Лекция: Наследование классов и виртуальные функции

Базовые идеи объектно-ориентированного программирования

Как вы уже видели, класс - это тип данных, которые вы определяете для удовлетворения требований вашего приложения. Классы в объектно-ориентированном программировании (ООП) также определяют объекты, с которыми имеет дело ваша программа. Вы программируете решение проблемы в терминах объектов, специфичных для этой проблемы, используя операции для непосредственной работы с этими объектами. Вы можете определить класс для представления некоторой абстракции, такой как комплексное число, представляющее собой математическую концепцию, или грузовик, который, несомненно, является механизмом (особенно если вы едете на одном из них по дороге). Поэтому, будучи типом данных, класс также может быть определением набора объектов реального мира определенного рода - по крайней мере, в той степени, которая необходима для решения данной проблемы.

Вы можете думать о классе как об определении характеристик конкретной группы вещей, которые определены общим набором параметров и разделяют общий набор операций, выполняемых над ними. Операции, которые вы применяете к объектам данного класса, определяются его интерфейсом, который соответствует функциям, находящимся в разделе *public* определения класса. Класс *СВох*, который мы использовали в предыдущей главе - это хороший пример; он определяет ящик в

терминах размеров плюс набор общедоступных функций, которые можно применять к объектам *СВох* для решения задачи.

Конечно, в реальном мире существует великое разнообразие всяческих ящиков: картонные коробки, упаковки для мониторов, конфетные коробки, коробки для крупы и многие-многие другие, с которыми вам, определенно, приходилось сталкиваться. Вы можете различать ящики по содержимому, которое может в них храниться, материалу, из которого они сделаны, и по множеству других параметров. Но даже несмотря на то, что существует много разнообразных видов ящиков, все они разделяют некоторый общий набор характеристик - некую сущность «ящичности» , если можно так выразиться. Поэтому вы можете визуализировать все виды ящиков в их отношениях друг к другу, невзирая на то, что они обладают многими отличительными особенностями. Вы можете определить некоторый конкретный вид ящиков как обладающий общими характеристиками всех ящиков - возможно, длиной, шириной и высотой. Затем вы можете добавить некоторые дополнительные характеристики к базовому типу ящика, чтобы отличать его от остальных. Вы можете также обнаружить, что существуют некоторые вещи, которые вы можете совершать со своим специфическим видом ящиков, но которые нельзя сделать ни с какими другими ящиками.

Также возможно, что некоторые объекты могут быть результатом комбинации определенных видов ящиков с другими типами объектов: например, коробка конфет или ящик пива. Чтобы описать все это, вы можете определить один вид ящиков как общий, обладающий базовыми характеристиками, присущими абсолютно всем ящи­кам , а затем специфицировать другую разновидность ящиков как уточняющую специализацию первой. На рисунке показан пример такого рода отношений, которые могут быть определены между разными видами ящиков.

Ящики становятся все более специализированными по мере движения вниз по диаграмме, а стрелки на ней направлены от данного типа ящика к другому, на котором он основан. На рисунке определены три разных типа ящиков, основанных на общем для всех базовом типе *СВох*. На нем также определен ящик пива (*CBeerCrate*) как специальная разновидность ящиков, предназначенных для хранения бутылок.

Это хороший способ приближения к описанию реального мира с использованием классов С++, благодаря способности языка определять классы, которые связаны между собой. Коробка конфет может рассматриваться как ящик, обладающий всеми характеристиками базового ящика, плюс несколько собственных характеристик, которые, идентифицируют то, что делает ее специальной.



Наследование в классах

Когда вы определяете один класс на базе другого, то такой класс называется производным. Производный класс автоматически получает поля того класса, который использован в качестве базового при его определении, и, с некоторыми ограничениями, также методы этого базового класса. Говорят, что класс наследует поля и методы класса, на котором он базируется.

Единственными членами базового класса, которые не наследуются производным классом, является деструктор, конструкторы и любые методы, перегружающие операцию присваивания. Все прочие методы, вместе со всеми

полями базового класса, наследуются производным классом. Конечно, причина того, что некоторые базовые члены не наследуются, состоит в том, что производный класс всегда имеет свои собственные конструкторы и деструктор. Если у базового класса есть операция присваивания, производный класс представляет ее собственную версию.

Когда говорят, что эти методы не наследуются, то имеют в виду, что они не суще­ствуют как члены объекта производного класса. Однако они по-прежнему существуют как часть объекта, относящаяся к базовому классу.

Что такое базовый класс?

Базовый класс - это любой класс, который вы используете в качестве основы для определения другого класса. Например, если вы определяете класс В непосредственно в терминах класса А, то об А говорят, что это прямой базовый класс класса В. На рисунке класс *CCrate* - прямой базовый класс для *CBeerCrate*. Когда такой класс, как *CBeer* , определяется в терминах другого класса CCrate, то о *CBeerCrate* говорят, что он унаследован от *CCrate*. Поскольку *CCrate* сам определен в терминах класса *СВох*, то о *СВох* можно сказать, что для *CBeerCrate* он является непрямым базовым классом. Очень скоро вы увидите, как это выражается в определении класса в программе. На следующем рисунке показан способ наследования членов базового класса в производном классе.

 Тот факт, что методы унаследованы, еще не означает, что вы не захотите заменить их в производном классе новыми версиями, и, конечно же, вы можете сде­лать это при необходимости.

Наследование классов от базового класса

Вернемся к исходному классу *СВох* с полями *public,* которые вы видели ранее.

class СВох

{

public:

double m\_Length;

double m\_Width;

double m\_Height;

CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0) :

m\_Length(lv), m\_Width(wv), m\_Height(hv){}

};

В классе есть конструктор, так что вы можете инициализировать объекты при их объявлении. Предположим, что вам понадобится другой класс объектов - *CCandyBox*, который во всем похож на *СВох*, но также имеет еще одно поле- указатель на текстовую строку, определяющую содержимое коробки.

