Глава 12

Полиморфизм

Предположим, что имеется указатель на объект типа базового класса. Возникает вопрос: указывает ли он действительно на объект базового класса или же на объект одного из его потомков?

А теперь обратимся к примеру со списком и множеством и допустим, что имеется несколько объектов класса `list` и объектов класса `set`. Пусть задача заключается в том, чтобы добавить некоторый элемент во все объекты как типа `list`, так и типа `set`:

```
void f() {
   void *x = 0;
   // здесь мы присваиваем x какое-то осмысленное значение
   //...
   list l1;
   set s1;
   //...
   g(l1, x);
   g(s1, x);
}

void g(list& l, void* x) {
   //...
   l.add(x);
   //...
}
```

Для класса `set` реализация функции `add` может отличаться от реализации метода `list::add`. Например, можно потребовать, чтобы перед добавлением элемента к множеству проверялось, есть ли уже во множестве такой элемент. Добавляться будут только те элементы, которых еще не было, увеличивая при этом мощность множества:

```
void set::add(void *d) {
   if (!is_member(d)) {
      list::add(d);
      power++;
   }
}
```

Компилятору известно, что переменная `l` в функции `g()` — это указатель на объект типа `list`. Когда вызывается функция `add()` объекта типа `list`, компилятор имеет достаточно информации только для того, чтобы произвести вызов функции базового класса, но ему неизвестно, о самом ли классе `list` или о классе-потомке (`set` или каком-то другом) идет речь. Как же тогда он определяет, какую функцию вызывать, встретив `l->add(x)`?

Ответ заключается в понимании одного из основных принципов объектно-ориентированного программирования — полиморфизма.

Свойства наследования, описанные в предыдущей главе, направлены на достижение повторной используемости: возможность строить классы как расширения уже существующих классов. Есть еще один аспект не меньшей важности, связанный с расширяемостью: концепция полиморфизма и динамическое связывание.

Полиморфизм — это свойство объекта принимать несколько форм. В объектно-ориентированном программировании это приводит, в частности, к тому, что переменная-

указатель имеет возможность ссылаться во время выполнения программы на экземпляры различных классов. Это обеспечивается и контролируется наследованием: как уже говорилось, можно разрешить переменной типа `list*` присваивать указатель на объект типа `set`. Но обратное неверно: переменной типа `set*` нельзя присваивать указатель на `list` без явного преобразования типов. Такое явное преобразование, кстати, не гарантирует, что указатель действительно настроен на объект требуемого типа! Правило совместимости типов гласит, что присвоение `x=y` корректно, если `x` — объект класса предка (или указатель на объект класса-предка), а `y` — объект класса потомка (или, соответственно, указатель на объект класса-потомка). То же правило применимо, если `x` — формальный параметр функции, а `y` — соответствующий фактический параметр в ее вызове.

Может показаться, что этим правилом полиморфизм "загнан" в слишком тесные рамки. Однако даже с точки зрения здравого смысла, если на просьбу заплатить в долларах вам предлагают тугрики на том основании, что это тоже деньги, становится ясно, что подобные ограничения не являются излишне жесткими.

Выполнение правила совместимости типов контролируется во время компиляции, и проверка в ходе работы программы не нужна. Компилятор не воспримет ни один фрагмент программы, нарушающий это правило.

В зависимости от решаемой задачи вы получаете возможность двояко использовать преимущества, предоставляемые механизмом наследования. С одной стороны, некоторые понятия, например понятие списка, можно детализировать, указав, что данный список есть множество. При выделении определенных специфических черт снижается уровень абстракции, что позволяет выполнять, так сказать, тонкую работу. И наоборот, в тех случаях, когда интерес представляют более глобальные задачи, существует возможность оперировать более общими понятиями.

Так, обратившись к тому же примеру со списком и множеством, можно увидеть, что поскольку множество имеет все свойства списка, в функции `g()` в качестве указателя на `set` можно использовать указатель на `list`. Таким образом, в этой функции понятиями списка и множества (а также любым другим уточнением или расширением простого списка, которое может быть получено путем наследования в будущем) можно оперировать совершенно одинаково.

