Лекция XI

Статические поля и методы в составных типах C++

Статические поля и методы

В примерах с последних лекций первого семестра 18-19 уч. года приводился пример для демонстрации *статических* локальных переменных:

```
1 double rand_a_b(double a, double b)
2 {
3   static mt19937_64 gnr{ time(nullptr) };
4   static const size_t max_gnr = gnr.max();
5
6   return a + (b - a) * ( double(gnr()) / max_gnr );
7 }
```

Здесь определена функция для создания псевдослучайного числа в интервале $[a;\ b]$ при использовании ГПСЧ из стандартной библиотеки C++ (<**random**>). Статическими являются переменные **gnr** и **max_gnr**, определённые с помощью ключевого слова **static**.

Код доступен тут:
https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/examples/2018_2019/lecture_8_fp_181218.cpp

Статические поля и методы

Статическая локальная переменная, по сути, имеет два отличия:

- её **время жизни** можно считать равным времени работы программы (до тех пор, пока из функции **main** не была вызвана инструкция **return**);
- она создаётся только один раз (строки **3, 4** в примере с предыдущего слайда будут выполнены единожды)

В противоположность, обычные локальные переменные создаются при каждом вызове функции и удаляются из памяти программы по её завершению.

Статические поля и методы

По аналогии с статическими локальными переменными, составные типы C++ могут иметь статические поля и методы. Их главное отличия от обычных полей/методов: **они не принадлежат объектам** класса (структуры). Определяются такие поля тоже с помощью ключевого слова **static**. Статическое поле - также создаётся только один раз и имеет время жизни, равное времени работы программы.

Далее рассмотрим пример статического класса на задачке, аналогичной третьему слайду: хотим иметь один ГПСЧ для использования в любой части кода, но без использования глобальных переменных.

Статические поля и методы

Статические поля (да и методы) точно также имеют модификатор доступа - **public**/**private**. Заведён два закрытых поля: под генератор случайных чисел т максимальное целое число, которое он может выдасть:

```
1 class Rand
2 {
3 private:
4     static mt19937_64 _gnr;
5     static size_t _gnr_max;
6 };
```

Статические поля и методы

Раз статические поля создаются только один раз, их надо инициализировать чем-нибудь полезным. Для ГПСЧ самое простое - задание зерна, зависящего от времени. По аналогии с 3-им слайдом, первый вариант такой:

```
1 class Rand
2 {
3 private:
4   static mt19937_64 _gnr{ time(nullptr) };
5   static size_t _gnr_max;
6 };
```

Но он не работает.

Когда работает?

Инициализировать **статические** поля внутри класса можно только для **целочисленных** типов (**int, size_t, long, char** и прочие).

Статические поля и методы

На практике это означает:

```
1  class Rand
2  {
3  private:
4    // Не компилируется следующая строка:
5    //static mt19937_64 _gnr{ time(nullptr) };
6    static size_t _gnr_max;
7
8    // А тут, компилятор не ругается:
9    static size_t _basic_seed = 455;
10 };
```

Статические поля и методы

C++ позволяет присваивать значения для статических полей нецелочисленных типов вне определения класса:

```
1 class Rand
2 {
3 private:
4   static mt19937_64 _gnr;
5   static size_t _gnr_max;
6 };
7
8 mt19937_64 Rand::_gnr{ time(nullptr) };
9 size_t Rand::_gnr_max = _gnr.max();
```

В 8 и 9 строках слово **static** уже не нужно. Плюс повторим, эти строки выполнятся только один раз. Кроме того, можно обратить внимание на то, что при инициализации статических полей класса можно вызывать любые функции (попробуйте вызвать конкретную функцию в глобальной области видимости).

Статические поля и методы

Доступ к статическим полям можно получить либо через название класса (для **открытых** статических полей), либо из любого (статического и обычного) метода данного класса.

