Институт информатики, математики и электроники
Кафедра геоинформатики и информационной безопасности

Технологии и методы программирования

Лабораторная работа № 2

Наследование и динамический полиморфизм

Самара – 2019

1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Литература:

- S. Lippman и др. C++. Базовый курс
  - *Глава 15. Объектно-ориентированное программирование*
- B. Eckel. Философия C++. Том 1. Введение в стандартный C++
  - *Глава 14. Наследование и композиция*
  - *Глава 15. Полиморфизм и виртуальные функции*

1.1 Наследование

Наследование – концепция объектно-ориентированного программирования (ООП), согласно которой тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию кода. Наряду с инкапсуляцией и полиморфизмом является одним из ключевых принципов ООП.

Рассмотрим пример, в котором класс `B` наследуется от класса `A`. Класс `A` при этом называют базовым классом (альтернативные термины: родительский класс, предок, суперкласс); класс `B` называют производным классом (альтернативные термины: дочерний класс, потомок, подкласс):

```
class A {
private:   // Секция закрытых (приватных) членов класса
    // ...
protected: // Секция защищённых членов класса
    // ...
public:
    void a() {
        std::cout << "A::a()" << std::endl;
    }
};

// Объявление открытого наследования от класса A
// Класс B неявно получает все поля и методы (в том числе приватные) класса A
class B : public A {
public:
    void b() {
        std::cout << "B::b()" << std::endl;
    }
};

int main() {
    B b;   // Тип B "работает как" тип A
    b.a(); // Вызов унаследованного метода A::a()
    b.b(); // Вызов собственного метода B::b()
}
```

В связи с введением понятия наследования, наряду с уже известными модификаторами доступа `private` и `public` появляется новый модификатор `protected`. Члены класса, объявленные как `protected`, видны только внутри класса и в классах-наследниках.

В примере выше объект класса `B` будет иметь («получит в наследство») все поля и методы `A`. (в том числе приватные). При этом доступа к приватным членам предка не будет. Но это не помешает их косвенному использованию через доступные (защищённые и открытые) методы предка.

Потомок может управлять доступностью унаследованных членов для внешнего кода. Для этого при объявлении наследования перед именем базового класса ставится один из трёх модификаторов доступа, который определяет **вид наследования**. Различают следующие виды наследования:

- **Открытое** (`public`) – доступность унаследованных методов не изменяется. Открытые члены предка остаются открытыми в потомке, защищённые остаются защищёнными. При этом говорят, что класс наследует как интерфейс, так и реализацию предка.
- **Защищённое** (`protected`) – доступность унаследованных методов не превышает уровень `protected`. То есть открытые члены предка становятся защищёнными в потомке, защищённые остаются защищёнными.
- **Закрытое** (`private`) – доступность унаследованных методов снижается до уровня `private`. То есть открытые и защищённые члены предка становятся закрытыми в потомке. При этом говорят, что класс наследует только реализацию. Это значит, что только сам потомок знает, что он потомок некого класса. Для любого внешнего кода это неизвестно, т. е. такой код при обращении к потомку не может использовать интерфейс предка.

В дальнейшем будем рассматривать только открытое наследование, представляющее наибольший практический интерес.

Рассмотрим процессы создания и уничтожения объектов унаследованных классов.

При создании объекта дочернего класса компилятор производит неявный вызов конструкторов по умолчанию всех базовых классов в <u>нисходящем</u> порядке.

Пусть дана иерархия классов `A ⇐ B ⇐ C`: `A` – базовый класс, `B` – наследник `A`, `C` – наследник `B`. Тогда при создании объекта `C` сначала выполнится конструктор по умолчанию класса `A`, затем – конструктор по умолчанию класса `B`, и только в последнюю очередь будет выполнен используемый конструктор класса `C`.

При необходимости конструктор дочернего класса может явно специфицировать вызов конкретного конструктора базового класса, вызвав его в списке инициализации (см. пример ниже).

При уничтожении объекта дочернего класса компилятор производит неявный вызов деструкторов всех базовых классов в <u>восходящем</u> порядке.