Так как проблема использования одной переменной для ссылки на объекты различных типов возникла уже давно, то, естественно, и раньше были попытки найти приемлемые решения путем ослабления требований статической типизации. В языке C существует (и, к счастью, продолжает существовать в C++) специальный тип указателя, который уже упоминался ранее, называемый указателем на неопределенный тип:

```
void *pointer;
```

Такой указатель может быть использован для ссылки на объект любого типа. Но для того чтобы оперировать таким указателем или объектом, на который он указывает, необходимо явно задавать (путем приведения типов) требуемый тип в каждой операции. Кроме того, для работы с различными объектами через `void*`, вероятно, придется завести у них поле типа, которое смогут проверять функции для установления истинного типа объекта. Однако определение истинного типа сущности во время выполнения программы на основе поля типа чревато ошибками и трудностями при сопровождении программы, да и просто неудобно. В этом контексте решение для списков, где элементы списка — указатели на неопределенный тип, не самое удачное. Гораздо приятнее было бы работать с указателями на объекты какого-то определенного класса, возможно, являющегося базовым для ряда других классов. Именно для преодоления предполагаемых проблем, связанных с неопределенными типами, достаточно часто в проектах реализуется класс с традиционным именем `object`, и все остальные классы являются его прямыми или непрямыми потомками.

Следует заметить, что полиморфизм, как уже упоминалось, реализует некоторые аспекты инкапсуляции. Так, пользователя, который пополняет контейнеры `list` и `set`, совершенно

не интересует их внутреннее устройство, для него существуют только определенные интерфейсы, позволяющие абстрагироваться от "заключенной в капсулу", т.е. инкапсулированной части кода обоих классов.

Виртуальные функции

Операции, определенные для иерархии классов, не обязаны реализовываться идентично в каждом из них. В анализируемом примере базовый класс `list` и класс-потомок `set` имеют различные версии функции `add()`. Можно представить себе и иные варианты для других специальных видов списков. Тогда немедленно встает фундаментальный вопрос: что происходит с полиморфной переменной, когда к ней применяется функция, имеющая более одной версии?

С помощью *виртуальных функций* есть возможность разрешить возникшую проблему применения функции к полиморфной переменной. В C++ для этого в базовом классе особым образом описываются функции, которые могут быть переопределены в любом производном классе. В представленном примере с функцией `add()` это выглядело бы следующим образом:

```
class list {
// ...
public:
   virtual void add(void* d);
// ...
};
```

Слово `virtual` указывает на то, что функция `add()` может иметь различные варианты реализации в классах-потомках. Теперь функция `g()` отработает корректно и даст нужный результат даже в том случае, если она была оттранслирована до того, как был создан производный класс `set`. Метод, известный как динамическое связывание, ведет к тому, что динамическая форма объекта определяет применяемую версию операции. Эта способность операций автоматически адаптироваться к объектам, к которым они будут применены, является одной из наиважнейших особенностей объектно-ориентированных систем. Принцип полиморфизма в реализации языка C++ предлагает гибкий механизм для решения означенной проблемы — позднее (или динамическое) связывание для виртуальных функций.

Вернувшись к примеру с плавучими ресторанами из предыдущей главы, нетрудно заметить, что если функции `order` для классов `restaurant` и `ship` определены как виртуальные, то вместо создания двух разных функций-"оберток" в классе-наследнике можно создать одну функцию с тем же названием `order`:

```
class floating_restaurant : public restaurant, ship {
public:
   virtual void order (int count, data d, time t) {
      restaurant::order (count, d, t);
      ship::order (count, d, t);
   }
//...
};
```

Механизм виртуализации

Давайте более подробно рассмотрим механизм, осуществляющий *динамическое связывание*. В традиционном C вызов функции происходит при помощи так называемого механизма *раннего* или *статического связывания*. Компилятор генерирует код вызова заранее известной конкретной функции по ее имени, и при сборке этот вызов связывается с реализацией функции. При раннем связывании адрес вызываемой функции всегда известен во время сборки программы. Но подобный механизм раннего связывания не всегда позволяет легко добавлять к имеющимся типам данных новые наследуемые типы, использующие собственные реализа-

ции унаследованных функций. Проблема состоит в том, что если добавить наследуемый тип данных, в котором переопределяются некоторые функции, то в этом случае нельзя использовать старый код программы для их вызова.

