Курс c++

Царьков Олег

Описание тикета OTSN-11

Наследование классов.

Наследование классов — это такой механизм, при котором содержимое одного из классов целиком переходит внутрь содержимого другого класса(со всеми переменными и всеми функциями).

Это полезно тогда, когда мы не хотим переписывать это содержимое несколько раз. Например, имеется несколько классов, в которых должна быть написана одна и та же функция. Чтобы не писать ее 10 раз, нужно создать вспомогательный класс, в котором она есть, а потом во всех остальных классах "отнаследоваться" от него. Этот вспомогательный класс будет называться БАЗОВЫМ классом.

Рассмотрим пример кода:

```
#include <iostream>
#include <cstdlib>

class Book {
private:
    size_t pages_number;
public:
    explicit Book(const size_t num)
     : pages_number(num)
    {}

    size_t GetPagesNumber() const {
        return pages_number;
    }

    virtual void WhatItIs() const = 0;
};

class MathBook : public Book {
public:
    MathBook(const size_t num)
     : Book(num)
    {}

    void WhatItIs() const {
        std::cout << "It is mathematics book\n";
```

```
    }
};

class PhysicsBook : public Book {
public:
    PhysicsBook(const size_t num)
    : Book(num)
    {}

    void WhatItIs() const {
        std::cout << "It is physics book\n";
    }
};

int main() {
//    Book book(5);
//    std::cout << "Number of pages = " << book.GetPagesNumber() << "\n";
//    book.WhatItIs();

    MathBook book2(6);
    std::cout << "Number of pages = " << book2.GetPagesNumber() << "\n";
    book2.WhatItIs();

    PhysicsBook book3(7);
    std::cout << "Number of pages = " << book3.GetPagesNumber() << "\n";
    book3.WhatItIs();

    return 0;
}
```

В этом примере мы хотим описать два класса, один из которых $MathBook$, другой $PhysBook$, и мы хотим, чтобы у них был один и тот же констуктор, принимающий количество страниц, и один и тот же метод $GetPagesNumber$, который возвращает количество страниц. А также мы хотим, чтобы был метод $WhatItIs$ печатающий, что это за книга, но он будет разный у этих двух классов.

Для того, чтобы два раза не писать один и тот же констуктор, одну и ту же переменную $pages\_number$, одну и ту же функцию $GetPagesNumber$, мы пишем БАЗОВЫЙ класс $Book$, в котором уже есть и констуктор, и нужные нам методы.

Перед методами, которые мы хотим впоследствии переопределять, мы пишем слово $virtual$. В данном случае это относится к строчке

$$virtual\ void\ WhatItIs()\ const = 0;$$

Этот метод будет разный во всех классах, поэтому мы будем его переопределять. Вместо тела у него написано = 0. Это означает, что у этой функции нет тела, тем самым мы запрещаем создавать объект класса *Book* и говорим, что можно создавать только объекты тех классов, где будет прописано тело этой функции при наследовании(нельзя создать объект класса, если у какой-то его функции нет тела).

Само наследование содержится в строчках:

$$class\ MathBook\ :\ public\ Book\ \{$$

и

$$class\ PhysicsBook\ :\ public\ Book\ \{$$

Эти строчки заставляют компилятор влить содержимое класса *Book* в классы *MathBook* и *PhysBook*. Слово *public*, которое там написано, подсказывает, как именно сливать области *private* и *public* в классах. Если написано *public*, то *public* у *Book* сливается с *public* у *MathBook*, а *private* сливается с *private*.

Если написать

$$class\ MathBook\ :\ private\ Book\ \{$$

то все содержимое *Book* войдет в область *private* у класса *MathBook*.

Есть еще область *protected*, она как область *private* для данного класса и она видна его наследникам тоже как *private*(*protected* — то есть защищенная от внешнего использования, но передающаяся по наследству). При наследовании можно написать

$$class\ MathBook\ :\ protected\ Book\ \{$$

тогда содержимое полностью переходит в область *protected*.

Поскольку содержимое БАЗОВОГО класса скопировалось в КЛАСС-НАСЛЕДНИК, то при вызове констуктора в наследнике нужно в списке инициализации вызвать конструктор базового класса, как это сделано в примере.

В результате всего этого имеем, что в классах *MathBook* и *PhysBook* есть функция *GetPagesNumber*, возвращающая количество страниц, хотя мы такую функцию не писали в этих классах.

Задание 1. Создать БАЗОВЫЙ класс *Animal*, и унаследовать от него классы *Cat* и *Dog*. Требуется, чтобы у каждого класса *Cat* и *Dog* была фукнция *Sound*, которая в случае *Cat* печатает на экран *meow*, а в случае *Dog* печатает на экран *bark*!, причем сама фукнция *Sound* должна быть в базовом класса, и не должна определяться в классах-наследниках.

Множественное наследование.

Можно унаследоваться от нескольких классов, тогда все их содержимое перейдет к наследнику.

Ничего существенно нового для раздумий не возникает. Рассмотрим пример:

```
#include <iostream>
#include <string>

class Stupid {
public:
    void AreYouStupid() const {
        std::cout << "Yes, I am very stupid\n";
    }
};

class Girl {
private:
    std::string name;
public:
    Girl(const std::string& name)
     : name(name)
    {}

    void WhoAreYou() const {
        std::cout << "My name is " << name << ", i am a girl\n";
    }
};

class StupidGirl : public Stupid, public Girl {
public:
    StupidGirl(const std::string& name)
     : Stupid()
     , Girl(name)
    {}
};

int main() {
    StupidGirl stupid_girl("N.");
    stupid_girl.WhoAreYou();
    stupid_girl.AreYouStupid();
}
```

Программа выведет на экран

My name is N., i am a girl
Yes, I am very stupid

Задание 2. Написать любой пример с двумя БАЗОВЫМИ классами и одним НАСЛЕДНИКОМ, похожий на то, что выше.

Требуется, чтобы в обоих базовых классах была хотя бы одна виртуальная функция, и хотя бы одна невиртуальная.

Шаблон проектирования Visitor.

Для контейнеров есть итераторы, которые умеют пробегаться по всем элементам какого-либо контейнера.

Каждый раз при использовании итератора приходилось писать такой код

$$
\begin{array}{l}
for\ (Container :: iterator\ it\ =\ var.begin();\ it\ !=\ var.end();\ ++it)\ \{ \\
\quad //something \\
\}
\end{array}
$$

Возникает вопрос: зачем сделано так, что все время надо писать один и тот же цикл?

Действительно, можно сделать так, чтобы этого не надо было делать с помощью шаблона проектирования $Visitor$. Он заключается в следующем:

• Имеется базовый класс $ContainerVisitor$ с функцией $on\_visited$, которая принимает элемент контейнера и что-то с ним делает(или у которой нет тела)

• внутри класса $Container$ имеется функция $visit$, которая принимает объект класса $ContainerVisitor$, перебирает все элементы контейнера и для каждого вызывает функцию $on\_visited$

• Остальные визиторы наследуются от класса $ContainerVisitor$ и как-то переопределяют функцию $on\_visited$, чтобы делать с объектами класса что-то другое.

В результате, для того, чтобы что-то массовое сделать со всеми элементами, нужно всего лишь вызвать функцию $visit$ у контейнера и отдать ей правильный визитор.

Задание 3. Написать визиторы у классов $Array, Matrix$ и $List$.