УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационной безопасности

Кафедра инфокоммуникационных технологий

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ИНФОКОММУНИКАЦИОННЫХ СИСТЕМ
Часть 1

Лабораторная работа 6
Наследование и механизм виртуальных функций. Клиент-серверное приложение Qt

Минск 2022

Макейчик Е.Г.

Содержание

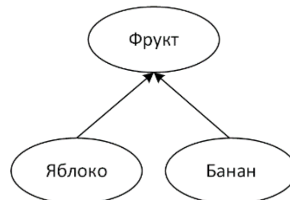
Лабораторная работа 6
Наследование и механизм виртуальных функций. Клиент-серверное приложение Qt 3
Базовое наследование 3
Порядок построения классов в цепочке наследования 7
Конструкторы и инициализация дочерних классов 7
Наследование и спецификатор доступа protected 10
Типы наследований. Доступ к членам 10
Переопределение методов родительского класса 13
Виртуальные функции и полиморфизм 13
Чистые виртуальные функции и абстрактные классы 14
Клиент-сервер в QT 15
Задание к лабораторной работе 6 23

Лабораторная работа 6
Наследование и механизм виртуальных функций.
Клиент-серверное приложение Qt

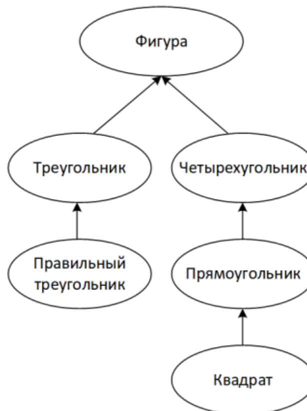
Цель работы: Изучить концепцию наследования и механизм виртуальных функций. Научиться создавать клиент-серверное приложение Qt.

Базовое наследование

Наследование в C++ происходит между классами. Класс, от которого наследуют, называется родительским (или "базовым", "суперклассом"), а класс, который наследует, называется дочерним (или "производным", "подклассом").



В диаграмме, представленной выше, Фрукт является родительским классом, а Яблоко и Банан – дочерними классами.



В этой диаграмме Треугольник является дочерним классом (родитель – Фигура) и родительским (для Правильный треугольник) одновременно.

Дочерний класс наследует как поведение (методы), так и свойства (поля) от родителя (с учетом некоторых ограничений доступа). Эти методы и поля становятся членами дочернего класса.

Поскольку дочерние классы являются полноценными классами, то они могут (конечно) иметь и свои собственные члены.

Класс Human

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

using namespace std;

class Human
{
public:
    string m_name;
    int m_age;
    Human(string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
    string getName() const { return m_name; }
    int getAge() const { return m_age; }
};
```

В этом классе определены только те члены, которые являются общими для всех объектов этого класса. Каждый Человек (независимо от пола, профессии и т.д.) имеет Имя и Возраст.

В примере, приведенном выше, все поля и методы класса открытые. Это сделано ради простоты примера. Обычно поля нужно делать private.

Класс BasketballPlayer

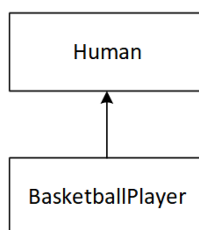
Предположим, что нужно написать программу, которая будет отслеживать информацию о баскетболистах, с возможностью сохранять средний уровень игры баскетболиста и количество очков.

```
class BasketballPlayer
{
public:
    double m_gameAverage;
    int m_points;
    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : m_gameAverage(gameAverage), m_points(points)
    {
    }
};
```

Также нужно знать Имя и Возраст баскетболиста, а эта информация уже есть: она хранится в классе Human.

Чтобы класс BasketballPlayer унаследовал информацию от класса Human, нужно после объявления BasketballPlayer (class BasketballPlayer) использовать двоеточие, ключевое слово public и имя класса, от которого хотим унаследовать. Это называется *открытым наследованием*.

```
// BasketballPlayer открыто наследует Human
class BasketballPlayer : public Human
{
public:
    double m_gameAverage;
    int m_points;
    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : m_gameAverage(gameAverage), m_points(points)
    {
    }
};
```



Когда BasketballPlayer наследует свойства класса Human, то BasketballPlayer приобретает методы и поля класса Human. Кроме того, BasketballPlayer имеет еще два своих собственных члена: m_gameAverage и m_points. Здесь есть смысл, так как эти свойства специфичны только для BasketballPlayer, а не для каждого Human.

Таким образом, объекты BasketballPlayer будут иметь 4 члена:

- m_gameAverage и m_points от BasketballPlayer;
- m_name и m_age от Human.

В коде ниже serge является объектом класса BasketballPlayer, а все объекты класса BasketballPlayer имеют поле m_name и метод getName, унаследованные от класса Human.

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

using namespace std;

class Human
{
public:
    string m_name;
    int m_age;
    Human(string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
    string getName() const { return m_name; }
    int getAge() const { return m_age; }
};

// BasketballPlayer открыто наследует Human
class BasketballPlayer : public Human
{
public:
    double m_gameAverage;
    int m_points;
    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : m_gameAverage(gameAverage), m_points(points)
    {
    }
};

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

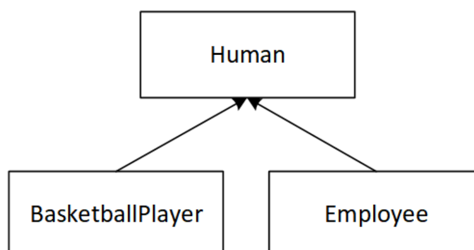
```
Serge
```

Дочерний класс Employee

Напишем еще один класс, который также будет наследовать свойства Human. Например, класс Employee (Работник). Работник "является" Человеком, поэтому использовать наследование здесь уместно.

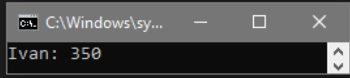
```cpp
// Employee открыто наследует Human
class Employee: public Human
{
public:
    int m_wage;
    long m_employeeID;
    Employee(int wage = 0, long employeeID = 0)
        : m_wage(wage), m_employeeID(employeeID)
    {
    }
    void printNameAndWage() const
    {
        cout << m_name << ": " << m_wage << '\n';
    }
};
```

Работник наследует m_name и m_age от Human (а также два метода) и имеет еще два собственных поля и один метод. Метод printNameAndWage использует переменные как из класса, к которому принадлежит (Employee::m_wage), так и с родительского класса (Human::m_name).



Классы Employee и BasketballPlayer не имеют прямых отношений, хотя оба наследуют свойства класса Human.

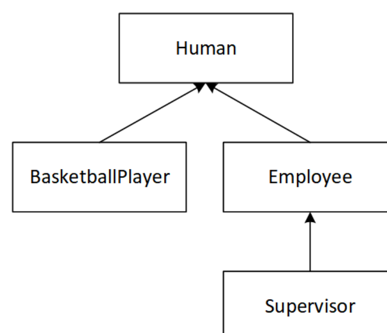
```cpp
#include <iostream>
#include <string>

using namespace std;

class Human
{
public:
    string m_name;
    int m_age;
    Human(string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
    string getName() const { return m_name; }
    int getAge() const { return m_age; }
};

// Employee открыто наследует Human
class Employee: public Human
{
public:
    int m_wage;
    long m_employeeID;
    Employee(int wage = 0, long employeeID = 0)
        : m_wage(wage), m_employeeID(employeeID)
    {
    }
    void printNameAndWage() const
    {
        cout << m_name << ": " << m_wage << '\n';
    }
};

int main()
{
    Employee ivan(350, 787);
    ivan.m_name = "Ivan"; // Мы можем это сделать, так как m_name является public
    ivan.printNameAndWage();
    return 0;
}
```



Цепочка наследований

Можно наследовать от класса, который сам наследует от другого класса. При этом ничего примечательного или чего-нибудь особенного не происходит – все аналогично. Например, напишем класс Supervisor (Супервайзер). Супервайзер – это Работник, который "является" Человеком. Класс Employee, будем использовать в качестве родительского класса.

