Лекция. Классы. Указатель this

В классе СВох вы написали функцию Volume () в терминах имен членов класса внутри определения класса. Конечно, каждый объект типа СВох, созданный вами, содержит эти члены, так что в функции должен быть механизм обращения к конкретному объекту, для которого вызвана данная функция.

Когда любой метод выполняется, он автоматически получает скрытый указатель по имени this, указывающий на объект, использованный с вызовом функ­ции. Таким образом, когда во время выполнения функции Volume () осуществляется обращение к члену m\_Length, то на самом деле идет обращение к this->m\_Length, что представляет собой ссылку на член используемого объекта. Компилятор берет на себя заботу о добавлении необходимого имени указателя this к именам членов в функции.

При необходимости, вы можете использовать указатель this явно внутри метода. Так, например, вы можете вернуть указатель на текущий объект.

Явное использование указателя this

Можно добавить в класс СВох общедоступную функцию, которая будет сравнивать объемы двух объектов СВох.

//listing 43

//Явное использование this

#include <iostream.h>

class CBox //Определение класса в глобальном контексте

{

public:

//определение конструктора

CBox (double lv =1.0, double bv =1.0, double hv =1.0)

{

cout <<endl << "constructor work";

m\_Length = lv; //установить значения полей

m\_Width = bv;

m\_Height = hv;

}

//Функция для вычисления объема ящика

double Volume()

{

return m\_Length\*m\_Width\*m\_Height;

}

//Функция сравнения объемов двух ящиков возвращает true(1)

//если первый больше второгоб и false (0) - в противном случае

int Compare (CBox xBox)

{

return this -> Volume() >xBox.Volume ();

}

private:

double m\_Length; //Длина ящика

double m\_Width; //Ширина ящика

double m\_Height; //Высота ящика

};

int main ()

{

CBox match(2.2,1.1,0.5); //объявление match типа CBox

CBox cigar(8.0,5.0,1.0); //объявление cigar типа CBox

if (cigar.Compare (match))

cout <<endl

<<"match < cigar" ;

else

cout <<endl

<<" match >= cigar" ;

cout <<endl;

return 0;

}

Результат

constructor work

constructor work

match < cigar

Указатели и ссылки на объекты классов

Использование указателей, и в частности, ссылок на объекты классов - очень важно в объектно-ориентированном программировании, в частности, для спецификации параметров функций. Объекты классов могут включать в себя значительные объемы данных, поэтому использование механизма передачи по значению параметров объектов в функции может оказаться очень дорогостоящим по затратам времени и неэффективным, поскольку при этом должно выполняться копирование каждого объекта-аргумента. Существует также техника использования ссылок, которая незаменима при некоторых операциях с классами.

Указатели на объекты класса

Вы объявляете указатель на объект класса точно так же, как и любой другой указатель. Например, указатель на объект класса СВох объявляется следующим образом:

*СВох\* рВох = 0; // Объявление указателя на СВох*

Теперь вы можете использовать его для хранения адреса объекта *СВох* путем присваивания, применяя операцию получения адреса:

*рВох = &cigar; // Сохранить адрес объекта СВох cigar в рВох*

Как видите, при использовании указателя this в определении метода

Compare () вы можете вызвать функцию, применяя указатель на объект.

Например, функцию Volume () для указателя рВох можно вызвать с помощью такого оператора:

*cout <<pBox->Volume(); //Отобразить объем объекта, на который указывает рВох*

Опять-таки, здесь применяется операция непрямого обращения к члену. Это - типичная нотация, используемая большинством программистов для операций подобного типа, поэтому с настоящего момента будем применять ее повсеместно.

