Конструктор и деструктор
(продолжение)

Деструктор

- Деструкторы не могут быть перегружены.
- У класса может быть только один деструктор.
- Если вы забудете реализовать деструктор, компилятор создаст и вызовет фиктивный деструктор, т.е. пустой деструктор, который не осуществляет никакого освобождения зарезервированной динамической памяти.

Деструктор

- Деструкторы классов, в которых используется операция delete, становятся необходимыми, когда в конструкторах классов применяется операция new.

Конструктор

- Это специальная функция-член класса, которая вызывается всякий раз при создании объекта данного класса

- Имеет то же имя, что и класс, но благодаря возможностям перегрузки функций, существует возможность создавать более одного конструктора с одним и тем же именем и разным набором аргументов

Конструктор

- Конструктор не имеет объявленного типа
- Обычно конструктор используется для инициализации членов объекта класса.

  Ваша инициализация должна соответствовать списку аргументов конструктора.

Конструктор

Предположим, что класс Bozo имеет следующий прототип для конструктора:

```
Bozo(const char * fname, const char * lname);      // прототип конструктора
```

Конструктор

В этом случае его можно использовать для инициализации объекта следующим образом:

```
Bozo bozetta = Bozo("Bozetta", "Biggens");      // основная форма
Bozo fufu("Fufu", "O'Dweeb");                   // сокращенная форма
Bozo *pc = new Bozo("Popo", "Le Peu");          // динамический объект
```

Конструктор

В C++11 можно взамен применять списковую инициализацию:

```
Bozo bozetta = {"Bozetta", "Biggens"};          // C++11
Bozo fufu{"Fufu", "O'Dweeb"};                   // C++11
Bozo *pc = new Bozo{"Popo", "Le Peu"};          // C++11
```

Конструктор

Когда конструктор имеет только один аргумент, он вызывается в случае инициализации объекта значением, которое имеет тот же тип, что и аргумент конструктора.

Например, предположим, что существует следующий прототип конструктора:

```
Bozo(int age) ;
```

Конструктор

Тогда в коде можно использовать любую из следующих форм инициализации объекта:

```
Bozo dribble = Bozo(44);  // первичная форма
Bozo roon(66);            // вторичная форма
Bozo tubby = 32;      // специальная форма для конструктора с одним аргументом
```

Конструктор

- **Внимание!**
  Конструктор, который принимает один аргумент, позволяет использовать синтаксис присваивания для инициализации объекта значением:

  ***имяКласса объект = значение*;**

  Эта возможность может привести к возникновению проблем, но ее можно заблокировать.

Конструктор

- Конструктор по умолчанию не имеет аргументов и используется, когда вы создаете объект без явной его инициализации.

Конструктор

- Если вы не предоставляете ни одного конструктора, то компилятор создаст конструктор по умолчанию самостоятельно.

  В противном случае вы обязаны определить собственный конструктор по умолчанию. Он может либо не иметь аргументов, либо предусматривать значения по умолчанию для всех аргументов:

```
Bozo();                                  // прототип конструктора по умолчанию
Bistro(const char *s = "Chez Zero");     // значение по умолчанию для класса
                                                                    Bistro
```

Конструктор

- Программа использует конструкторы по умолчанию для неинициализированных объектов:

```
Bozo bibi;                  // используется конструктор по умолчанию
Bozo *pb = new Bozo;        // используется конструктор по умолчанию
```

Конструктор копий
Поверхностное копирование и связанные с ним проблемы

- Такие классы, как MyString ,например, представленный в листинге прошлой лекции, содержат в качестве члена указатель, который указывает на область в динамически распределяемой памяти, зарезервированную в конструкторе при помощи оператора new и освобождаемую в деструкторе с использованием оператора delete [ ].