Пример открытых полей:

```
1  class Rand
2  {
3  public:
4    static mt19937_64 _gnr;
5    static size_t _gnr_max;
6  };
7
8  mt19937_64 Rand::_gnr{ size_t( time(nullptr) ) };
9  size_t Rand::_gnr_max = _gnr.max();
10
11 // Используем статическое поле напрямую
12 // через имя класса
13 cout << Rand::_gnr << endl;
14 cout << "Максимальное случайное число: "
15      << Rand::_gnr_max << endl;
```

Статические поля и методы

Но **открытые** статические поля для данной обёртки на ГПСЧ неинтересны. Лучше добавить статические методы:

```
1  class Rand
2  {
3  public:
4    static double get_0_1();
5    static double get_0_PI();
6    static double get_0_PI_2();
7    static double get_a_b(double a, double b);
8
9    static void seed();
10   static void seed(size_t new_seed);
11
12 private:
13   static mt19937_64 _gnr;
14   static size_t _gnr_max;
15 };
16
17 mt19937_64 Rand::_gnr{ size_t( time(nullptr) ) };
18 size_t Rand::_gnr_max = _gnr.max();
```

Статические поля и методы

Статические методы с предыдущего слайда задуманы для:

- **строка 4**: создание случайного числа в интервале $[0.0; 1.0]$;
- **строка 5**: интервал $[0.0; \pi]$;
- **строка 6**: интервал $[0.0; \pi/2]$;
- **строка 7**: интервал $[a; b]$;
- **строка 9**: задать новое случайное зерно ГСПЧ;
- **строка 10**: задать конкретное зерно ГСПЧ.

Статические поля и методы

Правила определения статических методов не отличаются от обычных: тело метода либо пишется внутри определения класса, либо вне его.

```
1  double Rand::get_0_1()
2  { return double( _gnr() ) / _gnr_max; }
3
4  double Rand::get_0_PI()
5  {
6    #ifndef M_PI
7      #define M_PI 3.14159265358979323846
8    #endif
9    return M_PI * get_0_1();
10 }
11
12 void Rand::seed()
13 { _gnr.seed( size_t(time(nullptr)) ); }
14
15 void Rand::seed(size_t new_seed)
16 { _gnr.seed( new_seed ); }
```

Остальные методы реализуются аналогично.

Статические поля и методы

Вызываются открытые статические методы с помощью имени класса:

```
1 cout << "Примеры случайных чисел:\n";
2 cout << Rand::get_0_PI() << endl;
3 cout << Rand::get_0_PI_2() << endl;
4 cout << Rand::get_a_b(-3.0; 14.25) << endl;
5
6 Rand::seed(555);
7 // При каждом запуске вывод будет одинаков:
8 cout << Rand::get_o_PI() << endl;
```

По сути, для открытых статических методов, имя класса аналогично определнию нового простарнства имён. Разве что, с помощью **using** имя класса никак не убрать.

Статические поля и методы

В данной задаче (обёртка над ГПСЧ) с помощью статических полей и методов достигли:

- возможность получать случайное число в любом месте программы, без передачи конкретного объекта-генератора;
- сам объект-ГПСЧ не является глобальной переменной, а скрыт в статическом поле;
- методы для получения случайных чисел объеденены по смыслу приставкой **Rand::**;
- в любой момент в программе, использующей подобный класс, можно менять зёрна генератора по своему усмотрению.

Естественно, данная реализация не покрывает случая, когда в программе принудительно должно присутствовать более одного ГПСЧ.

Статические поля и методы

И покажем, как статические поля/методы взаимодействуют с обычными. Пусть в класс **Rand** добавлена пара методов:

```
1  class Rand
2  {
3  public:
4    ...
5    double tricky_rnd()
6    { return -2.5 + (2.0 * _gnr()) / _gnr_max; }
7
8    double want_0_PI()
9    { return get_0_PI(); }
10   ...
11 };
12
13 Rand r1;
14 cout << r1.tricky_rnd() << endl;
15 cout << r1.want_0_PI() << endl;
16 // Ошибка компиляции: статический метод не может
17 // быть вызван через объект r1
18 // cout << r1.get_0_PI() << endl;
```

Статические поля и методы

На предыдущем слайде просто демонстрация вызова статических полей/методов из нестатических методов. Практический смысл искать не рекомендуется.