```cpp
class Supervisor: public Employee
{
public:
    // Этот Супервайзер может наблюдать максимум за 5-тью Работниками
    long m_nOverseesIDs[5];
    Supervisor()
    {
    }
};
```



Все объекты Supervisor наследуют методы и поля от Employee и Human, а также имеют свою собственное поле m_nOverseesIDs.

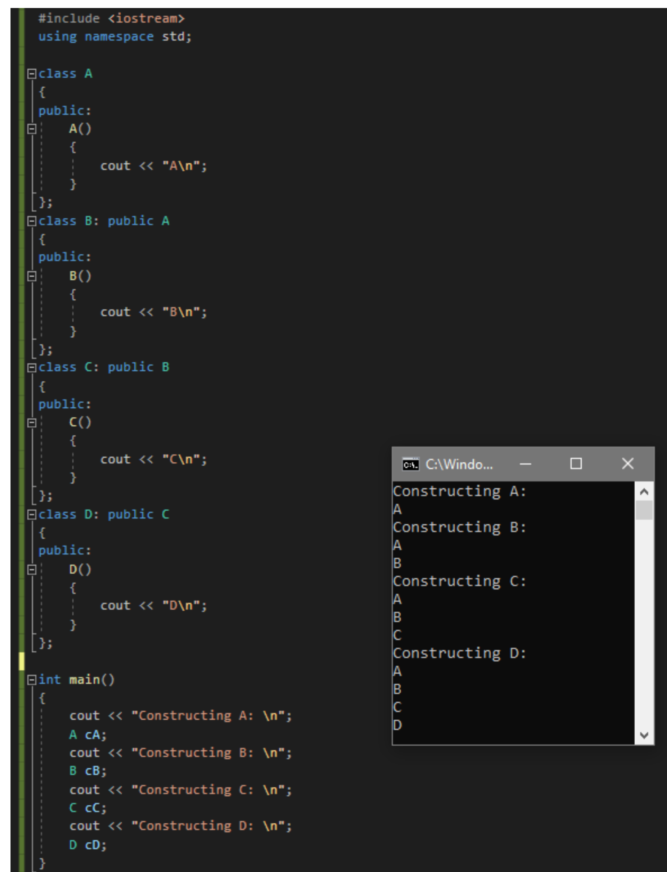
Построив такие цепочки наследований, можно создать набор повторно используемых классов, которые будут иметь общие свойства вверху и

становиться все более специфичными на каждом последующем уровне наследования.

Порядок построения классов в цепочке наследования

Часто случаются ситуации, когда одни классы наследуют свойства других классов, которые, в свою очередь, наследуют свойства своих предыдущих (родительских) классов.

В С++ всегда идет построение с "первого" или "топового" класса иерархии. Затем С++ переходит к следующему классу в иерархии и выполняет его построение. Этот процесс последовательный.



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

class A
{
public:
    A()
    {
        cout << "A\n";
    }
};
class B: public A
{
public:
    B()
    {
        cout << "B\n";
    }
};
class C: public B
{
public:
    C()
    {
        cout << "C\n";
    }
};
class D: public C
{
public:
    D()
    {
        cout << "D\n";
    }
};

int main()
{
    cout << "Constructing A: \n";
    A cA;
    cout << "Constructing B: \n";
    B cB;
    cout << "Constructing C: \n";
    C cC;
    cout << "Constructing D: \n";
    D cD;
}
```

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Конструкторы и инициализация дочерних классов

Конструкторы и инициализация

Детально рассмотрим роль конструкторов в инициализации дочерних классов.

```
class Parent
{
public:
    int m_id;
    Parent(int id = 0)
        : m_id(id)
    {
    }
    int getId() const { return m_id; }
};

class Child: public Parent
{
public:
    double m_value;
    Child(double value = 0.0)
        : m_value(value)
    {
    }
    double getValue() const { return m_value; }
};
```

С обычными классами (не дочерними) конструктору нужно заморачиваться только с членами своего класса. Например, объект класса Parent создается следующим образом.

```
int main()
{
    Parent parent(7); // Вызывается конструктор Parent(int)
    return 0;
}
```

Вот что на самом деле происходит при инициализации объекта parent:

- выделяется память для объекта parent;
- вызывается соответствующий конструктор класса Parent;
- список инициализации инициализирует переменные;
- выполняется тело конструктора;
- точка выполнения возвращается обратно в caller.

С дочерними классами дела обстоят несколько сложнее.

```
int main()
{
    Child child(1.5); // Вызывается конструктор Child(double)
    return 0;
}
```

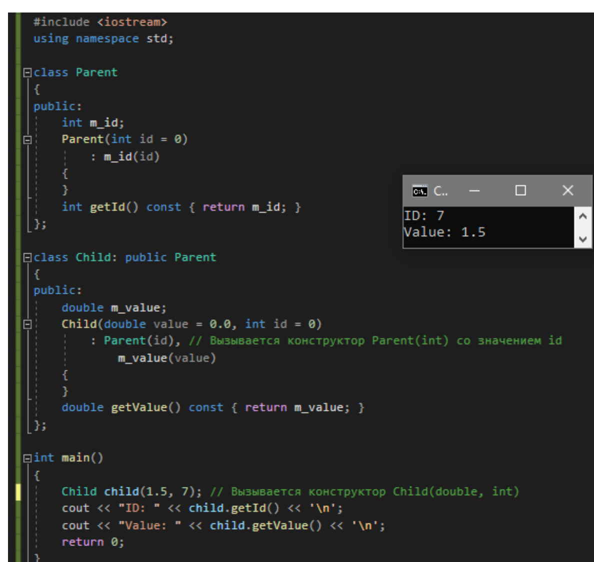
Вот что происходит при инициализации объекта child:

- выделяется память для объекта дочернего класса (достаточная порция памяти для части Parent и части Child объекта класса Child);
- вызывается соответствующий конструктор класса Child;
- создается объект класса Parent с использованием соответствующего конструктора класса Parent. Если такой конструктор программистом не предоставлен, то будет использоваться конструктор по умолчанию класса Parent;
- список инициализации инициализирует переменные;
- выполняется тело конструктора класса Child;
- точка выполнения возвращается обратно в caller.

Единственное различие между инициализацией объектов обычного и дочернего класса заключается в том, что при инициализации объекта дочернего класса, сначала выполняется конструктор родительского класса (для инициализации части родительского класса) и только потом уже выполняется конструктор дочернего класса.

Инициализация членов родительского класса

Язык C++ предоставляет возможность явно выбирать конструктор класса Parent для выполнения инициализации части Parent.



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

class Parent
{
public:
    int m_id;
    Parent(int id = 0)
        : m_id(id)
    {
    }
    int getId() const { return m_id; }
};

class Child: public Parent
{
public:
    double m_value;
    Child(double value = 0.0, int id = 0)
        : Parent(id), // Вызывается конструктор Parent(int) со значением id
          m_value(value)
    {
    }
    double getValue() const { return m_value; }
};

int main()
{
    Child child(1.5, 7); // Вызывается конструктор Child(double, int)
    cout << "ID: " << child.getId() << '\n';
    cout << "Value: " << child.getValue() << '\n';
    return 0;
}
```

Конструктор Parent(int) будет использоваться для инициализации m_id значением 7, а конструктор дочернего класса будет использоваться для инициализации m_value значением 1.5.

Рассмотрим детально, что происходит:

- Выделяется память для объекта child. Вызывается конструктор Child(double, int), где value = 1.5, а id = 7.
- Компилятор смотрит, запрашиваем ли мы какой-нибудь конкретный конструктор класса Parent. И видит, что запрашиваем. Поэтому вызывается Parent(int) с параметром id, которому мы до этого присвоили значение 7.
- Список инициализации конструктора класса Parent присваивает для m_id значение 7.
- Выполняется тело конструктора класса Parent, которое ничего не делает.
- Завершается выполнения конструктора класса Parent.
- Список инициализации конструктора класса Child присваивает для m_value значение 1.5.
- Выполняется тело конструктора класса Child, которое ничего не делает.
- Завершается выполнения конструктора класса Child.

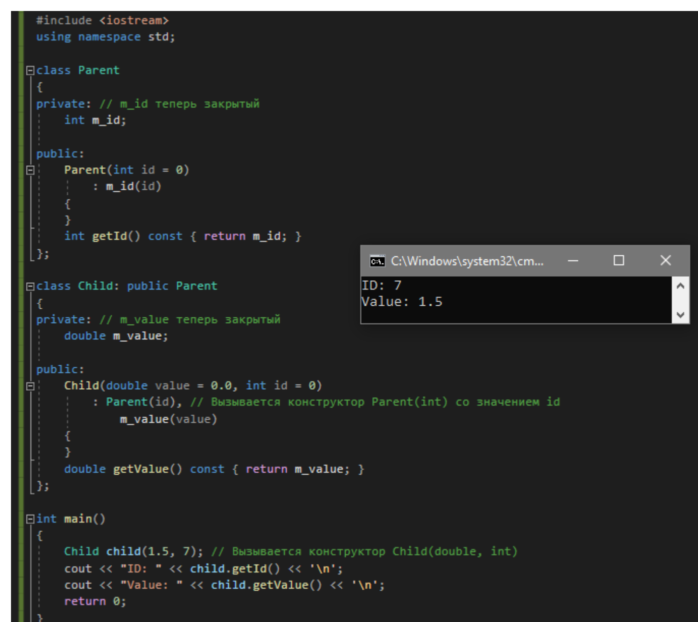
Все, что происходит – это вызов конструктором класса Child конкретного конструктора класса Parent для инициализации части Parent объекта класса Child. Поскольку m_id находится в части Parent, то только конструктор класса Parent может инициализировать это значение.

Не имеет значения, где в списке инициализации конструктора класса Child вызывается конструктор класса Parent – он всегда будет выполняться первым.

Члены private

Знаем о инициализации членов родительского класса, нет никакой необходимости сохранять поля открытыми. Сделаем их private, как и должно быть.

Доступ к public-членам открыт для всех. Доступ к private-членам открыт только для других членов этого же класса. Это означает, что дочерние классы не могут напрямую обращаться к закрытым членам родительского класса. Дочерним классам нужно использовать геттеры и сеттеры для доступа к этим членам.



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

class Parent
{
private: // m_id теперь закрытый
    int m_id;

public:
    Parent(int id = 0)
        : m_id(id)
    {
    }
    int getId() const { return m_id; }
};

class Child: public Parent
{
private: // m_value теперь закрытый
    double m_value;

public:
    Child(double value = 0.0, int id = 0)
        : Parent(id), // Вызывается конструктор Parent(int) со значением id
            m_value(value)
    {
    }
    double getValue() const { return m_value; }
};

int main()
{
    Child child(1.5, 7); // Вызывается конструктор Child(double, int)
    cout << "ID: " << child.getId() << '\n';
    cout << "Value: " << child.getValue() << '\n';
    return 0;
}
```