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

```
1: #include <iostream>
2: using namespace std;
3:
4: class MyString
5: {
6: private:
7:     char* Buffer;
8:
9: public:
```

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

```
 9:      // Конструктор
10:      MyString(const char* InitialInput)
11:      {
12:          if(InitialInput != NULL)
13:          {
14:              Buffer = new char [strlen(InitialInput) + 1];
15:              strcpy(Buffer, InitialInput);
16:          }
17:          else
18:              Buffer = NULL;
19:      }
20:
```

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

```
21:     // Деструктор
22:     ~MyString()
23:     {
24:         cout << "Invoking destructor, clearing up" << endl;
25:         if (Buffer != NULL)
26:             delete [] Buffer;
27:     }
28:
29:     int GetLength()
30:     {
31:         return strlen(Buffer);
32:     }
33:
34:     const char* GetString()
35:     {
36:         return Buffer;
37:     }
38: };
```

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

```
39:
40: void UseMyString(MyString Input)
41: {
42:     cout << "String buffer in MyString is " << Input.GetLength();
43:     cout << " characters long" << endl;
44:
45:     cout << "Buffer contains: " << Input.GetString() << endl;
46:     return;
47: }
48:
49: int main()
50: {
51:     MyString SayHello("Hello from String Class");
52:
53:     // Передать SayHello функции как параметр
54:     UseMyString(SayHello);
55:
56:     return 0;
57: }
```