При той же самой иерархии `A ⇐ B ⇐ C` при уничтожении объекта `C` сначала выполнится деструктор `C`, затем – деструктор `B`, и в самом конце – деструктор `A`.

Указанные правила проиллюстрированы на следующем примере:

```
struct A {
    A() {
        std::cout << "A::A()" << std::endl;
    }

    ~A() {
        std::cout << "A::~A()" << std::endl;
    }
};

struct B : public A {
    B(int) {
        // До выполнения конструктора происходит неявный вызов
        // конструктора по умолчанию базового класса A::A()
        std::cout << "B::B(int)" << std::endl;
    }

    ~B() {
        std::cout << "B::~B()" << std::endl;
        // После выполнения деструктора происходит неявный вызов
        // деструктора базового класса A::~A()
    }
};

struct C : public B {
    C() : B(0) {
        // До выполнения конструктора происходит явный вызов
        // конструктора с параметром базового класса B::B(int)
        std::cout << "C::C()" << std::endl;
    }

    ~C() {
        std::cout << "C::~C()" << std::endl;
        // После выполнения деструктора происходит неявный вызов
        // деструктора базового класса B::~B()
    }
};

int main() {
    C c;
}
```

Указанный код приведёт к следующему выводу:

```
A::A()
B::B(int)
C::C()
C::~C()
B::~B()
A::~A()
```

1.2 Виртуальные методы

Рассмотрим следующий пример.

```
class A {
public:
    void f() const {                        // Объявление невиртуального метода
        std::cout << "A::f()" << std::endl;
    }

    virtual void vf() const {               // Объявление виртуального метода
        std::cout << "A::vf()" << std::endl;
    }
};

class B : public A {
public:
    void f() const {                        // Скрытие унаследованного метода
        std::cout << "B::f()" << std::endl;
    }

    virtual void vf() const override {      // Переопределение метода
        std::cout << "B::vf()" << std::endl;
    }
};
```

Здесь предоставлена иерархия классов `A ⇐ B`. В классе `A` объявлен обычный (невиртуальный) метод `f` и виртуальный метод `vf`. Класс `B` наследует от `A`, предоставляя при этом собственные реализации обоих методов.

Про невиртуальный метод `f` говорят, что в классе `B` происходит **скрытие** («name shadowing») унаследованного метода. **Вызываемая реализация невиртуального метода всегда известна на этапе компиляции и определяется формальным типом объекта** (механизм раннего связывания; «static binding»).

Про виртуальный метод `vf` говорят, что в классе `B` происходит **переопределение** («override») метода. **Вызываемая реализация виртуального метода может быть неизвестна на этапе компиляции и определяется фактическим типом объекта во время выполнения программы** (механизм позднего связывания, «dynamic binding»).

Изложенные ключевые правила для вышеприведённой иерархии классов `A ⇐ B` иллюстрируются следующим кодом:

```
/* Формальный тип: A */   A *paa = new A;   // Фактический тип: A
/* Формальный тип: A */   A *pab = new B;   // Фактический тип: B
/* Формальный тип: B */   B *pbb = new B;   // Фактический тип: B

// Механизм раннего связывания
// Вызываемая реализация НЕВИРТУАЛЬНОГО метода определяется ФОРМАЛЬНЫМ типом
paa->f(); // Выведет A::f() (т. к. формальный тип: A)
pab->f(); // Выведет A::f() (т. к. формальный тип: A)
pbb->f(); // Выведет B::f() (т. к. формальный тип: B)

// Механизм позднего связывания
// Вызываемая реализация ВИРТУАЛЬНОГО метода определяется ФАКТИЧЕСКИМ типом
paa->vf(); // Выведет A::vf() (т. к. фактический тип: A)
pab->vf(); // Выведет B::vf() (т. к. фактический тип: B)
pbb->vf(); // Выведет B::vf() (т. к. фактический тип: B)


// ... // Удаление созданных объектов
```

Обычно сокрытие никогда не используется и может являться признаком ошибки, т. к. при наличии иерархии классов принципиально важно определение вызываемой реализации по фактическому (реальному) типу объекта, а не по формальному типу указателя или ссылки. Но это не означает, что все методы должны быть виртуальными! Метод должен быть виртуальным только тогда, когда базовый класс подразумевает возможное предоставление наследником собственной реализации. Если логика метода не подразумевает его переопределение в наследнике, то делать такой метод виртуальным смысла нет, т. к. это приведёт лишь к дополнительным издержкам.