Вы можете определить *CCandyBox* как производный класс, имеющий *СВох* в каче­стве базового, как показано ниже:

class CCandyBox: CBox

{

public:

char\* m\_Contents;

CCandyBox(char\* str = "Candy") // Конструктор

{

m\_Contents = new char [ strlen(str) + 1 ];

strcpy\_s(m\_Contents, strlen(str) + 1, str);

}

~CCandyBox() // Деструктор

{ delete[] m\_Contents; };

};

Имя базового класса *СВох* появляется после имени производного класса *CCandyBox* и отделяется от него двоеточием. Во всех остальных отношениях он выглядит как нормальное определение класса. Вы добавили новый член *m\_Contents*, и, поскольку это указатель на строку, нужно, чтобы конструктор инициализировал ее, а деструктор освобождал память, занятую этой строкой. Вы также должны предусмотреть в конструкторе значение по умолчанию для строки, описывающей содержимое объекта *CCandyBox*. Объекты типа класса *CCandyBox* содержат все члены базового класса *СВох* плюс дополнительный член данных *m\_Contents*.

Обратите внимание на использование функции *strcpy\_s ().* Она принимает три аргумента: место назначения операции копирования, длина принимающего буфера и строка-источник. Если оба массива будут статическими - то есть, не размещенными в куче - вы можете опустить второй аргумент и задать только указатели на источник и исходную строку. Это возможно благодаря тому, что функция *strcpy\_ s ()* также доступна как функция-шаблон, определяющая длину строки назначения автоматически. Поэтому, работая со статическими строками, вы можете вызывать эту функцию только с двумя аргументами.

Использование производного класса

Теперь посмотрим, как работает производный класс, на соответствующем примере.

//listing 48

// Использование производного класса

*#include <string.h> //Для strlen() и strcpy()*

*#include <iostream.h> //Для потокового ввода-вывода*

*class CBox*

*{*

*public:*

*double m\_Length;*

*double m\_Width;*

*double m\_Heiht;*

*CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0):*

*m\_Length(lv), m\_Width(wv), m\_Heiht(hv) {}*

*};*

*class CCandyBox: CBox*

*{*

*public:*

*char\* m\_Contents;*

*CCandyBox (char\* str = "Candy") //Конструктор*

*{*

*m\_Contents = new char[ strlen(str) + 1];*

*strcpy(m\_Contents, str);*

*}*

*~CCandyBox() //Деструктор*

*{ delete [] m\_Contents; };*

*};*

*int main()*

*{*

*CBox myBox(4.0, 3.0, 2.0); //Создать объект CBox*

*CCandyBox myCandyBox;*

*CCandyBox myMintBox("Wafer Thin Mints");*

*cout << endl*

*<<"size of myBox " << sizeof myBox*

*<<endl*

*<<" size of myMintBox " << sizeof myMintBox;*

*cout << endl*

*<<" Length of myBox "<< myBox.m\_Length;*

*myBox.m\_Length = 10;*

*// myCandyBox.m\_ Length =10.0; //Ошибка*

*cout << endl;*

*return 0;*

*}*

*Результат*

*size of myBox 24*

*size of myMintBox 32*

*Length of myBox 4*

Описание полученных результатов

После объявления объекта *СВох* и двух объектов *CCandyBox* выводится количество байт памяти, занятых каждым объектом

Объект *СВох* включает три поля типа double, каждый из которых имеет размер 8 байт, что в сумме составляет 24 байта. Оба объекта *CCandyBox* имеют одинаковые размеры - 32 байта. Длина строки не влияет на размер объекта, поскольку память для размещения строки выделяется динамически в свободном хранилище. 32 байта получаются из 24 байт, которые занимают три поля типа *double*, унаследованные от базового класса *СВох*, плюс 4 байта для поля-указателя

m\_Contents, что дает в сумме 28 байт. Откуда же берутся еще 4 байта? Это объясняется тем, что компилятор выравнивает поля класса по адресам, кратным четырем байтам. Вы можете убедиться в этом, добавив к классу *CCandyBox* дополнительное поле, скажем, типа *int*. После этого размер объекта класса по-прежнему составит 32 байта.

В этой программе также выводится значение поля *m\_Length* объекта *mуВох* типа *СВох*. Даже несмотря на то, что нет никаких проблем с доступом к этому члену в объекте *СВох*, если вы удалите комментарий со следующего оператора в *main ():*

*// myCandyBox.m\_Length =10.0; // удалите комментарий — получите ошибку*

то программа перестанет компилироваться. Компилятор выдаст следующее сообще­ние:

*error С2247: 'СВох::m\_Length' not accessible because 'CCandyBox' uses 'private' to inherit from 'CBox'*

*ошибка C2247: 9СВох::m\_Length' недоступен, поскольку 'CCandyBox' использует 'private' для наследования от 'СВох'*

Это ясно указывает на то, что член *m\_Length* из базового класса недоступен, потому что *m\_Length* в производном классе стал *private.* Это произошло потому, что для базового класса по умолчанию устанавливается спецификатор доступа *private*, когда вы определяете производный класс. Это все равно, как если бы первая строка определения класса-наследника выглядела бы следующим образом:

*class CCandyBox: private СВох*

Всегда необходимо указывать спецификатор доступа к базовому классу, который определяет состояние унаследованных членов в производном классе. Если вы опустите спецификатор доступа к базовому классу, то компилятор подразумевает *private*. Если изменить определение класса *CCandyBox* следующим образом:

class CCandyBox: public СВох

{

public:

char\* m\_Contents;

CCandyBox(char\* str = "Candy") // Конструктор

{

m\_Contents = new char[ strlen(str) + 1 ];

strcpy\_s(m\_Contents, strlen(str) + 1, str);

}

~CCandyBox() //Деструктор

{ delete[] m\_Contents; };

}

то член *m\_Length* будет унаследован в производном классе как *public* и станет до­ступным функции *main ()*. Со спецификатором доступа *public* к базовому классу все унаследованные члены, изначально специфицированные в базовом классе как *public*, будут иметь тот же уровень доступа в классе-наследнике.