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

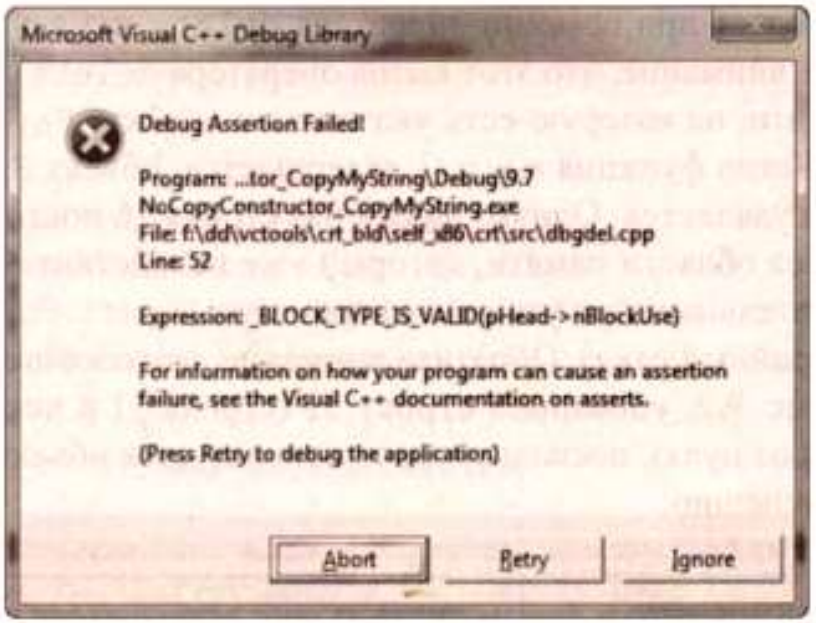
Результат

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
```

<отказ, как можно заметить на рис.

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

РИС.1 Снимок экрана аварийного отказа, произошедшего при выполнении кода листинга 1



ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Анализ

Почему класс, который только что прекрасно работал в листинге, привел к отказу? Единственное различие между листингами в том, что задача использования объекта S a y H e l l o класса M y S t r i n g , созданного в функции m a i n (), была
делегирована функции U s e M y S t r i n g (), вызываемой в строке 54. Делегирование работы этой функции привело к тому, что объект S a y H e l l o в функции m a i n () копируется в аргумент параметра I n p u t, используемого в функции U s e M y S t r i n g ( ) .

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Анализ

Эта копия создается компилятором, поскольку функция была объявлена как получающая параметр I n p u t по значению, а не по ссылке. Компилятор создает двоичную копию простых старых данных, таких, как целые числа, символы и указатели. Таким образом, значение, содержащееся в указателе
S a y H e l l o . B u f f e r , было просто скопировано в параметр I n p u t, т.е. он теперь указывает на ту же область памяти, что и I n p u t . B u f f e r (рис. 2)

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

РИС. 2 Поверхностное копирование объекта **SayHello** в параметр **Input** при вызове функции **UseMyString ()**

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Двоичная копия не обеспечивает ***глубокого копирования*** (deep copy) и не распространяется на указываемую область памяти, поэтому теперь есть два объекта класса M y S t r i n g , указывающих на ту же область в памяти. Таким образом, по завершении работы функции U s e M y S t r i n g () переменная I n p u t выходит из области видимости и удаляется. При этом вызывается деструктор класса M y S t r i n g , и его код в строке 26

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Двоичная копия не обеспечивает ***глубокого копирования*** (deep copy) и не распространяется на указываемую область памяти, поэтому теперь есть два объекта класса M y S t r i n g , указывающих на ту же область в памяти. Таким образом, по завершении работы функции

U s e M y S t r i n g () переменная I n p u t выходит из области видимости и удаляется. При этом вызывается деструктор класса M y S t r i n g , и его код в строке 26 листинга 1 освобождает при помощи оператора d e l e t e память, зарезервированную для буфера.

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Обратите внимание, что этот вызов оператора d e l e t e объявляет недействительной область памяти, на которую есть указатель в объекте

S a y H e l l o , находящемся в функции m a i n (). Когда функция m a i n () завершается, объект S a y H e l l o выходит из области видимости и удаляется. Однако на сей раз строка 26 повторно вызывает оператор

d e l e t e для адреса области памяти, который уже недействителен (уже освобожден и объявлен недействительны м при удалении параметра

I n p u t).

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Результатом повторного удаления и будет аварийный отказ.

Обратите внимание, что сообщение режима отладки,
представленное на рис. 1, упоминает строку 52 (строка 51 в листинге, ведь строки в книге отсчитываются от нуля), поскольку здесь используется объект S a y H e l l o , который не был освобожден успешно.

ЛИСТИНГ 1. Проблема передачи объекта класса, такого, как MyString, по значению

Компилятор в данном случае не смог автоматически обеспечить глубокое копирование, поскольку на момент компиляции ему неизвестно ни количество байтов, на которые указывает указатель-член **MyString:: Buffer,** ни характер резервирования.

Обеспечение глубокого копирования с использованием конструктора копий

- *Конструктор копий* (copy constructor) — это специальный перегруженный конструктор, который должен предоставить разработчик класса.

Обеспечение глубокого копирования с использованием конструктора копий

- Компилятор использует конструктор копий каждый раз, когда объект класса копируется, включая передачу объекта в функцию по значению.

Обеспечение глубокого копирования с использованием конструктора копий

- Конструктор копий для класса MyString можно объявить так:

```
class MyString
{
    MyString(const MyString& CopySource); // конструктор копий
};

MyString::MyString(const MyString& CopySource)
{
    // Код реализации конструктора копий
}
```

Обеспечение глубокого копирования с использованием конструктора копий

- Таким образом, конструктор копий получает как параметр по ссылке объект того же класса. Этот параметр — псевдоним исходного объекта, используемый при написании собственного специального кода копирования (где вы гарантировали бы глубокое копирование всех буферов оригинала), как показано в листинге 9.

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
0: #include <iostream>
1: using namespace std;
2:
3: class MyString
4: {
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
private:
    char* Buffer;

public:
    // конструктор
    MyString(const char* InitialInput)
    {
        cout << "Constructor: creating new MyString" << endl;
        if(InitialInput != NULL)
        {
            Buffer = new char [strlen(InitialInput) + 1];
            strcpy(Buffer, InitialInput);

            // Отображение адреса области памяти локального буфера
            cout << "Buffer points to: 0x" << hex;
            cout << (unsigned int*)Buffer << endl;
        }
        else
            Buffer = NULL;
    }
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
// Конструктор копий
MyString(const MyString& CopySource)
{
    cout << "Copy constructor: copying from MyString" << endl;

    if(CopySource.Buffer != NULL)
    {
        // гарантировать глубокое копирование, создав сначала
        // собственный буфер
        Buffer = new char [strlen(CopySource.Buffer) + 1];

        // копирование из оригинала в локальный буфер
        strcpy(Buffer, CopySource.Buffer);

        // Отображение адреса области памяти локального буфера
        cout << "Buffer points to: 0x" << hex;
        cout << (unsigned int*)Buffer << endl;
    }
    else
        Buffer = NULL;
}
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
        // Деструктор
        ~MyString()
        {
            cout << "Invoking destructor, clearing up" << endl;
            if (Buffer != NULL)
                delete [] Buffer;
        }

        int GetLength()
        {
            return strlen(Buffer);

58:     }
59:
60:     const char* GetString()
61:     {
62:         return Buffer;
63:     }
64: };
65:
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
66: void UseMyString(MyString Input)
67: {
68:     cout << "String buffer in MyString is " << Input.GetLength();
69:     cout << " characters long" << endl;
70:
71:     cout << "Buffer contains: " << Input.GetString() << endl;
72:     return;
73: }
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
74:
75: int main()
76: {
77:     MyString SayHello("Hello from String Class");
78:
79:     // Передача SayHello по значению (с копированием)
80:     UseMyString(SayHello);
81:
82:     return 0;
83: }
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

Результат

