Урок 2

Наследование

Управление доступом к базовому классу. Конструкторы и наследование. Создание многоуровневой иерархии классов. Указатели на производные классы. Простое и множественное наследование.

Константные объекты классов

По аналогии с фундаментальными типами данных (**int**, **float**, **char**) объекты классов также могут быть константными. Для таких объектов запрещается изменять переменные-члены, а также вызывать их напрямую (даже если они **public**). Но есть способ получить значение переменной: использовать константный метод, который также должен быть объявлен в классе (например, константная **get**-функция-член). Константный метод гарантирует, что объект не будет изменяться. Рассмотрим следующий пример:

```
class Time
{
public:
	int m_hours;

	Time() { m_hours = 0; }

	void resetHours() { m_hours = 0; }
	void setHours(int value) { m_hours = value; }

	int getHours() const { return m_hours; }
	// ключевое слово const находится после списка параметров, но перед телом функции
};
```

В данном примере мы можем вызвать метод **getHours()** через любой константный объект.

Старайтесь делать все методы, которые не изменяют данные объекта, константными.

Если нужно передать внешней функции объект, сделаем это по ссылке, так как по значению будет нерационально. Передадим константную ссылку, чтобы функция случайно не изменила значение своего параметра.

```
#include <iostream>

class Date
{
private:
    int m_day;
    int m_month;
    int m_year;

public:
    Date(int day, int month, int year)
    {
        setDate(day, month, year);
    }

    void setDate(int day, int month, int year)
    {
        m_day = day;
        m_month = month;
        m_year = year;
    }

    int getDay() const { return m_day; }
    int getMonth() const { return m_month; }
    int getYear() const { return m_year; }
};

// Мы передаем объект date по константной ссылке, чтобы избежать создания копии объекта date
void printDate(const Date &date)
{
    std::cout << date.getDay() << "." << date.getMonth() << "." << date.getYear() << '\n';
}

int main()
{
    Date date(12, 11, 2018);
    printDate(date);

    return 0;
}
```

В данном примере передаем функции **printDate()** константный объект. Через него вызываем методы **getDay()**, **getMonth()**, **getYear()**, которые тоже должны быть константными.

Определение методов вне класса

Функции-члены могут быть определены как внутри класса, так и за его пределами. Первый способ определения мы уже рассматривали — поговорим о втором.

Чтобы функцию-член класса определить вне класса, в нем определяют только прототип самой функции, а ее тело — вне класса, используя в качестве префикса имя класса с оператором разрешения области видимости (::).

```
class Date
{
private:
    int m_day;
    int m_month;
    int m_year;

public:
    Date(int day, int month, int year);

    void SetDate(int day, int month, int year);

    int getDay() { return m_day; }
    int getMonth() { return m_month; }
    int getYear()  { return m_year; }
};

// Конструктор класса Date
Date::Date(int day, int month, int year)
{
    SetDate(day, month, year);
}

// Метод класса Date
void Date::SetDate(int day, int month, int year)
{
    m_day = day;
    m_month = month;
    m_year = year;
}
```

Скрытый указатель this

Может возникнуть вопрос: как при вызове метода класса C++ отслеживает, какой объект его вызвал? Ответ: C++ использует скрытый указатель **this**!

Пусть объект **date** класса **Date** вызывает метод **setDate()**: **date.setDate(4, 5, 2019**). Хотя кажется, что здесь только три аргумента, на самом деле их четыре. Во время компиляции эта строчка будет конвертирована в следующую: **setDate(&date, 4, 5, 2019)**.

Теперь это всего лишь стандартный вызов функции, а объект **date** теперь передается по адресу в качестве аргумента функции.

Поскольку в вызове функции теперь четыре аргумента, то и метод нужно изменить соответствующим образом — чтобы он их принимал:

```
void setDate(Date* const this, int day, int month, int year) {

 this->m_day = day;

this->m_month = month;

this->m_year = year; }
```

При компиляции обычного метода компилятор неявно добавляет к нему параметр **this** — это скрытый константный указатель, который содержит адрес объекта, который вызывает метод класса.

Еще одна деталь: внутри метода также необходимо обновить все члены класса (функции и переменные), чтобы они ссылались на объект, который вызывает этот метод. Это легко сделать, добавив префикс **this->** к каждому из них. Таким образом, в теле функции **setDate()**, **m_day** (переменная-член класса) будет конвертирована в **this->m_day** и так далее. И когда ***this** указывает на адрес **date**, то **this->m_day** будет указывать на **date.m_day**.