В коде, приведенном выше, m_id и m_value закрытые. Для их инициализации используются соответствующие конструкторы, а для доступа – открытые функции доступа (геттеры).

Наследование и спецификатор доступа protected

В языке C++ есть третий спецификатор доступа, он полезен только в контексте наследования. Спецификатор доступа protected открывает доступ к членам класса дружественным и дочерним классам. Доступ к protected-члену вне тела класса закрыт.

```
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()
{
    Parent parent;
    parent.m_public = 1; // Разрешено: доступ к открытым членам класса извне
    parent.m_private = 2; // Запрещено: доступ к закрытым членам класса извне
    parent.m_protected = 3; // Запрещено: доступ к защищенным членам класса извне
}
```

Член m_protected класса Parent напрямую доступен дочернему классу Child, но доступ к нему для членов извне – закрыт.

Когда следует использовать спецификатор доступа protected?

К protected-членам родительского класса доступ открыт для членов дочернего класса, а это означает, что если позже изменить что-либо в protected-члене (тип данных, значение и пр.), то придется внести изменения как в родительский, так и во все дочерние классы. Поэтому использование спецификатора доступа protected наиболее полезно, когда будут наследоваться только свои же классы и количество дочерних классов будет небольшое.

Типы наследований. Доступ к членам

Существует три типа наследований классов:

- public;
- private;
- protected.

Для определения типа наследования нужно просто указать нужное ключевое слово возле наследуемого класса.

```
// Открытое наследование
class Pub: public Parent
{
};

// Закрытое наследование
class Pri: private Parent
{
};

// Защищенное наследование
class Pro: protected Parent
{
};

class Def: Parent // По умолчанию язык C++ устанавливает закрытое наследование
{
};
```

В языке C++ по умолчанию будет выбран тип наследования private (аналогично и для членов класса, которые по умолчанию являются private, если не указано иначе).

Это дает 9 комбинаций: 3 спецификатора доступа (public, private и protected) и 3 типа наследования (public, private и protected).

Так в чем же разница между ними? Если вкратце, то при наследовании спецификатор доступа члена родительского класса может быть изменен в дочернем классе (в зависимости от типа наследования). Другими словами, члены, которые были public или protected в родительском классе, могут стать private в дочернем классе.

Правила:

- Класс всегда имеет доступ к своим (не наследуемым) членам.
- Доступ к члену класса основывается на его спецификаторе доступа.
- Дочерний класс имеет доступ к унаследованным членам родительского класса на основе спецификатора доступа этих членов в родительском классе.

Наследование типа public

Открытое наследование является одним из наиболее используемых типов наследования. Открытое наследование является самым легким и простым из всех типов. Когда открыто наследуется родительский класс, то унаследованные public-члены остаются public, унаследованные protected-члены остаются protected, а унаследованные private-члены остаются недоступными для дочернего класса.

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

```
class Parent
{
public:
    int m_public;

private:
    int m_private;

protected:
    int m_protected;
};

class Pub: public Parent // Открытое наследование
{
    // Открытое наследование означает, что:
    // - public-члены остаются public в дочернем классе;
    // - protected-члены остаются protected в дочернем классе;
    // - private-члены остаются недоступными в дочернем классе.

public:
    Pub()
    {
        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
    Pub pub;
    pub.m_public = 1; // Разрешено: m_public доступен извне через дочерний класс
    pub.m_private = 2; // Запрещено: m_private недоступен извне через дочерний класс
    pub.m_protected = 3; // Запрещено: m_protected недоступен извне через дочерний класс
}
```

Наследование типа private

При закрытом наследовании все члены родительского класса наследуются как закрытые. Это означает, что private-члены остаются недоступными, а protected-члены и public- члены становятся private в дочернем классе.

Это не влияет на то, как дочерний класс получает доступ к членам родительского класса. Это влияет только на то, как другими объектами осуществляется доступ к этим членам через дочерний класс.

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

```
class Parent
{
public:
    int m_public;

private:
    int m_private;

protected:
    int m_protected;
};

class Priv: private Parent // Закрытое наследование наследование
{
    // Закрытое наследование означает, что:
    // - public-члены становятся private (m_public теперь private) в дочернем классе;
    // - protected-члены становятся private (m_protected теперь private) в дочернем классе;
    // - private-члены остаются недоступными (m_private недоступен) в дочернем классе.

public:
    Priv()
    {
        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

    Priv priv;
    priv.m_public = 1; // Запрещено: m_public недоступен извне через дочерний класс
    priv.m_private = 2; // Запрещено: m_private недоступен извне через дочерний класс
    priv.m_protected = 3; // Запрещено: m_protected недоступен извне через дочерний класс
}
```

Закрытое наследование может быть полезно, когда дочерний класс не имеет очевидной связи с родительским классом, но использует его в своей реализации. На практике наследование типа private используется редко.

Наследование типа protected

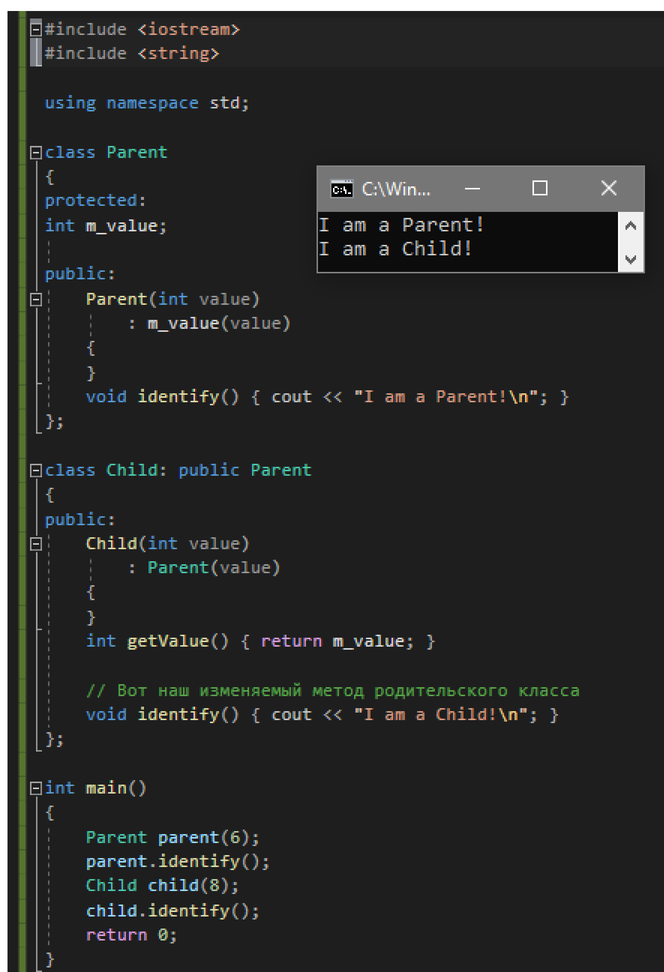
Этот тип наследования почти никогда не используется, за исключением особых случаев. С защищенным наследованием, public-члены и protected-члены становятся protected, а private-члены остаются недоступными.

