Лабораторная работа 4. Конструкторы и деструкторы

1. Цель работы

Изучить возможность инициализации объектов класса с помощью конструкторов и уничтожение их с помощью деструкторов.

2. Краткие теоретические сведения

При создании объектов одной из наиболее широко используемых операций является инициализация элементов данных объекта. Единственным способом, с помощью которого вы можете обратиться к частным элементам данных, является использование функций класса. Чтобы упростить процесс инициализации элементов данных класса, C++ использует специальную функцию, называемую конструктором, которая запускается для каждого создаваемого вами объекта. Подобным образом C++ обеспечивает функцию, называемую деструктором, которая запускается при уничтожении объекта. Основными концепциями конструктора и деструктора являются:

- Конструктор представляет собой метод класса, который облегчает инициализацию элементов данных класса.
- Конструктор имеет такое же имя, как и класс.
- Конструктор не имеет возвращаемого значения.
- Каждый раз, когда программа создает переменную класса, C++ вызывает конструктор класса, если конструктор существует.
- Многие объекты могут распределять память для хранения информации; когда вы уничтожаете такой объект, C++ будет вызывать специальный деструктор, который может освобождать эту память, очищая ее после объекта.
- Деструктор имеет такое же имя, как и класс, за исключением того, что вы должны предварять его имя символом тильды (~).
- Деструктор не имеет возвращаемого значения.

Представьте конструктор как функцию, которая помогает строить (конструировать) объект. Подобно этому, деструктор представляет собой функцию, которая помогает уничтожать объект. Деструктор обычно используется, если при уничтожении объекта нужно освободить память, которую занимал объект.

Создание простого конструктора

Конструктор представляет собой метод класса, который имеет такое же имя, как и класс. Например, если вы используете класс с именем employee, конструктор также будет иметь имя employee. Конструктор не возвращает никакого значения, несмотря на то, что он не объявляется как void.

```
class employee
{
public:
  employee(char *, long, float); //Конструктор
  void show_employee(void);
  int change_salary(float);
  long get_id(void);
```

```
private:
  char name [64];
  long employee_id;
  float salary;
};

employee::employee(char *name, long employee_id, float salary)
{
  strcpy(employee::name, name) ;
  employee::employee_id = employee_id;
  if (salary < 50000.0)
  employee::salary = salary;
  else // Недопустимый оклад
  employee::salary = 0.0;
}


void main(void)
{
  employee worker("Happy Jamsa", 101, 10101.0);
  worker.show_employee();
}
```

Обратите внимание, что за объявлением объекта worker следуют круглые скобки и начальные значения, как и при вызове функции. Когда вы используете конструктор с параметрами, передавайте ему параметры при объявлении объекта:

```
employee worker("Happy Jamsa", 101, 10101.0);
```

Конструкторы и параметры по умолчанию

C++ позволяет указывать значения по умолчанию для параметров функции. Если пользователь не указывает каких-либо параметров, функция будет использовать значения по умолчанию. Конструктор не является исключением. Например, следующий конструктор employee использует по умолчанию значение оклада равным 10000.0, если программа не указывает оклад при создании объекта. Однако программа должна указать имя служащего и его номер:

```
employee::employee(char *name, long employee_id, float salary = 10000.00)
{
  strcpy(employee::name, name);
  employee::employee_id = employee_id;
  if (salary < 50000.0)
    employee::salary = salary;
  else // Недопустимый оклад
    employee::salary = 0.0;
}
```