Управление доступом при наследовании

Вопрос о доступе к унаследованным членам в производном классе требует более тщательного изучения. Рассмотрим состояние *private*-членов базового класса в производном классе.

Существует веская причина для выбора в предыдущем примере версии класса

*СВох* с полями *public*, а не более безопасной поздней версии с полями *private*. Причина в том, что хотя *private*-члены базового класса являются также членами производного класса, они остаются *private* по отношению к базовому классу, поэтому методы, добавленные к производному классу, не могут получить к ним доступ. Они могут быть доступны в производном классе через методы базового класса, которые не находятся в разделе *private* базового класса.

Это очень легко продемонстрировать, сделав все поля класса *СВох* приватными (*private*) и добавив в производный класс *CCandyBox* функцию *Volume*(), так что определения классов станут такими:

// Версии классов, которые не компилируются

class СВох

{

public:

СВох (double lv » 1.0, double wv = 1.0, double hv = 1.0):

m\_Length(lv), m\_Width(wv), m\_Height(hv){}

private:

double m Length;

double m\_Width;

double m\_Height;

};

class CCandyBox: public CBox

{

public:

char\* m\_Contents;

// Функция вычисления объема объекта CCandyBox

double Volume() const // Ошибка — член не доступен

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

CCandyBox(char\* str = "Candy") // Конструктор

{

m\_Contents = new char[ strlen(str) + 1 ];

strcpy\_s(m\_Contents, strlen(str) + 1, str);

}

~CCandyBox() // Деструктор

{ delete[] m\_Contents; }

};

Программа, использующая эти классы, компилироваться не будет. Функция Volume() в классе CCandyBox пытается обратиться к *private*-полям базового класса, что недопустимо.

**Обращение к приватным членам базового класса**

Однако совершенно законно использовать функцию *Volume ()* в базовом классе, так что если вы перенесете определение функции *Volume ()* в раздел public базового класса *СВох*, то программа не только скомпилируется, но вы также сможете вызвать эту функцию для получения объема объекта *CCandyBox*.

Если метод не должен влиять на состояние объекта, то такой метод имеет смысл делать константным.

//listing 49

//Обращение к приватным полям базового класса

*#include <iostream.h> //Для потокового ввода-вывода*

*#include <string.h> //Для strlen() b strcpy()*

*class CBox*

*{*

*public:*

*CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0):*

*m\_Length(lv), m\_Width(wv), m\_Heiht(hv) {}*

***//Функция для вычисления объема объекта CBox***

***double Volume() const***

***{return m\_Length\* m\_Length\* m\_Heiht;}***

*private:*

*double m\_Length;*

*double m\_Width;*

*double m\_Heiht;*

*};*

*class CCandyBox: public CBox*

*{*

*public:*

*char\* m\_Contents;*

*CCandyBox (char\* str = "Candy") //Конструктор*

*{*

*m\_Contents = new char[ strlen(str) + 1];*

*strcpy(m\_Contents, str);*

*}*

*~CCandyBox() //Деструктор*

*{ delete [] m\_Contents; };*

*};*

*//Использование функции, унаследованной от базового класса*

*int main()*

*{*

*CBox myBox(4.0, 3.0, 2.0); //Создать объект CBox*

*CCandyBox myCandyBox;*

*CCandyBox myMintBox("Wafer Thin Mints");*

*cout << endl*

*<<"size of myBox = " << sizeof myBox*

*<<endl*

*<<" size of myMintBox = " << sizeof myMintBox;*

*cout << endl*

*<<" V myMintBox = " << myMintBox.Volume();*

*cout << endl;*

*return 0;*

*}*

*Результат*

*size of myBox = 24*

*size of myMintBox = 32*

*V myMintBox = 1*

}

Интересен дополнительный вывод в последней строке. Он показывает значение, порожденное функцией *Volume ()*, которая теперь находится в разделе *public* базового класса. Внутри производного класса она работает с членами производного класса, которые унаследованы от базового. Это полноценный член производного класса, поэтому она может свободно использоваться с объектами производного класса.

Значение объема объекта производного класса равно 1, потому что при создании объекта *CCandyBox* конструктор по умолчанию *СВох ()* был вызван первым, чтобы создать часть объекта, относящуюся к базовому классу, и он установил значения размеров *СВох* по умолчанию равными 1

Работа конструктора в производном классе

Хотя конструкторы базового класса не наследуются в производном классе, все же они существуют в базовом классе и используются для создания той части объекта производного класса, которая относится к базовому классу. Это объясняется тем, что создание этой части объекта производного класса - действительно задача конструктора базового класса, а не конструктора производного класса. В конце концов, вы видели, что приватные члены базового класса недоступны в объекте про­изводного класса, даже несмотря на то, что они унаследованы, поэтому ответствен­ность за них лежит на конструкторах базового класса.

В последнем примере конструктор базового класса по умолчанию вызывается автоматически, чтобы создать часть объекта производного класса, которая относиться к базовому классу. Вы можете вызвать определенный конструктор базового класса из конструктора производного класса. Это позволит инициализировать поля базового класса конструктором, отличным от конструктора по умолчанию, или в самом деле выбрать определенный конструктор базового класса, в зависимости от данных, переданных конструктору производного класса.

