Объектно-ориентированное программирование с использованием языка C++

компьютерная академия ШАГ

Урок №2

Объектно-ориентированное программирование с использованием языка C++

Содержание

Перегруженные конструкторы

В прошлом уроке вы познакомились с основами ООП — и узнали о понятии класса. Сегодня мы с вами продолжим это приятное знакомство. Мы уже выяснили, конструкторы могут иметь параметры. Для этого просто нужно добавить эти параметры в объявление и определение конструктора, а затем, при создании объекта, задать их в качестве аргументов. Теперь к нашим знаниям добавим еще одно — конструкторов может быть несколько. Рассмотрим пример:

```
#include <iostream>
using namespace std;

class _3D
{
     double x, y, z;
     public:
     _3D ();
     _3D (double initX, double initY, double initZ);
};

//конструктор класса _3D с параметрами
_3D::_3D(double initX, double initY, double initZ)
{
     x = initX;
     y = initY;
     z = initZ;
     cout << “\nWith arguments!!!\n”;
}
```

```
//конструктор класса _3D без параметров
_3D::_3D()
{
     x=y=z=0;
     cout << “\nNo arguments!!!\n”;
}

void main()
{
     //создается объект A, вызывается
     //конструктор без параметров
     //все члены класса инициализируются нулем
     //на экране надпись "No arguments!!!"
     _3D A;

     //создается объект B, вызывается
     //конструктор с параметрами
     //все члены класса инициализируются
     //соответствующими переменными
     //на экране надпись "With arguments!!!"
     _3D B (3,4,0);
}
```

Примечание: Кстати!!! В отличие от конструктора, деструктор не может быть перегружен, так как не имеет параметров. Это вполне логично, поскольку отсутствует механизм передачи параметров удаляемому объекту.

Комментарии к примеру и особенности использования

1. Каждому способу объявления объекта класса должна соответствовать своя версия конструкторов класса. Если это не будет обеспечено, то при компиляции программы обнаружится ошибка на этапе компиляции.

2. На этом примере легко понять, чем может быть вызвана необходимость перегрузки конструкторов.(именно перегрузки, поскольку речь здесь идет о функциях, имеющих одинаковые имена, но различные списки параметров) Итак, главный смысл перегрузки конструкторов состоит в том, чтобы предоставить программисту наиболее подходящий метод инициализации объекта.
3. В примере представлен наиболее распространенный вариант перегрузки конструкторов, т.е. конструктор без параметров и конструктор с параметрами. Как правило, в программе бывают необходимы оба эти вида, поскольку конструктор с параметрами более удобен при работе с одиночными объектами, но не может использоваться при инициализации объектов-элементов динамического массива.
4. Хотя конструктор можно перегружать столько раз, сколько захотите, лучше не стоит этим злоупотреблять. Конструктор стоит перегружать лишь для наиболее часто встречающихся ситуаций.

Полезная информация

Обратите внимание на то, что тела конструкторов описаны за пределами класса. В класс помещены только прототипы. Данная форма записи может быть использована и для обычных методов класса. Напомним, что в примерах прошлого урока тела методов описывались прямо в определении класса.

Как лучше и грамотнее, спросите вы?! Способ из прошлого урока используется для простых и коротких мето-

дов, которые в дальнейшем не предполагается изменять. Так поступают отчасти из-за того, что описания классов помещают обычно в файлы заголовков, включаемые затем в прикладную программу с помощью директивы #include. Кроме того, при этом способе машинные инструкции, генерируемые компилятором при обращении к этим функциям, непосредственно вставляются в оттранслированный текст. Это снижает затраты на их исполнение, поскольку выполнение таких методов не связано с вызовом функций и механизмом возврата, увеличивая в свою очередь размер исполняемого кода (то есть такие методы становятся inline или встраиваемыми).

Способ, используемый в описанном выше примере, предпочтительнее для сложных методов. Объявленные таким образом функции автоматически заменяются компилятором на вызовы подпрограмм.