Классы и динамические объекты

Динамические объекты в C++

- Все объекты в C++ могут быть динамическими.
- Главным свойством динамического объекта является его **время жизни** в программе: объект существует до тех пор, пока не будет удалён вручную (либо программа не завершится).
- Создание и удаление динамических объектов происходит с помощью операторов **new** и **delete** соответственно (и их версий для массивов объектов).
- Действия с динамическими объектами выполняются с использованием **указателей** на нужный тип (класс, структуру).

Динамические объекты в C++

Для повторения: пример использования **new** и **delete** для **фундаментальных** типов

```
1 double *p_real;
2 p_real = new double;
3
4 *p_real = 145.897;
5 std::cout << *p_real << '\n';
6
7 delete p_real;
```

Короткое приключение одинокого указателя-на-**double**. По сути:

строка 2: выделяем **одну** динамическую переменную типа **double**. Её адрес записываем в **p_real**;

строка 4: используя оператор разыменования (*****) в дин. переменную записываем значение **145.897**;

строка 7: удаляем динамическую переменную вручную.

Для подобных типов больше значения имеет создание динамических массивов, о них ещё вспомним позже.

Динамические объекты в C++

В предыдущем примере технически оператор **new** во **второй строке** отработал следующим образом:

1. посмотрел на запрашиваемый тип (в данном примере - **double**);
2. узнал, сколько нужно памяти под **одну** переменную этого типа (язык C++ всегда в курсе);
3. запросил требуемое количество памяти у ОС;
4. если выделение прошло успешно, вернул адрес выделенного блока (который сохранился в указателе **p_real**).

В дополнении к общей схеме при динамическом создании одной переменной **фундаментальных** типов, её можно присвоить конкретное значение. Делается это так:

```
2 p_real = new double{};
3 // создаётся динамическая перменная со значением 0.0
```

или так

```
2 p_real = new double{-1.4};
3 // создаётся динамическая перменная со значением -1.4
```

Динамические объекты в C++

Общая схема работы **new** для **динамических объектов** (переменных пользовательских типов данных) немного изменится:

1. оператор смотрит на запрашиваемый тип;
2. узнаёт, сколько нужно памяти под **одну** переменную этого типа (для пользовательских типов - сумма блоков памяти под каждое поле);
3. запрашивает требуемое количество памяти у ОС;
4. если выделение прошло успешно, **вызывает конструктор**;
5. возвращает адрес динамического объекта.

Обратите внимание

Чётвёртый пункт является **ключевым** для создания динамических объектов: оператор **new** всегда попытается вызвать **конструктор** после выделения памяти под конкретный объект.

Динамические объекты в C++

Рассмотрим на примере класса **DynArray1D** из предудущей лекции:

```
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      ~DynArray1D();
9      ...
10
11 private:
12     double *_arr = nullptr;
13     size_t _length = 0;
14     size_t _capacity = 0;
15 };
```

Динамические объекты в C++

И пробуем создать динамический объект класса **DynArray1D**

```
1 DynArray1D *p_arr = new DynArray1D;
```

Здесь:

- переменная **p_arr** имеет тип **указатель на DynArray1D**;
- оператор **new** после выделения памяти под объект типа **DynArray1D** пытается вызвать конструктор. Поскольку не указаны никакие параметры, он ищет либо **конструктор без параметров**, либо обращается к **конструктору по умолчанию**;
- так как класс **DynArray1D** содержит в себе **конструктор без параметров** (*двадцать третий слайд, строка 4*) - он и вызывается.

Итого: указатель **p_arr** ссылается на объект класса **DynArray1D**, который был создан с помощью **конструктора без параметров**.

Динамические объекты в C++

Нет никаких препятствий для вызова других конструкторов класса **DynArray1D**. Синтаксис следующий:

```
1 DynArray1D *p_arr1 = new DynArray1D{16};
2
3 DynArray1D *p_arr2 = new DynArray1D{8, 555.555};
4
5 (*p_arr1) << -1.4;
6 cout << "Длина массива: " << p_arr1->length() << endl;
7 // ...
```

строка 1: создаём динамический объект с помощью конструктора с одним параметром;
строка 3: создаём динамический объект с помощью конструктора с двумя параметрами;
строки 5-6: демонстрация работы с объектом через указатель.
Стоит помнить: синтаксис C++ позволяет вместо *фигурных* скобок, использовать *круглые*. Но первый вариант в настоящее время *предпочтительней*.