**Вызов конструкторов**

Можно увидеть , как работает механизм, на измененной версии последнего примера. Чтобы сделать производный класс удобным в применении, необходимо предоставить конструктор для производного класса, который позволит определить значения полей объекта. Для этого вы можете добавить дополнительный конструктор в производный класс и вызывать в нем явно конструктор базового класса, чтобы установить значения полей, которые унаследованы от базового класса.

//listing 50

//вызов конструкторов

*#include <iostream.h>*

*#include <string.h>*

***class CBox***

*{*

*public:*

***CBox(double lv = 1., double wv = 1.0, double hv = 1.0)***

***: m\_Length(lv), m\_Width(wv), m\_Height(hv)***

***{***

***cout << endl << "Constructor CBox";***

***}***

*double Volume() const*

*{*

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

*}*

*private:*

*double m\_Length;*

*double m\_Width;*

*double m\_Height;*

*};*

***class CCandyBox : public CBox***

*{*

*public:*

*char \*m\_Contents;*

***//Конструктор для установки размеров и содержимого с явным вызовом //конструктора CBox***

***CCandyBox(double lv, double wv, double hv, char \* str = "Candy")***

***:CBox(lv, wv, hv)***

***{***

***cout << endl << "Constructor2 CCandyBox";***

***m\_Contents = new char[strlen(str) + 1];***

***strcpy(m\_Contents, str);***

***}***

***//Конструктор для установки только содержимого, автматически вызывает //CBox по умолчанию***

***CCandyBox(char\* str = "Candy")***

***{***

***cout << endl << "Constructor1 CCandyBox";***

***m\_Contents = new char[strlen(str) + 1];***

***strcpy(m\_Contents, str);***

***}***

*~CCandyBox()*

*{*

*delete[] m\_Contents;*

*}*

*};*

***//Вызов конструктора базового класса из конструктора производного //класса***

*int main()*

*{*

*CBox myBox(4.0, 3.0, 2.0);*

*CCandyBox myCandyBox;*

*CCandyBox myMintBox(1.0, 2.0, 3.0, "Wafer Thin Mints");*

*cout << endl << "myBox size: " << sizeof(myBox) << " bytes"*

*<< endl << "myCandyBox size: " << sizeof(myMintBox) << " bytes"*

*<< endl << "Volume of myMintBox equals " << myMintBox.Volume() << endl;*

*return 0;*

*}*

*Результат*

***Constructor CBox***

***Constructor CBox***

***Constructor1 CCandyBox***

***Constructor CBox***

***Constructor2 CCandyBox***

***myBox size: 24 bytes***

***myCandyBox size: 32 bytes***

***Volume of myMintBox equals 6***

Описание полученных результатов

Вместе с добавлением дополнительного конструктора к производному классу, мы добавили операторы вывода в каждый конструктор, чтобы наблюдать, когда каждый из них вызывается. Явный вызов конструктора класса *СВох* появляется после двоеточия в заголовке функции конструктора производного класса. Вы должны заметить, что нотация та же самая, как и использованная ранее для инициализации членов в конструкторе:

*// Вызов конструктора базового класса*

*CCandyBox(double lv, double wv, double hv, char\* str= "Candy"):*

*CBox(lv, wv, hv)*

*{*

*…….*

*}*

Это отлично согласовано с тем, что сделано здесь, потому что вы, по сути, ини­циализируете подобъект *СВох* объекта производного класса. В первом случае вы явно вызывали конструкторы по умолчанию для *double*-полей m\_*Length, m\_Width* и *m\_Height* в списке инициализации. Во втором случае вызывается конструктор *СВох*. Это запускает выбранный вами специфический конструктор *СВох* перед выполнени­ем конструктора *CCandyBox*.

Если вы скомпилируете и запустите этот пример, он выдаст следующий вывод:

*Constructor CBox*

*Constructor CBox*

*Constructor1 CCandyBox*

*Constructor CBox*

*Constructor2 CCandyBox*

*myBox size: 24 bytes*

*myCandyBox size: 32 bytes*

*Volume of myMintBox equals 6*

Вызовы конструкторов объясняются в таблице

|  |  |
| --- | --- |
| **экранный вывод** | **конструируемый объект** |
| *Constructor CBox*  *Constructor CBox*  *Constructor1 CCandyBox*  *Constructor CBox*  *Constructor2 CCandyBox* | *myBox*  *myCandyBox*  *myCandyBox*  *myMintBox*  *myMintBox* |

Первая строка вывода сформирована вызовом конструктора *СВох*, происходящим в момент объявления объекта *myBox* типа *СВох*.

Вторая строка вывода получается в результате автоматического вызова конструктора базового класса, происходящего при объявлении объекта *myCandyBox типа CCandyBox.*

Обратите внимание, что конструктор базового класса всегда вызывается перед конструктором производного класса.

Следующая строка получается в результате вызова конструктора производного класса, для объекта *MyCandyBox*. Этот конструктор вызывается потому, что объект не инициализирован. Четвертая строка вывода происходит от явной идентификации конструктора класса *СВох*, вызываемого из нашего нового конструктора для объектов *CCandyBox*. Значения аргументов, которые определяют значения полей объекта *CCandyBox*, передаются конструктору базового класса.

Далее идет вывод от самого нового конструктора производного класса, так что конструкторы, опять же, вызываются в такой последовательности - сначала для базового класса, затем для производного.

Из сказанного должно быть ясно, что когда вызывается конструктор производно­го класса, то конструктор базового класса всегда вызывается для

построения той части объекта производного класса, которая относится к базовому классу. Если вы не определяете используемый конструктор базового класса, то компилятор вызовет автоматически конструктор базового класса по умолчанию.

