Наследование. Обработка исключений

Наследование

Код искать тут: 8_1_Inheritance.cpp

Что такое наследование и зачем оно нужно?

Давайте вернемся к нашему старому примеру с двумя стульями. Закройте глаза и представьте два типичных стула из наших аудиторий. Первый — так называемый стул школьный, ученический, с четырьмя ногами и жесткими фанерными сиденьем и спинкой. Второй более удобный для сидения, потому что у него мягкие спинка и сиденье, но гораздо хуже влезает под парту. Вспомнили? Если нет, то см. рисунок 1.

Рисунок 1. Типичные стулья из кабинета 12-403

Теперь представьте, что вам нужен класс для описания обоих стульев. Если использовать один и тот же класс, то придется создавать излишне подробные описания свойств и, возможно, методов, потому что иначе будет сложно отобразить различия. Можно, конечно, сделать два совершенно разных класса, но тогда придется дублировать слишком многие свойства и методы, да и объекты будет проблематично сравнивать между собой. А если нам потребуется потом добавить стулья на колесиках?

Выходом в данном случае будет создать сначала класс Стул, в котором описать самые общие свойства и методы, а потом на его основе еще два класса — ШкольныйСтул и ОфисныйСтул. Для этих классов можно будет задать и уникальные свойства. С стулом на колесиках тоже проблем не будет. Мы просто сделаем еще один класс на основе обычного, а может быть даже офисного, и зададим ему не только новые свойства, но еще и новый метод — КачениеСтула.

В программировании это называется наследованием. Для удобства понимания родства обычно рисуют так называемые диаграммы классов[1]* (см.рис. 2).

Рисунок 2. Простейшая диаграмма классов

Немного терминологии. Класс, от которого происходит другой класс, называют **родительским, базовым, надклассом** или суперклассом (кривой перевод), а класс, от него производный — **ребенком,**

[1] На самом деле диаграмма классов выглядит не совсем так, но мы пока притворимся, что этого не знаем

сыновним/дочерним классом, подклассом и даже суб- или сабклассом (снова кривой перевод). В более сложных конструкциях используются также термины **предок** и **потомок**.

Синтаксис:

```
class имя: [private|protected|public] имя_базового_класса
{
//тело класса
}
```

[ ] здесь означают необязательность параметра, по умолчанию для классов `private`, для структур — `public`.

С этими двумя параметрами мы уже знакомы, а что такое `protected`? Это защищенное наследование. Для единичного класса и внутри класса оно работает аналогично закрытому, а вот при комбинации параметров доступность меняется. Типы наследования работают согласно таблице 1.

Таблица 1. Сравнение типов наследования в C++

Базовый класс CBase	Публичное	Защищенное	Закрытое
	Производный класс CDerived: public CBase	Производный класс CDerived: protected CBase	Производный класс CDerived: private CBase
public:	public:	protected:	private:
protected:	protected:		
private:	недоступно	недоступно	недоступно

Как мы с вами помним из предыдущей лекции, конструкторы и деструкторы не наследуются, поэтому их необходимо явно определить, если нужно. При создании объекта последовательно вызываются конструкторы всех родительских классов, начиная с базового. При уничтожении объекта последовательно вызываются деструкторы в обратном порядке по отношению к вызову конструкторов. Также не наследуются дружественные функции и перегруженная операция присваивания.

Рассмотрим пример.

```
class figure{
 protected:
      double x, y;
 public:
      figure(double a=0, double b=0){x=a; y=b;}
      double area(){return 0;}
};
```

Здесь `figure` — это базовый класс, который содержит два защищенных поля, конструктор с параметрами по умолчанию и метод вычисления площади, возвращающий ноль.

Создадим класс для работы с прямоугольниками, используя открытое наследование от базового класса. Переопределим в нем метод вычисления площади.

```
class rectangle: public figure{
 public:
      rectangle(double a, double b): figure(a, b){};
      double area(){return x*y;}
};
```

Также мы определили в нем новый конструктор, который вызывает конструктор базового класса и передает ему свои аргументы.