Указатель ***this** является скрытым параметром, который неявно добавляется к каждому методу класса. В большинстве случаев нам не нужно обращаться к нему напрямую, но при необходимости это можно сделать. Стоит отметить, что **this** является константным указателем: вы можете изменить значение исходного объекта, но нельзя заставить **this** указывать на что-то другое.

```
class Day {
private:
    int day;
public:
    Day(int day) {
        this->day = day;
    }
};

int main() {
    Day(4);
}
```

Базовое наследование

Идею наследования в C++ легко понять через аналогию с реальной жизнью. Яблоки и груши — это фрукты, и они унаследовали все свойства, которые имеют такие плоды (цвет, размер и подобное). Но и у яблок, и у груш есть индивидуальные свойства.

Вернемся к программированию. Наследование в C++ происходит между классами. **Родительский (базовый)** класс — это тот, от которого наследуются свойства и методы. **Дочерний (производный)** класс — это тот, который наследует.

Определим родительский класс **Human()**:

```
#include <string>

class Human
{
public:
    std::string m_name;
    int m_age;

    Human(std::string name = "", int age = 0)
    : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

В нем определены только общие свойства, которые есть у любого человека: имя и возраст.

Теперь определим дочерний класс **Employee()**:

```
// Employee открыто наследует Human
class Employee : public Human
{
public:
    string m_employer;
    double m_wage;

    Employee(string employer, double wage)
    : m_employer(employer), m_wage(wage)
    { }
};
```

В этом классе мы добавили еще две переменные-члена, которые являются специфичными для данного класса. При этом переменные класса **Human()** также являются переменными-членами класса **Employee()**. Обратите внимание, что тип наследования **public** (об этом поговорим дальше), а родительский класс указывается через двоеточие.

```
int main()
{
    // Создаем нового сотрудника
    Employee anton;
    // Присваиваем ему имя (мы можем делать это напрямую, так как m_name
является public)
    anton.m_name = "Anton";
    // Выводим имя сотрудника
    std::cout << anton.getName() << '\n';
    // используем метод getName(), который мы унаследовали от класса Human
    return 0;
}
```

Результат выполнения этого кода: **Anton**

От одного класса могут наследовать несколько классов: от **Human()** может наследовать **Employee()** и другие.

Также можно создавать цепочки наследований: от класса **Employee()** может наследовать класс **Supervisor()**. Для него родительским классом будет **Employee()**.

Построение дочернего класса начинается с самого верхнего родительского класса и заканчивается нижним классом иерархии. По мере построения выполняются и конструкторы: сначала конструктор топового класса, а в конце — дочернего. Деструкторы выполняются в обратном порядке.

Чтобы создать объект дочернего класса, необязательно создавать объект базового. А чтобы передать значения конструктору базового класса, не создавая сам объект, следует вызвать конструктор базового класса с нужными параметрами в списке инициализации производного класса. Пример такого объявления:

```
#include <string>
using namespace std;
class Human
{
private:
    string m_name;
    int m_age;

public:
    Human(string name = "", int age = 0) : m_name(name), m_age(age )
    { }

    string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
class Employee : public Human
{
private:
    string m_employer;
    double m_wage;
public:
    Employee(string name = "", int age = 0, string employer, double wage)
    : Human(name, age), // вызывается Human(std::string, int) для инициализации
членов name и age
      m_employer(employer), m_wage(wage)
    { }
    string getEmployer() const { return m_employer; }
    double getWage() const { return m_wage; }
};
```

Преимущество наследования в том, что не надо переопределять информацию из родительских классов в дочерние. Это эффективный способ разработки сложных программ. Например, все производные классы наследуют изменения родительского.

Дружественные функции

Дружественные функции имеют доступ к закрытым членам класса. Это могут быть как обычные функции, так и методы других классов. Дружественные функции объявляются с помощью ключевого слова **friend** перед прототипом функции в том классе, дружественной для которого вы хотите ее сделать. Тело функции может быть объявлено в любом другом месте программы. Рассмотрим пример:

```
class Cat
{
private:
    int m_age;
    // Делаем функцию resetAge() дружественной классу Cat
    friend void resetAge(Cat &cat);
};

// resetAge() теперь является другом класса Cat
void resetAge(Cat &cat)
{
    // И мы имеем доступ к закрытым членам объектов класса Cat
    cat.m_age = 0;
}

int main()
{
    Cat Frisky;
    resetAge(Frisky);
    return 0;
}
```

Отметим, что в качестве параметра функция **resetAge()** принимает объект класса **Cat**.

Можно создавать дружественные функции для нескольких классов.

