Объекты

Шокуров Антон В.
shokurov.anton.v@yandex.ru
http://машинноезрение.рф

29 сентября 2017 г.

Версия: 0.10

Аннотация

В данной заметке будут рассмотрены пара ключевых объектов из библиотеки `stl`. Также показано как взаимодействовать с динамической памятью.

Цель. Изучить как создавать объект типа массив, динамический массив и как работать с памятью. Умный указатель.

Предварительный вариант!

1 Объекты

В языке Си++ расширяется понятие структуры (`struct`). Теперь вводятся действия, которые непосредственно связаны с самим объектом, а часть полей наоборот скрыто от пользователя объекта. Последнее фактически позволяет взаимодействовать с объектом как с черной коробкой. У объекта для взаимодействия открыто только то, что считается нужным. Такой подход позволяет мыслить объектами, которые сами знаю как себя сохранить в корректном состоянии.

Далее будут рассмотрены два объекта связанных с понятием массив. Один массив имеет фиксированный размер, а другой динамический, т.е. его размер может меняться с течением работы программы.

1.1 Массив фиксированного размера

В языке Си можно создать массив фиксированного размера:

```
1 int a[5];
```

Как мы помним из первого семестра последнее действие создало переменную a, имеющую тип массив, который состоит из intов в количестве 5 штук. Таким образом a[0], a[1], a[2], a[3] и a[4] являются переменными типа int. Но, зная a сложно понять сколько в этом массиве элементов. Можно конечно написать такой код:

```
1 sizeof(a)/sizeof(a[0])
```

Здесь sizeof вычисляет полный размер массива и делит на размер одного элемента. Такое возможно благодаря тому, что компилятор знает какой размер у массива, т.е. sizeof вычисляется при компиляции программы (compile time), а не при выполнении (run time). Следовательно сама переменная a не знает из скольких элементов она состоит. Ввиду последнего такой подход некрасив.

Объекты в Си++, как было упомянуто выше, призваны устранить данный недостаток. В частности, в Си++ (точнее в библиотеке STL) есть объект, который соответствует массиву в Си, но он уже сам про себя много чего знает:

```
1 #include <array>
2 ...
3 stl::array<int, 5> b;
```

Первая строчка показывает, что для работы примера нужно добавить данный заголовочный файл в начале программы. В противном случае компилятор скажет, что он не знает об stl::array. Также подчеркну, что заголовочный файл называется именно array, а не array.h, т.е. расширения h у него нету.

Последняя строчка фрагмента кода объявляет переменную b, по аналогии с Си у неё тип stl::array<int, 5>. Угловые скобки относятся к понятию шаблона, который будет изучен в одной из заметок. Сейчас же можно отметить, что они позволяют специализировать объект. Так, сам массив фиксированного размера задается stl::array, а вот тип элементов указывается в качестве первого аргумента, а размер второго. Например, если бы мы хотели сделать массив размера 10 из double, то мы бы написали:

```
1 stl::array<double, 10> bb;
```

Как следует из первой заметке в C++ добавлены некоторые расширения, которые призваны упростить написание кода. В частности, в Си++ можно написать собственную функцию для оператора квадратные скобки. Фактически последнее будет являться полиморфизмом так как переопределяется понятие квадратной скобки для другого объекта (т.е. не встроенных в сам язык массивов). Так вот все эти вещи позволяют после создания объекта array оперировать с ним как с массивом. Например:

```
1 b[2] = 5;
2 b[3] = b[2] +1;
```

т.е. можно использовать выражения `b[0]`, ..., `b[4]` как переменные, в точности как ранее описано это было в Си.

Его также можно инициализировать следующим образом:

```
1 stl::array<int, 5> b = {5, 9, -1, 2};
2 //В последнем элементе будет мусор!
```

Но в отличии от Си этот массив знает свой размер:

```
1 b.size()
```

Данное выражение даст размер массива b, в данном случае 5. Такая конструкция вызова функции является продолжением понятия struct языка Си. Напомню, что

```
1 typedef struct
2 {
3   double x, y;
4 }mypoint;
```

объявляет новый тип данных соответствующей двумерной точки. Если нужно будет далее с ним взаимодействовать, то это делается так:

```
1 mypoint p0, p1;//Объявили две точки
2 ...
3 p0.x = p1.y + 3;
4 p0.y = p1.x + p1.y * 1.1;
```

Таким образом доступ к полям осуществляется через оператор точка: пишется точка после переменной являющейся объектом, а далее имя поля.