```
Constructor: creating new MyString
Buffer points to: 0x0040DA68
Copy constructor: copying from MyString
Buffer points to: 0x0040DAF8
String buffer in MyString is 17 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
```

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

Анализ

Для начала сосредоточимся на функции main( ), которая создает объект S a y H e l l o в строке 77.

Создание объекта S a y H e l l o приводит к отображению первой строки вывода, оператор cout которой расположен в строке 12 конструктора M y S t r i n g .

Для удобства конструктор отображает также адрес области памяти, на которую указывает B u f f e r .

Затем, в строке 80, функция main() передает объект S a y H e l l o по значению функции U s e M y S t r i n g (), что автоматически приводит к вызову конструктора копий, как свидетельствует вывод.

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

Анализ

Код в конструкторе копий очень похож на таковой в конструкторе.

Основная идея та же — выяснить длину строки в стиле C, которая содержится в буфере оригинала (строка 34), зарезервировать достаточно памяти в собственном экземпляре.

ЛИСТИНГ 9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

ВНИМАНИЕ!

Использование ключевого слова const в объявлении конструктора копий гарантирует, что он не изменит то, на что указывает исходный объект.

Кроме того, параметр должен передаваться в конструктор копий по ссылке.

Если бы он не передавался по ссылке, то конструктор сам вызвал бы копирование значения, приведя таким образом к поверхностному копированию данных оригинала, а именно этого мы и намеревались избежать.

Обеспечение глубокого копирования с использованием конструктора копий

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Всегда создавайте конструктор копий и оператор присвоения копии, когда ваш класс содержит *простой указатель* (raw pointer) (например, `char*` и т.п.)	**Не используйте** простой указатель как член класса, если в этом нет абсолютно неизбежной необходимости
Всегда создавайте конструктор копий с константным параметром ссылки на оригинал	
Используйте как члены такие классы строк, как `std::string`, и классы интеллектуальных указателей вместо простых указателей, поскольку они реализуют конструкторы копий и экономят ваше время	

Обеспечение глубокого копирования с использованием конструктора копий

- **ПРИМЕЧАНИЕ**

  Класс **Mystring** с простым указателем **char* Buffer** в качестве члена используется как пример для объяснения необходимости конструктора копий.

  Если вам нужно создать класс, который должен содержать строковые данные для хранения имен, например, то используйте класс **std::string,** а не **char*,** ведь при отсутствии простых указателей даже конструктор копий не нужен. Дело в том, что стандартный конструктор копий, вставленный компилятором, гарантирует вызов всех доступных конструкторов копий членов класса объектов, таких как **std::string.**

Конструкторы копирования

- Конструктор копирования служит для копирования некоторого объекта в создаваемый объект.

Конструкторы копирования

- Другими словами, он используется во время инициализации — в том числе при передаче функции аргументов по значению — но не во время обычного присваивания.

Конструкторы копирования

- Конструктор копирования для класса обычно имеет следующий прототип:

***Имя_класса*(const *Имя_класса* &);**

Конструкторы копирования

- Обратите внимание, что в качестве аргумента он принимает константную ссылку на объект класса.

  Например, конструктор копирования для класса **String** будет выглядеть так:

  StringBad(const StringBad &);

Когда используется конструктор копирования

Конструктор копирования вызывается всякий раз, когда создается новый объект, и для его инициализации берется значение существующего объекта того же типа.

Когда используется конструктор копирования

Это происходит в нескольких ситуациях. Наиболее очевидный случай — когда новый объект явно инициализируется существующим объектом.

Когда используется конструктор копирования

Например, если **motto** является объектом **StringBad,** то следующие четыре объявления вызывают конструктор копирования:

```
StringBad ditto(motto);          // вызывает StringBad(const StringBad &)
StringBad metoo = motto;         // вызывает StringBad(const StringBad &)
StringBad also = StringBad(motto); // вызывает StringBad(const StringBad &)
```