При удалении объекта для определения реализации деструктора, начиная с которого начнётся цепочка восходящих вызовов, применяются те же самые правила, что и для невиртуальных и виртуальных методов. Это означает, что **открытый деструктор базового класса должен быть виртуальным**, чтобы уничтожение объекта по ссылке или указателю на базовый класс привело к полной цепочке восходящих вызовов деструкторов, начиная с деструктора фактического типа. Деструктор может быть защищённым и невиртуальным, если не предполагается удаление объекта по ссылке или указателю на базовый класс.

Если базовый класс подразумевает, что потомок **обязан** переопределить виртуальный метод, то такой метод можно объявить **чисто виртуальным**:

```
class A {
public:
    virtual void pvf() const = 0; // Объявление чисто виртуального метода
    virtual ~A() = default;       // Объявление открытого виртуального деструктора
};

class B : public A {
public:
    virtual void pvf() const override {
        std::cout << "B::pvf()" << std::endl;
    }
};
```

В базовом классе для чисто виртуального метода можно не предоставлять реализацию (обычно так и делают; предоставление чисто виртуального метода с реализацией – очень редкий сценарий). Класс, содержащий хотя бы один чисто виртуальный метод, называют **абстрактным**. Экземпляры таких классов не могут быть созданы. При этом в цепочке наследо-

вания все производные классы также становятся абстрактными до тех пор, пока все чисто виртуальные методы не будут переопределены.

При разработке программ часто применяются классы, содержащие только чисто виртуальные методы. Такие классы называют **интерфейсами** или **классами протокола**. Они используются для определения строгого протокола взаимодействия между компонентами программы.

Использование интерфейсов позволяет устранить зависимость (абстрагироваться) от конкретных типов данных, что *значительно/невероятно/неимоверно/чрезвычайно (!)* упрощает как разработку, так и последующее сопровождение программы. Это позволяет писать код, работающий с объектами на уровне базового класса, не заботясь при этом о реальных типах.

Разберём, как использованный материал может быть применён для абстрагирования различных операций над математическим объектом «вектор»:

```
class Vector {
public:
    // Чисто виртуальные методы, обеспечивающие абстрактную работу с вектором
    virtual size_t getSize() const = 0;
    virtual float get(size_t index) const = 0;
    virtual void set(size_t index, float value) = 0;

    // Открытый виртуальный деструктор
    virtual ~Vector() = default;

protected:
    // 1. Операции копирования сделаны защищёнными, т. к. на уровне базового класса
    //    нельзя обеспечить для них виртуальное поведение.
    // 2. При этом они не запрещены, чтобы объекты конкретных типов-наследников
    //    могли быть скопированы непосредственно (если они допускают такую возможность).
    // 3. При необходимости копирования на уровне базового класса
    //    можно предоставить виртуальную функцию clone().
    Vector() = default;
    Vector(const Vector&) = default;
    Vector& operator=(const Vector&) = default;
};
```

При этом можно предоставить несколько наследников с различными реализациями. Например, могут быть предоставлены следующие типы-наследники:

- `BufferedVector`, который хранит данные в динамическом массиве (оперативной памяти).
- `SubVector`, который вообще не хранит данные сам, а представляет из себя лишь обёртку для удобной работы с частью другого вектора. Например, если есть некоторый вектор $\{0, 1, 2, 3\}$, `SubVector` может представлять часть изначального вектора, например, $\{1, 2\}$. Естественно, все изменения с таким подвектором отображаются на изначальном векторе.
- `FileVector`, который не хранит данные в оперативной памяти, а работает напрямую с некоторым файлом на диске. В методе `get` при этом происходит считывание значения с нужной позиции в файле, в методе `set` – запись. Такая реализация может пригодиться при огромных размерах вектора для экономии оперативной памяти.