```
class Cat; //это прототип, сам класс определяем позже

class Dog
{
// ...
    friend void getAge(const Cat &cat, const Dog &dog);
};

class Cat
{
// ...
    friend void getAge(const Cat &cat, const Dog &dog);
};

void getAge(const Cat &cat, const Dog &dog)
{
// ...
}
```

В данном случае необходимо в самом начале объявить прототип класса **Cat**. Без этой строчки компилятор выдаст ошибку, так как упоминание об этом классе есть в дружественной функции. Обратите внимание, что параметры в функцию передаются по ссылке. Это необходимо для того, чтобы не создавались копии объектов, так как этот процесс может быть неэффективным. **Const** обеспечивает, что данные классы не будут изменяться в функции.

Дружественные классы

Один класс можно сделать дружественным другому. Это откроет всем членам первого класса доступ к закрытым членам второго:

```
#include <iostream>
using namespace std;

class Dog
{
private:
    int m_age;
public:
    Dog (int age) : m_age(age)
    { }
    // Делаем класс Cat другом классу Dog
    friend class Cat;
};

class Cat
{
private:
    int m_age;
public:
    Cat (int age) : m_age(age)
    { }
    bool compareAge(Cat &cat, Dog &dog) {
        // у нас есть доступ к private переменным класса Dog
        return dog.m_age<cat.m_age;
    }
};

int main() {
    Cat Frisky(4);
    Dog Spike(3);
    if (Frisky.compareAge(Frisky, Spike))
        cout << "Кот старше собаки";
    else
        cout << "Кот не старше собаки";
    return 0;
}
```

Поскольку класс **Cat** — другом класса **Dog**, то любой из членов **Cat** имеет доступ к private-членам Dog.

Обратите внимание: **Cat** — друг **Dog**, но **Cat** не имеет прямого доступа к указателю **this** * объектов **Dog**. И это не означает, что **Dog** также является другом **Cat**. Если хотите сделать оба класса дружественными, то они должны указать друг друга в таком качестве.

Спецификатор доступа protected

Спецификатор **protected** открывает доступ к члену класса для дружественных функций и дочерних классов.

```
class Parent
{
public:
	int m_public;		// доступ к этому члену открыт для всех объектов
private:
	int m_private;		// доступ к этому члену открыт только для других членов класса Parent и для дружественных классов/функций (но не для дочерних классов)
protected:
	int m_protected;	// доступ к этому члену открыт для других членов класса Parent, дружественных классов/функций, дочерних классов
};

class Child : public Parent
{
public:
	Child()
	{
		m_public = 1;	// разрешено: доступ к открытым членам родительского класса из дочернего класса
		m_private = 2;	// запрещено: доступ к закрытым членам родительского класса из дочернего класса
		m_protected = 3;	// разрешено: доступ к защищенным членам родительского класса из дочернего класса
	}
};

int main()
{
	Child child;
	child.m_public = 1; // разрешено: доступ к открытым членам класса извне
	child.m_private = 2; // запрещено: доступ к закрытым членам класса извне
	child.m_protected = 3; // запрещено: доступ к защищенным членам класса извне
}
```

Спецификатор доступа **protected** имеет смысл применять, только если количество дочерних классов невелико. Если в члене **protected** появятся изменения, их придется вносить как в родительский класс, так и во все дочерние.

Типы наследования

Тип наследования определяется с помощью спецификатора доступа, который указывается возле наследуемого класса. По умолчанию тип наследования определяется как **private**.

Всего есть 9 комбинаций наследования для членов класса:

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе	Спецификатор доступа при наследовании типа private в дочернем классе	Спецификатор доступа при наследовании типа protected в дочернем классе
Public	Public	Private	Protected
Private	Недоступен	Недоступен	Недоступен
Protected	Protected	Private	Protected

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Child: public Parent   // открытое наследование
{
    // Открытое наследование означает, что:
    // члены public остаются public в дочернем классе
    // члены protected остаются protected в дочернем классе
    // члены private остаются недоступными в дочернем классе
public:
    Child()
    {
        m_public = 1;     // разрешено: доступ к m_public открыт
        m_private = 2;    // запрещено: доступ к m_private в дочернем классе из
родительского класса закрыт
        m_protected = 3;  // разрешено: доступ к m_protected в дочернем классе
из родительского класса открыт
    }
};

int main()
{
    Parent parent;
    parent.m_public = 1;    // разрешено: m_public доступен извне через
родительский класс
    parent.m_private = 2;   // запрещено: m_private недоступен извне через
родительский класс
    parent.m_protected = 3; // запрещено: m_protected недоступен извне через
родительский класс

    Child pub;
    child.m_public = 1;     // разрешено: m_public доступен извне через
дочерний класс
    child.m_private = 2; // запрещено: m_private недоступен извне через
дочерний класс
    child.m_protected = 3; // запрещено: m_protected недоступен извне через
дочерний класс
}
```

Самое распространенное наследование — открытое, оно же — самое легкое. Когда дочерний класс открыто наследует родительский, унаследованные члены **public** остаются **public**, **protected** — **protected**, а унаследованные члены **private** по-прежнему недоступны для дочернего класса. Используйте именно этот тип наследования, если нет острой необходимости делать иначе.