Указатели на объекты

До сих пор доступ к членам объекта осуществлялся, с использованием операции ".". Это правильно, если вы работаете с объектом. Однако доступ к членам объекта можно осуществлять и через указатель на объект. В этом случае обычно применяется операция стрелка "->". Похоже на структуру, не правда ли?

Указатель на объект объявляется точно так же, как и указатель на переменную любого типа. А, для получения адреса объекта, перед ним необходим оператор &.

```
#include <iostream>
using namespace std;

class _3D
{
     double x, y, z;
public:
     _3D ();
     _3D (double initX, double initY, double initZ);
     void Show(){
     cout<<x<<” “<<y<<z<<”\n”;
     }
};

//конструктор класса _3D с параметрами
_3D::_3D(double initX, double initY, double initZ)
{
     x = initX;
     y = initY;
     z = initZ;
     cout << “\nWith arguments!!!\n”;
}
```

```
//конструктор класса _3D без параметров
_3D::_3D()
{
     x=y=z=0;
     cout << “\nNo arguments!!!\n”;
}

void main()
{
     //создается объект A, вызывается
     //конструктор с параметрами
     //все члены класса инициализируются
     //соответствующими переменными
     //на экране надпись "With arguments!!!"
     _3D A (3,4,0);

     //создается указатель на объект типа
     //_3D и в этот указатель записывается
     //адрес объекта A
     _3D*PA=&A;

     //через указатель вызывается функция
     //Show()
     PA->Show();
}
```

Динамическое выделение памяти под объект

Если класс имеет конструктор без аргументов, то обращение к операции new полностью совпадает с тем, что используется для выделения памяти под обычные типы данных без инициализирующего выражения.

```
#include <iostream>
using namespace std;
```

```
class Point
{
     double x, y;
public:
     Point(){
          x=y=0;
          cout << “\nNo arguments!!!\n”;
     }
     void Show(){
          cout<<x<<” “<<y<<”\n”;
     }
};

void main()
{
     //создание объекта
     Point A;
     //показ содержимого на экран
     A.Show();

     cout<<”****************************”;
     //создание указателя на объект
     Point*PA;

     //Динамическое выделение памяти под один
     //объект типа Point
     PA=new Point;

     //проверка, выделилась ли память
     //и выход, если не выделилась
     if(!PA) exit(0);

     //через указатель вызывается функция
     //Show()
     PA->Show();

     cout<<”****************************”;
```

```
      //создание указателя на объект
      Point*PB;

      //Динамическое выделение памяти под массив
      //объектов типа Point
      PB=new Point[10];

      //проверка, выделилась ли память
      //и выход, если не выделилась
      if(!PB) exit(0);

      //Вызов функции Show() для каждого элемента
      //массива PB
      for(int i=0;i<10;i++){
            PB[0].Show();
      }

      //Удаление объекта PA
      delete PA;

      //Удаление массива PB
      delete[]PB;
}
```

Если же конструктор класса имеет аргументы, то список аргументов помещается там же, где при работе со стандартными типами данных находится инициализирующее выражение.

```
#include <iostream>
using namespace std;

class Point
{
      double x, y;
```

```
public:
     //конструктор с параметрами
     //по умолчанию
     Point(double iX=1,double iY=1){
          x=iX;
          y=iY;
          cout << "\nWith arguments!!!\n";
     }
     void Show(){
          cout<<x<<" "<<y<<"\n";
     }
};

void main()
{
     //создание объекта
     Point A(2,3);
     //показ содержимого на экран
     A.Show();

     cout<<"****************************";
     //создание указателя на объект
     Point*PA;

     //Динамическое выделение памяти под один
     //объект типа Point
     //в круглых скобках - параметры для конструктора
     PA=new Point(4,5);

     //проверка, выделилась ли память
     //и выход, если не выделилась
     if(!PA) exit(0);

     //через указатель вызывается функция
     //Show()
     PA->Show();

     cout<<"****************************";
```

```
        //создание указателя на объект
        Point*PB;

        //Динамическое выделение памяти под массив
        //объектов типа Point
        //параметры не передаются
        //используются параметры
        //конструктора по умолчанию
        PB=new Point[10];

        //проверка, выделилась ли память
        //и выход, если не выделилась
        if(!PB) exit(0);

        //Вызов функции Show() для каждого элемента
        //of the PB array
        for(int i=0;i<10;i++){
             PB[0].Show();
        }

        //Deleting the PA object
        delete PA;

        //Deleting the PB array
        delete[]PB;
}
```