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

Переопределение методов родительского класса

Дочерние классы по умолчанию наследуют все методы родительского класса.

Переопределение родительского метода в дочернем классе происходит, как обычное определение метода.



При переопределении родительского метода в дочернем классе, дочерний метод не наследует спецификатор доступа родительского метода с тем же именем. Используется тот спецификатор доступа, который указан в дочернем классе. Таким образом, метод, определенный как private в родительском классе, может быть переопределен как public в дочернем классе, или наоборот.

Виртуальные функции и полиморфизм

Виртуальная функция в языке C++ – это особый тип функции, которая, при ее вызове, выполняет "наиболее" дочерний метод, который существует между родительским и дочерними классами. Это свойство еще известно, как *полиморфизм*. Дочерний метод вызывается тогда, когда совпадает сигнатура (имя, типы параметров и является ли метод константным) и тип возврата дочернего метода с сигнатурой и типом возврата метода родительского класса. Такие методы называются *переопределениями* (или "*переопределенными методами*").

Чтобы сделать функцию виртуальной, нужно просто указать ключевое слово virtual перед объявлением функции.

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

using namespace std;

class Animal
{
protected:
    string m_name;
    // Мы делаем этот конструктор protected так как не хотим,
    // чтобы пользователи имели возможность создавать объекты класса Animal напрямую,
    // но хотим, чтобы в дочерних классах доступ был открыт
    Animal(string name)
        : m_name(name)
    {
    }
public:
    string getName() { return m_name; }
    virtual const char* speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name)
        : Animal(name)
    {
    }
    virtual const char* speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string name)
        : Animal(name)
    {
    }
    virtual const char* speak() { return "Woof"; }
};
void report(Animal &animal)
{
    cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat("Barsik");
    Dog dog("Matros");
    report(cat);
    report(dog);
}
```

```
C:\Windows\system32\cmd.exe
Barsik says Meow
Matros says Woof
```

При обработке animal.speak(), компилятор видит, что метод Animal::speak() является виртуальной функцией. Когда animal ссылается на часть Animal объекта cat, то компилятор просматривает все классы между Animal и Cat, чтобы найти наиболее дочерний метод speak(). И находит Cat::speak(). В случае, когда animal ссылается на часть Animal объекта dog, компилятор находит Dog::speak().

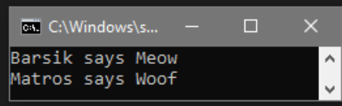
Чистые виртуальные функции и абстрактные классы

С++ позволяет создавать особый вид виртуальных функций, так называемых *чистых виртуальных функций* (или "*абстрактных функций*"), которые вообще не имеют определения. Переопределяют их дочерние классы.

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, нужно присвоить ей значение 0.

Использование чистой виртуальной функции имеет два основных последствия. Во-первых, любой класс с одной и более чистыми виртуальными функциями становится *абстрактным классом*, объекты которого создавать нельзя. Во-вторых, все дочерние классы абстрактного родительского класса должны переопределять все чистые виртуальные функции, в противном случае – они также будут считаться абстрактными классами

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

using namespace std;

class Animal // Теперь Animal является абстрактным родительским классом
{
public:
    string m_name;
    // Мы делаем этот конструктор protected так как не хотим,
    // чтобы пользователи имели возможность создавать объекты класса Animal напрямую,
    // но хотим, чтобы в дочерних классах доступ был открыт
    Animal(string name)
        : m_name(name)
    {
    }
    string getName() { return m_name; }
    virtual const char* speak() = 0; // speak() является чистой виртуальной функцией
};

class Cat: public Animal
{
public:
    Cat(std::string name)
        : Animal(name)
    {
    }
    virtual const char* speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string name)
        : Animal(name)
    {
    }
    virtual const char* speak() { return "Woof"; }
};
void report(Animal &animal)
{
    cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat("Barsik");
    Dog dog("Matros");
    report(cat);
    report(dog);
}
```



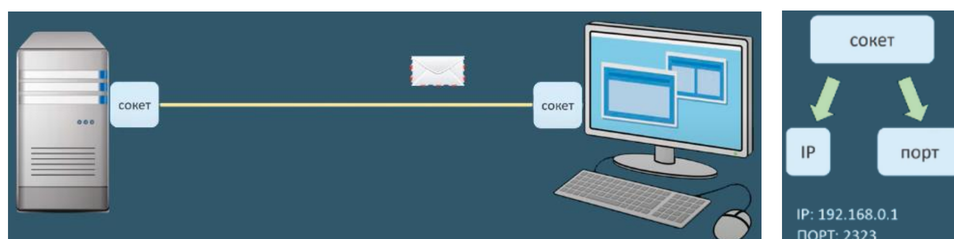
Чистая виртуальная функция полезна, когда есть функция, которую нужно поместить в родительский класс, но реализацию оставить дочерним классам. Чистая виртуальная функция абстрактного родительского класса вынуждает дочерние классы переопределить эту функцию, иначе объекты этих классов создавать будет невозможно.

Клиент-сервер в QT

Создадим клиент-серверный чат, в котором сообщения от каждого клиента будут приходить на сервер. А сервер будет рассылать эти сообщения всем подключенным клиентам.

Связь между сервером и клиентом обеспечивается с помощью сокетов.

Сокет – внутренняя конечная точка для отправки и получения данных в узле сети. Другими словами, это программный интерфейс, представляющий собой совокупность ip и порта.



IP-адрес – это уникальный адрес, идентифицирующий устройство в интернете или локальной сети. *Порт* – это то, что позволяет операционной системе понять для какой программы, запущенной на ПК, предназначаются полученные по сети данные.

Определить порт можно по его номеру. В протоколе TCP под номер порта отведено 16 бит, поэтому номер порта представляет собой число в диапазоне от 1 до 65535.

Реализация сервера

Для реализации сервера нет необходимости в графическом интерфейсе, поэтому создаем консольное приложение.

В файле .pro необходимо подключить network (Qt += core network).

Включение сервера осуществляется методом listen (файл server.cpp), в нем устанавливается с какого адреса и в каком порту сервер будет принимать сигналы. Клиент запускается и создает свой сокет, со своим адресом и портом. Теперь клиент готов подключиться к серверу.

Он отправляет запрос с помощью метода connectToHost (файл mainwindows.cpp), в котором указан адрес и порт сервера. Как только на сервер приходит сигнал о новом подключении, вызывается функция incommingConnection (файл server.cpp), где это подключение обрабатывается. Сервер выделяет новый сокет для этого подключения.

Сервер также может отправлять клиенту сообщение об успешном подключении. Для этого нужно написать функцию. Сокеты передают и принимают массив байтов QByteArray, поэтому все сообщения необходимо приводить к этому типу данных. При получении данных от сервера клиент автоматически вызывает сигнал ReadyRead. Необходимо написать в нем слот и сделать обработку этих данных, т.е. перевести их из QByteArray.