//listing 45

//УКАЗАТЕЛИ НА КЛАССЫ

//испытание операции непрямого доступа к полю

#include <iostream.h>

class CBox //Определение класса в глобальном контексте

{

public:

//определение конструктора

CBox (double lv = 1.0, double bv =1.0, double hv =1.0)

{

cout <<endl << "constructor work";

m\_Length = lv; //установить значения полей

m\_Width = bv;

m\_Height = hv;

}

//Функция сравнения объемов двух ящиков возвращает true(1)

//если первый больше второго и false (0) - в противном случае

int Compare (CBox\* pBox)

{

return this -> Volume() >pBox->Volume ();

}

//Функция для вычисления объема ящика

double Volume()

{

return m\_Length\*m\_Width\*m\_Height;

}

private:

double m\_Length; //Длина ящика

double m\_Width; //Ширина ящика

double m\_Height; //Высота ящика

};

int main ()

{

CBox boxes[5]; //объявление массива объектов CBox

CBox match(2.2,1.1,0.5); //объявление match типа CBox

CBox cigar(8.0,5.0,1.0); //объявление cigar типа Cbox

//инициализация указателя адресом объекта cigar

CBox\* pB1 = &cigar;

CBox\* pB2 = 0; //указатель на CBox

cout << endl

<< "adress cigar = "<<pB1;//Отображение адреса

cout<< endl

<< "V cigar= " <<pB1->Volume();//Объем объекта на котором //установлен указатель

pB2 = &match;

if (pB2->Compare(pB1)) //Сравнение через указатель

cout <<endl

<<"match more then cigar";

else

cout <<endl

<<"match less or equal then cigar";

pB1 = boxes; //Установка адреса массива

boxes[2] = match; //Установка 3-го элемента равным match

cout << endl //Доступ через указатель

<< "V boxes[2] = "<<(pB1 + 2)->Volume();

cout << endl;

return 0;

}

Результат

constructor work

constructor work

constructor work

constructor work

constructor work

constructor work

constructor work

adress cigar = 0x001

V cigar= 40

match less or equal then cigar

V boxes[2] = 1.21

Описание полученных результатов

Единственное изменение в этом определении класса не слишком существенно. Вы только модифицировали функцию Compare (), чтобы она принимала указатель на объект СВох в качестве аргумента. Функция main () только испытывает указатели на объекты СВох различными, довольно произвольными способами.

Внутри функции main () объявлены два указателя на объекты СВох после объяв­ления массива boxes, а также объекты СВох с именами cigar и match. Первый, pB1, инициализируется адресом объекта cigar, а второй, рВ2, инициализируется NULL. Использование этих указателей осуществляется в точности таким способом, каким вы применяете указатели на базовые типы. Тот факт, что указатель установлен на объект типа, который вы определили самостоятельно, не имеет значения.

Вы используете рВ1 с операцией непрямого доступа к члену для получения объема объекта, на который он указывает, и результат отображается на экране. Затем вы присваиваете адрес match указателю рВ2 и используете оба указателя при вызове функции сравнения. Поскольку аргументом функции Compare () является указатель на объект СВох, функция применяет операцию непрямого выбора члена для вызова функции Vo1ume () для этого объекта.

Чтобы продемонстрировать возможность использования арифметики адресов над указателем pBl в случае применения его для выбора метода, вы устанавливаете в pBl адрес первого элемента массива boxes элементов типа СВох. В данном случае выбирается третий элемент массива и вычисляется его объем. Это то же самое, что и объем match.

В выводе программы можно видеть, что выполняется семь вызовов конструктора объектов СВох: пять для элементов массива boxes и по одному для каждого из объектов cigar и match.

В целом почти нет разницы между использованием указателей на объекты класса и использованием указателей на базовый тип, такой как double.

Ссылки на объекты класса

Практическая польза ссылок особенно проявляется, когда они используются с классами. Как и в случае указателей, почти нет разницы между способом объявления и использования ссылок на объекты класса и способом объявления и использования ссылок на переменные базовых типов. Чтобы объявить ссылку на объект cigar, например, вы должны написать следующее:

СВох& rcigar = cigar; // Определение ссылки на объект cigar

Чтобы применить ссылку для вычисления объема объекта cigar, вы должны просто использовать имя ссылки вместо имени объекта:

cout << rcigar.Volume(); // Вывод объема cigar через ссылку

Как вы должны помнить, ссылка действует в качестве псевдонима объекта, на который она ссылается, поэтому ее применение в точности совпадает с применением исходного имени объекта.