Примечание: Обратите внимания, что в данном примере использован конструктор с параметрами по умолчанию. Это связано с тем, что при динамическом выделении памяти под массив объектов НЕВОЗМОЖНО передать параметры в конструктор. В нашем примере мы этого и не делаем. Для массива объектов используются параметры по умолчанию. Эту проблему можно было решить иначе, создав конструктор без параметров.

Статические массивы

В отличие от динамики, при создании статического массива параметры в конструктор передать можно. Рассмотрим синтаксис этого действия на примере:

```
#include <iostream>
using namespace std;

class Point
{
     double x, y;
public:
     //конструктор с параметрами
     Point(double iX,double iY){
          x=iX;
          y=iY;
          cout << “\nWith arguments!!!\n”;
     }
     void Show(){
          cout<<x<<” “<<y<<”\n”;
     }
};

void main()
{
     //создание массива объектов
     //передача параметров в конструктор
     Point AR[2]={Point(2,3),Point(4,5)};

     //Вызов функции Show() для каждого элемента
     //массива AR
     for(int i=0;i<2;i++){
          AR[i].Show();
     }
}
```

Указатель this

В прошлом уроке мы выяснили, что любой метод класса самостоятельно определяет, для какого объекта он был вызван и "видит" другие члены класса без передачи их в качестве параметров. Зададимся вопросом: как это происходит?!

Ответ на этот вопрос не является секретом. Дело в том, что когда функция, принадлежащая классу, вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет фиксированное имя this и незаметно для программиста определен в каждой функции класса.

Итак, кое-что о this...

1. Указатель this инициализируется значением адреса объекта, для которого вызван метод, перед началом выполнения кода этого метода.
2. Имя this является служебным (ключевым) словом.
3. Явно описать или определить указатель this нельзя.
4. В соответствии с неявным определением this является константным указателем, т.е. изменять его нельзя, однако в каждой принадлежащей классу функции он указывает именно на тот объект, для которого функция вызывается.

5. Объект, который адресуется указателем this, становится доступным внутри принадлежащей классу функции именно с помощью указателя this.
6. Внутри функции — члена класса можно явно использовать этот указатель.

Указатель this является очень полезным, а иногда просто незаменимым. Например, в следующем коде указатель this позволяет компилятору разобраться в ситуации, когда имя компонента класса совпадает с именем формального параметра, принадлежащего методу:

```
#include <iostream>
using namespace std;
class Student //Класс студент.
{
     char name[50]; //имя
     char surname[50]; //фамилия
     int age; //возраст
public:
     //Конструктор:
     Student(char name[],char surname[],int age)
     {
          //Компоненты и одноименные параметры:
          strcpy(this->name,name);
          strcpy(this->surname,surname);
          this->age=age;
     }
     void Show()
     {
          //Здесь this является необязательным,
          //однако использовать его можно
          cout << "\nNAME - " << this->name;
          cout << "\nSURNAME - " << this->surname;
          cout << "\nAGE - " << this->age;
```

```
            cout << “\n\n”;
      }
};

void main(void)
{
      Student A(“Ivan”,”Sidoroff”,25);
      A.Show();
}
```

Сейчас мы только познакомились с указателем this. Более широкое применение он еще найдет в последующих уроках.

Copy constructor

Прежде чем обсуждать "загадочный" конструктор копирования, давайте поговорим о простых истинах. Итак.

