasons::summer;
    std::cout << season << std::endl; //  Ошибка!
}
```

Как вывести enum class?

Такое поведение предусмотрено для улучшения безопасности типов - например, чтобы программист случайно не передал значение перечисления в функцию, которая должна работать с целыми числами. Поэтому лучше всегда работать с enum class - это безопаснее

Тем не менее, иногда такое поведение может быть желательно

Как вы думаете

как превратить значение перечисления в целое число, которое лежит в его основе?

Напишите в чат

Выводим enum class?

Правильно! Нужно использовать приведение типов (static_cast)

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = seasons::summer;
    std::cout << static_cast<int>(season) << std::endl; // 1
}
```

Обратное преобразование

Иногда нам нужно число превратить в соответствующе значение перечисления. Попробуем:

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = 1;
    std::cout << static_cast<int>(season); // Ошибка!
}
```

Обратное преобразование

А если без квалификатора class?

```
enum seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = 1;
    std::cout << static_cast<int>(season); // Ошибка!
}
```

Обратное преобразование

Преобразование из типа int в тип перечисления может быть только явным:

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons season = static_cast<seasons>(1);
    std::cout << static_cast<int>(season); // 1
}
```

Пользовательские значения

При объявлении перечисления вы можете задать некоторым (или всем) значениям перечисления свои значения типа int

```
enum seasons
{
    spring = 1,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    std::cout << seasons::spring << std::endl; // 1
    std::cout << seasons::summer << std::endl; // 2
    std::cout << seasons::autumn << std::endl; // 3
    std::cout << seasons::winter << std::endl; // 4
}
```

Пользовательские значения

Можно задать для произвольных значений произвольные числа

```
enum seasons
{
    spring,
    summer = 4,
    autumn = 8,
    winter
};
int main(int argc, char** argv)
{
    std::cout << seasons::spring << std::endl; // 0
    std::cout << seasons::summer << std::endl; // 4
    std::cout << seasons::autumn << std::endl; // 8
    std::cout << seasons::winter << std::endl; // 9
}
```

Пользовательские значения

Они даже не обязаны следовать по возрастанию

```
enum seasons
{
    spring,
    summer = 4,
    autumn = -2,
    winter
};
int main(int argc, char** argv)
{
    std::cout << seasons::spring << std::endl; // 0
    std::cout << seasons::summer << std::endl; // 4
    std::cout << seasons::autumn << std::endl; // -2
    std::cout << seasons::winter << std::endl; // -1
}
```

Пользовательские значения

Они даже могут пересекаться. Но так делать в целом не стоит, можно запутаться:

```
enum seasons
{
    spring,
    summer,
    autumn = 0,
    winter
};
int main(int argc, char** argv)
{
    std::cout << seasons::spring << std::endl; // 0
    std::cout << seasons::summer << std::endl; // 1
    std::cout << seasons::autumn << std::endl; // 0
    std::cout << seasons::winter << std::endl; // 1
}
```

Использование перечислений

Перечисления обычно используются как набор связанных именованных констант.
И чаще всего они используются для сравнения

```
enum class seasons
{
    spring,
    summer,
    autumn,
    winter
};
int main(int argc, char** argv)
{
    seasons s = seasons::summer;
    if (s == seasons::summer)
        std::cout << “На дворе лето!” << std::endl;
}
```

Перерыв

Структуры

2

В чём задача?

Довольно часто нам хочется, чтобы некоторые данные хранились **вместе**

Например, если наша программа хранит данные о сотрудниках, то нам неудобно будет держать отдельно массив фамилий, массив имён и массив возрастов

Тогда данные человека будут связаны только одинаковыми индексами в массиве, и эта связь может легко нарушиться

Как её решить?