Деструкторы классов

Хотя заголовок этого раздела ссылается на деструкторы, он в такой же мере посвящен и динамическому распределению памяти. Когда выделяется память в свободном хранилище для членов класса, вы обязаны использовать деструктор в дополнение к конструктору и применение динамического выделения памяти для методов класса также требует написания собственного конструктора копирования.

Что такое деструктор?

Деструктор (destructor) - это функция, которая уничтожает объект, когда необходимость в нем отпадает или когда он выходит из области видимости. Разрушение объекта включает освобождение памяти, занятой полями объекта (за исключением статических , которые продолжают существовать, даже когда не остается ни одного экземпляра класса).

Деструктор класса - это метод с именем, совпадающим с именем класса, но предваренным символом "тильда" Деструктор класса не возвращает значения и не принимает параметров. Для класса СВох прототип деструктора класса выглядит следующим образом:

*~СВох(); // Прототип деструктора класса*

Поскольку деструктор не имеет параметров, в классе может существовать лишь один деструктор.

Определять возвращаемое значение или параметры деструктора - ошибка!

Деструктор по умолчанию

Все объекты, которые вы использовали до настоящего момента, уничтожались автоматически с помощьюдеструктора по умолчанию**.** Деструктор класса по умолчанию всегда генерируется компилятором автоматически, если только вы не определяете собственного деструктора для этого класса. Деструктор по умолчанию не удаляет объекты или члены объекта, которые были размещены в свободном хранилище операцией new. Если место для членов класса было выделено в конструкторе динамически, вы должны определить собственный деструктор, который явно воспользуется операцией delete, чтобы освободить память, выделенную в конструкторе операцией new - так же, как вы это делаете с обычными переменными. Для правильного написания деструкторов нужна практика, так что давайте попробуем.

Простой деструктор

Чтобы получить представление о том, когда вызывается деструктор класса, можно включить деструктор в класс СВох. Ниже приведен пример, включающий класс СВох с деструктором.

//listing 46

//ПРОСТОЙ ДЕСТРУКТОР

//класс с явным деструктором

#include <iostream.h>

class CBox //Определение класса в глобальном контексте

{

public:

//определение деструктора

~CBox ()

{

cout << "destructor work" <<endl;

}

//определение конструктора

CBox (double lv = 1.0, double bv =1.0, double hv =1.0)

{

cout <<endl << "constructor work";

m\_Length = lv; //установить значения полей

m\_Width = bv;

m\_Height = hv;

}

//Функция сравнения объемов двух ящиков возвращает true(1)

//если первый больше второгоб и false (0) - в противном случае

int Compare (CBox\* pBox)

{

return this -> Volume() >pBox->Volume ();

}

//Функция для вычисления объема ящика

double Volume()

{

return m\_Length\*m\_Width\*m\_Height;

}

private:

double m\_Length; //Длина ящика в дюймах

double m\_Width; //Ширина ящика в дюймах

double m\_Height; //Высота ящика в дюймах

};

//демонстрация работы деструктора класса CBox

int main ()

{

CBox boxes[5]; //массив объявленных объектов CBox

CBox match(2.2,1.1,0.5); //объявление match типа CBox

CBox cigar(8.0,5.0,1.0); //объявление cigar типа CBox

CBox\* pB1 = &cigar; //инициализация указателя на объект cigar

CBox\* pB2 = 0; //указатель инициализированный нулем

cout<< endl

<< "V cigar= " <<pB1->Volume();//Объем объекта на котором //установлен указатель

pB2 = boxes; // указатель- адресс массива

boxes[2] = match; //Установка 3-го элемента равным match

cout << endl //Доступ через указатель

<< "V boxes[2] = "<<(pB2 + 2)->Volume();

cout << endl;

return 0;

}

Результат

constructor work

constructor work

constructor work

constructor work

constructor work

constructor work

constructor work

V cigar= 40

V boxes[2] = 1.21

destructor work

destructor work

destructor work

destructor work

destructor work

destructor work

destructor work

Вы получили по одному вызову деструктора в конце программы для каждого из созданных объектов. Каждому вызову конструктора соответствует свой вызов деструктора. Вы не должны вызывать деструктор явно. Когда объект класса выходит из области видимости (контекста), компилятор автоматически помещает в эту точку вызов деструктора. В нашем примере вызов деструктора случается после того, как main () завершает выполнение, поэтому вполне возможно, что ошибка в деструкторе вызовет крах программы после благополучного завершения main ().