Когда создавалась программа, она еще "не знала" о том, что в дальнейшем появятся какие-то новые функции, к которым ей придется обращаться. И даже если при выполнении каких-то действий с объектом нового типа программа, по логике, должна обратиться к переопределенной функции унаследованного типа, все равно произойдет вызов базовой функции типа, так как именно эта функция жестко привязана к конкретному коду программы. Это приводит к тому, что при добавлении новых типов приходится изменять код программы, который обращается к функциям, переопределенным в классах-потомках, и затем проводить повторную компиляцию всей программы.

Механизм *позднего связывания* позволяет передавать сообщение о вызове виртуальной функции тому объекту, над которым выполняется действие. При этом виртуальная функция не связывается с объектом так, как это происходит с обычными функциями в процессе компиляции. Можно создать новый класс, унаследованный от базового, и написать для него другую подходящую функцию, соответствующую виртуальной функции базового типа. Затем можно скомпилировать свой код и скомпоновать его с уже готовыми объектными файлами кода функций, обращающихся к виртуальным функциям объектов базового типа. Вызов необходимой функции (для объектов базового или унаследованного типа) осуществляется уже в процессе выполнения программы и полностью зависит от типа объекта, участвующего в вызове. Еще раз подчеркнем, что адрес функции, соответствующей данному объекту, определяется во время выполнения программы, что именно и характеризует понятие позднего связывания.

Во время компиляции в C++ осуществляется проверка того, что функция определена, и аргументы и возвращаемое значение имеют верный тип. Однако в момент компиляции неизвестно, какой именно код будет выполняться при вызове виртуальной функции. Поэтому на место реального вызова функции компилятор помещает в программу несколько специальных операторов. В процессе работы программы эти операторы извлекают указатель, используемый для вычисления адреса нужной функции, из самого объекта. По этому указателю, который в C++ называется VPTR (Virtual PoinTeR), из специальной таблицы выбирается физический адрес вызываемой функции. Эта таблица, называемая в C++ VTABLE, содержит адреса всех виртуальных функций. Каждый класс, имеющий или унаследовавший виртуальные функции, имеет и соответствующую ему таблицу VTABLE. Любой объект имеет свой указатель VPTR. При вызове виртуальной функции самим объектом определяется, какой код должен выполняться.

Исключения из правила наследования

У правила, гласящего, что все свойства родителя автоматически передаются потомку, есть одно исключение — конструктор. Он никогда не наследуется прямо. Конструктор — единственный метод, обладающий таким свойством.

Причина этого исключения ясна: поскольку обычно потомки имеют больше данных и методов, чем родители, у каждого конструктора различные аргументы. Иными словами, для правильного построения объекта конструктор обязан знать его истинный тип. В C++ конструктор существенно отличается от обычных функций-членов. Он неявно взаимодействует с функциями управления памятью и никогда не вызывается для объектов, которые уже созданы.

Правило построения нового объекта гласит, что класс, не определяющий конструктор, имеет стандартную процедуру создания экземпляров этого класса, инициализирующую все данные значениями по умолчанию, но он никогда не наследует конструктор родителя.

Поскольку конструктор никогда не наследуется, его нельзя "переопределить" так же, как обычную функцию. Его надо просто определять для каждого класса в наиболее удобном виде, — включая вид по умолчанию, если конструктор не объявлен. Конструктор потомка

никак не связан с конструкторами родителей, в частности, аргументы могут отличаться по числу и типам.

Однако довольно часто возникает ситуация, когда конструкторы класса-предка и класса-потомка имеют абсолютно одинаковый вид. Конечно, выделение памяти под объекты происходит по-разному, но по содержанию задачи конструкторов абсолютно идентичны. Возможность переопределения конструктора в потомке позволила бы инициализировать объекты суперклассов и классов-потомков единым способом. С подобной проблемой приходится сталкиваться и в случае создания нового объекта, когда неизвестен его истинный тип. Но эти ограничения вполне можно преодолеть, определив, например, функцию, содержащую вызов конструктора и возвращающую указатель на созданный объект. Эта функция может наследоваться и при необходимости заменит вызов конструктора. Допустим, что класс `list` имеет метод `new_list()`, определенный следующим образом:

```
class list {
//...
public:
   virtual list *new_list() { return new list(); }
};
```

переопределенный в классе `set`:

```
class set: public list {
//...
public:
   list * new_list() { return new set(); }
};
```

Теперь пользователь имеет возможность написать, например, следующую функцию:

```
void f(list *l1, list *l2)
{
   list *l3, *l4;

   l3 = l1->new_list();
   l4 = l2->new_list();
}
```

причем, если в качестве фактического параметра функции, допустим `l1`, будет указатель на объект типа `set`, то `l3` также будет указывать на объект типа `set`.