Теперь определим еще один производный класс, в котором также переопределим метод вычисления площади и определим конструктор. Обратите внимание, что мы здесь также вызываем конструктор базового класса, но передаем ему только один параметр. Это возможно благодаря тому, что у параметров конструктора базового класса заданы значения по умолчанию

```
class circle: public figure{
public:
      circle(double a): figure(a){};
      double area(){return 3.1416*x*x;}
};
```

Протестируем созданные классы внутри функции main()

```
      figure A(2, 3);
      cout<<A.area()<<endl;
      rectangle B(2, 3);
      cout<<B.area()<<endl;
      circle C(3);
      cout<<C.area()<<endl;
```

Для каждого класса вызывается свой метод вычисления площади

```
Успешно #stdin #stdout 0s 4324KB
0
6
28.2744
```

Абстрактные классы и виртуальные функции

Виртуальные функции — это особый вид членов класса. Отличие виртуальной функции от обычной состоит в том, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции, а для виртуальных функций это происходит во время выполнения программы. Этот механизм получил название динамического (позднего) связывания.

Виртуальная функция объявляется в базовом классе с использованием ключевого слова virtual. В производном классе его писать уже не нужно.

В нашем примере мы можем создать два динамических объекта базового класса

```
      figure *X, *Y;
      X=&B;
      Y=&C;
      cout<<X->area()<<endl;
      cout<<Y->area()<<endl;
```

При попытке вычисления площади будет вызван метод базового класса, возвращающий 0

```
Успешно #stdin #stdout 0s 4552KB
0
6
28.2744
0
0
```

Но если мы сделаем метод `area()` базового класса виртуальным

```
class figure{
 protected:
      double x, y;
 public:
      figure(double a=0, double b=0){x=a; y=b;}
```

```
        virtual double area(){return 0;}
};
```

то для X и Y будут вызваны реализации методов из производных классов.

```
Успешно #stdin #stdout 0s 4452KB
0
6
28.2744
6
28.2744
```

Чистая виртуальная функция — это метод класса, тело которого не определено.

```
virtual заголовок функции = 0;
```

Если в классе есть хотя бы один чисто виртуальный метод, то такой класс считается абстрактным. В C++ нельзя создавать объекты абстрактных классов. Чисто абстрактный класс содержит только виртуальные методы. В производных от него классах все методы должны быть обязательно переопределены.

Если в классе есть хотя бы одна виртуальная функция, то деструктор тоже должен быть виртуальным. Более подробном об этом можно почитать, например, здесь.

Множественное наследование

В C++ может существовать множественное наследование, т.е. один класс может быть потомком сразу нескольких классов (см.рис. 3). Базовые классы в таком случае перечисляются через запятую.

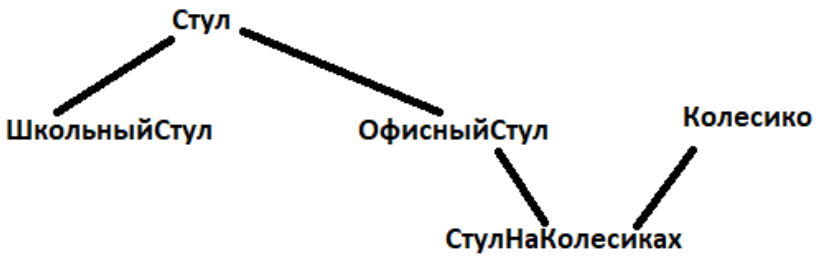

Рисунок 3. Множественное наследование

Главное не допускать так называемого «ромбовидного наследования» (см.рис.4).

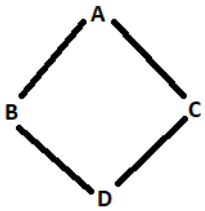

Рисунок 4. Ромбовидное наследование. Не надо так!

Ромб можно не заметить, когда он появляется в более сложных конструкциях, поэтому нельзя пренебрегать проектированием при создании сложных систем!

Инкапсуляция, наследование и полиморфизм

И в заключение немного о так называемых трех китах, на которых стоит объектно-ориентированное программирование — инкапсуляция, наследование и полиморфизм (см. рис. 5).