Динамические объекты в C++

Когда динамический объект становится ненужным, от него следует избавиться. С технической точки зрения - высвободить используемую под него память. Для этого применяется оператор **delete**. Общая схема его работы для динамических объектов:

1. оператор получает объект для удаления;
2. определяет тип объекта;
3. вызывает **деструктор** объекта;
4. высвобождает память, используемую под объект.

В виде примера:

```
1 DynArray1D *p_arr1 = new DynArray1D{16};
2 DynArray1D *p_arr2 = new DynArray1D{8, 555.555};
3 // ... что-то делаем
4
5 delete p_arr1; // удалили первый объект
6 delete p_arr2; // а теперь и второй
```

Динамические объекты в C++

Аналогично **фундаментальным** типам, можно создавать массивы динамических объектов с помощью операторов **new[]** и **delete[]**. Для примера:

```
1 DynArray1D *p_arrays = new DynArray1D[5];
2
3 p_arrays[2] << 9.999;
4
5 delete[] p_arrays;
```

строка 1: создаём пятиэлементный динамический массив объектов **DynArray1D**. Для каждого объекта в данном случае была выделена память под него и был вызван **конструктор без параметров**.

строка 3: после работы с динамическим массивом, удалаляем все его элементы. Для каждого из пяти объектов будет вызван деструктор и выделенная под объект память возращена ОС.
Повторение: для массивов должен быть вызван оператор **delete[]**, а не **delete**

Динамические объекты в C++

А что, если для каждого объекта массива хочется вызывать конструктор с параметрами. «Нет каких преград» - отвечает Вам C++:

```
1 DynArray1D *p_arrays = new DynArray1D[3] {3, 4, 5};
2
3 for (size_t i = 0; i < 3; i++) {
4   cout << "Длина массива: " << p_arrays[i].length()
5        << endl;
6 }
7
8 delete[] p_arrays;
```

В **первой строке** создаём массив на три элемента, для каждого из которых вызывается **конструктор с одним параметром**. Причём для первого объекта в конструктор передаётся число **три**, во второй - **четыре**, в третий - **пять**.

Поскольку для **DynArray1D** есть ещё конструктор с двумя параметрами, и его можно вызвать при создании динамического массива объектов:

```
1  DynArray1D *p_arrays =
2              new DynArray1D[3] { {3, 1.5},
3                                  {4, -0.4},
4                                  {5, 900.03} };
5
6  cout << p_arrays[0][0] << "\n";
7  cout << p_arrays[-1][0] << "\n";
8  cout << p_arrays[-1][0] << "\n";
9
10 delete[] p_arrays;
```

Первый объект массива будет создан с вызовом конструктора **DynArray1D(3, 1.5)**, второй - **DynArray1D(4, -0.4)**, третий - **DynArray1D(5, 900.03)**.
Итого: параметры в конструктор каждого объекта динамического массива можно передать с использованием пары *фигурных* скобок.

Наследование в C++

Наследование в C++

Наследование - специальный механизм в языках программирования, позволяющий создавать новые классы на основе существующих. В C++ это реализуется через добавление полей некоторого имеющегося класса и получение частичного доступа к его методам в некоторый новый. Стоит заметить, что слово «добавить» не равно словосочетанию «получить полный доступ».
Терминология следующая:

- Класс, от которого берутся поля и методы называется **базовым**.
- Класс, который получает поля и методы другого класса называется **производным**.

Наследование в C++

Основные моменты наследования в C++:

- при наследовании **все** поля базового класса переходят к производному;
- у производного класса есть полный доступ к открытым полям базового класса и **закрытым полям** с модификатором **protected**;
- методы также передаются по определённым правилам.

Наследование в C++

Введение в наследование в C++ наиболее полно описано на github'e: **https://github.com/posgen/OmsuMaterials/wiki** (подпункт *«Несколько статей, касающихся объектно-ориентированного программирования в C++»*)
Желательно для ознакомления группой ФП (тем, кто действительно хочет разобраться - *обязательно*).