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Child: private Parent  // закрытое наследование
{
    // Закрытое наследование означает, что:
    // члены public становятся private (m_public теперь private) в дочернем
классе
    // члены protected становятся private (m_protected теперь private) в
дочернем классе
    // члены private остаются недоступными (m_private недоступен) в дочернем
классе
public:
    Child()
    {
        m_public = 1;        // разрешено: m_public теперь private в Priv
        m_private = 2;       // запрещено: дочерние классы не имеют доступ к
закрытым членам родительского класса
        m_protected = 3;     // разрешено: m_protected теперь private в Priv
    }
};

int main()
{
    Parent parent;
    parent.m_public = 1;     // разрешено: m_public доступен извне через
родительский класс
    parent.m_private = 2;    // запрещено: m_private недоступен извне через
родительский класс
    parent.m_protected = 3;  // запрещено: m_protected недоступен извне через
родительский класс

    Child child;
    child.m_public = 1;     // запрещено: m_public недоступен извне через
дочерний класс
    child.m_private = 2;    // запрещено: m_private недоступен извне через
дочерний класс
    child.m_protected = 3;  // запрещено: m_protected недоступен извне через
дочерний класс
}
```

При закрытом наследовании все члены родительского класса наследуются как закрытые и поэтому невозможно к ним обратиться извне дочернего класса. Данный тип наследования полезен, когда дочерний класс слабо связан с родительским и мы не хотим, чтобы открытые члены были доступны через объекты дочернего класса.

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Child: private Parent  // закрытое наследование
{
    // Закрытое наследование означает, что:
    // члены public становятся private (m_public теперь private) в дочернем
классе
    // члены protected становятся private (m_protected теперь private) в
дочернем классе
    // члены private остаются недоступными (m_private недоступен) в дочернем
классе
public:
    Child()
    {
        m_public = 1;    // разрешено: m_public теперь private в Priv
        m_private = 2;   // запрещено: дочерние классы не имеют доступ к
закрытым членам родительского класса
        m_protected = 3; // разрешено: m_protected теперь private в Priv
    }
};

int main()
{
    Parent parent;
    parent.m_public = 1;    // разрешено: m_public доступен извне через
родительский класс
    parent.m_private = 2;   // запрещено: m_private недоступен извне через
родительский класс
    parent.m_protected = 3; // запрещено: m_protected недоступен извне через
родительский класс

    Child child;
    child.m_public = 1; // запрещено: m_public недоступен извне через дочерний
класс
    child.m_private = 2; // запрещено: m_private недоступен извне через дочерний
класс
    child.m_protected = 3; // запрещено: m_protected недоступен извне через
дочерний класс
}
```

Тип наследования **protected** почти никогда не используется. С защищенным наследованием члены **public** и **protected** становятся **protected**, а члены **private** остаются недоступными.

Указатели и ссылки на производные классы

Если создан указатель на один тип данных, он не сможет указывать на другой тип. Но в ООП указатель на объект базового класса также может указывать на объект производного. Покажем это на примере:

```
Animal *p_animal;               // указатель на объект типа Animal
Animal animal;                  // объект типа Animal
Dog dog;                        // объект типа Dog
                               // Следующие присвоения будут безошибочными
p_animal = &animal;             // p_animal указывает на объект типа Animal
p_animal = &dog;                // p_animal указывает на объект типа Dog,
являющийся объектом, порожденным от Animal
```

В данном случае указатель **p_animal** может хранить адрес как объекта класса **Animal**, так и объекта класса **Dog**.

Но указатель **p_animal** может получить доступ только к членам класса **Dog**, которые были унаследованы от класса **Animal**. К специфичным членам класса **Dog** указатель не имеет доступа.