Ну и напоследок заметим, что не следует путать механизм работы конструкторов и деструкторов! (Предупреждение, к сожалению, не столь уж и бесполезное, как может показаться.) Как уже отмечалось в предыдущей главе, работа деструктора отличается от работы других функций: при вызове деструктора класса-наследника выполняется не только его код, но и вызывается деструктор его базового класса. Кроме того, в отличие от конструкторов, деструкторы, с точки зрения виртуализации, ведут себя совершенно иначе: деструктор не только может быть виртуальным, но и, зачастую, должен быть таковым. Пояснить разумность такой реализации деструкторов можно на примере матрешек из предыдущей главы. Допустим, имеется следующая иерархия классов:

```
class matreshka_1 {
public:
   matreshka_1 () : size_1 (new int(20)) { }
   ~ matreshka_1 () {
      printf("matreshka_1 destructor\n");
      delete size_1;
   }
   int *size_1;
};

class matreshka_2 : public matreshka_1 {
```

```
public:
   matreshka_2 () : size_2 (new int(10)) { }
   ~ matreshka_2 () {
      printf("matreshka_2 destructor\n");
      delete size_2;
   }
   int *size_2;
};
```

Давайте создадим объект класса `matreshka_2` и сразу же его уничтожим:

```
matreshka_2 m2 = new matreshka_2 ();
delete m2;
```

Если бы в производном классе `matreshka_2` не был определен деструктор, это привело бы к "утечке памяти", так как память, выделенная под переменную `size_2`, не освободилась бы. В этом случае вызвался бы только деструктор базового класса `matreshka_1`. Но поскольку деструктор был переопределен, то вызвались деструкторы обоих классов, и базового, и производного, что является вполне удовлетворительным результатом. Однако предположим, что для удаления объектов используется некая функция, параметром которой является указатель на базовый класс `matreshka_1`:

```
void f(matreshka_1 *m) {
   delete m;
}
```

В этом случае при отработке следующего фрагмента кода:

```
matreshka_2 *m2 = new matreshka_2 ();
f(m2);
```

произойдет нежелательное — вызовется деструктор только базового класса `matreshka_1`, объект `m2` будет уничтожен, а память, выделенная под переменную `size_2`, не освободится. Единственный способ избежать этого — определить деструктор базового класса `matreshka_1` виртуальным, что приведет к последовательному вызову деструкторов: сперва порожденного класса `matreshka_2`, затем базового класса `matreshka_1`.

Абстрактные классы

Как отмечалось ранее, функции могут быть переопределены в классах-потомках. Однако иногда возникает необходимость создания такой иерархии классов, при которой весьма сложно написать разумный вариант реализации функции-члена базового класса. Можно, конечно, на данном уровне отказаться от такой функции, добавив ее классам-потомкам, но при этом теряется возможность однообразного обращения ко всем ее реализациям у потомков. Возникновение такой проблемы является основанием для применения очень важного инструмента проектирования — абстрактных классов.

Чтобы нагляднее продемонстрировать достоинства абстрактных классов, рассмотрим их на примере. Допустим, что построена иерархия классов, состоящая из базового класса `stack` и двух классов-потомков: `stack_array` и `stack_list`, которые представляют собой стеки на основе массивов и списков соответственно. Применяя механизм полиморфизма и динамического связывания, можно написать такой код:

```
stack *s;
stack_array *sa;
stack_list *sl;
void* item;

if (нужен стек как массив)
   s = sa;
if (нужен стек как список)
```

```
   s = sl;

s->push(item);
```

В данном случае функция `push` применяется к телу `s`, а механизм динамического связывания выбирает подходящую версию, зависящую от требуемого вида `s`, который становится известен только во время выполнения (чтобы подчеркнуть это, предположим, что этот вид зависит от выбора при взаимодействии с пользователем). Смысл заключается в предположении, что каждый из классов `stack_array` и `stack_list` имеет свою версию `push`.