Так вот, в языке Си++ появляется ещё возможность использовать в качестве поля не только переменную, а функцию. Такие функции называются методами объекта (класса). Вызывается она по полной аналогии с обращением к полю структуры, разница в том, что это функция, т.е. у неё существуют, возможно пустой, список аргументов. В любом случае нужно использовать скобки, а не просто имя функции (иначе это было бы просто переменной поля структуры).

Поэтому размер массива как раз и возвращается после вызова такой вот функции. В качестве примера можно ещё привести следующее:

```
1 int i, n = b.size()
2 for( i = 0; i < n; i++)
```

```
3     printf("%d ", b.at(i));
4 printf("\n")
```

Здесь показан ещё один метод объекта `array`, `at`, который полностью аналогичен квадратным скобкам, т.е. он возвращает значение элемента массива с индексом указанным в качестве аргумента.

Взаимодействие с объектами в основном осуществляется через вызов методов. Благодаря этому можно контролировать корректное состояние объекта.

1.2 Массив динамического размера

Как создать умный массив фиксированного размера было показано в предыдущем подразделе. Как создать аналогичную вещь для массивов динамического размера?

В первом семестре вводились понятия необходимые для работы с динамической памятью. Напомню, что он создавался следующим образом:

```
1 int n;
2 ...
3 int *c = (int*)malloc( n * sizeof( int ) );
```

Но этот случай ещё хуже, чем даже массив фиксированного размера обсуждаемый ранее. Зная переменную a нет возможности вычислить размер массива. Например, код вида

```
1 sizeof(c)/sizeof(c[0])
```

не сработает, так как в данном случае `sizeof(c)` вернет размер самого указателя, а не самого массива.

Для создания объекта соответствующего динамическому массиву используется следующая конструкция:

```
1 #include <vector>
2 ...
3 stl::vector<int> d;
```

Размер массива можно узнать по аналогии с предыдущем объектом:

```
1 d.size()
```

Первоначально он конечно нулевого размера.

Также можно заранее проинициализировать массив:

```
1 stl::vector<int> b = {5, 9, -1, 2};
```

Обращаться с отдельным элементам массива можно как в Си, через квадратные скобки, как ранее было показано с объектом `array`.

Тонкость заключается в том, что если указать в качестве индекса элемент не лежащий в массиве, то это приведет к ошибке. Как же тогда обеспечивается рост массива? Последнее делается, например, с помощью метода `push_back`:

```
1 d.push_back(11);
```

После данного вызова число 11 добавиться в конец массива, а сам массив изменит свой размер. Таким образом, если до вызова массив состоял из элементов : 5, 9, -1, 2. То после данного вызова он будет содержать: 5, 9, -1, 2, 11.

1.3 Динамическая память

В языке Си++ разумеется можно выделять память динамически. В данном подразделе будет показано как это сделать.

В двух предыдущих подразделах память в явном виде не выделялась и не освобождалась.

Так, в первом подразделе это было и так понятно. В языке Си размер массива в случае фиксированного размера указывается в самом исходном коде, поэтому память под него выделяется иначе и освобождается тоже системой самой. Далее для других объектов это само собой подразумевалось.

Выделение памяти Автоматическое освобождение памяти является основополагающим и краеугольным. Такая возможность достигается благодаря введения понятия жизни объекта. Так он создается в момент объявления и уничтожается по завершению его использования. Последнее достигается благодаря таким понятиям как конструктор и деструктор. Мы их изучим более детально в отдельной заметке.

Сейчас перейдем к явному способу выделения памяти. Следует отметить, что в языке Си++ выделение памяти является частью синтаксиса языка, а не благодаря вспомогательной библиотеке. Напомню, как выше уже и было показано, что память в языке Си выделяется вызовом функции `malloc`. Эта функция не имеет отношение к самому языку Си прямого отношения, она относится к стандартной библиотеке. С точки зрения самого языка Си память уже вся выделена, а нам дана возможность взаимодействовать с определенными её кусками благодаря указателям. Таким образом, в языке Си как бы считается, что malloc является посредником, который передает нам некий указатель. К самому языку это отношение не имеет.

