ЛАБОРАТОРНАЯ РАБОТА №4
КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

Целью данной лабораторной работы является знакомство с конструкторами и деструкторами в языке C++.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

Конструкторы - это особые методы класса, используются для создания объектов и их инициализации, имеют имя, совпадающее с именем класса.

Конструктор существует для любого класса, причем он может быть создан без явных указаний программиста. Таким образом, для классов complex1 и goods существуют автоматически созданные конструкторы.

По умолчанию формируются конструктор без параметров и конструктор копирования вида.

```
1) T::T(const T&)
2) //где T - имя класса. Например,
3) class F {
4) ...
5) public: F(const F&);
6) ...
7) }
```

Такой конструктор существует всегда. По умолчанию конструктор копирования создается общедоступным.

Сформулируем несколько правил использования конструкторов.

1. В классе может быть несколько конструкторов (перегрузка), но только один с значениями параметров по умолчанию.
2. Нельзя получить адрес конструктора.

3. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него, как у конструктора копирования.
4. Конструктор нельзя вызывать как обычную компонентную функцию. Для явного вызова конструктора можно использовать две разные синтаксические формы:

```
1) имя_класса
   имя_объекта(фактические_параметры_конструктора);
2) имя_класса(фактические_параматры_конструктора);
```

Первая форма допускается только при непустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

```
1) complex SS(10.3,0.22); // SS.real == 10.3;
2) // SS.imag == 0.22;
3) complex EE(2.345);      // EE.real == 2.345;
4) //  По умолчанию EE.imag = 0.0.
5) complex DD() ; // Ошибка! Компилятор решит, что это
6) // прототип функции без параметров,
7) //  возвращающей значение типа complex.
```

Вторая форма явного вызова конструктора приводит к созданию объекта, не имеющего имени. Созданный таким вызовом безымянный объект может использоваться в тех выражениях, где допустимо использование объекта данного класса. Например:

```
1) complex ZZ = complex(4.0,5.0);
```

Этим определением создается объект ZZ, которому присваивается значение безымянного объекта (с элементами real == 4.0, imag == 5.0), созданного за счет явного вызова конструктора.

Существуют два способа инициализации данных объекта с помощью конструкторов. Первый способ, а именно передача значений параметров в тело конструктора, который уже продемонстрирован на примерах. Второй способ предусматривает применение списка инициализаторов данных объекта. Этот список помещается между списком параметров и телом конструктора:

```
1.имя-класса(список_параметров):
```

```
                список_инициализаторов_компонентных_данных
                                                          {
тело_конструктора  }
```

Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

```
  1) имя_компонента_данных (выражение)
```

Например:

```
1) class AZ
2) { int ii; float ee; char cc;
3) public:
4) AZ(int in, float en, char cn) : ii(5),
5) ee(ii * en + in), cc(cn) { }
6) };
7) AZ A(2,3.0 , 'd');
8) // Создается именованный объект A
9) //  с компонентами A.ii == 5,
10) //  A.ee == 17, A.cc == 'd'.
11) AZ X = AZ(0,2.0,'z');
12) // Создается безымянный объект, в
13) //  котором ii == 5, ee == 10,
14) //  cc == 'z', и копируется в объект X.
```

Перечисленные особенности конструкторов, соглашения о статусах доступа компонентов и новое понятие "деструктор" иллюстрирует следующее определение класса "символьная строка":

```
1) //STROKA.CPP - файл с определением класса
   "символьная
2) //строка"
3) #include <string.h>
4) // Для библиотечных строковых функций.
5) #include <iostream.h>
6) class stroka
7) { // Скрытые от внешнего доступа данные:
8) char *ch; // Указатель на текстовую строку.
9) int len;  // Длина текстовой строки.
10) public: // Общедоступные функции:
11) // Конструкторы объектов класса:
```

```
12) // Создает объект как новую пустую строку:
13) stroka(int N = 80):
14) // Строка не содержит информации:
15) len(0)
16) { ch = new char[N + 1];
17) // Память выделена для массива.
18) ch[0] = '\0';
19) }
20) // Создает объект по заданной строке:
21) stroka (const char *arch)
22) { len = strlen(arch);
23) ch = new char[len+1];
24) strcpy(ch,arch);
25) }
26) int& len_str(void)
27) // Возвращает ссылку на длину строки.
28) { return len; }
29) char *string(void)
30) // Возвращает указатель на строку.
31) { return ch; }
32) void display(void) // Печатает информацию о строке.
33) { cout << "\nДлина строки: " << len;
34) cout << "\nСодержимое строки: " << ch;
35) }
36) // Деструктор - освобождает память объекта:
37) ~stroka() { delete [] ch; }
38) };
```