Последняя строка таблицы показывает, что инициализация базовой части объекта *myMintBox* работает так, как должно, и приватные поля инициализируются конструктором класса *СВох*.

Наличие *private*-членов базового класса, доступных только методам базового класса, не всегда удобно. Должно существовать много случаев, когда может понадобиться иметь *private*-члены базового класса, которые могут быть доступны в производном классе. И, логично ожидать, что С++ предоставляет такую возможность.

Объявление членов класса как *protected*

В дополнение к спецификаторам доступа *public* и *private*, члены класса можно также объявлять как *protected* (защищенные). Внутри класса ключевое слово *protected* обеспечивает тот же эффект, что и слово *private*: члены класса с этой спецификацией могут быть доступны только методам этого класса и его дру­жественным функциям. Используя ключевое слово *protected*, вы можете переопределить класс *СВох* следующим образом:

//listing 51

//МЕТОД ДОСТУПА PROTECTED

*#include <iostream.h>*

*#include<string.h>*

*class CBox*

*{*

*public:*

*CBox(double lv = 1., double wv = 1.0, double hv = 1.0)*

*: m\_Length(lv), m\_Width(wv), m\_Height(hv)*

*{*

*cout << endl << "Constructor CBox";*

*}*

*//Деструктор CBox-для отслеживания вызова*

***~CBox()***

***{ cout <<"Destructor CBox"<<endl; }***

*protected:*

*double m\_Length;*

*double m\_Width;*

*double m\_Height;*

*};*

*class CCandyBox : public CBox*

*{*

*public:*

*char \*m\_Contents;*

***//функция производного класса, вычисляющая объем***

***double Volume()const***

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

*//Конструктор для установки размеров и содержимого с явным вызовом //конструктора CBox*

*CCandyBox(double lv, double wv, double hv, char \* str = "Candy")*

*:CBox(lv, wv, hv)*

*{*

*cout << endl << "Constructor2 CCandyBox" << endl;*

*m\_Contents = new char[strlen(str) + 1];*

*strcpy(m\_Contents, str);*

*}*

*//Конструктор для установки только содержимого, автм. вызывает CBox по //умолчанию*

*CCandyBox(char\* str = "Candy")*

*{*

*cout << endl << "Constructor1 CCandyBox"<<endl;*

*m\_Contents = new char[strlen(str) + 1];*

*strcpy(m\_Contents, str);*

*}*

***~CCandyBox()***

***{***

***delete[] m\_Contents;***

***cout <<"Destructor CCandyBox"<<endl;***

***}***

*};*

*//Использование спецификатора доступа protected*

*int main()*

*{*

*CCandyBox myCandyBox;*

*CCandyBox myToffeeBox(1.0, 2.0, 3.0, "Stickjaw Toffee");*

*cout << "Volume of myCandyBox equals " << myCandyBox.Volume() << endl*

*<< "Volume of myToffeeBox equals " << myToffeeBox.Volume()<< endl;*

*//cout <<endl << myToffeeBox. m\_Length; //ошибка*

*return 0;*

*}*

*Результат*

*Constructor CBox*

*Constructor1 CCandyBox*

*Constructor CBox*

*Constructor2 CCandyBox*

*Volume of myCandyBox equals 1*

*Volume of myToffeeBox equals 6*

*Destructor CCandyBox*

*Destructor CBox*

*Destructor CCandyBox*

*Destructor CBox*

Описание полученных результатов

В этом примере вычисляются объемы двух объектов *CCandyBox* путем вызова метода *Volume ()* - члена производного класса. Для вычисления результата эта функция обращается к унаследованным полям *m\_Length, m\_Width* и *m\_Height*. В ба­зовом классе эти поля объявлены как *protected* и остаются таковыми в производном классе

Результат показывает, что объем вычисляется правильно для обоих объектов *CCandyBox*. Первый объект имеет размеры по умолчанию, полученные вызовом конструктора по умолчанию *СВох*, поэтому его объем равен 1, а второй объект получает размеры, указанные при его объявлении.

В выводе также показана последовательность вызовов конструкторов и деструкторов, и вы можете видеть, как каждый объект производного класса уничтожается за два шага.

Деструкторы в объектах производных классов вызываются в порядке, обратном вызовам конструкторов объекта. Это - общее правило, которое действует всегда. Конструкторы вызываются, начиная с конструктора базового класса, затем производного, в то время как деструктор производного класса вызывается первым, когда уничтожается объект, а за ним идет вызов деструктора базового класса.

Вы можете убедиться, что *protected*-члены базового класса остаются *protected* и в производном классе, убрав комментарий со строки, предшествующей оператору *return* в функции *main ()*. Если это сделать, компилятор выдаст следующее сообще­ние об ошибке:

*error С2248: 'm\_Length': cannot access protected member declared in class 'CBox' ошибка C2248: 'm\_Length': невозможен доступ к члену protected, объявленному в классе 'СВох'*

Отсюда ясно, что член *m\_Length* недоступен.

Уровень доступа унаследованных членов класса

Вы знаете, что если не указан спецификатор доступа базового класса в определении производного класса, то по умолчанию принимается private. Эффект от этого заключается в том, что унаследованные *public*- и *protected*-члены базового класса становятся в производном классе *private*, и потому не доступны методам производного класса. Фактически они остаются *private* по отношению к базовому классу, независимо от того, как специфицирован базовый класс в определении про­изводного класса.

Вы также использовали *public* в качестве спецификатора для базового класса. Это оставляет в силе спецификации доступа членов базового класса в производном классе, так что *public*-члены остаются *public,* a *protected* - остаются *protected*.