Перегрузка конструкторов

C++ позволяет вашим программам перегружать определения функций, указывая альтернативные функции для других типов параметров. C++ позволяет вам также перегружать конструкторы. Следующая программа перегружает конструктор employee. Первый конструктор требует, чтобы программа указывала имя служащего, номер служащего и оклад. Второй конструктор запрашивает пользователя ввести требуемый оклад, если программа не указывает его:

```
#include <iostream.h>
#include <string.h>

class employee
{
public:
  employee(char *, long, float);
  employee(char *, long);
  void show_employee(void);
  int change_salary(float) ;
  long get_id(void);
private:
  char name [64];
  long employee_id;
  float salary;
};

employee::employee(char *name, long employee_id, float salary)
 {
  strcpy(employee::name, name);
  employee::employee_id = employee_id;
  if (salary < 50000.0) employee::salary = salary;
  else // Недопустимый оклад
    employee::salary = 0.0;
}

employee::employee(char *name, long employee_id)
{
  strcpy(employee::name, name);
  employee::employee_id = employee_id;
  do
  {
   cout << "Введите оклад для " << name << " меньше $50000: ";
   cin >> employee::salary;
  }
  while (salary >= 50000.0);
```

```
}

void employee::show_employee(void)
{
  cout << "Служащий: " << name << endl;
  cout << "Номер служащего: " << employee_id << endl;
  cout << "Оклад: " << salary << endl;
}

void main(void)
{
  employee worker("Happy Jamsa", 101, 10101.0);
  employee manager("Jane Doe", 102);
  worker.show_employee();
  manager.show_employee();
}
```

Если вы откомпилируете и запустите эту программу, на вашем экране появится запрос ввести оклад для Jane Doe. Когда вы введете оклад, программа отобразит информацию об обоих служащих.

Конструктор копирования

Конструктор копирования – это специальный конструктор, который позволяет получить идентичный заданному объект. То есть, с помощью конструктора копирования можно получить копию уже существующего объекта. Конструктор копирования еще называется инициализатором копии (copy initializer). Конструктор копирования должен получать входным параметром константную ссылку (**&**) на объект такого же класса.

(Параметр лучше принимать по ссылке, потому что при передаче по значению компилятор будет создавать копию объекта, а для создания копии объекта будет вызываться конструктор копирования, что приведет бесконечной рекурсии.)

Конструктор копирования необходимо использовать в тех классах, где осуществляется *динамическое выделение памяти* для данных. Другими словами, если в классе есть указатель, для которого память выделяется динамически с помощью оператора new (или других функций), то такой класс обязательно должен иметь конструктор копирования. В противном случае, в программе будут возникать проблемы, связанные с недостатками **побитового копирования**. Кроме того, такой класс должен содержать операторную функцию operator=(), перегружающую оператор копирования.

Если в классе нет динамического выделения памяти для данных, то конструктор копирования можно не использовать. В этом случае побитового копирования (по умолчанию) достаточно для корректной работы класса. Исключение, если при инициализации объекта другим объектом нужно установить некоторые специальные условия копирования.

Важно:

Если в классе не объявлен конструктор копирования, то используется конструктор копирования, который автоматически генерируется компилятором. Этот

конструктор копирования реализует побитовое копирование для получения копии объекта.

Побитовое копирование есть приемлемым для классов, в которых нет динамического выделения памяти. Однако, если в классе есть динамическое выделение памяти (класс использует указатели), то побитовое копирование приведет к тому, что указатели обоих объектов будут указывать на один и тот же участок памяти. А это ошибка.

Поэтому если в программе требуется использование копии объектов с динамической памятью, то нужно написать конструктор копирования. В остальных случаях нет надобности писать его программный код.

Конструктор копирования вызывается в случаях, когда нужно получить полную копию объекта. В C++ полная копия объекта нужна в трех случаях.

Случай 1. В момент объявления нового объекта и его инициализации данными другого объекта с помощью оператора =. Следующий фрагмент кода демонстрирует данную ситуацию для некоторого класса ClassName

```
// объявление экземпляра (объекта) класса ClassName
ClassName obj1;

// объявление объекта obj2 с одновременной инициализацией данными объекта obj2
ClassName obj2 = obj1;  // вызывается конструктор копирования
```

В этом случае нужно скопировать данные из объекта obj1 в объект obj2. То есть, нужно создать копию объекта obj1 так, чтобы этот объект мог в дальнейшем корректно использоваться в программе.