Независимо от количества наследников с различными реализациями, наличие базового класса предоставляет возможность писать код на абстрактном уровне, не привязываясь к конкретным типам. Например, напишем функцию, вычисляющую сумму двух векторов:

```
void sum(const Vector* lhs, const Vector* rhs, Vector* result) {
    const auto size = result->getSize();
    if (size != lhs->getSize() || size != rhs->getSize()) {
        throw std::logic_error("Размерности не совпадают!");
    }
    for (size_t i = 0; i < size; ++i) {
        result ->set(i, lhs->get(i) + rhs->get(i));
    }
}
```

Предоставленная функция позволяет получить сумму двух векторов, ничего не зная о конкретных типах входных векторов `lhs` и `rhs` и выходном типе `result` (кроме того факта, что все они реализуют интерфейс `Vector`). Более того, на этапе компиляции конкретные типы указанных объектов могут быть неизвестными!

Механизм виртуальных методов является средством реализации концепции **динамического полиморфизма** (т. е. полиморфизма времени выполнения – когда поведение объекта определяется не на этапе компиляции, а по фактическому типу объекта во время работы программы).

1.3 Итоговый обзор ключевых принципов объектно-ориентированного программирования

Абстракция. Заключается в выделении наиболее значимых характеристик объекта, доступных остальной программе. Позволяет работать с объектами, не вдаваясь в особенности их реализации.

Инкапсуляция. Заключается в сокрытии любых деталей реализации от внешнего кода. Другими словами, любой класс должен рассматриваться как чёрный ящик, которым можно управлять только посредством предоставленного им интерфейса (открытых методов). Инкапсуляция исключает возможную зависимость внешнего кода от деталей реализации, что приводит к меньшей связности различных компонентов программы и позволяет менять реализацию, не ломая при этом внешний код.

Наследование. Способность класса (потомка) наследовать все члены другого класса (предка). Приводит к возникновению **иерархии классов**. Является механизмом повторного использования кода, но при этом важно понимать следующее: **открытое наследование не должно применяться для повторного использования кода, находящегося в базовом классе; открытое наследование необходимо для того, чтобы быть повторно использованным существующим кодом, который полиморфно использует объекты базового класса**. Подробнее см. рекомендацию № 37 в книге «Стандарты программирования на C++» (Herb Sutter, Andrei Alexandrescu). Другими словами, открытое наследование всегда должно моделировать отношение «работает как» (то есть для любого предка `A` и его наследника `B` выражение «`B` работает как `A`» должно быть истинным). В любом коде, который формально использует `A`, использование фактического типа `B` также должно являться корректным.

Полиморфизм. Предоставляет мощнейшие средства для абстракции посредством единообразной работы с различными типами данных. Условно разделяется на два вида:

- **Динамический полиморфизм**. Реализуется в иерархии классов с помощью механизма виртуальных методов. Основной принцип заключается в определении вызываемой реализации метода по фактическому типу объекта. Момент определения вызываемой реализации происходит во время выполнения программы. **Преимущества**: единообразная работы с объектами, фактический тип которых известен только во время выполнения программы. **Недостатки**: издержки времени выполнения.
- **Статический полиморфизм**. Реализуется с помощью обобщённых функций и классов, которые работают с некоторыми обобщёнными типами. Подстановка конкретных типов происходит на этапе компиляции. **Преимущества**: сокращение размера исходного кода, генерация кода на этапе компиляции (возможность оптимизации компилятором, отсутствие издержек времени выполнения). **Недостатки**: конкретный тип должен быть известен на этапе компиляции, размер объектного кода не сокращается.

1.4 Темы для самостоятельного изучения

- множественное наследование: проблема ромба и механизмы её разрешения;
- приведение типов:
  - `static_cast`,
  - `dynamic_cast`,
  - `const_cast`,
  - `reinterpret_cast`.