Передача объекта в функцию

Объекты класса можно передавать в функции в качестве аргументов точно так же, как передаются данные других типов. Однако, следует помнить, что в языках C и C++ методом передачи параметров, по умолчанию является передача объектов по значению. Это означает, что внутри функции создается копия объекта — аргумента, и эта копия, а не сам объект, используется функцией. Следовательно, изменения копии объекта внутри функции не влияют на сам объект.

Вот несколько утверждений, которые характеризуют действия, происходящие с объектом в этом случае:

При передаче объекта в функцию появляется новый объект. Когда работа функции, которой был передан объект, завершается, то удаляется копия аргумента.

Когда удаляется копия объекта, вызывается деструктор копии, поскольку эта копия выходит из своей области видимости.

Объект внутри функции — это побитовая копия передаваемого объекта, а это значит, что если объект содержит в себе, например, некоторый указатель на динамически выделенную область памяти, то при копировании создается объект, указывающий на ту же область памяти. И как только вызывается деструктор копии, где, как правило, принято

высвобождать память, то высвобождается область памяти, на которую указывал объект-"оригинал", что приводит к разрушению исходного объекта.

```
#include <iostream>

using namespace std;
class ClassName
{
public:
     ClassName ()
     {
          cout << “ClassName!!!\n”;
     }
     ~ClassName ()
     {
          cout << “~ClassName!!!\n”;
     }
};

void f (ClassName o)
{
     cout << “Function f!!!\n”;
}

void main()
{
     ClassName c1;
     f(c1);
}

Program output:

ClassName!!!
Function f!!!
~ClassName!!!
~ClassName!!!
```

Конструктор вызывается только один раз. Это происходит при создании c1. Однако деструктор срабатывает дважды: один раз для копии o, второй раз для самого объекта c1. Тот факт, что деструктор вызывается дважды, может стать потенциальным источником проблем, например, (как уже говорилось) для объектов, деструктор которых высвобождает динамически выделенную область памяти.

Возврат объекта из функции

Похожая проблема возникает и при использовании объекта в качестве возвращаемого значения.

Для того чтобы функция могла возвращать объект, нужно: во-первых, объявить функцию так, чтобы ее возвращаемое значение имело тип класса, во-вторых, возвращать объект с помощью обычного оператора return. Однако если возвращаемый объект содержит деструктор, то в этом случае возникают проблемы, связанные с "неожиданным" разрушением объекта.

```
#include <iostream>

using namespace std;
class ClassName
{
public:
     ClassName ()
     {
          cout << “ClassName!!!\n”;
     }
     ~ClassName ()
     {
         cout << “~ClassName!!!\n”;
     }
};
```

```
ClassName f()
{
     ClassName obj;
     cout << “Function f\n”;
     return obj;
}

void main()
{
     ClassName c1;
     f();
}

Program output:

ClassName!!!
ClassName!!!
Function f
~ClassName!!!
~ClassName!!!
~ClassName!!!
```

Конструктор вызывается два раза: для c1 и obj. Однако деструкторов здесь три. Как же так? Ясно, что один деструктор разрушает c1, еще один - obj. "Лишний" вызов деструктора (второй по счету) вызывается для так называемого временного объекта, который является копией возвращаемого объекта. Формируется эта копия, когда функция возвращает объект. После того, как функция возвратила свое значение, выполняется деструктор временного объекта. Естественно, что если деструктор, например, высвобождает динамически выделенную память, то разрушение временного объекта приведет к разрушению возвращаемого объекта.