Пример:

Пусть задан класс CMyPoint, описывающий точку на координатной плоскости. В классе объявляются конструктор по умолчанию, конструктор с параметрами, конструктор копирования

```
// класс CMyPoint
class CMyPoint
{
     int x, y;

     public:
     CMyPoint(void);  // конструктор класса по умолчанию
     CMyPoint(int nx, int ny);  // конструктор класса с двумя параметрами
     CMyPoint(const CMyPoint & ref_Point);  // конструктор копирования

   // методы доступа - реализованы в классе
     int GetX(void) { return x; }
     int GetY(void) { return y; }
};

// реализация конструкторов (методов) класса
// конструктор класса CMyPoint
CMyPoint::CMyPoint(void)
{
```

```
    x = y = 0;
}

// конструктор класса CMyPoint с двумя параметрами
CMyPoint::CMyPoint(int nx, int ny)
{
    x = nx;
    y = ny;
}

// конструктор копирования класса CMyPoint
// передается ссылка на CMyPoint
CMyPoint::CMyPoint(const CMyPoint & ref_Point)
{
  // копирование данных из одного объекта в другой
    x = ref_Point.x;
    y = ref_Point.y;
}
```

Использования конструктора копирования в некотором программном коде:

```
// демонстрация использования конструктора копирования
CMyPoint p1(5, 8); // создание объекта p1
CMyPoint p2; // создание объекта p2 - вызывается конструктор по умолчанию

// проверка
int d;
d = p1.GetX(); // d = 5
d = p2.GetX(); // d = 0

p2 = p1; // побитовое копирование, конструктор копирования не вызывается
d = p2.GetX(); // d = 5 - данные были скопированы, но без использования конструктора
копирования

// код, который вызывает конструктор копирования
CMyPoint p3 = p1; // инициализация объекта => вызывается конструктор копирования
d = p3.GetX(); // d = 5
```

Случай 2. Когда нужно передать объект в функцию как параметр-значение. В этом случае создается полная копия объекта.

Пример 1:

В примере передается объект класса CMyPoint (см. пример случая 1) в функцию GetLength(), которая вычисляет расстояние от точки CMyPoint к началу координат. Текст функции:

```
// функция, которая вычисляет расстояние от точки до начала координат
// точка есть входящим параметром
double GetLength(CMyPoint mp)
{
    double length;
    int tx, ty;

    tx = mp.GetX();
    ty = mp.GetY();
    length = Math::Sqrt(tx*tx + ty*ty);
```

```
    return length;
}
```

Пример 2:

```
CMyPoint p1(5,5);  // объявить экземпляр класса CMyPoint
double len;

// передача точки p1 в функцию, вызывается конструктор копирования, len = 7,07...
len = GetLength(p1);
```

Случай 3. Когда нужно вернуть объект из функции по значению. В этом случае также создается полная копия объекта.

Пример:

Реализовать функцию GetCenterPoint(), которая возвращает точку, которая есть серединой отрезка, проведенного между точкой CMyPoint и началом координат.

Объявление класса см. в случае 1.

Реализация двух вариантов функций GetCenterPoint() и GetCenterPoint2().

```
// функция, которая возвращает середину отрезка
CMyPoint GetCenterPoint(CMyPoint mp)
{
    int tx, ty;
    tx = mp.GetX() / 2;
    ty = mp.GetY() / 2;

  // возврат из функции, конструктор копирования не вызывается,
  // вместо него вызывается конструктор с двумя параметрами
    return CMyPoint(tx, ty);
}

// функция, которая возвращает середину отрезка, заданного точками
CMyPoint GetCenterPoint2(int x, int y)
{
    CMyPoint mp(x/2, y/2);

    // создается временный объект, который инициализируется значением mp,
  // в результате вызывается конструктор копирования
  return mp;

  // в этом случае конструктор копирования не вызывается
    // return CMyPoint(x/2,y/2);
}
```