Посредством механизма переопределения предложенный вариант решения отработает хорошо. Но сложность создавшегося положения состоит в том, что переопределять-то нечего! Класс `stack` — слишком общее понятие, чтобы иметь осмысленное определение виртуальной функции `push`, так как невозможно написать общий вариант `push`, не имея дополнительной информации о рассматриваемом стеке.

Таким образом, получена ситуация, когда код будет корректно выполнен благодаря динамическому связыванию, но статически некорректен, поскольку `push` — несуществующий метод для `stack`. Механизм контроля типов определит вызов `s->push(x)` как некорректную операцию.

Конечно, можно ввести для `stack` функцию `push`, не делающую ничего. Но это приведет к достаточно бессмысленному понятию класса `stack`, к тому же нарушающему семантику метода `push`. Гораздо лучше описать функции класса `stack` как чисто виртуальные. В C++ это можно сделать, добавив инициализатор `=0`:

```
class stack {
//...
public:
   virtual void push(void* item) = 0;
   virtual void pop() = 0;
// ...
};
```

Класс, в котором есть чисто виртуальные функции, называется *абстрактным классом*. Эти функции могут быть объявлены чисто виртуальными как в самом классе, так и быть унаследованы от класса-предка и не переопределены. Абстрактный класс можно использовать только как базовый для других классов, и этим он несколько напоминает понятие интерфейса в Java. Создавать объекты абстрактного класса запрещено, но использовать их, как было показано выше, вполне возможно.

Необходимо отметить важную особенность абстрактных классов. Абстрактные классы гораздо ближе стоят к понятию абстрактного типа данных, чем обыкновенные классы, которые являются, скорее, реализацией абстрактного типа. Абстрактные классы в принципе покрывают весь спектр между абстрактным типом данных, например `stack`, и пригодными к использованию реализациями, типа `stack_array`.

Как уже отмечалось, наличие различий между проектированием и реализацией несет смертельную угрозу качеству программного обеспечения. Необходимость перехода от одного формализма к другому может привести к ошибкам и угрожает целостности системы.

Напротив, использование абстрактных классов для программирования позволяет применять одинаковый язык к проектированию, реализации и этапам развития. Таким образом устраняется разрыв, и переход от проекта к реализациям может происходить гладко, в пределах одного формализма.

Виртуальные базовые классы

При множественном наследовании достаточно часто возникает ситуация, когда у базовых классов существуют одинаковые свойства. Как эта проблема разрешается в C++, было рас-

смотрено выше. Однако иногда, когда связь между базовыми классами достаточно тесная, может оказаться целесообразным разовое вхождение общего свойства в класс-потомок.

Так как C++ строго типизированный язык, общность информации возможна только при явном указании того, что является общим в этих классах. Для этой цели и используется виртуальный базовый класс.

Допустим, для представления общей информации о работниках служит класс `worker`:

```
class worker {
   char *name;
   int age;
//...
};
```

Определим два его подкласса — инженеров и управляющих:

```
class engineer : public virtual worker {
   list *machines; // список машин и механизмов, обслуживаемых
инженером
//...
};

class manager : public virtual worker {
   list *workers;  // список подчиненных работников
//...
};
```

Тогда работающих на руководящих должностях инженеров можно описать с помощью следующего класса:

```
class manager_engineer : public manager, public engineer {
//...
};
```

Подобное описание избавит новый класс `manager_engineer` от двойного вхождения информации полей `name` и `age` из головного базового класса.

Доступ к компонентам класса

Теперь, "вооружившись" наследованием и полиморфизмом, снова обратимся к инкапсуляции, а точнее, к проблеме ограничения доступа. Абстрагирование уделяет основное внимание внешним особенностям объектов, скрывая детали их реализации. С помощью ограничения доступа можно оградить себя от проблем, возникающих вследствие зависимости частей системы от подробностей внутреннего устройства классов. Таким образом, абстракция и ограничение доступа являются взаимодополняющими компонентами: абстракция сосредоточена на интерфейсе объекта, который объединяет все, что относится к взаимодействию данного объекта с другими объектами, а ограничение доступа — на внутреннем представлении, скрывающем детали реализации от других объектов. Подобное разделение интерфейса и реализации, с точки зрения модулей, позволяет разрешить проблемы малого числа интерфейсов или слабой связности, затронутые ранее. Практическим следствием ограничения доступа является возможность внесения изменений в объект, не затрагивая других объектов, связанных с ним.