Клиент отправляет на сервер сообщения, для этого их тоже нужно привести к QByteArray. И по аналогии с клиентом, сервер при получении запросов вызовет сигнал ReadyRead, для которого нужно написать слот. Полученное сообщение сервер разошлет всем полученным клиентам. При отключении от сервера и клиент и сервер должны удалить свои сокеты.

server.pro

```
QT -= gui
QT += core network
# core network для работы с сетевыми классами

CONFIG += c++17 console
CONFIG -= app_bundle

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
        main.cpp \
        server.cpp

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

HEADERS += \
    server.h
```

server.h

```
#ifndef SERVER_H
#define SERVER_H
#include <QTcpServer> // Подключение класса QTcpServer
#include <QTcpSocket> // Подключение класса QTcpSocket
#include <QVector> // Подключение класса QVector

class Server : public QTcpServer // Наследование класса QTcpServer
{
    Q_OBJECT

public:
    Server(); // Создание конструктора класса Server
    QTcpSocket *socket; // Создание указателя на объект класса QTcpSocket

private:

    /* Сервер будет создавать новый сокет для каждого нового подключения.
     * Все сокеты будут записываться в вектор */

    QVector <QTcpSocket*> Sockets; // Создание вектора для сокетов

    /* Сокеты передают и принимают массив байтов QBateArray,
     * поэтому все сообщения необходимо будет приводить к этому типу данных.
     * Для этого создаем объект Data класса QByteArray */

    QByteArray Data;

    void SendToClient(QString str); // Функция SendToClient, для отправки сообщений об подключении

 public slots:
    void incomingConnection(qintptr socketDescriptor); // Слот для обработки новых подключений
    void slotReadyRead(); // Слот для обработки полученных от клиента сообщений
};

#endif // SERVER_H
```

server.cpp

```
#include "server.h"
Server::Server() // Описание конструктора
{
    // Включение сервера осуществляется методом listen, в нем устанавливается с какого адреса и в каком порту сервер будет принимать сигналы.
    // Клиент должен запуститься и создать свой сокет, со своим адресом и портом. После этого клиент готов подключиться к серверу
    if(this->listen(QHostAddress::Any,2323))
    // Первый аргумент метода QHostAddress::Any означает, что сервер будет принимать сигналы пришедшие с любого адреса, второй аргумент метода 2323, что сервер прослушивает порт номер 2323
    {
        qDebug() << "start"; // Если сервер запустится, то выведем сообщение start
    }
    else
    {
        qDebug() << "error"; // Если сервер не запустится, то выведем сообщение error
    }
}
// Как только на сервер приходит сигнал о новом подключении, вызывается функция incommingConnection, где это подключение обрабатывается. Сервер выделяет новый сокет для этого подключения
void Server::incomingConnection(qintptr socketDescriptor)
{
    socket = new QTcpSocket; // Создание нового сокета
    socket->setSocketDescriptor(socketDescriptor); // Устанавливаем в сокет дескриптор
    connect(socket, &QTcpSocket::readyRead, this, &Server::slotReadyRead); // Объединяем сигнал readyRead с соответствующим слотом
    connect(socket, &QTcpSocket::disconnected, socket, &QTcpSocket::deleteLater); // Соединяем сигнал disconnected с функцией deleteLater. Теперь при отключении клиента приложение сразу удалит сокет
    Sockets.push_back(socket); // Помещаем сокет в push_back. Функция push_back будет вызываться при каждом новом подключении клиента
    qDebug() << "client connected" << socketDescriptor; // Выводим сообщение о подключении
}
void Server::slotReadyRead() // Описание слота для чтения сообщений
{
    socket = (QTcpSocket*)sender(); // Запись сокета с которого пришел запрос
    // QDataStream класс для работы с потоковым вводом/выводом данных
    QDataStream in(socket); // Создание объекта in (ввод). С помощью объекта in будем работать с данными находящимися в socket
    in.setVersion(QDataStream::Qt_6_3); // Указываем Qt_6_3, для исключения ошибок версий
    if(in.status() == QDataStream::Ok) // Проверка состояния объекта in
    {
        qDebug() <<"read..."; // Если ошибок нет, продолжаем обработку
        QString str; // Клиент и сервер будут принимать только строки
        in >> str;
        qDebug() << str;
        SendToClient(str); // Вызов SendToClient
    }
    else
    {
        qDebug() << "DataStream error"; // Если ошибки есть, выводим сообщение об ошибке в консоль
    }
}
void Server::SendToClient(QString str) // Описание функции отправки сообщений клиенту
{
    Data.clear();
    QDataStream out(&Data, QIODevice::WriteOnly); // Создание объекта out (вывод). В конструкторе указываем массив байтов и режим работы WriteOnly (только для записи)
    out.setVersion(QDataStream::Qt_6_3); // Указываем Qt_6_3, для исключения ошибок версий
    out << str;
    // Сообщение необходимо отправить всем клиентам
    for(int i = 0; i < Sockets.size(); i++) // Пишем цикл от 0 до размера вектора, и в каждый сокет вектора записываем сообщение
    {
        Sockets[i]->write(Data);
    }
}
```

main.cpp

```
#include <QCoreApplication>
#include "server.h" // Подключение заголовочного файла

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Server s; // Создание объекта s класса Server
    return a.exec();
}
```

Реализация клиента

Для реализации клиента создадим приложение с графическим интерфейсом.

В файле client.pro необходимо подключить network (Qt += core gui network).

client.pro

```
QT += core gui network
# network для работы с сетевыми классами

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++17

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h

FORMS += \
    mainwindow.ui

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

В файле mainwindows.h необходимо подключить класс QTcpSocket. Создать указатель на объект этого класса и создать объект класса QByteArray.

Подготовим форму, смотреть.

С помощью команды Перейти к слоту… необходимо выбрать сигнал clicked() для кнопок и сигнал returnPressed() для Line Edit.

mainwindows.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpSocket> // Подключение класса QTcpSocket

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

    void on_pushButton_2_clicked();

    void on_lineEdit_returnPressed();

private:
    Ui::MainWindow *ui;
    QTcpSocket *socket; // Создание указателя на объект класса QTcpSocket
    QByteArray Data; // Создание объекта класса QByteArray
    void SendToServer(QString str);

public slots:
    void slotReadyRead();
};
#endif // MAINWINDOW_H
```

mainwindows.cpp

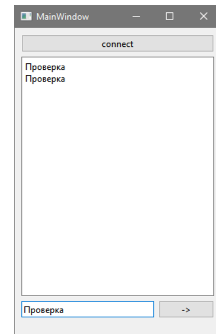
```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    socket = new QTcpSocket(this); // Создание сокета в конструкторе
    connect(socket, &QTcpSocket::readyRead, this, &MainWindow::slotReadyRead); // Объединяем сигнал readyRead с соответствующим слотом
    connect(socket, &QTcpSocket::disconnected, socket, &QTcpSocket::disconnected); // Соединяем сигнал disconnected с функцией deleteLater
}
MainWindow::~MainWindow()
{
    delete ui;
}
void MainWindow::on_pushButton_clicked()
{
    // Клиент отправляет запрос с помощью метода connectToHost,  в котором указан адрес и порт сервера
    socket->connectToHost("127.0.0.1", 2323); // Подключаем сокет к серверу
}
void MainWindow::SendToServer(QString str) // Описание функции отправки сообщений
{
    Data.clear();
    QDataStream out(&Data, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_6_3);
    out << str;
    socket->write(Data);
    ui->lineEdit->clear();
}
void MainWindow::slotReadyRead() // Описываем слот ReadyRead так же как и на сервере
{
    QDataStream in(socket);
    in.setVersion(QDataStream::Qt_6_3);
    if(in.status()==QDataStream::Ok)
    {
        QString str;
        in >> str;
        ui->textBrowser->append(str); // Добавляем строку в textBrowser
    }
    else
    {
       ui->textBrowser->append("read error");
    }
}
void MainWindow::on_pushButton_2_clicked() // SendToServer будет вызываться при нажатии кнопки ->
{
    SendToServer(ui->lineEdit->text());
}
void MainWindow::on_lineEdit_returnPressed() // SendToServer еще будет вызываться при нажатии клавиши Enter
{
    SendToServer(ui->lineEdit->text());
}
```

Собираем проекты, осуществляем отладку и запуск.

Сервер

```
18:14:41: Starting C:/Users/Piglet/Documents/build-untitled23-Desktop_Qt_6_3_1_MSVC2019_64bit-Debug/debug/untitled23.exe...
start
client connected 936
read...
"Проверка"
read...
"Проверка"
```

Клиент



Но есть нюанс, в сети данные передаются по частям. Получается, что данные можно обрабатывать до того, как они пришли. Из этого следует, что отправитель должен в начале сообщения указать размер передаваемого блока, а получатель должен сравнить размер имеющихся данных с высланным размером, смотреть принцип работы алгоритма.