Использование этих функций

```
CMyPoint mp(18,-8);

CMyPoint mpC;

// вызывается конструктор копирования при передаче mp в функцию
mpC = GetCenterPoint(mp);

int cx, cy;
cx = mpC.GetX();  // cx = 9
```

```
cy = mpC.GetY(); // cy = -4

mpC = GetCenterPoint2(-9, 13); // вызывается конструктор копирования
cx = mpC.GetX(); // cx = -4
cy = mpC.GetY(); // cy = 6
```

В первом варианте GetCenterPoint() конструктор копирования вызывается только при передаче параметра mp по значению. При возврате из функции GetCenterPoint() с помощью оператора return, конструктор копирования не вызывается. Вместо него вызывается конструктор с двумя параметрами, объявленными в классе.

Во втором варианте GetCenterPoint2() конструктор копирования вызывается при возвращении из функции оператором return. В операторе return создается временный объект класса CMyPoint, который сразу инициализируется значением mp, в результате этого вызывается конструктор копирования.

Конструктор копирования необходимо использовать в тех классах, где осуществляется *динамическое выделение памяти* для данных. Другими словами, если в классе есть указатель, для которого память выделяется динамически с помощью оператора new (или других функций), то такой класс обязательно должен иметь конструктор копирования. В противном случае, в программе будут возникать проблемы, связанные с недостатками **побитового копирования**. Кроме того, такой класс должен содержать операторную функцию operator=(), перегружающую оператор копирования.

Если в классе нет динамического выделения памяти для данных, то конструктор копирования можно не использовать. В этом случае побитового копирования (по умолчанию) достаточно для корректной работы класса. Исключение, если при инициализации объекта другим объектом нужно установить некоторые специальные условия копирования.

```
#include <iostream>

class Person
{
    std::string name;
    unsigned age;
public:
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
    Person(const Person &p)
    {
        name = p.name;
```

```
        age = p.age + 1;    // для примера
    }
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person tomas{tom};    // создаем объект tomas на основе объекта tom
    tomas.print();      // Name: Tom       Age: 39
}
```

Представление о деструкторе

Деструктор автоматически запускается каждый раз, когда программа уничтожает объект.

При завершении программ C++ уничтожает объекты. Если определен деструктор, C++ будет автоматически вызывать деструктор для каждого объекта, когда программа завершается (т.е. когда объекты уничтожаются). Подобно конструктору, деструктор имеет такое же имя, как и класс объекта. Однако в случае деструктора его имя предваряется символом тильды (~), как показано ниже:

```
~class_name (void) //---------->указывает деструктор
{
  // Операторы деструктора
}
```

В отличие от конструктора деструктору параметры не передаются. Деструктор для класса employee:

```
void employee::-employee(void)
{
  cout << "Уничтожение объекта для " << name << endl;
}
```

В данном случае деструктор просто выводит на ваш экран сообщение о том, что C++ уничтожает объект. Программа автоматически вызывает деструктор, без какого-либо явного вызова функции деструктора.

Что нужно знать?

Конструкторы и деструкторы представляют собой специальные функции класса, которые программа автоматически вызывает при создании или уничтожении объекта. Большинство программ используют конструктор для инициализации элементов данных класса.