Однако на этом пути нас подстерегает целый ряд трудностей. Каким образом отделять общедоступную информацию от скрытой? Что следует предоставлять для общего пользования, а что — сохранить для внутреннего? И наконец, у всех ли пользователей одинаковые права доступа к скрытой информации? Попытаемся ответить на эти вопросы.

Как уже отмечалось, принято различать три типа отношений между классами: отношения разновидности, включения и ассоциативности. Различные языки программирования предоставляют несколько механизмов для отражения этих трех типов отношений. Чаще всего выделяют наследование, использование, принадлежность и метаклассы.

Проблема ограничения доступа тесно связана с вопросами определения отношений между абстракциями. Если вы хотите, чтобы объект класса `A` вызывал функцию `f` объекта класса `B`, то прямо или косвенно класс `B` должен быть доступен для класса `A`, иначе операцию `f` невозможно использовать в классе `A`. Под доступностью понимается возможность одной абстракции обращаться к структурным компонентам другой абстракции. Таким образом взаимоотношения являются мерой допустимости.

В C++ проблемы сокрытия информации разрешаются посредством механизма дружественных функций, как уже отмечалось ранее, и особой стратегии наследования. Для любого класса обычно существует два вида классов-пользователей: классы, которые имеют доступ к элементам данного класса, и подклассы, полученные в результате наследования.

С точки зрения сокрытия информации при реализации наследования возникают следующие проблемы:

- видимость внутренней структуры классов-предков на более низком уровне иерархии;
- видимость внутренней структуры классов-потомков классами-пользователями суперкласса.

Как уже упоминалось, в C++ правила доступа к данным класса основаны на делении обращающихся к классу функций на три вида: функции, реализующие класс (это функции-члены и так называемые функции-друзья, которые часто являются методами клиентов данного класса); функции, реализующие производный класс (функции-друзья и функции-члены производного класса); и все остальные функции. В соответствии с этим делением члены класса подразделяются на три группы по праву доступа к ним: `private` (частные), `protected` (защищенные) и `public` (общие). Функции-члены и функции-друзья класса имеют доступ ко всем членам класса, в том числе и частным. Функциям-друзьям и функциям-членам производного класса доступны защищенные и общие члены класса-предка. Всем остальным функциям доступны только общие члены класса.

Часто возникает вопрос: почему в собственных функциях-членах производного класса нельзя напрямую обращаться к унаследованным свойствам, которые в базовом классе были описаны как частные? Ведь вполне естественно, казалось бы, унаследовав свойства суперкласса, даже если они из частной области, “унаследовать” и правила доступа в подклассе, то есть разрешить доступ к унаследованным частным членам хотя бы функциям-членам подкласса. Однако наличие подобной возможности делает бессмысленным само понятие частного члена, ведь чтобы получить к нему доступ, достаточно было бы просто произвести класс-потомок. И в этом случае, например, будет весьма сложно локализовать ошибку, появившуюся вследствие доступа к свойству класса, поскольку сделать это могла любая функция как базового, так и всех его производных классов.

Давайте рассмотрим на примере, к каким плачевным результатам могла бы привести возможность в классе-наследнике оперировать защищенными данными класса-предка. Предположим, на вашем предприятии введена единая система начисления премий. Даже если появляются новые типы работников, бонусы им должны начисляться так же, как и всем остальным: в зависимости от отработанного времени, коэффициента сложности работ, опыта, наличия ученой степени и т.п. Функция расчета премий `calc_bonus` вызывается из другой, защищенной функции, производящей расчет всех полагающихся работнику выплат. А уже эта функция, в свою очередь, вызывается для всех работников предприятия раз в месяц из соответствующего бухгалтерского пакета, причем не напрямую (поскольку она защищена), а посредством вызова определенной внешней функции, являющейся другом класса, описывающего работников предприятия.