Внесем изменения в файлы клиента.

mainwindows.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpSocket> // Подключение класса QTcpSocket

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

    void on_pushButton_2_clicked();

    void on_lineEdit_returnPressed();

private:
    Ui::MainWindow *ui;
    QTcpSocket *socket; // Создание указателя на объект класса QTcpSocket
    QByteArray Data; // Создание объекта класса QByteArray
    void SendToServer(QString str);

    quint16 nextBlockSize; // Объявляем переменную nextBlockSize, quint16 это целое положительное число размером 16 бит или 2 байта

public slots:
    void slotReadyRead();
};
#endif // MAINWINDOW_H
```

mainwindows.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    socket = new QTcpSocket(this); // Создание сокета в конструкторе
    connect(socket, &QTcpSocket::readyRead, this, &MainWindow::slotReadyRead); // Объединяем сигнал readyRead с соответствующим слотом
    connect(socket, &QTcpSocket::disconnected, socket, &QTcpSocket::disconnected); // Соединяем сигнал disconnected с функцией deleteLater

    nextBlockSize = 0; // Обнуляем переменную
}
MainWindow::~MainWindow()
{
    delete ui;
}
void MainWindow::on_pushButton_clicked()
{
    // Клиент отправляет запрос с помощью метода connectToHost,  в котором указан адрес и порт сервера
    socket->connectToHost("127.0.0.1", 2323); // Подключаем сокет к серверу
}
void MainWindow::SendToServer(QString str) // Описание функции отправки сообщений
{
    Data.clear();
    QDataStream out(&Data, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_6_3);

    out << quint16(0) << str; /* Пока сообщение не записано полностью, мы не можем определить размер блока, поэтому записываем 0.
                               * При этом в начале сообщения выделяется 2 байта, и следующая часть будет записываться с 17 бита, а первые 16 остаются в резерве */
    out.device()->seek(0); // Переходим в начало блока
    out << quint16(Data.size() - sizeof(quint16)); // Записываем в начало блока размер сообщения, как разность между длиной всего блока и выделенными байтами для размера

    socket->write(Data);
    ui->lineEdit->clear();
}
void MainWindow::slotReadyRead() // Описываем слот ReadyRead так же как и на сервере
{
    QDataStream in(socket);
    in.setVersion(QDataStream::Qt_6_3);
    if(in.status()==QDataStream::Ok)
    {
        for (;;) // Бесконечный цикл. Необходимо считать размер блока при двух условиях
        {
            if(nextBlockSize == 0) // Размер блока неизвестен, т.е. равен 0
            {
                if(socket->bytesAvailable() < 2) // Для чтения доступно не менее 2 байт
                {
                    break; // Если условие не выполняется - выходим из цикла
                }
                in >> nextBlockSize; // Если условие выполняется - считываем размер блока
            }
            if(socket->bytesAvailable() < nextBlockSize) // Когда известен размер блока, сравниваем его с количеством байт, которое пришло от сервера
            {
                break; // Если размер блока больше, то данные пришли не полностью. Выходим из цикла
            }
```

```
            QString str;
            in >> str;

            nextBlockSize = 0; // Обнуляем размер блока, для приема новых сообщений

            ui->textBrowser->append(str); // Добавляем строку в textBrowser
        }
    }
    else
    {
        ui->textBrowser->append("read error");
    }
}
void MainWindow::on_pushButton_2_clicked() // SendToServer будет вызываться при нажатии кнопки ->
{
    SendToServer(ui->lineEdit->text());
}
void MainWindow::on_lineEdit_returnPressed() // SendToServer еще будет вызываться при нажатии клавиши Enter
{
    SendToServer(ui->lineEdit->text());
}
```

В файлы сервера необходимо внести такие же изменения.

server.h

```
#ifndef SERVER_H
#define SERVER_H
#include <QTcpServer> // Подключение класса QTcpServer
#include <QTcpSocket> // Подключение класса QTcpSocket
#include <QVector> // Подключение класса QVector

class Server : public QTcpServer // Наследование класса QTcpServer
{
    Q_OBJECT

public:
    Server(); // Создание конструктора класса Server
    QTcpSocket *socket; // Создание указателя на объект класса QTcpSocket

private:

    /* Сервер будет создавать новый сокет для каждого нового подключения.
     * Все сокеты будут записываться в вектор */

    QVector <QTcpSocket*> Sockets; // Создание вектора для сокетов

    /* Сокеты передают и принимают массив байтов QBateArray,
     * поэтому все сообщения необходимо будет приводить к этому типу данных.
     * Для этого создаем объект Data класса QByteArray */

    QByteArray Data;

    void SendToClient(QString str); // Функция SendToClient, для отправки сообщений об подключении

    quint16 nextBlockSize; // Объявляем переменную nextBlockSize, quint16 это целое положительное число размером 16 бит или 2 байта

 public slots:
    void incomingConnection(qintptr socketDescriptor); // Слот для обработки новых подключений
    void slotReadyRead(); // Слот для обработки полученных от клиента сообщений
};

#endif // SERVER_H
```

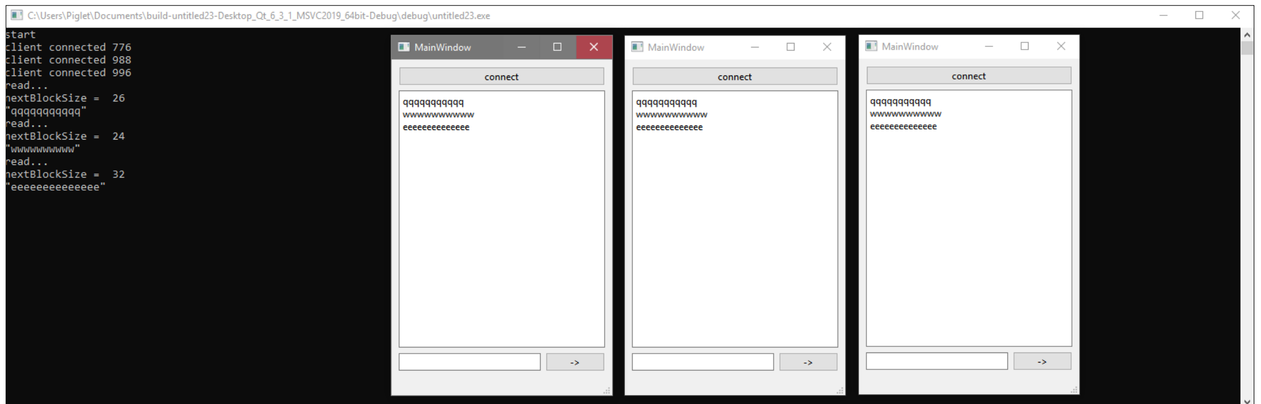
server.cpp

```
#include "server.h"
Server::Server() // Описание конструктора
{
    // Включение сервера осуществляется методом listen, в нем устанавливается с какого адреса и в каком порту сервер будет принимать сигналы.
    // Клиент должен запуститься и создать свой сокет, со своим адресом и портом. После этого клиент готов подключиться к серверу
    if(this->listen(QHostAddress::Any,2323))
    // Первый аргумент метода QHostAddress::Any означает, что сервер будет принимать сигналы пришедшие с любого адреса, второй аргумент метода 2323, что сервер прослушивает порт номер 2323
    {
        qDebug() << "start"; // Если сервер запустится, то выведем сообщение start
    }
    else
    {
        qDebug() << "error"; // Если сервер не запустится, то выведем сообщение error
    }
    nextBlockSize = 0; // Обнуляем переменную
}
// Как только на сервер приходит сигнал о новом подключении, вызывается функция incommingConnection, где это подключение обрабатывается. Сервер выделяет новый сокет для этого подключения
void Server::incomingConnection(qintptr socketDescriptor)
{
    socket = new QTcpSocket; // Создание нового сокета
    socket->setSocketDescriptor(socketDescriptor); // Устанавливаем в сокет дескриптор
    connect(socket, &QTcpSocket::readyRead, this, &Server::slotReadyRead); // Объединяем сигнал readyRead с соответствующим слотом
    connect(socket, &QTcpSocket::disconnected, socket, &QTcpSocket::deleteLater); // Соединяем сигнал disconnected с функцией deleteLater. Теперь при отключении клиента приложение сразу удалит сокет
    Sockets.push_back(socket); // Помещаем сокет в push_back. Функция push_back будет вызываться при каждом новом подключении клиента
    qDebug() << "client connected" << socketDescriptor; // Выводим сообщение о подключении
}
void Server::slotReadyRead() // Описание слота для чтения сообщений
{
    socket = (QTcpSocket*)sender(); // Запись сокета с которого пришел запрос
    // QDataStream класс для работы с потоковым вводом/выводом данных
    QDataStream in(socket); // Создание объекта in (ввод). С помощью объекта in будем работать с данными находящимися в socket
    in.setVersion(QDataStream::Qt_6_3); // Указываем Qt_6_3, для исключения ошибок версий
    if(in.status() == QDataStream::Ok) // Проверка состояния объекта in
    {
        qDebug() <<"read..."; // Если ошибок нет, продолжаем обработку

        for (;;) // Бесконечный цикл. Необходимо считать размер блока при двух условиях
        {
            if(nextBlockSize == 0) // Размер блока неизвестен, т.е. равен 0
            {
                if(socket->bytesAvailable() < 2) // Для чтения доступно не менее 2 байт
                {
                    qDebug() << "Data < 2, break"; // Сообщение о состоянии чтения данных
                    break; // Если условие не выполняется - выходим из цикла
                }
                in >> nextBlockSize; // Если условие выполняется - считываем размер блока
                qDebug() << "nextBlockSize = " << nextBlockSize; // Сообщение о состоянии чтения данных
            }
            if(socket->bytesAvailable() < nextBlockSize) // Когда известен размер блока, сравниваем его с количеством байт, которое пришло от сервера
            {
                qDebug() << "Data not full, break "; // Сообщение о состоянии чтения данных
                break; // Если размер блока больше, то данные пришли не полностью. Выходим из цикла
            }
            QString str;
            in >> str;
            nextBlockSize = 0; // Обнуляем размер блока, для приема новых сообщений
            qDebug() << str;
            SendToClient(str);
            break; // Принудительный выход из цикла
        }
    }
    else
    {
        qDebug() << "DataStream error"; // Если ошибки есть, выводим сообщение об ошибке в консоль
    }
}
```

```
void Server::SendToClient(QString str) // Описание функции отправки сообщений клиенту
{
    Data.clear();
    QDataStream out(&Data, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_6_3);

    out << quint16(0) << str; /* Пока сообщение не записано полностью, мы не можем определить размер блока, поэтому записываем 0.
                               * При этом в начале сообщения выделяется 2 байта, и следующая часть будет записываться с 17 бита, а первые 16 остаются в резерве */
    out.device()->seek(0); // Переходим в начало блока
    out << quint16(Data.size()-sizeof(quint16)); // Записываем в начало блока размер сообщения, как разность между длиной всего блока и выделенными байтами для размера
    // Сообщение необходимо отправить всем клиентам
    for(int i = 0; i < Sockets.size(); i++) // Пишем цикл от 0 до размера вектора, и в каждый сокет вектора записываем сообщение
    {
        Sockets[i]->write(Data);
    }
}
```