Последняя возможность - объявить базовый класс как *protected*. Это меняет доступ унаследованных *public*-членов базового класса на *protected* в производном классе. Унаследованные члены с уровнем доступа *protected* (и *private*) сохраняют свою спецификацию доступа в производном классе. Все это резюмировано на рисунке.



Этот механизм может показаться несколько сложным, но можно свести все, что касается унаследованных членов, к следующим трем утверждениям.

* Члены базового класса, которые объявлены как private, никогда не доступны в производном классе.
* Определение базового класса как public не изменяет уровней доступа его чле­нов в производном классе.
* Определение базового класса как protected изменяет его public-члены на protected в производном классе.

Возможность изменять уровень доступа унаследованных членов в производном классе предлагает вам определенную степень гибкости, но не забывайте, что вы не можете ослабить уровень, специфицированный в базовом классе; вы можете только сделать доступ более ограниченным. Это предполагает, что ваш базовый класс должен объявлять как *public* те члены, доступ к которым вы хотите варьировать в производных классах. Это утверждение может показаться противоречащим идее инкапсуляции данных в классе для предотвращения неавторизованного доступа, но как вы вскоре убедитесь, часто приходится объявлять в подобной манере базовые классы, единственное назначение которых - служить базой для других классов, и которые не предназначены для того, чтобы создавать объекты их собственного типа.

Виртуальные функции

Давайте присмотримся к поведению унаследованных методов и их отно­шениям с методами производного класса. Вы можете добавить функцию к классу *СВох*, чтобы вывести объем объекта *СВох*. Упрощенная версия класса станет такой:

*#include <iostream.h>*

*class CBox //Базовый класс*

*{*

*public:*

*//Функция для отображения объема объекта*

*void ShowVolume() const*

*{*

*cout << endl <<"Volume of CBox equals " << Volume();*

*}*

*//Функция вычисления объема объекта CBox*

*double Volume() const*

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

*//Конструктор*

*CBox(double lv = 1., double wv = 1.0, double hv = 1.0)*

*: m\_Length(lv), m\_Width(wv), m\_Height(hv) { }*

*protected:*

*double m\_Length;*

*double m\_Width;*

*double m\_Height;*

*};*

Теперь вы можете вывести полезный объем объекта *СВох*, просто вызвав функцию *ShowVolume ()* с любым объектом, который вам нужен. Конструктор устанавливает значения полей в списке инициализации, так что никакие операторы в теле функции не нужны. Поля остаются прежними и специфицируются как *protected*, поэтому они доступны методам любого производного класса.

Предположим, что вы хотите создать производный класс для моделирования ящиков другого вида под названием *CGlassBox*, чтобы хранить стеклянную посуду.

Содержимое хрупко, и поскольку для его предохранения добавляется упаковочный материал, емкость такого ящика будет меньше, чем базового объекта *СВох*. Поэтому вам понадобится другая функция *Volume (),* чтобы учесть это обстоятельство, и вы добавляете ее к производному классу:

*class CGlassBox : public Cbox //Производный класс*

*{*

*public:*

*//функция производного класса для вычисления объема CGlassBox резервирующая 15% на упаковку*

*double Volume()const*

*{ return 0.85\*m\_Length\* m\_Width\* m\_Height; }*

*//Конструктор*

*CGlassBox (double lv, double wv, double hv):CBox(lv, wv, hv) { }*

*};*

Вероятно, в производном классе будут и другие дополнительные члены, но для простоты мы не будем их добавлять, а сосредоточимся на том, как работают унаследованные функции. Конструктор для объектов производного класса просто вызывает конструктор базового класса в своем списке инициализации, чтобы установить значе­ния полей. Никаких дополнительных операторов в его теле не требуется. Вы включили новую версию функции *Volume ()* взамен версии из базового класса. Идея состоит в том, чтобы заставить унаследованную функцию *ShowVolume ()* обращаться к версии функции вычисления объема *Volume ()* производного класса, когда вы запускаете ее с объектом производного класса *CGlassBox.*

**Использование унаследованной функции**

Теперь посмотрим, как производный класс работает на практике. Вы можете проверить это очень просто - создав объект базового класса и объект производного

класса одного и того же размера и затем сравнив правильность вычисления объемов в первом и втором.

//listing 52

//Использование унаследованной функции

#include <iostream.h>

class CBox //Базовый класс

{

public:

//Функция для отображения объема объекта

void ShowVolume() const

{

cout << endl <<"Volume of CBox equals " << Volume();

}

//Функция вычисления объема объекта CBox

double Volume() const

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

//Конструктор

CBox(double lv = 1., double wv = 1.0, double hv = 1.0)

: m\_Length(lv), m\_Width(wv), m\_Height(hv) { }

protected:

double m\_Length;

double m\_Width;

double m\_Height;

};

class CGlassBox : public Cbox //Производный класс

{

public:

//функция производного класса для вычисления объема CGlassBox резервирующая 15% на упаковку

double Volume()const

{ return 0.85\*m\_Length\* m\_Width\* m\_Height; }

//Конструктор

CGlassBox (double lv, double wv, double hv):CBox(lv, wv, hv) { }

};

int main()

{

CBox myBox (2.0, 3.0, 4.0); //Объявление базового ящика

CGlassBox myGlassBox(2.0, 3.0, 4.0)//Объявление производного ящика

myBox. ShowVolume(); //Отобразить объем базового ящика

myGlassBox. ShowVolume(); //Отобразить объем производного ящика

cout << endl;

return 0;

}

Результат

Volume of CBox equals 24

Volume of CBox equals 24

Описание полученных результатов

В результате программа вообще не работает так, как вы хотите, и единственное, что интересует - почему? Очевидно, что второй вызов не принимает во внимание, что он выполнен для объекта производного класса *CGlassBox*. Это видно из неправильного вывода вычисленного объема. Объем объекта *CGlassBox* по определению должен быть меньше, чем объем базового класса *СВох* с теми же внешними размерами.