Основные концепции:

- Конструктор представляет собой специальную функцию, которую ваша программа автоматически вызывает каждый раз при создании объекта. Конструктор имеет такое же имя, как и класс объекта.
- Конструктор не имеет возвращаемого значения, но вы не указываете ему тип void. Вместо этого вы просто не указываете возвращаемое значение вообще.
- Когда ваша программа создает объект, она может передать параметры конструктору во время объявления объекта.
- C++ позволяет вам перегружать конструкторы и разрешает использовать значения по умолчанию для параметров.
- Деструктор представляет собой специальную функцию, которую ваша программа вызывает автоматически каждый раз при уничтожении объекта. Деструктор имеет такое же имя, как и класс объекта, но его имя предваряется символом тильды (~)

3. **Индивидуальное задание (5 баллов)**

Требования к выполнению:

- пользовательский класс **MyString** должен содержать необходимые элементы-данные, которые создаются в динамической области памяти,
- конструкторы (без параметров, с параметрами, копирования) для создания строк: **MyString (…)**,
- деструктор: **~MyString(),**
- метод ввода исходной строки: **set()**,
- метод изменения исходной строки согласно варианту (исходная и измененная строка должны сохраняться в файле): **update()**,
- метод вывода на экран: **print(…)**,
- каждый вызов методов (в том числе конструкторов и деструктора) сопровождается выдачей соответствующего сообщения,
- код методов – вне пространства определения класса.

Написать демонстрационную программу, в которой показать использование конструкторов и деструктора для объектов созданного класса.

Варианты:

1. Длина L нечетная, то удаляется символ, стоящий посередине строки;
2. Длина L четная, то удаляются 2 первых и 2 последних символа;
3. Длина L кратна 2-м, то удаляются все цифры, которые делятся на 2;
4. Длина L кратна 3-м, то удаляются все цифры, делящиеся на 3;
5. Длина L >10, то удаляются все цифры;
6. Длина L >15, то удаляются все a..z;
7. Длина L=10, то удаляются все A..Z;
8. Длина L кратна 4-м, то первая часть строки меняется местами со второй;
9. Длина L кратна 5-и, то подсчитывается количество скобок всех видов;
10. Длина L >5-и, то выделяется подстрока до первого пробела;
11. Длина L >6-и, то выделяется подстрока { } скобках;
12. Длина L >10-и, то удаляется подстрока в [] скобках;
13. Длина L >12-и, то удаляется подстрока до первой (скобки;
14. Длина L кратна 4-м, то выделяется подстрока после последнего пробела;
15. Длина L >5, то удаляются все точки.

16. Длина L четная, то выделяется подстрока до первого пробела
17. Длина L четная, то удаляется подстрока до первого пробела
18. Длина L четная, то выделяется подстрока со второго пробела
19. Длина L нечетная, то выделяется подстрока после первого пробела
20. Длина L нечетная, то удаляется подстрока со второго пробела
21. Длина L кратна 3, то удаляется каждый 3-й символ
22. Длина L четная, то удаляется каждый 2-й символ
23. Длина L нечетная, то
24. Длина L четная, то выделяется подстрока до последнего пробела
25. Длина L нечетная, то выделяется подстрока от последней цифры

4. Контрольные вопросы

1. Что такое конструктор
2. Как задается имя конструктора?
3. Может ли класс иметь более одного конструктора?
4. Что такое деструктор?
5. В чем состоит преимущество определения конструктора со списком инициализации элементов?
6. Какие виды конструкторов создаются по умолчанию?
7. В каком порядке инициализируются поля в классе?
8. Какая ошибка в следующей реализации конструктора?

```
…
typedef unsigned int dlina;
const dlina n=30;

class Mouse{
    dlina rost;
protected :
    char * name;
public :
    Mouse (const char * a="None"):rost(1)
      {name=new char[n]; strcpy(name,a);}
    ~Mouse (){rost=0;}
    Mouse (Mouse & A):rost(A.rost), name(A.name){}
…
};
```

9. Может ли деструктор иметь аргументы?
10. Какая ошибка в следующей реализации деструктора?

```
…
typedef unsigned int dlina;
const dlina n=30;

class Mouse{
    dlina rost;
protected :
    char * name;
public :
    Mouse (const char * a="None"):rost(1)
      {name=new char[n]; strcpy(name,a);}
    ~Mouse (){rost=0;}
    Mouse (Mouse & A):rost(A.rost), name(A.name){}
…
};
```