Хотелось бы, чтобы был какой-нибудь тип данных, в котором можно было бы сохранить имя, фамилию и возраст сотрудника - и держать массив переменных такого типа данных

Проблема в том, что таких типов можно придумать бесконечное количество, и заранее все создать невозможно

Вместо этого C++ предлагает вам инструмент для **создания** новых типов данных - **структуры** и **классы** (сегодня мы рассмотрим структуры)

Структура

это пользовательский тип данных,
объединяющий разнородные данные

Структура

Данные, которые содержит в себе структура, называются **полями** - каждое поле - это как переменная, только существующая в рамках структуры

У структуры есть определение - в нём описывается, из каких полей состоит структура

И есть **экземпляры** структуры, построенные с помощью определения

Синтаксис

Определение структуры имеет следующий синтаксис:

```
struct <название структуры>
{
        <тип данных поля 1> <название поля 1>;
        <тип данных поля 2> <название поля 2>;
        ...
        <тип данных поля N> <название поля N>;
};
```

Пример

Создадим структуру Person, которая будет содержать в себе фамилию, имя и возраст сотрудника. Как вы думаете, какие типы данных должны быть у этих полей?

Пример

Правильно, фамилия и имя - это строки (std::string), а возраст - это int

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
```

Как работать со структурами

3

Экземпляры структуры

Сама структура - это новый, пользовательский (то есть созданный пользователем языка) тип данных. Как std::string, int, float и другие. Но без переменных этого типа от него мало пользы

Переменные, которые имеют тип данных созданной вами структуры, называются экземплярами этой структуры. Экземпляр структуры создаётся точно так же, как переменная любого другого типа

Экземпляры структуры

Нужно просто указать тип структуры и имя переменной - будет создан экземпляр структуры

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
int main(int argc, char** argv)
{
     person p; // теперь в переменной p хранится экземпляр
                  структуры person
}
```

Инициализация экземпляра структуры

При создании экземпляра структуры его поля будут содержать значения по умолчанию. Чтобы сразу заполнить поля экземпляра структуры нужными значениями, можно использовать синтаксис инициализации структуры

Для инициализации структуры после объявления переменной структуры нужно написать знак присвоения (=) и список инициализации (в фигурных скобках). В списке инициализации нужно указать значения для всех полей в том порядке, в котором они определены в вашей структуре

Инициализация экземпляра структуры

Нужно просто указать тип структуры и имя переменной - будет создан экземпляр структуры

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
int main(int argc, char** argv)
{
     person p = {“Дмитрий”, “Иванов”, 20};
}
```

Доступ к полям структуры

Однако мы создавали структуру для того, чтобы иметь возможность работать с данными, которые заключены внутри неё. Чтобы получить доступ к полям структуры - нужно поставить точку после экземпляра структуры и указать название поля

В поля можно записывать новую информацию - для этого после доступа к полю нужно поставить оператор присвоения (знак =) и указать информация, которую вы хотите внести в поле

Можно также прочитать информацию, хранящуюся в поле - для этого нужно просто написать конструкцию доступа к полю и дальше что-то с этим сделать - например, запомнить в переменную или вывести на консоль

Доступ к полям структуры

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
int main(int argc, char** argv)
{
    person p;
    p.first_name = “Дмитрий”;
    p.last_name = “Иванов”;
    p.age = 20;
    std::cout << p.first_name << “ “ << p.last_name << “ “ <<
p.age; // Дмитрий Иванов 20
}
```

Доступ к полям структуры через указатель

Переменная, содержащая экземпляр структуры, ведёт себя так же, как другие переменные, и тоже хранится в памяти - а значит, имеет адрес. К переменной структуры можно применить оператор взятия адреса (&) и получить указатель на структуру

Имея указатель на структуру, как можно получить доступ к полям экземпляра структуры, на который указывает указатель? Первый вариант - разыменовать структуру и воспользоваться оператором . (точка)

Доступ к полям структуры через указатель

Второй вариант более удобен - для доступа к полям структуры через указатель существует специальный оператор -> (стрелочка)

Иными словами - если у вас есть указатель на структуру (например, person* p_pers), то доступ к полям удобнее осуществлять через ->, чем через * и . (p_pers->age удобнее, чем (*p_pers).age)