Причина некорректного вывода в том, что вызов *Volume ( )* в методе *ShowVolume ()* установлен компилятором раз и навсегда по версии, определенной в базовом классе. *ShowVolume ()* -метод базового класса, и когда компилируется класс *СВох*, то вызов *Volume ()* в нем разрешается в момент компиляции как вызов метода *Volume ()* базового класса; компилятор не имеет понятия ни о какой другой функции *Volume ()*. Это называется статическим разрешением вызова функции, поскольку вызов фиксирован до выполнения программы. Иногда это также называют ранним связыванием, поскольку определенная выбранная функция

привязывается к вызову из функции *ShowVolume ()* во время компиляции программы.

В этом примере мы с вами надеялись, что решение о том, какая функция *Volume ()* будет вызвана для каждого конкретного экземпляра объекта, будет принято во время выполнения программы. Операции подобного рода называются динамическим или поздним связыванием. Нам хотелось, чтобы конкретная версия функции *Volume ()*, вызываемая из *ShowVolume О* , определялась типом обрабатываемого объекта, а не была произвольно фиксированной компилятором перед выполнением программы.

Несомненно, вы не опять удивитесь, узнав, что С++ на самом деле позволяет вам добиться этого, поскольку в противном случае вся эта дискуссия была бы ни к чему. Итак, вы должны применить нечто, называемое виртуальной функцией.

Что такое виртуальная функция?

Виртуальная функция - это функция в базовом классе, которая объявлена с использованием ключевого слова *virtual*. Если вы определите функцию базового класса как *virtual*, и в производном классе имеется другое определение этой функции, это сообщит компилятору, что вам не нужна статическая компоновка этой функции. Что вам нужно - так это чтобы выбор функции, которая должна быть вызвана в любой заданной точке программы, был основан на типе объекта, для которого она вызывается.

Исправление CGlassBox

Чтобы этот пример оправдал надежды, вы должны просто добавить ключевое слово *virtual* к определению функции *Volume ()* в этих двух классах

//listing 53

//Использование виртуальной функции

#include <iostream.h>

class CBox //Базовый класс

{

public:

//Функция для отображения объема объекта

void ShowVolume() const

{

cout << endl <<"Volume of CBox equals " << Volume();

}

//Функция вычисления объема объекта CBox

**virtual double Volume() const**

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

//Конструктор

CBox(double lv = 1., double wv = 1.0, double hv = 1.0)

: m\_Length(lv), m\_Width(wv), m\_Height(hv) { }

protected:

double m\_Length;

double m\_Width;

double m\_Height;

};

class CGlassBox : public CBox //Производный класс

{

public:

//функция производного классадля вычисления объема CGlassBox //резервирующая 15% на упаковку

**virtual double Volume()const**

{ return 0.85\*m\_Length\* m\_Width\* m\_Height; }

//Конструктор

CGlassBox (double lv, double wv, double hv):CBox(lv, wv, hv){ }

};

int main()

{

CBox myBox (2.0, 3.0, 4.0); //Объявление базового ящика

CGlassBox myGlassBox(2.0, 3.0, 4.0) //Объявление производного ящика

myBox. ShowVolume(); //Отобразить объем базового ящика

myGlassBox. ShowVolume(); //Отобразить объем производного ящика

cout << endl;

return 0;

}

Результат

Volume of CBox equals 24

Volume of CBox equals 20.4

Описание полученных результатов

Вот теперь программа делает то, что вы хотели. Первый вызов функции *ShowVolume ()* с объектом *myBox* типа *СВох* обращается к версии *Volume ()* из класса *СВох*. Второй вызов с объектом *myGlassBox* типа *CGlassBox* активизирует версию, определенную в производном классе.

Обратите внимание, что, несмотря на то, что вы поместили ключевое слово *virtual* в определение производного класса функции *Volume ()*, делать это не обязательно. Определения базовой версии функции как *virtual* вполне достаточно. Однако все же желательно, чтобы вы указывали это ключевое слово для виртуальных функций в производных классах, поскольку это сделает ясным любому, кто будет читать определение производного класса, что речь идет о виртуальных функциях, которые при выполнении выбираются динамически.

Чтобы функция вела себя как виртуальная, она должна иметь то же имя, список параметров и тип возврата во всех производных классах, как в базовом классе, и если в базовом классе функция объявлена как *const*, функция производного класса также должна быть *const*. Если вы попытаетесь использовать другие типы параметров или возврата или же объявить функцию в одном месте *const*, а в другом - нет, то в этом случае механизм виртуальных функций работать не будет. Функция будет компонована статически и фиксирована во время компиляции.

Работа виртуальных функций - исключительно мощный механизм. Вы, должно быть, слышали термин полиморфизм в связи с объектно-ориентированным программированием; этот термин имеет отношение к возможностям виртуальных функций. Иногда полиморфизм может проявляться в разных формах, таких как оборотень, или в поведении политиков перед выборами и после них. Вызов виртуальных функций приводит к разному эффекту в зависимости от вида объекта, для которого она вызвана.

Следует отметить, что функция *Volume ()* в производном классе *CGlassBox* на самом деле скрывает версию этой функции из базового класса от функций производного класса. Если вы хотите вызвать версию *Volume ()* базового класса из функции производного класса, то должны будете использовать операцию разрешения контекста для обращения к функции, например, *СВох:: Volume ().*

Использование указателей на объекты классов

Применение указателей с объектами базового класса или производного класса - важная техника. Указателю на объект базового класса можно присвоить адрес объ­екта производного класса, равно как и объекта базового класса. Поэтому вы можете использовать указатель типа "указатель на объект базового класса", чтобы получить разное поведение с виртуальными функциями, в зависимости от того, на объект какого типа установлен указатель. Чтобы лучше понять, как это работает, рассмотрим пример.