Инициализация одного объекта другим при создании

В программировании есть еще один случай побитовогокопирования — инициализация одного объекта другим при создании:

```
#include <iostream>
using namespace std;

class ClassName
{
public:
     ClassName ()
     {
          cout << “ClassName!!!\n”;
     }
     ~ClassName ()
     {
          cout << “~ClassName!!!\n”;
     }
};

void main()
{
     ClassName c1;
     //Вот он!!! Момент побитового копирования.
     ClassName c2=c1;
}

Результат работы программы:

ClassName!!!
~ClassName!!!
~ClassName!!!
```

Конструктор вызывается один раз: для c1. Для c2 конструктор не срабатывает. Однако деструктор сраба-

тывает для обоих объектов. А, поскольку, c2 является точной копией c1, деструктор, высвобождающий динамически выделенную память, вызывается дважды для одного и того же фрагмента этой памяти. Это неминуемо приведет к ошибке.

Решение проблемы

Одним из способов обойти такого рода проблемы является создание особого типа конструкторов, — конструкторов копирования. Конструктор копирования или конструктор копии позволяет точно определить порядок создания копии объекта. Любой конструктор копирования имеет следующую форму:

```
class_name (const class_name & obj)
{
... //тело конструктора
}
```

Здесь obj — это ссылка на объект или адрес объекта. Конструктор копирования вызывается всякий раз, когда создается копия объекта. Таким образом, в конструкторе копирования можно выделить "свою" память под новый объект.

```
#include <iostream>
using namespace std;

class ClassName
{
public:
     ClassName ()
     {
```

```
            cout << “ClassName!!!\n”;
      }
      ClassName (ClassName&obj){
            cout << “Copy ClassName!!!\n”;
      }
      ~ClassName ()
      {
            cout << “~ClassName!!!\n”;
      }
};

void f(ClassName o){
      cout<<”Function f!!!\n”;
}

ClassName r(){
      ClassName o;
      cout<<”Function r!!!\n”;
      return o;
}

void main()
{
      //инициализация одного объекта другим
      ClassName c1;
      ClassName c2=c1;

      //передача объекта в функцию
      ClassName a;
      f(a);

      //возврат объекта из функции
      r();
}

Результат работы программы:
```

```
//создался объект c1
ClassName!!!

//инициализация объекта c2 объектом c1
Copy ClassName!!!

//создался объект a
ClassName!!!

//передача a в функцию по значению
//создалась копия o
Copy ClassName!!!

//отработала функция f
Function f!!!

//уничтожилась копия o
~ClassName!!!

//создался объект o
//внутри функции r
ClassName!!!

//отработала функция r
Function r!!!

//возврат из функции
//создалась копия объекта o
Copy ClassName!!!

//уничтожился объект o
~ClassName!!!

//уничтожилась его копия
~ClassName!!!

//уничтожился объект a
~ClassName!!!
```

```
//уничтожился объект c2
~ClassName!!!

//уничтожился объект c1
~ClassName!!!
```

Примечание: Конструктор копирования не влияет на операцию присваивания вида A=B. Здесь также срабатывает побитовое копирование, однако эту проблему решают в C++ иначе.

Теперь, когда есть конструктор копирования, можно смело передавать объекты в качестве параметров функций и возвращать объекты. При этом количество вызовов конструкторов будет совпадать с количеством вызовов деструкторов, а поскольку процесс образования копий теперь стал контролируемым, существенно снизилась вероятность неожиданного разрушения объекта.

Примечание: Кстати!!! Помимо создания конструктора копирования есть другой способ организации взаимодействия между функцией и программой, передающей объект. Этот способ — передача объекта по ссылке или по указателю.

Домашнее задание

1. Разработать класс Person, который содержит соответствующие члены для хранения:
   - имени,
   - возраста,
   - пола и
   - телефонного номера.

   Напишите функции-члены, которые смогут изменять эти члены данных индивидуально. Напишите функцию-член `Person::Print()`, которая выводит отформатированные данные о человеке.
2. Разработать класс String, который в дальнейшем будет использоваться для работы со строками. Класс должен содержать:
   - конструктор по умолчанию, позволяющий создать строку длиной 80 символов;
   - конструктор, позволяющий создавать строку произвольного размера;
   - конструктор, который создаёт строку и инициализирует её строкой, полученной от пользователя.

   Класс должен содержать методы для ввода строк с клавиатуры и вывода строк на экран.