Класс QDataStream позволяет работать с разными типами данных. С его помощью можно передавать изображения, дату, время и пр.

Собираем проекты, осуществляем отладку. Запускаем сервер и три клиента, смотреть.



Задание к лабораторной работе 6

Для выполнения лабораторной работы необходимо установить и настроить Visual Studio и инструменты Qt. Задание к лабораторной работе 6 состоит из нескольких задач. Задачи выполняются по вариантам (например, Вариант 1 – номера по списку в группе: 1, 9, 17, 25; Вариант 2 – номера по списку в группе: 2, 10, 18, 26 и т.д.).

Вариант 1

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

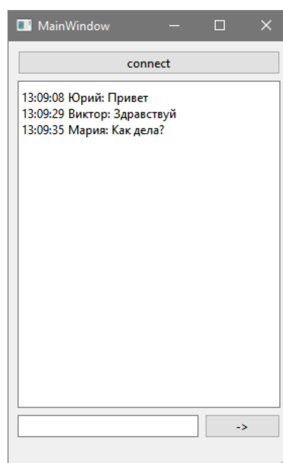
После того, как вы успешно разобрались с читерами в игре "Sword, Magic, Shield, Bow, Arrow and something else" вас попросили помочь с разработкой нового патча обновления для этой игры. Это глобальный патч в который планируют добавить очень многое: новых монстров, героев, NPC, локации, оружие, заклинания, механику и даже новое название самой игры. Конкретно вы занимаетесь разработкой новых монстров. Так как монстров необходимо добавить очень много, то вы решили описать базовый класс монстра и создать три производных класса монстров на основе базового.

Вам необходимо создать базовый класс Monsters с полями: HP, damage, armour и двумя виртуальными функциями print_info() и scream(). Так же создать три дочерних класса от класса Monsters (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и scream(). Переопределенная функция print_info() должна вывести поля: HP, damage, armour умноженные на константное значение (на какое значение умножать решайте сами). А переопределенная функция scream() должна вывести крик этого монстра (как будет кричать ваш монстр придумайте сами) к примеру "Life for Nerzula".

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Какая же игра может обойтись без средств общения между игроками. Заказчик попросил вас написать клиент-серверное приложение для обмена сообщениями. Каждое сообщение должно содержать время отправки (класс QTime) и имя игрока.



Вариант 2

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

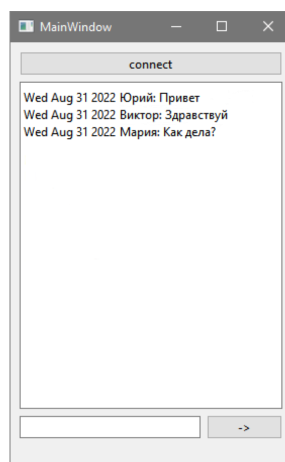
Вас попросили подготовить реферат про число Эйлера и число Пи для лекции студентам. Во время подготовки реферата вам пришла интересная мысль округлить эти числа (число Пи (3,14…), число Эйлера (2,718…). Проделав это, вы пришли к выводу что число Пи == числу Эйлера. После того как вы поделились этим открытием с коллегами, вас попросили немедленно покинуть лабораторию и никогда на нее не возвращаться. Не зная куда податься, вы решили пойти в администрацию центра чтобы они помогли вам найти новое место, где вы могли бы работать. Вам предложили поработать на планово-финансовый отдел. Вашей задачей будет подсчет суммы расходов на каждую кафедру. Подумав немного, вы решили написать программу, которая сама бы делала все необходимые расчеты.

Вам необходимо создать базовый класс Faculty с полями: number_of_employees, employee_salary, surcharge и двумя виртуальными функциями print_info() и costs_with_surcharge(). Так же создать три производных класса от класса Faculty (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и costs_with_surcharge(). Переопределенная функция print_info() должна вывести поля number_of_employees, employee_salary и сумму расходов на кафедру, которая равна $\text{number_of_emloyees} \times \text{employee_salary}$. А переопределенная функция costs_with_surcharge() должна вывести поля number_of_employees, employee_salary, surcharge и сумму расходов на кафедру, которая равна $\text{number_of_emloyees} \times ((\text{employee_salary} + \text{surcharge}) \times 0.13)$.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Программа получила одобрение у начальника планово-финансового отдела. Ее установили всем сотрудникам. У сотрудников возникают вопросы по работе с ней, поэтому нужно осуществлять техническую поддержку. Вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать дату отправки (класс QDate) и имя сотрудника.



Вариант 3

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

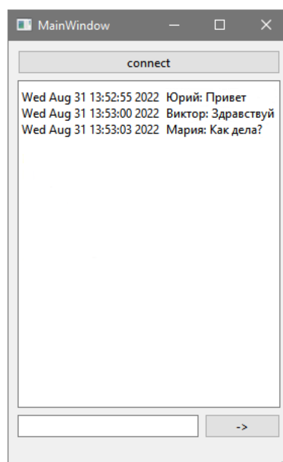
Вы уже целый месяц работаете в больнице главврачом и уже многое успели сделать. Единственное что вы не успели – это выдать сотрудникам зарплаты. Пока они этого не заметили вам нужно срочно посчитать сколько и кому необходимо платить. Так как необходимо все это сделать быстро, а времени считать столбиком у вас нет, то вы решили написать программу, которая все посчитает за вас.

Вам необходимо создать базовый класс Employees с полями: hourly_payment, worked_hours, surcharge и двумя виртуальными функциями print_info() и costs_with_surcharge(). Так же создать три производных класса от класса Employees (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и costs_with_surcharge(). Переопределенная функция print_info() должна вывести поля hourly_payment, worked_hours и зарплату сотрудника, которая равна $\text{hourly_payment} \times \text{worked_hours}$. А переопределенная функция costs_with_surcharge() должна вывести поля worked_hours, hourly_payment, surcharge и зарплату сотрудника которая равна $\text{worked_hours} \times ((\text{hourly_payment} + \text{surcharge}) \times 0.13)$.

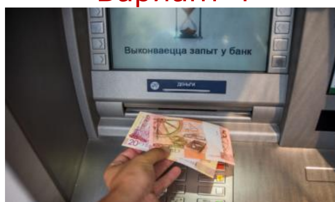
Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Вы решили внедрить свое ПО в бухгалтерию. У сотрудников бухгалтерии возникают вопросы по работе с ПО и дополнительные предложения о добавлении функционала. Поэтому вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать дату и время отправки (класс QDateTime) и имя сотрудника.