Мы применим те же классы, что и в предыдущем примере, но внесем небольшое изменение в функцию main (), так что она будет использовать указатель на объект базового класса.

//listing 54

//Использование указателя базового класса для вызова виртуальной //функции

#include <iostream.h>

class CBox //Базовый класс

{

public:

//Функция для отображения объема объекта

void ShowVolume() const

{

cout << endl <<"Volume of CBox equals " << Volume();

}

//Функция вычисления объема объекта CBox

virtual double Volume() const

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

//Конструктор

CBox(double lv = 1., double wv = 1.0, double hv = 1.0)

: m\_Length(lv), m\_Width(wv), m\_Height(hv) { }

protected:

double m\_Length;

double m\_Width;

double m\_Height;

};

class CGlassBox : public CBox //Производный класс

{

public:

//функция производного классадля вычисления объема CGlassBox резервирующая 15% на упаковку

virtual double Volume()const

{ return 0.85\*m\_Length\* m\_Width\* m\_Height; }

//Конструктор

CGlassBox (double lv, double wv, double hv):CBox(lv, wv, hv) { }

};

int main()

{

CBox myBox (2.0, 3.0, 4.0); //Объявление базового ящика

CGlassBox myGlassBox(2.0, 3.0, 4.0); //Объявление производного ящика

**CBox\* pBox = 0; //Объявление указателя на объекты базового класса**

**pBox = &myBox; //Установить указатель на адрес базового объекта**

**pBox ->ShowVolume(); //Отобразить объем базового ящика**

**pBox = &myGlassBox; //Установить указатель на адрес объекта**

**//производного класса**

**pBox ->ShowVolume(); //Отобразить объем производного ящика**

cout << endl;

return 0;

}

Результат

Volume of CBox equals 24

Volume of CBox equals 20.4

Описание полученных результатов

Здесь используются те же самые классы, что и в предыдущем примере, но функция *main ()* изменена, чтобы использовать указатель для вызова *ShowVolume ().* Поскольку применяется указатель, с ним нужно использовать операцию непрямого выбора члена, ->, для вызова функции. Функция *ShowVolume ()* вызывается дважды, и оба вызова используют один и тот же указатель на объект базового класса - *рВох*. В первом случае указатель содержит адрес объекта базового класса *myВох*, а при втором - адрес объекта производного класса, то есть *myGlassBox*.

Результат работы программы аналогичен результату в предыдущем примере, где использовались явные обращения к объектам для вызова функции.

На основании этого примера вы можете сделать вывод, что механизм виртуальных функций работает одинаково хорошо и через указатель на базовый класс; при этом выбирается специфическая функция на основе действительного типа объекта, на который он указывает. Смотрим на рисунок.

Он показывает, что даже когда вы точно не знаете тип объекта, на который установлен указатель базового класса в программе (например, когда указатель передается в функцию в качестве аргумента), механизм виртуальных функций гарантирует вызов правильной функции. Это чрезвычайно могучее средство, поэтому убедитесь, что вы хорошо понимаете его.

Полиморфизм - фундаментальный механизм в С++, который вы, несомненно, будете использовать постоянно.



Мы можем увидеть, что происходит, модифицировав *main ()* из последнего примера, чтобы вызывалась функция, принимающая ссылку как параметр.

//listing 55

//Использование ссылок с виртуальными функциями

#include <iostream.h>

class CBox //Базовый класс

{

public:

//Функция для отображения объема объекта

void ShowVolume() const

{

cout << endl <<"Volume of CBox equals " << Volume();

}

//Функция вычисления объема объекта CBox

**virtual double Volume() const**

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

//Конструктор

CBox(double lv = 1., double wv = 1.0, double hv = 1.0)

: m\_Length(lv), m\_Width(wv), m\_Height(hv) { }

protected:

double m\_Length;

double m\_Width;

double m\_Height;

};

class CGlassBox : public CBox //Производный класс

{

public:

//функция производного классадля вычисления объема CGlassBox резервирующая 15% на упаковку

**virtual double Volume()const**

**{ return 0.85\*m\_Length\* m\_Width\* m\_Height; }**

//Конструктор

CGlassBox (double lv, double wv, double hv):CBox(lv, wv, hv) { }

};

//Использование ссылки для вызова виртуальной функции

**void Output (const CBox& aBox); //Прототип функции**

int main()

{

CBox myBox (2.0, 3.0, 4.0); //Объявление базового ящика

CGlassBox myGlassBox(2.0, 3.0, 4.0); //Объявление производного //ящика

**Output (myBox); //Вывод объема объекта базового класса**

**Output (myGlassBox);//Вывод объема объекта производного класса**

cout << endl;

return 0;

}

**void Output (const CBox& aBox)**

**{**

**aBox.ShowVolume();**

**}**

Результат

Volume of CBox equals 24

Volume of CBox equals 20.4

Описание полученных результатов

Функция *main ()* теперь в основном состоит из двух вызовов функции *Output ()*, первого с аргументом - объектом базового класса, и второго с аргументом - объектом производного класса. Поскольку параметр представляет собой ссылку на базовый класс, *Output ()* может принимать объекты обоих классов в качестве аргумента, и соответствующая версия виртуальной функции *Volume ()* вызывается в зависимости от объекта, инициализировавшего ссылку.

Программа генерирует точно тот же вывод, что и предыдущий пример, демон­стрируя, что механизм виртуальных функций работает и со ссылочными параметрами.