**Деструкторы и динамическое распределение памяти**

Нередко возникает необходимость в динамическом выделении памяти для полей класса. Вы можете воспользоваться операцией *new* в конструкторе, чтобы выделить место в памяти для поля объекта. В таком случае ответственность за освобождение этого места в памяти, когда объект уже не нужен, должна быть возложена на подходящий деструктор. Давайте для начала определим простой класс, в котором сможем это сделать.

Предположим, что нужно определить класс, в котором каждый объект содержит некоторое сообщение, например, текстовую строку. Класс должен использовать память наиболее эффективным образом, поэтому вместо определения члена как массива char, достаточно большого, чтобы уместилась строка максимальной длины, какая может потребоваться, мы будем распределять память для сообщения в области свободного хранилища во время создания объекта. Ниже показано определение класса.

class CMessage

{

private;

char\* pmessage;

public:

//функция для вывода сообщений

void ShowIt() const

{

cout << endl << pmessage;

}

//Определение конструктора

CMessage(const char\* text = “default message”)

{

//выделить место для текста

pmessage = new char[strlen(text) + 1];

//Копировать text в новую память

strcpy(pmessage, text);

}

~ CMessage(); //Прототип деструктора

};

Этот класс имеет только одино поле данных - *pmessage*, который является указателем на текстовую строку. Он определен в разделе *private* класса, так что не может быть доступен извне класса.

В разделе public находится функция Showlt (), которая выводит объект CMessage на экран. Есть также определение конструктора и прототип деструктора класса, ~CMessage (), к которому вы вскоре вернемся.

Конструктор класса принимает строку в качестве аргумента, но если не передано ничего, использует строку по умолчанию, определенную для этого параметра. Конструктор получает длину строки, переданной в аргументе, исключая завершающий NULL, используя для этого библиотечную функцию strlen (). Чтобы конструктор мог вызывать библиотечную функцию, в файле должен присутствовать оператор #include с заголовочным файлом <cstring.h>. Конструктор определяет количество байт памяти, необходимых для размещения строки в области свободного хранилища, добавляя 1 байт к значению, возвращенному функцией strlen ().

Конечно, если выделение памяти не удастся, будет возбуждено исключение, которое прервет выполнение программы. Если вы хотите обработать такой сбой и обеспечить более элегантное завершение программы, то должны будете перехватить исключение внутри кода конструктора .

Получив память для строки, выделенную операцией new, вы используете библиотечную функцию strcpy (), которая также объявлена в заголовочном файле < cstring.h >, чтобы скопировать переданную в аргументе конструктора строку в область памяти, выделенную для нее. Функция strcpy () копирует строку, определенную вторым аргументом-указателем, по адресу, содержащемуся в первом аргументе-указателе.

Теперь вы должны написать деструктор класса, чтобы он освобождал память, выделенную для текста сообщения. Если вы не представите деструктора класса, то не будет никакой возможности освободить память, выделенную объекту. Если вы используете этот класс в таком виде, как он есть в программе, в которой создается большое количество объектов CMessage, то память свободного хранилища будет постепенно «съедаться» - до тех пор, пока программа не завершится аварийно. И это легко может произойти в ситуациях, когда совсем не очевидно, что подобное возможно. Например, если вы создаете временный объект CMessage в функции, вызываемой в программе многократно, то можете ожидать, что объект уничтожается при возврате из функции. Конечно, вы правы - сами указатели уничтожаются, но память из свободного хранилища, на которую они указывали, остается занятой. Поэтому при каждом вызове функции будет занято все больше и больше памяти свободного хранилища отброшенными объектами CMessage.