В следующей программе создаются объекты класса stroka и выводится информация на дисплей об их компонентах:

```
1)//PR6_2.CPP - программа с классом "символьные строки".
2)#include "stroka.cpp" // Текст определения класса.
3)void main()
4){
5)stroka LAT ("Non Multa, Sed Multum!");
6)stroka RUS ("Не много, но многое!");
7)stroka CTP(20);
8)LAT.display();
9)cout << "\nВ объекте RUS: " << RUS.string();
10) CTP.display();
11) }
```

Результат выполнения программы:

```
Длина строки: 22
Содержимое строки: Non Multa, Sed Multum!
В объекте RUS: Не много, но многое!
Длина строки: 0
Содержимое строки:
```

Так как класс stroka введен с помощью служебного слова class, то элементы char *ch и int len недоступны для непосредственного обращения. Чтобы получить значение длины строки из конкретного объекта, нужно использовать общедоступную компонентную функцию len_str(). Указатель на строку, принадлежащую конкретному объекту класса stroka, возвращает функция string(). У класса stroka два конструктора - перегруженные функции, при выполнении каждой из которых динамически выделяется память для символьного массива. При вызове конструктора с параметром int N массив из N+1 элементов остается пустым, а длина строки устанавливается равной 0. При вызове с параметром char *arch длина массива и его содержание определяются уже существующей строкой, которую адресует фактический параметр, соответствующий указателю-параметру arch.

Деструкторы

Заслуживает внимания компонентная функция ~stroka(). Это деструктор. Объясним его назначение и рассмотрим его свойства.

Динамическое выделение памяти для объектов какого-либо класса создает необходимость в освобождении этой памяти при уничтожении объекта. Например, если объект некоторого класса формируется как локальный внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена системе. Желательно, чтобы освобождение памяти происходило автоматически и не требовало вмешательства программиста. Такую возможность обеспечивает специальный компонент класса - деструктор (разрушитель объектов) класса. Для него предусматривается стандартный формат:

```
1.~имя_класса() { операторы_тела_деструктора };
```

Название деструктора в C++ всегда начинается с символа тильда '~', за которым без пробелов или других разделительных знаков помещается имя класса.

Основные правила использования деструкторов следующие:

1. У деструктора не может быть параметров (даже типа void).
2. Деструктор не имеет возвращаемого значения (даже типа void).
3. Вызов деструктора выполняется неявно, автоматически, как только объект класса уничтожается.

В нашем примере в теле деструктора только один оператор, освобождающий память, выделенную для символьного массива при создании объекта класса stroka.

Итак, бывают:

- Конструктор пустой stroka(){}
- Конструктор с параметрами stroka(int);
- Инициализирующий конструктор stroka(int N = 80) : len(0)
- Конструктор копирования stroka (const stroka &obj)

Конструктор копирования

Конструктор копирования нужен нам для того, чтобы создавать «реальные» копии объектов класса, а не побитовую копию объекта. Иногда это принципиально важно. Такую «реальную» копию объекта надо создавать в нескольких случаях:

- когда мы передаем объект в какую-либо функцию в виде параметра;
- когда какая-либо функция должна вернуть объект класса в результате своей работы;
- когда мы в главной функции один объект класса инициализируем другим объектом класса.