```
StringBad * pStringBad = new StringBad(motto);
     // вызывает StringBad(const StringBad &)
```

Когда используется конструктор копирования

В зависимости от реализации, два объявления в середине могут использовать конструктор копирования либо непосредственно для создания объектов **metoo** и **also,** либо для генерирования временных объектов, содержимое которых затем присваивается объектам **metoo** и **also.** Приведенный выше код инициализирует анонимный объект значением **motto** и присваивает адрес нового объекта указателю **pstring.**

Когда используется конструктор копирования

Менее очевидно то, что компилятор использует конструктор копирования при каждом генерировании копии объекта в программе.

В частности, он применяется, когда функция передает объект по значению или когда функция возвращает объект. Ведь передача по значению подразумевает создание копии исходной переменной.

Компилятор также использует конструктор копирования при генерировании временных объектов.

Когда используется конструктор копирования

Различные компиляторы могут вести себя по-разному при создании временных объектов, но все они вызывают конструктор копирования при передаче объектов по значению и при их возврате.

Что делает конструктор копирования по умолчанию

Конструктор копирования по умолчанию выполняет ***почленное копирование*** нестатических членов, также иногда называемое ***поверхностным копированием.***

Каждый член копируется по значению.

Что делает конструктор копирования по умолчанию

Если член сам является объектом класса, для копирования одного объекта-члена в другой используется конструктор копирования этого класса.

Конструктор копирования

Совет

Если в классе имеются статические данные-члены, значение которых изменяется при создании новых объектов, должен быть предусмотрен явный конструктор копирования, который принимает это во внимание.

Конструктор копирования

- Для устранения проблем в структуре класса следует выполнять ***глубокое копирование***

Конструктор копирования

- **Внимание!**
  Если класс содержит члены, которые являются указателями, инициализированными операцией **new,** потребуется определить конструктор копирования, копирующий данные, на которые указывают указатели, а не сами указатели. Это называется *глубоким копированием.* Альтернативная форма копирования *(почленное* или *поверхностное копирование)* просто копирует значения указателей. Поверхностная копия — это только “наружное соскабливание" информации указателя для копирования, а не “глубокая добыча”, требующая копирования конструкций, на которые указывают указатели.

Указатель this

- Указатель t h i s — это важнейшая концепция языка C++; зарезервированное ключевое слово

  t h i s применимо в рамках класса, который содержит адрес объекта.

Указатель this

- Другими словами, значение указателя

  t h i s — это & o b j e c t.

Указатель this

- В пределах метода класса, когда вы вызываете другой метод, компилятор неявно передает ему в вызове указатель t h i s как невидимый параметр:

```
class Human
{
private:
    // ... объявления закрытых членов
    void Talk (string Statement)
    {
        cout << Statement;
    }
public:
    void IntroduceSelf()
    {
        Talk("Bla bla");
    }
};
```

Указатель this

- Здесь представлен метод

  I n t r o d u c e S e l f (); использующий закрытый член T a l k () для вывода на экран выражения. В действительности компилятор внедряет указатель t h i s в вызов метода T a l k (), который выглядит как T a l k (t h i s , " B l a b l a ").

Указатель this

- С точки зрения программирования у указателя t h i s не слишком много областей применения, но иногда он оказывается удобным.

  Например, у кода доступа к переменной Age в пределах функции S e t A g e( ), может быть такой вариант:

```
void SetAge(int HumansAge)
{
    this->Age = HumansAge; // то же, что и Age = HumansAge
}
```

Указатель this

- **ПРИМЕЧАНИЕ**

  Указатель **this** не передается в статические методы класса. Как и статические функции, они не связаны с экземпляром класса. Статические методы совместно используются всеми экземплярами.

  Если хотите использовать переменные экземпляра в статической функции, явно объявите параметр, используемый вызывающей стороной, для передачи указателя **this** как аргумента.

Указатель this

На заметку!
Каждая функция-член, включая конструкторы и деструкторы, имеет указатель this.
Специфическим свойством является то, что он указывает на вызывающий объект.
Если метод нуждается в получении ссылки на вызвавший объект в целом, он может использовать * this.
Применение квалификатора const после скобок с аргументами заставляет трактовать this как указатель на const; в этом случае вы не можете использовать this для изменения значений объекта.

Указатель this

Однако то, что необходимо вернуть из метода — это не this, поскольку this представляет собой адрес объекта. Вам нужно вернуть сам объект, а это обозначается выражением * this. (Вспомните, что применение операции разыменования * к указателю дает значение, на которое он указывает.)