Рисунок 5. Для наглядности 😊

Что такое наследование, мы только что рассмотрели. С инкапсуляцией мы тоже неоднократно сталкивались. **Инкапсуляция** представляет собой размещение в одном компоненте данных и методов, которые с ними работают. Именно так устроены классы. Особенностью инкапсуляции в C++ является то, что она тесно связана с сокрытием деталей реализации. Например, поля в классах мы, как правило, делаем скрытыми, а для доступа к ним используем специализированные методы.

Полиморфизм — это возможность использовать одну и ту же функцию для решения похожих, но разных задач. Например, оператор сложения + используется и для сложения чисел различных типов, так и для логически сходного действия — склеивания строк. В C++ с помощью механизма перегрузки можно сделать так, чтобы + использовался и для собственноручно создаваемых типов данных. Что мы с вами и сделали на предыдущих занятиях, когда создали класс для работы с комплексными числами и добавили туда операцию сложения.

Полиморфизм в C++ достаточно хорошо реализован, однако его практически нет в языке-предшественнике C. Примером здесь может служить функция abs(). Наверняка в начале изучения программирования многие из вас задавались вопросом, почему в различных примерах то fabs(), то просто abs(), и какая между ними разница? В C на самом деле таких функций не две, а больше — abs(), labs() и fabs(). Первая возвращает абсолютную величину для целых чисел, вторая для длинных целых (и результаты будут тоже целыми!), а третья работает с числами с плавающей точкой. Если в C выбрать не ту функцию, результат может быть не совсем таким, каким нужно, если код вообще скомпилируется, в C++ таких проблем нет, мы просто используем одну и ту же функцию abs() для всех чисел. Аналогичный пример можно привести для функций sqrt() и sqrtf().

Обработка исключительных ситуаций

Код искать тут: 8_2_Exceptions.cpp

Что такое исключительная ситуация в программе? Бывает, что в некоторых случаях программа вылетает. Например, если пользователь ввел некорректные данные и в результате получилось деление на ноль. Или под корнем стоит отрицательное выражение. Конечно, эти ситуации можно отследить разными способами, и что-то с ними сделать, но есть удобный встроенный механизм.

Для этого используются три новых ключевых слова — `throw`, `try`, `catch`. Давайте разбираться в синтаксисе.

С throw все просто — это бросок исключения. Вы просто пишете throw [значение]; в том месте, где может возникнуть исключение. [] как и прежде означают необязательность параметра.

Бросок должен проходить внутри контролируемого блока кода. Просто блок кода это {}, контролируемый блок -- try{} .

После этого блока надо исключение поймать. делается это при помощи catch.

Синтаксис

```
try{

}
catch(){

}
```

Внутри скобок у catch() может быть три варианта:

1. тип имя (для ловли исключений определенного типа с возможностью использовать параметр)
2. тип (для ловли всех исключений заданного типа данных)
3. можно записать также catch(...) (для ловли всех остальных исключений, этот вариант должен идти последним, если он есть)

Давайте попробуем пример. Я возьму задачу деления на ноль.

```
int a, b, c;
cin>>a>>b;
try{
   if (b==0)
     throw 0;
   c=a/b;
   cout<<"a/b="<<c<<endl;
}
catch(int err){
   switch (err){
       case 0: cout<<"Division by zero!"<<endl; break;
   }
}
```

Сначала запустила для случая, когда исключения не возникает, а теперь для случая, когда оно есть.

switch для одной ошибки, конечно, не нужен, но это заготовка для более сложного проекта, где могут возникать ошибки разного рода и в разных количествах.

Можно использовать как собственные исключения, что мы только что делали, так и стандартные. В лабораторной работе вам нужно будет снабдить ваш класс обработкой исключений. Не всегда место, где могут быть исключения, лежит на поверхности. Это к вопросу о том, что сделать, если в вашей задаче не видно, какие можно придумать исключения. Наша задача в первую очередь понять механизм их работы, так что исключения просто нужно использовать для вашего варианта.