Код деструктора класса CMessage выглядит следующим образом:

*// Деструктор для освобождения памяти, выделенной new*

*CMessage::~CMessage()*

*{*

*cout << "destructor work" ; // Просто чтобы видеть, что происходит*

*<<endl;*

*delete [] pmessage; // Освобождением памяти, на которую установлен указатель*

*}*

Поскольку деструктор определен вне определения класса, вы должны определить имя деструктора именем класса, CMessage. Все, что делает деструктор - отображает сообщение, чтобы было видно, что происходит, и вызывает операцию delete для освобождения памяти, на которую указывает член класса pmessage. Обратите внимание на добавление квадратных скобок рядом с delete, поскольку уда­ляется массив (типа char).

Использование класса CMessage.

Рассмотрим небольшой пример с использованием класса Cmessage

//listing 47

//Использование деструктора для освобождения памяти

#include <iostream.h>

#includ<cstring.h> // Для strlen() и strcpy()

class CMessage

{

private;

char\* pmessage; //Указатель на объект, текстовую строку

public:

//функция для вывода сообщений

void ShowIt() const

{

cout << endl << pmessage;

}

//Определение конструктора

CMessage(const char\* text = “default message”)

{

//Выделить место для текста

pmessage = new char[strlen(text) + 1];

//Копировать text в новую память

strcpy(pmessage, text);

}

~ CMessage(); //Прототип деструктора

};

**// Деструктор для освобождения памяти, выделенной new**

**CMessage::~CMessage()**

**{**

**cout << " destructor work" ;// для иллюстрациии**

**<<endl;**

**delete [] pmessage; // Освобождением памяти, на которую установлен указатель**

**}**

int main()

{

// Объявление объекта

CMessage motto(“large seen from a distance”);

// Динамический объект

CMessage\* pM = new CMessage (“cat, which looked at the queen”);

motto. ShowIt(); //отобразить 1-е сообщение

pM -> ShowIt(); //отобразить 2-е сообщение

cout << endl;

//delete pM; //вручную освободить объект, выделенный new

return 0;

}

large seen from a distance

cat, which looked at the queen

destructor work

Описание полученных результатов

В начале *main ()* в обычной манере объявляется, определяется и инициализируется объект *motto* класса *CMessage*. В следующем объявлении определен указатель на объект *CMessage - рМ*, и выделяется память для объекта *CMessage*, на который он указывает, с помощью операции new. Вызов new активизирует конструктор класса CMessage, в котором снова вызывается new, чтобы выделить память для текста сообщения, на которую указывает поле *pmessage*. Если вы соберете и запустите этот пример, он выдаст такой вывод:

*large seen from a distance*

*cat, which looked at the queen*

*destructor work*

Как видим, деструктор вызван лишь однажды, несмотря на то, что вы создали два объекта *CMessage*. Компилятор не отвечает за объекты, созданные в свободном хранилище. Компилятор вставил вызов деструктора для объекта *motto*, потому что это нормальный автоматический объект, даже несмотря на то, что память для его члена была выделена конструктором в свободном хранилище. Объект, на который указывает *рМ*, не таков. Вы выделили память для объекта в свободном хранилище, поэтому должны использовать *delete* для удаления его. Поэтому необходимо убрать комментарий со следующего оператора, находящегося перед оператором *return* в *main():*

*// delete рМ; // Вручную освободить объект, выделенный new*

Теперь, запустив код, вы увидите такой вывод:

*large seen from a distance*

*cat, which looked at the queen*

*destructor work*

*destructor work*

В этом случае добавился дополнительный вызов деструктора. И это несколько неожиданно. Ясно, что *delete* имеет дело только с памятью, выделенной *new* в функции *main ().* То есть освобождает память, на которую указывает *рМ*. Но поскольку указатель *рМ* установлен на объект *CMessage* (для которого определен деструктор), *delete* также вызывает ваш деструктор, чтобы обеспечить освобождение память, занятую членами объекта. Поэтому когда вы используете *delete* для объекта,

динамически созданного с помощью *new*, объекта перед освобождением памяти, занятой объектом, всегда вызывается деструктор.