В следующем примере показывается использование указателя на базовый класс.

```
#include <iostream>
#include <string.h>
using namespace std;

class Animal {
protected:
    int m_age;
public:
    void set_age(int age) {m_age=age;}
    void show_age() {cout << m_age << endl; }
};

class Dog: public Animal {
    char m_breed[80];              // кличка собаки
public:
    void set_breed(char *breed) {
        strcpy(m_breed, breed);
    };
    void show_breed () {
        cout << m_breed << endl;
    }
};

int main()
{
    Animal *p_animal;
    Animal animal;
    Dog dog;
    p_animal = &animal;            // адрес объекта базового класса
                                   // доступ к Animal через указатель
    p_animal->set_age(5);
                                   // доступ к Dog через указатель
    p_animal = &dog;
    p_animal->set_age(10);
                                   // показать каждое имя соответствующего
объекта
    animal.show_age();
    dog.show_age();
                                   // dog.set_breed("poodle");
                                  //((Dog *)p_animal) ->show_breed();


    cout << "\n";
    return 0;
}
```

p_animal может указывать на члены класса **Animal**, а также на члены класса **Dog**, которые были определены в базовом классе. Вместе с этим нужно помнить, что указатель нельзя использовать для доступа к члену **show_breed()**, пока не выполнено приведение типов:

```
((Dog *)p_animal) ->show_breed();
```

Но такую конструкцию употреблять не рекомендуется, так как это может вызвать дополнительные ошибки в коде.

Множественное наследование

В C++ существует множественное наследование: когда у одного дочернего класса несколько родительских. Например, нужно создать класс **Teacher**. Этот класс может наследовать от классов **Employee** и **School**. Синтаксис множественного наследования:

```
#include <string>
using namespace std;

class School
{
private:
    int m_number;
        string m_type;
public:
    School(int number, string type) : m_number(number), m_type(type)
    { }
};

class Employee
{
private:
    string m_employer;
    double m_wage;
public:
    Employee(string employer, double wage)
    : m_employer(employer), m_wage(wage)
    { }
};

// класс Teacher открыто наследует свойства классов Human и Employee
class Teacher: public School, public Employee
{
private:
    int m_teachesGrade;
public:
    Teacher(int number, string type, string employer, double wage, int
teachesGrade)
    : School(number, type), Employee(employer, wage),
m_teachesGrade(teachesGrade)
    { }
};
```

У класса **School** есть переменные-члены **m_number**, **m_type**. Это то, что есть у любой школы. У класса **Employee** есть переменные-члены **m_employeer**, **m_wage**. Эти свойства присущи любому работнику. Поскольку у учителя есть все свойства школы и работника, то класс **Teacher** наследует свойства классов **School** и **Employee**. Но у педагога есть еще своя квалификация, поэтому в классе **Teacher** есть переменная-член **m_teachesGrade**.

Множественное наследование — мощное средство языка C++, но оно же может привести к серьезным проблемам:

1. ***Когда дочерний класс пытается вызвать метод, который есть в обоих родительских классах.*** Компилятор сообщит об ошибке, так как не будет знать, какой из методов вызвать. Эту проблему можно устранить путем явного указания класса, которому принадлежит метод.

2. ***Когда один класс наследует от двух классов, которые в свою очередь наследуют от одного и того же родительского класса (алмаз смерти)***. Проблемы заключаются в

неоднозначности вызова методов, а также повторном копировании членов базового класса в дочерний класс.

Множественное наследование запрещено использовать в некоторых языках программирования (Java и C#) для обычных классов, но для интерфейсных такое ограничение снято.

Таким образом, множественное наследование следует применять только в случаях крайней необходимости, если другие решения недоступны или слишком сложны.

Практическое задание

1. Создать класс **Person** (человек) с полями: имя, возраст, пол и вес. Определить методы переназначения имени, изменения возраста и веса. Создать производный класс **Student** (студент), имеющий поле года обучения. Определить методы переназначения и увеличения этого значения. Создать счетчик количества созданных студентов. В функции main() создать несколько студентов. По запросу вывести определенного человека.

2. Создать классы **Apple** (яблоко) и **Banana** (банан), которые наследуют класс **Fruit** (фрукт). У **Fruit** есть две переменные-члена: **name** (имя) и **color** (цвет). Добавить новый класс **GrannySmith**, который наследует класс **Apple**.

```
int main()
{
    Apple a("red");
    Banana b;
    GrannySmith c;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
    std::cout << "My " << c.getName() << " is " << c.getColor() << ".\n";

    return 0;
}
```

Код, приведенный выше, должен давать следующий результат:

```
My apple is red.
My banana is yellow.
My Granny Smith apple is green.
```

3. Изучить правила игры в Blackjack. Подумать, как написать данную игру на C++, используя объектно-ориентированное программирование. Сколько будет классов в программе? Какие классы будут базовыми, а какие производными? Продумать реализацию игры с помощью классов и записать результаты.

Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования C++.

2. Стивен Прата. Язык программирования C++. Лекции и упражнения.

3. Роберт Лафоре. Объектно-ориентированное программирование в C++.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Наследование классов в C++.
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.