Доступ к полям структуры через указатель

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
int main(int argc, char** argv)
{
    person pers = {“Василий”, “Чапаев”, 20};
    person* p_pers = &pers;
    (*p_pers).first_name = “Дмитрий”; // доступ с помощью * и .
    p_pers->last_name = “Иванов”; // доступ с помощью ->
    std::cout << pers.first_name << “ “ << pers.last_name << “ “
<< pers.age; // Дмитрий Иванов 20
}
```

Работа со структурами в функциях

Вы можете передавать экземпляры структуры в функции и возвращать их из функции

Однако, будьте внимательны!

Если вы передаёте экземпляр структуры в качестве аргумента, то внутри функции у вас создаётся **новый** экземпляр структуры, значения полей которого копируются из полей исходного экземпляра. Поэтому если ваша структура содержит простые типы данных (например, int), то изменение значения такого поля в функции не приведёт к изменению значения этого же поля в исходном экземпляре структуры

Работа со структурами в функциях

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
void increment_age(person p)
{ p.age++; }
int main(int argc, char** argv)
{
    person pers;
    pers.age = 20;
    increment_age(pers);
    std::cout << pers.age; // 20
}
```

Работа со структурами в функциях

как можно изменить значение поля age исходного экземпляра структуры внутри функции?

Напишите в чат

Работа со структурами в функциях

Чтобы повлиять на значение поля в исходном экземпляре структуры внутри функции, исходный экземпляр нужно передать с помощью указателя или с помощью ссылки - ссылка на структуру выглядит так же, как ссылка на любую другую переменную

Работа со структурами в функциях

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
void increment_age(person& p) // ссылка
{ p.age++; }
int main(int argc, char** argv)
{
    person pers;
    pers.age = 20;
    increment_age(pers);
    std::cout << pers.age; // 21
}
```

Работа со структурами в функциях

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
void increment_age(person* p) // указатель
{ p->age++; }
int main(int argc, char** argv)
{
    person pers;
    pers.age = 20;
    increment_age(&pers);
    std::cout << pers.age; // 21
}
```

Массивы
экземпляров структур

Можно даже создавать массивы из экземпляров структур

Как создать массив для 20 экземпляров структур person?

Напишите в чат

Массивы экземпляров структур

Нужно использовать синтаксис new[]

```
struct person
{
    std::string first_name;
    std::string last_name;
    int age;
};
int main(int argc, char** argv)
{
    person* person_array = new person[20];
    person_array[0].first_name = “Анатолий”;
    person_array[0].last_name = “Петров”;
    person_array[0].age = 20;
    std::cout << person_array[0].first_name << “ “ <<
person_array[0].last_name << “ “ << person_array[0].age; // Анатолий
Петров 20
    delete[] person_array;
}
```

Итоги

Итоги занятия

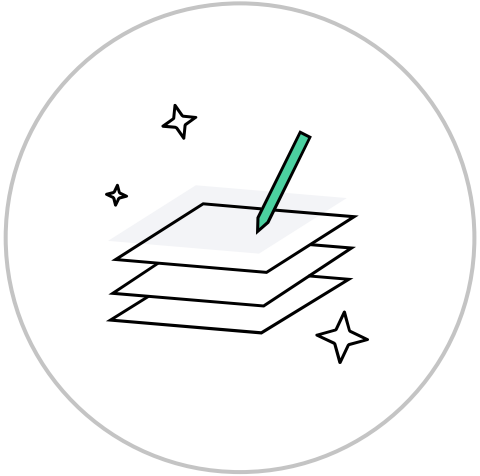
Сегодня мы

1. Разобрались, что такое перечисления
2. Познакомились с использованием перечислений в программах
3. Узнали, что такое структуры
4. Выяснили, как работать со структурами

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Перечисления
- Структуры

Задавайте вопросы и пишите отзыв о лекции

Михаил Марков
C++ - разработчик