Лекция 3. Конструкторы и деструкторы

При создании объектов одной из наиболее широко используемых операций которую выполняет в программах, 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.

Конструктор представляет собой метод класса, который облегчает вашим программам инициализацию элементов данных класса.

- Конструктор имеет такое же имя, как и класс.
- Конструктор не имеет возвращаемого значения.
- Каждый раз, когда ваша 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.

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

1. Конструкторы классов

После того как класс определен и заданы объекты этого класса, как правило, возникает необходимость выполнения каких-либо действий по инициализации каждого из объектов. Под инициализацией в данном случае понимается выполнение некоторых начальных действий в программе, для того, чтобы объект мог успешно функционировать. При этом для разных классов могут понадобиться существенно различные способы инициализации. Такими действиями могут быть, например, открытие файлов, загрузка драйверов, динамический заказ дополнительной оперативной памяти, присвоение начальных значений элементам данных. Для выполнения действий такого рода можно было бы воспользоваться какой-либо специально определенной программистом функцией-членом класса, например InitObject или SetObject. Вместе с тем, это налагает на программиста дополнительные обязанности, например, записывать вызов этих функций для каждого вновь определяемого объекта. Преодолеть это неудобство в C++ довольно просто, используя конструкторы классов. Для некоторого класса конструктор - это функция, являющаяся его членом и имеющая имя, совпадающее с именем самого класса, а также не содержащая типа возвращаемого значения. Особенностью функции является ее автоматический вызов для каждого из объектов класса в тот момент, когда по естественному ходу выполнения программы встречается описание объекта:

```
class Vectors{
int A[25], B[25], C[25];
public:
Vectors ( );
void VectorsSum ( Vectors *, Vectors * );
// Другиеметоды
};
Vectors::Vectors( )
{
memset ( A, 0, 25);
```

```
memset ( B, 0, 25);
memset ( C, 0, 25);
}
voidmain( )
{
Vectors First; // В этом месте будут вызваны
Vectors Second; // конструкторы для First и Second.
// Операторы программы
}
```

Конструктор представляет собой специальную функцию, которую C++ автоматически вызывает каждый раз при создании объекта. Обычное назначение конструктора заключается в инициализации элементов данных объекта. Конструктор имеет такое же имя, как и класс. Класс с именем file использует конструктор с именем file. Вы определяете конструктор внутри своей программы так же, как и любой метод класса. Единственное различие заключается в том, что конструктор не имеет возвращаемого значения. Когда вы позже объявляете объект, вы можете передавать параметры конструктору. Одним из важных свойств конструктора является его автоматический вызов при описании любого объекта какого-либо класса, использующего конструктор, что снимает с программиста задачу своевременного отслеживания инициализации вновь вводимых объектов. В общем случае конструкторы классов могут иметь списки пара- метров, которые могут потребоваться при инициализации. При этом программист будет обязан задать список инициализации при описании каждого нового объекта, например, рассмотрим класс дат с соответствующим конструктором:

```
class Date{
int Month, Day, Year;
public:
Date(int , int, int);
void GetDate( );
};
Date::Date ( int M, int D, int Y )
{
Month = M; Day = D; Year = Y;
}
void main( )
{
Date MemDay( 10, 15, 1993 ); // Обязательная инициали-
DateNewDate = MemDay; // зация, иначе :
Date Another; // Ошибка !
// Операторы программы
}
```

Ограничением использования конструкторов является запрет использования его имени в качестве явного аргумента внутри самого этого класса:

```
class Vector{
int Vec[5];
Vector V; // Ошибка
public:
Vector ( Vector ); // Ошибка
// Другиеметоды
};
```

В данном случае определение объекта V как объекта этого же класса Vector должно было привести к бесконечному рекурсивному выделению памяти, однако транслятор с языка C++ просто выдаст соответствующее сообщение о синтаксической ошибке. 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; ваша программа может указать для него значения по умолчанию так же, как и для любой другой функции. 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. Конструкторы, не имеющие параметров, называются конструкторами по умолчанию:

```
#include "iostream.h" // для cin, cout см.следующие главы
class X{
public:
char *PointX;
X ( ) { cout << "Объявлен объект класса X!";
PointX = (char* 0);
}; // X( ) - конструктор по умолчанию
};
X NewX;
void main( )
{
cout << "Конец работы.";
}
```

Результатом работы этой программы будет:
Объявлен объект класса X!
Конец работы.

Как и другие методы класса конструкторы могут быть перегружаемыми, т.е. могут использовать несколько определений с различными списками параметров:

```
class Intg {
char Number[5];// Число можно хранить символьно
int N;// И целым типом
public:
Intg ( char *Str ) {// Конструктор для 1-го случая
if ( strlen(Str) > 5 )
cout<< "Превышение размера числа";
else strcpy( Number, Str );
};
Intg ( int L ) { N = L }; // Конструктор для второго случая
};
void main( )
{
Intg First ( "125" ); // Вызов первого конструктора
Intg Second ( 125 ); // Вызов второго конструктора
// Другие операторы программы
}
```

2. Операция ссылки

Операция ссылки "&" в базовом языке C использовалась для взятия адреса объекта (переменных, массивов, структур, функций и т.п.). BC++ расширены возможности операции ссылки. При этом появилась новая концепция ссылки в операторах объявления. Рассмотримпример.

```
int Handle;
int *New = &Handle;
int&Next = Handle;
```