Таким образом, класс `worker` в части расчета премий принимает следующий вид:

```
class worker {
//...
private:
```

```
   money bonus;
   // функция, рассчитывающая премию для данного работника
   void calc_bonus () {
      money b = 0;
      // расчет премии
      bonus += b; //обратите внимание, что премия может быть расчитана,
                  //но не выплачена, поскольку на момент выплаты работник
                  //мог быть в командировке, в отпуске или на больничном
   }
   // ...
protected:
   virtual void calc_payment() {
      calc_bonus();
      //...
   }

   // функция-друг, производящая расчет всех выплат для всех
   // работников предприятия
   friend void calc_payment (list* all_workers);
//...
};

void calc_payment (list* all_workers) {
   do {
      worker* w = (worker*)all_workers->getdata();
      w-> calc_payment();
   } while (all_workers->next());
}
```

Предположим, что на предприятии появилась новая категория работников, допустим, водителей, и программисты получили задание должным образом расширить программное обеспечение. Если бы в классах-наследниках можно было бы обращаться к частным членам базового класса, это могло бы привести к следующей ситуации:

```
class driver : public worker {
//...
protected:
   void calc_payment() {
      calc_bonus();
      calc_bonus();
      //...
   }
//...
};
```

Неважно, умышленно или по ошибке в переопределенной функции `calc_payment` была дважды вызвана функция расчета премий `calc_bonus`, однако это привело бы к удвоению суммы премии для всех водителей предприятия.

Подобно тому как члены класса делятся на частные, защищенные и общие, так и базовый класс в C++ в классе-потомке можно описать как частный, защищенный или общий:

```
class A {
//...
};

class B : public /* protected или private */ A {
//...
};
```

Такая запись устанавливает правило для неявного преобразования указателей на объект класса-потомка к указателям на объект класса-предка. Если базовый класс описан как общий, то преобразование корректно во всех функциях. Если же класс описан как частный, то преобразование возможно только в функциях, реализующих класс-потомок. Когда класс `A` — защищенный суперкласс класса `B`, то такие операции могут выполняться в функциях, реализующих класс `B` и все классы-потомки `B`. Поэтому функция `f()`, определенная следующим образом:

```
void f(A *a, B *b)

{
   a = b;
}
```

отработает правильно в трех случаях:

- `A` — общий класс-предок `B` и `f()` — произвольная функция;
- `A` — частный класс-предок `B`, а функция `f()` реализует класс `B` (напомним, что класс реализуют функции-члены и функции-друзья);
- `A` — защищенный класс-предок `B`, а функция `f()` реализует либо класс `B`, либо любой из его потомков.

Продемонстрируем вышеприведенные правила на простой иерархии классов:

```
class A {
public:
   int val;
   // компилируется
   virtual void show_val(){ printf("class A, val = %i", val); }
};

class A_1 : protected A {
public:
   // компилируется
   virtual void show_val(){ printf("class A_1, val = %i", val); }
};

class A_2 :  private A {
public:
   // компилируется
   virtual void show_val(){ printf("class A_2, val = %i", val); }
};

class A_1_1 : public A_1 {
public:
   // компилируется
   virtual void show_val(){ printf("class A_1_1, val = %i", val); }
};

class A_2_1 : public A_2 {
public:
   // не компилируется, поскольку val здесь уже частная переменная.
   // Ошибка: "'val' not accessible because 'A_2' uses 'private'
   // to inherit from 'A'"
   virtual void show_val(){ printf("class A_2_1, val = %i", val); }
};

int main(int argc, char* argv[])
{
   A a;
```

```
   a.val = 0;      // компилируется

   A_1 a_1;
   a_1.val = 0;    // не компилируется

   A_2 a_2;
   a_2.val = 0;    // не компилируется

   A_1_1 a_1_1;
   a_1_1.val = 0; // не компилируется
}
```

Поскольку наследование можно рассматривать двояко — как задание определенного отношения между классом-потомком и классом-предком и как удобный способ передачи части кода одной структуры другой — то полезной может оказаться возможность отражения подобной двойственности понятия наследования. Если базовый класс объявлен как общий для данного подкласса, то такой подкласс можно считать подтипом суперкласса. Если же суперкласс объявлен как частный либо защищенный для подкласса, то такой подкласс не является подтипом суперкласса, хотя поведение и данные полностью наследуются. Действительно, может возникнуть ситуация, когда класс-потомок сохраняет все свойства предка, являясь, по сути, качественно новой сущностью (например, автомобиль можно считать наследником телеги, поскольку он имеет четыре колеса, однако заявление о том, что автомобиль является телегой, вероятно, будет излишне смелым). Поэтому обращение с ней должно отличаться от обращения с сущностью, имеющей тип суперкласса, и уж ни в коем случае нельзя допустить их взаимозаменяемости. Именно для таких целей и может пригодиться возможность определения базового класса как частного для класса-потомка.