Например, мы передаем объект в функцию в виде параметра. Функция будет работать не с самим переданным объектом, а с его побитовой копией. Допустим в конструкторе класса, при создании объекта, выделяется определенный объем памяти, а деструктор класса

эту память освобождает. Указатель побитовой копии объекта будет хранить тот же адрес памяти, что и оригинальный объект. И, когда при завершении работы функции и уничтожении побитовой копии объекта, сработает деструктор, он обязательно освободит память, которая была занята объектом-оригиналом. В придачу, еще и при завершении работы программы, деструктор сработает повторно и попытается еще раз освободить этот объем памяти, что неизбежно приведет к ошибкам программы. Та же участь постигнет и память, выделенную для указателя объекта, если будет удаляться побитовая копия возвращаемого функцией объекта, и побитовая копия при инициализации объекта класса другим объектом.

Чтобы избежать этих проблем и ошибок существует конструктор копирования. Его работа заключается в том, чтобы создать реальную копию объекта со своей личной выделенной динамической памятью. Синтаксис конструктора копирования следующий:

```
1) ИмяКласса (const имяКласса & object)
2) {
3) //код конструктора копирования
4) }
```

Рассмотрим простой пример. В нем создадим класс, который будет содержать обычный конструктор, конструктор копирования и деструктор. В этом примере будут описаны все три случая, для которых надо применять конструктор копирования. Для того, чтобы пример не был слишком большим, мы не будем заставлять конструкторы выделять память, а деструктор освобождать память. Пропишем в них только вывод определенного сообщения на экран. Таким образом, можно будет посмотреть, сколько раз сработают конструкторы и деструктор. Как вы понимаете, если бы деструктор освобождал память, то количество его вызовов не должно превышать количество вызовов конструкторов.

Пример:

```
1) #include <iostream>
2) using namespace std;
3) class SomeClass
4) {
5) int *ptr; // указатель на какой-либо участок памяти
```

```
6) public:
7) SomeClass() //  конструктор
8) {
9) cout << "\nОбычный конструктор\n";
10)  }
11)  SomeClass(const SomeClass &obj)
12)  {
13)  cout << "\nКонструктор копирования\n";
14)  }
15)  ~SomeClass()
16)  {
17)  cout << "\nДеструктор\n";
18)  }
19)  };
20)  void funcShow(SomeClass object)
21)  {
22)  cout << "\nФункция принимает объект, как
  параметр\n";
23)  }
24)  SomeClass funcReturnObject()
25)  {
26)  SomeClass object;
27)  cout <<   "\nФункция возвращает объект\n";
28)  return object;
29)  }
30)  int main()
31)  {
32)  setlocale(LC_ALL,"rus");
33)  cout << "1 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";
34)  SomeClass obj1; // создаем объект класса
35)  cout << "2 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";
36)  funcShow(obj1); // передаем объект в функцию
37)  cout << "3 - 4 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";
38)  funcReturnObject(); // эта функция возвращает объект
39)  cout << "5 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";
40)  SomeClass obj2 = obj1;
41)  // инициализация объекта при создании
42)  }
```

Определены две функции funcShow() и funcReturnObject(). Первая принимает объект в виде параметра, вторая при вызове во-первых создает новый объект (тут сработает обычный конструктор), а во-вторых возвращает объект (тут сработает конструктор копирования

т.к. при возврате объекта из функции создается его временная копия). Сравните работу конструкторов

```
1.class B { ... };
2.class A
3.{
4....
5.public:
6.// Constructor конструктор
7.A() { ... }
8.// Copy constructor - копирующий
9.A(const A& a_obj) { ... }
10. // Constructor overloading перегруженный
11. A(const B& b_obj) { ... }
12. ...
13. A& operator =(const A& a_src) { ... }
14. ...
15. };
16. // Примеры:
17. A    a1;              // конструктор вида A()
18. A    a2(a1);     // конструктор копирования A(a1)
19. A    a3 = a2;     // конструктор копирования A(a2)
20. B    b;
21. A    a4(b);     // перегруженный конструктор A(b)
22. A    a5 = b;     // перегруженный конструктор A(b)
23. a1 = a5;     // operator=(const A&) то есть
  operator=(a5)
24. a2 = b;     // Шаг1: создается временный объект
  класса a A
25. // с помощью конструктора вида A(b),
26. // Шаг2: -> operator=(const A&) то есть
  operator=(A(b))
```

ЗАДАНИЕ

Из прошлой лабораторной работы дополнить всем видами конструкторов и деструкторами классы:

1. Complex
2. Vector
3. Tiles
4. Child