Иначе дело обстоит в C++. Выделение памяти является частью самого языка, самого синтаксиса. Так, память выделяется следующим образом:

```
1 int n;
2 ...
3 int *a = new int[n];//Создали массив размера n.
```

После этой конструкции выделяется память под массив из `int` длины `n`. В данном случае `new` является частью языка C++, а не какой-то функцией.

Когда память выделяется в явном виде, то её освобождать тоже нужно явным образом.

В Си, напомню, это делалось вызовом функции `free`:

```
1 free( a );//Освобождение памяти под массив a.
```

В языке C++ для этого же используется конструкция:

```
1 delete[] a;//Используется []!
```

Здесь важно обратить внимание на то, что это не просто `delete`, а `delete` с квадратными скобками `[]`.

Предыдущие показывает как выделить память для массива. Но в C++ существует вариант конструкции для выделения памяти и для одного единственного объекта. Она чуть проще:

```
1 int *a = new int;//Создали единственный int.
2 ...
3 delete a;//Удалили данный объект. Без []!
```

Подчеркну ещё раз. Что при явном выделении памяти её необходимо и явно освободить! Иначе это приведет к утечки памяти.

1.4 Умные указатели

Ввиду того, что язык C++ достаточно наворочен в нем существует конструкция для забывчивых. Так, можно создать объект, который автоматически освободит выделенную памяти сам за нас. Он называется умных указателем.

Он существует в разным вариациях.

Уникальный указатель Мы можем создать указатель, к нему обращаться, но не может его скопировать. Зато в нужным момент он сам освободит память.

Рассмотрим такой код:

```
1 int main()
2 {
```

```
3   int *a = new int;//Создали единственный int.
4   ...//взаимодействуем с a, но не удаляем.
5   return 0;
6 }
```

Такой код приведет к утечки памяти. Исходя из предыдущего нужно писать код так:

```
1 int main()
2 {
3   int *a = new int;//Создали единственный int.
4   ...//взаимодействуем с a, но не удаляем.
5   delete a;//Удаляем a.
6   return 0;
7 }
```

Такой код сработает, но уклюжий (не в идеологии Си++). Тем более если учесть, что мы должны освобождать объект из любой точки где есть return (подробнее об этом в другой заметке).

Более элегантно с точки зрения языка C++ будет код:

```
1  ...
2  #include <memory>
3  ...
4
5  int main()
6  {
7    int *a = new int;//Создали единственный int.
8    std::unique_ptr<int> p(aaa);
9    ...//взаимодействуем с a, но не удаляем.
10   return 0;
11 }
```

Тогда память будет освобождения автоматически при уничнотжении объекта `p` типа `unique_ptr`.

Для ещё большей аккуратности следует избегать создание самой переменной с указателем и писать код так:

```
1 int func()
2 {
3   std::unique_ptr<int> p(new int);
4   ...//взаимодействуем с a, но не удаляем.
```

```
5   return 0;
6 }
```

Разделяемый указатель Не всегда удобно то, что указатель нельзя копировать (как в предыдущем случае). Проблема связанная с копированием в том, что если её поддерживать, то нужно учитывать сколько существуют действующих указателей. Последнее означает, что если указатель копируется, потом первый указатель прекращает свое существование, то это не означает, что объект нужно удалить. Видь второй указатель по-прежнему существует.

Для решения данной ситуации приходится усложнять умный указатель: он учитывает сколько указателей действительно существуют. Для этого вводится счетчик, и при каждом копировании и уничтожении он соответственно увеличивается или уменьшается. Сам объект уничтожается, когда счетчик становится равным нулю.

Для взаимодействия с таким умным указателем нужно сделать следующее:

```
1 int func()
2 {
3   std::shared_ptr<int> p(new int);
4   ...//взаимодействуем с a, но не удаляем.
5   return 0;
6 }
```