Проблема наследования реализации

Поскольку C++ обладает весьма солидным арсеналом средств для разработки качественного программного обеспечения, существует проблема выбора средств, необходимых именно для вашего проекта, или, иными словами, перед программистом всегда стоит вопрос выбора адекватного стиля программирования. Так, например, для разработчика на C++ нет технических препятствий для построения системы исключительно в процедурном стиле. Однако, не рассматривая таких крайне радикальных решений, затронем один вопрос, напрямую связанный именно с объектно-ориентированной парадигмой.

Многие программисты при разработке программного обеспечения всячески стараются избегать явления, называемого наследованием реализации. Благо, существует такое средство, как абстрактные классы, которые позволяют ограничиться исключительно наследованием интерфейса. Не вдаваясь в философские рассуждения и споры о преимуществах того или иного стиля программирования, попытаемся на примере пояснить, что такое наследование реализации, и какую потенциальную угрозу оно может представлять.

Наследование реализации — явление, при котором производный класс заимствует код, созданный для базового класса, именно таким образом достигается повторная используемость кода. Давайте рассмотрим следующий базовый класс `A`:

```
class A {
public:
   virtual void f1 () {
      printf ("class A, function f1\n");
   }
   void f2 () {
      printf ("class A, function f2\n");
      f1 ();
```

```
   }
};
```

Создадим класс-наследник `B`, переопределив виртуальную функцию `f1`:

```
class B :public A {
public:
   virtual void f1 () {
      printf ("class B, function f1\n");
      f2 ();
   }
};
```

В этом случае синтаксически корректный вызов функции `f1` для экземпляра класса `B` приведет к переполнению стека:

```
B b;
b.f1();
```

Вызвано это чрезмерной связанностью классов `A` и `B`. В приведенном примере демонстрируется весьма одиозная ошибка, которую выявить и исправить не так уж и сложно, но, к сожалению, зачастую ошибки, вызванные наследованием реализации, гораздо тоньше и их обнаружение может превратиться в серьезную проблему. Пример всего лишь иллюстрирует следующее: при наследовании реализации порожденный класс нельзя понять без понимания реализации унаследованных методов в его базовом классе, а базовый класс, в свою очередь, нельзя понять, не уяснив, какие из его методов будут переопределены в классах-наследниках.

Осторожно — переиспользуемый код!

И напоследок хотелось бы рассказать одну программистскую легенду. Насколько правдива эта история, никто не берется утверждать, однако следует признать, что она очень характерна для разработок, основывающихся на объектно-ориентированной парадигме. Рассказывают, что в австралийской армии для тренировок экипажей боевых вертолетов активно используются симуляторы. Поскольку кенгуру, разбегающиеся от вертолетного шума, представляли определенную угрозу для военных, так как своим поведением могли выдавать расположение воинских частей, командование распорядилось смоделировать поведение кенгуру при приближении к ним вертолетов. Программисты профессионально подошли к выполнению задания и решили переиспользовать код, который отвечал за поведение пехоты в подобной ситуации. Каково же было удивление вертолетчиков, когда во время симуляции низко пройдясь над стадом кенгуру, они увидали, что животные, как и ожидалось, сперва разбежались, а затем, перегруппировавшись, нанесли по вертолетам мощный удар из стингеров. Говорят, что с тех пор летчики боятся кенгуру как огня, чего, в общем-то, и хотели добиться. Однако подобные истории не всегда заканчиваются столь благостно, и программисту надо быть особенно бдительным при наследовании и переопределении методов.

Контрольные вопросы

1. Какова роль полиморфизма в реализации идей инкапсуляции?
2. Что такое виртуальная функция и каков механизм позднего связывания?
3. Какие классы называются абстрактными и когда возникает необходимость в их использовании?
4. Имеется иерархия классов и созданы объекты этих классов:

```
class A {
public:
   virtual void print() {
```

```
      printf("class A");
   }
};

class B : public A {
public:
   virtual void print() {
      printf("class B");
   }
};

   A a;
   B b;
   A* pa = new A();
   A* pb = new B();
```

Какие экземпляры виртуальной функции отработают при следующих вызовах?

```
   a.print();
   b.print();
   pa->print();
   pb->print();
```