Вариант 4



Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

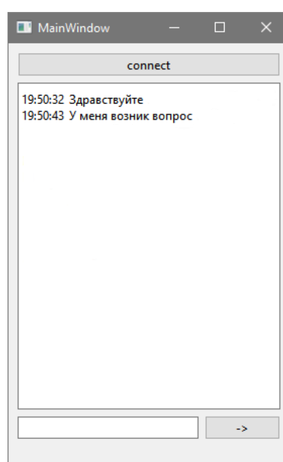
Ваш начальник обожает инновационные технологии и на этот раз он попросил вас написать программу, которая бы показывала сколько денег в месяц нужно будет платить человеку, который берет ипотеку в банке с учетом процента этого банка. Ваш начальник владеет несколькими банками и у каждого из них свой процент по ипотеке и это нужно учитывать.

Вам необходимо создать базовый класс Bank с полями: amount_of_debt, number_of_months и двумя виртуальными функциями print_info() и debt(). Так же создать три производных класса от класса Bank (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и debt(). Переопределенная функция print_info() должна вывести поля amount_of_debt, number_of_months. А переопределенная функция debt() должна вывести сколько в месяц будет платить человек именно этому банку. Сумма оплаты в месяц считается по формуле $\text{amount_of_debt} + \left(\text{amount_of_debt} \times \left[(\text{процент банка любой})/100\right]\right) / \text{number_of_months}$.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Программа понравилась и вашему начальнику, и клиентам банка. Но клиентам еще нужно дать возможность задать интересующие их вопросы. Вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать время отправки (класс QTime).



Вариант 5

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

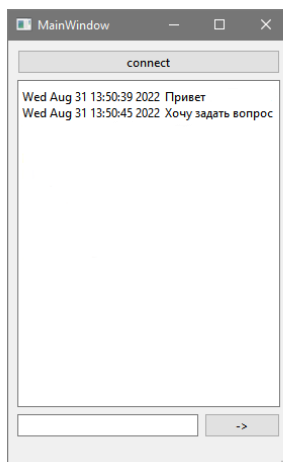
Скоро в ваш полицейский участок придут школьники на экскурсию. Вы будете рассказывать про разные виды преступлений и что за эти преступления грозит. Так как экскурсий будет много, а вам еще и преступления раскрывать то вы решили поставить в холле интерактивное табло, на котором бы дети просто выбирали вид преступления, и в ответ им выводилась информация о том, сколько лет заключения грозит за это деяние. Осталось только запрограммировать это табло.

Вам необходимо создать базовый класс Criminals с полями: years_imprisonment, severity_of_crime, monetary_penalty_for_crime и двумя виртуальными функциями print_info() и how_many_days(). Так же создать три производных класса от класса Criminals (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и how_many_days(). Переопределенная функция print_info() должна вывести поля years_imprisonment, severity_of_crime, monetary_penalty_for_crime. А переопределенная функция how_many_days() должна вывести количество дней, которое необходимо отсидеть преступнику. Если $severity_of_crime < 4$ то количество дней вычисляется по формуле $(years_imprisonment \times 365)/2$. Если $4 <= severity_of_crime <= 10$ то количество дней вычисляется по формуле $years_imprisonment \times 365$. Для всех остальных случаев формула для расчета $years_imprisonment \times 365 \times 2$.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Интерактивное табло пользуется вниманием школьников, но они все ровно подходят и задают дополнительные вопросы. Чтобы избежать этого, вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать дату и время отправки (класс QDateTime).



Вариант 6

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

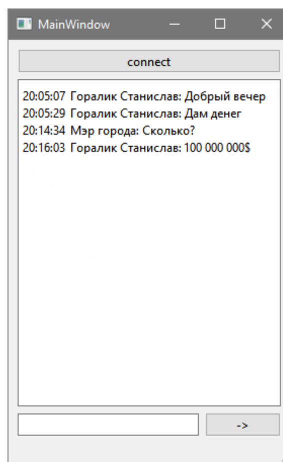
Вы хотите начать грандиозную стройку в городе, но, чтобы что-то построить нужны деньги. Сегодня вы пригласили самых богатых людей своего города, для того, чтобы попросить их инвестировать в развитие города. Чтобы не спугнуть инвесторов одними лишь большими числами вы решили поставить интерактивное табло, на котором бы выводилась красивое изображение будущего строения и расход на постройку одного такого строения. Осталось только запрограммировать это табло.

Вам необходимо создать базовый класс Building с полями: cost, monthly_profit и двумя виртуальными функциями print_info() и when_will_pay_off(). Так же создать три производных класса от класса Building (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и when_will_pay_off(). Переопределенная функция print_info() должна вывести поля cost, monthly_profit. А переопределенная функция when_will_pay_off() должна вывести информацию о том через сколько месяцев новое здание окупит свою стоимость: cost/monthly_profit.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

К вам обратились люди, которые родились в этом городе, но давно проживают за пределами страны и тоже хотят инвестировать в развитие. Поэтому вы решили написать мобильное приложение с теми же возможностями, что и у интерактивного табло и добавлением нового функционала "Обратная связь". Вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать время отправки (класс QTime) и имя отправителя.



Вариант 7

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

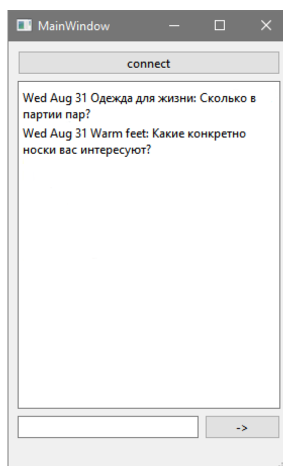
Ваша фабрика по производству всякой всячины каждый день производит все больше и больше товара, что не может вас не радовать. Но вот только теперь вам нужно найти новые рынки сбыта для этого товара. Для привлечения внимания магазинов вы решили написать программу, которая будет выдавать информацию о товарах, производимых на вашей фабрике их количестве и цене, а также производить элементарные вычисления, чтобы продавцам не тратить время на подсчеты столбиком.

Вам необходимо создать базовый класс Product с полями: cost, quantity и двумя виртуальными функциями print_info() и price_for_bulk_purchases(). Так же создать три производных класса от класса Product (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и price_for_bulk_purchases(). Переопределенная функция print_info() должна вывести поля cost, quantity и общую сумму за весь товар, равную $\text{cost} \times \text{quantity}$. А переопределенная функция price_for_bulk_purchases() должна вывести сколько нужно заплатить при покупке всего товара оптом: $\left(\text{cost} \times \left[(\text{процент скидки любой})/100\right]\right) \times \text{quantity}$.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Вашу программу начали использовать магазины, но как оказалось им не хватает обратной связи. Вам необходимо написать клиент-серверное приложение с возможностью отправки сообщений. Каждое сообщение должно содержать дату отправки (класс QDate) и название магазина.



Вариант 8

Задача 1

Задача 1 выполняется с использованием Visual Studio. Необходимо написать объектно-ориентированную программу используя концепцию наследования. Базовый класс программы должен содержать конструктор с параметрами для инициализации полей. Производные классы программы должны содержать конструкторы с параметрами и переопределенные функции.

Ваша блестящая идея "Своя комната" принесла много новых клиентов, но отель из-за низкой стоимости комнат начал нести убытки, именно поэтому вам пришлось вернуть стандартные номера и фиксированные цены на них. Чтобы клиент мог посмотреть информацию об номерах, вы установили в холле отеля интерактивное табло, где представлены фото номеров и их стоимость. Осталось только запрограммировать это табло.

Вам необходимо создать базовый класс Room с полями: cost, number_of_residents, convenience и двумя виртуальными функциями print_info() и how_much_is_days(). Так же создать три производных класса от класса Room (название классам придумайте самостоятельно) в которых будет происходить переопределение функций print_info() и how_much_is_days(). Переопределенная функция print_info() должна вывести поля cost, number_of_residents и convenience. А переопределенная функция how_much_is_days(). должна вывести цену за три дня проживания в данном номере. Если $convenience < 3$ то цена вычисляется по формуле $cost \times 3$. Если $3 < convenience <= 5$ то цена вычисляется по формуле $cost \times 3 + cost \times 0.15 \times number_of_residents$.

Задача 2

Задача 2 выполняется с использованием инструментов Qt.

Интерактивное табло пользуется вниманием клиентов, но они все ровно подходят к ресепшн и задают дополнительные вопросы. Чтобы избежать этого, вам необходимо написать клиент-серверное приложение "Онлайн поддержка" с возможностью отправки сообщений. Каждое сообщение должно содержать время отправки (класс QTime).