В этом примере переменная Next не является указателем на тип int, а носит название ссылки на объект типа int. Эта переменная должна быть проинициализирована при ее объявлении. Далее в программе она становится некоторым синонимом объекта Handle для использования этого объекта как единого целого. В общем случае можно определить ссылки и на более сложные объекты, например, структуры или объекты классов. Для приведенного примера следующие два оператора будут эквивалентными:

```
// Ранее, например, было определено int First = 0;
*New = First; Next = First;
```

Используя ссылку для более сложных типов данных можно производить быстрое копирование объектов:

```
struct R {
char L[20];
int Numb;
};
struct R First, Second;
struct R &New = Second;
void main( )
{
First. Numb = 10;
New = First; // Скопируетсявсяструктура
}
```

Ссылки удобно использовать в качестве параметров и возвращаемых значений в функциях.

При программировании конструкторов существует специальный тип конструкторов, использующий ссылки. Такие конструкторы называются конструкторами копирования-инициализации. Например, конструктор может создавать новый объект, копируя данные из старого объекта:

```
class MyOwn {
int Leng;
public:
MyOwn ( int L ) { Leng = L };
MyOwn ( MyOwn& );
};
MyOwn::MyOwn( MyOwn& Old )
{
Leng = Old.Leng;
}
```

3. Деструкторы классов

инициализация программа конструктор деструкция

Для выполнения действий, обратных совершаемым конструкторами, т.е., например, освобождение заказанной памяти, закрытие открытых конструктором файлов и т.п., в C++ введен механизм деструкторов. Деструктор класса вызывается автоматически для каждого из объектов класса при потере его из области видимости в программе. Это происходит при выходе программы из блока, в котором определен объект класса. Существование блока легко определяется по фигурным скобкам, открывающим и закрывающим каждый из блоков. Если объект класса определен глобально, например, перед функцией main(), деструктор для этого объекта будет вызван в самом конце программы.

Если для класса X конструктор класса называется X, то его деструктор называется ~X.

Чаще всего конструкторы и деструкторы классов используют стандартные операции C++ для заказа и освобождения динамически распределяемой оперативной памяти, соответственно new и delete.

Деструктор автоматически запускается каждый раз, когда программа уничтожает объект. В следующих уроках вы узнаете, как создать списки объектов, которые увеличиваются или уменьшаются по мере выполнения программы. Чтобы создать такие динамические списки, ваша программа для хранения объектов распределяет память динамически (что вы еще не научились делать). К настоящему моменту вы можете создавать и уничтожать объекты в процессе выполнения программы. В таких случаях имеет смысл применение деструкторов.

Каждая из созданных до сих пор программ создавала объекты в самом начале своего выполнения, просто объявляя их. При завершении программ C++ уничтожал объекты. Если вы определяете деструктор внутри своей программы, 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.

Деструктор представляет собой функцию, которую C++ автоматически запускает, когда он или ваша программа уничтожает объект. Деструктор имеет такое же имя, как и класс объекта; однако вы предваряете имя деструктора символом тильды (~), например ~employee. В своей программе вы определяете деструктор точно так же, как и любой другой метод класса.

В качестве примера рассмотрим некоторый класс String:

```
#include "iostream.h"
#include "string.h"
class String {
char *QuoteString;
int StringLength;
public:
String ( char * ); // Конструктор
~String ( ); // Деструктор
};
String::String ( char *InitString )
{
QuoteString = new char[strlen(InitString)+1];
strcpy(QuoteString, InitString);
if (!QuoteString)
cout<< "Недостаточнопамяти!";
StringLength = strlen(QuoteString);
}
String::~String( ) // Освобождениепамяти
{
cout<< "Строка" << QuoteString;
delete QuoteString;
QuoteString = (char *) 0;
cout<< "Освобождена\n";
}
void main( )
{
String First("Первая строка"); // Вызов конструктора First
{
String Second("Вторая строка"); // Вызов конструктора
// для Second
// Операторы программы
} // Вызов деструктора
// для Second
// Операторы программы
} // Вызов деструктора
// для First
```

Результатом работы этой программы будет следующее сообщение:

```
СтрокаПервая строкаОсвобождена
СтрокаВторая строкаОсвобождена
```

4. Пример программы с конструкторами и деструкторами

Продолжим выполнение задания из примера предыдущей главы для объектов класса "Комплексные числа" в части программирования конструкторов и деструкторов:

```
/**********************/
/* Constructors & */
/* Destructors of */
/* class Complex */
/**********************/
/* v.25.12.2002 */
#include "iostream.h" // Для cin, cout см.последующиеглавы
class Complex {
float Re; // Действительная и
float Im; // мнимая части числа
public:
Complex ( );
Complex ( int, int );
~Complex ( );
// Функциииарифметики
void Put ( ); // Функция ввода
};
void Complex::Put ( ) // Выводнаэкран
{
cout << "Действительная часть числа: " << Re;
cout << "\nMнимая часть числа: " << Im;
}
Complex::Complex(int R, int I)
{
Re = R; Im = I;
}
Complex::Complex( )
{
Re = Im = 0;
}
Complex::~Complex( )
{
Re = 0; Im = 0;
}
void main( )
{
{
Complex a,b; // Определение объектов a,b
Complex c(12,24); // Определение объекта c
a.Put( ), b.Put( ),c.Put ( ); // Вывод на экран
}
}
```