2 ТРЕБОВАНИЯ К ПРОЕКТИРОВАНИЮ И СТИЛЮ КОДА

Код лабораторной работы должен удовлетворять всем перечисленным ниже рекомендациям, а также всем требованиям, изложенных в **аналогичных разделах предыдущих лабораторных работ**. Отклонения допускаются, если вы сможете обосновать свой подход.

2.1 Herb Sutter, Andrei Alexandrescu. Стандарты программирования на C++

Проектирование классов и наследование	
32	Ясно представляйте, какой вид класса вы создаёте
33	Предпочитайте минимальные классы монолитным
34	Предпочитайте композицию наследованию
35	Избегайте наследования от классов, которые не спроектированы для этой цели
36	Предпочитайте предоставление абстрактных интерфейсов
37	Открытое наследование означает заменимость. Наследовать надо не для повторного использования, а чтобы быть повторно использованным
38	Практикуйте безопасное перекрытие
39	Виртуальные функции стоит делать неоткрытыми, а открытые – невиртуальными
Конструкторы, деструкторы и копирование	
49	Избегайте вызова виртуальных функций в конструкторах и деструкторах
50	Делайте деструкторы базовых классов открытыми и виртуальными либо защищёнными и невиртуальными
54	Избегайте срезки. Подумайте об использовании в базовом классе клонирования вместо копирования
55	Предпочитайте канонический вид присваивания
Шаблоны и обобщённость	
64	Разумно сочетайте статический и динамический полиморфизм

2.2 Scott Meyers. Эффективный и современный C++

Глава 3. Переход к современному C++	
3.6	Объявляйте перекрывающие функции как `override`

3 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать иерархию классов согласно варианту.
2. Независимо от варианта, в базовом классе должен быть предоставлен метод `print(…)` распечатки строкового представления объекта.
3. Реализовать основное меню приложения, которое обеспечивает работу пользователя с коллекцией объектов, реализованной посредством динамического массива указателей на экземпляр базового класса. Предоставить пользователю в дополнение к функционалу, требуемого согласно варианту, следующие пункты меню:
   - создание объекта указанного типа из иерархии классов с его последующей вставкой в конец коллекции;
   - вставка в конец коллекции случайно сгенерированного объекта (случайными должны быть как типы объектов, так и их содержимое);
   - удаление объекта из коллекции по индексу;
   - удаление всех объектов из коллекции;
   - вывод содержимого коллекции на экран согласно заданию.

3.1 Вариант 1

Описать и реализовать базовый класс **фигура** и классы-наследники **круг**, **треугольник**, **прямоугольник**. Предусмотреть методы расчёта площади и периметра, вывода информации о фигуре. Реализовать процедуру вывода списка фигур с указанием площади и периметра.

3.2 Вариант 2

Описать и реализовать базовый класс **фигура** и классы-наследники **куб**, **шар**, **конус**. Предусмотреть методы расчёта площади поверхности и объёма, вывода информации о фигуре. Реализовать процедуру вывода списка фигур с указанием площади поверхности и объема.

3.3 Вариант 3

Описать и реализовать базовый класс **функция** и классы-наследники **константа**, **линейная зависимость**, **квадратичная зависимость**. Предусмотреть методы расчёта значения функции для заданного аргумента, возврата объекта-производной, вывода функции на экран. Реализовать процедуру вывода списка функций, их производных и значений в заданной точке.

3.4 Вариант 4

Описать и реализовать базовый класс **функция** и классы-наследники **константа**, **показательная функция**, **степенная функция**. Предусмотреть методы расчёта значения функции для заданного аргумента, возврата объекта-производной, вывода функции. Реализовать процедуру вывода списка функций, их производных и значений в заданной точке.

3.5 Вариант 5

Описать и реализовать базовый класс **функция** и классы-наследники **константа**, и два класса **гармонические функции вида Acos(ωx+φ) и Asin(ωx+φ)**. Предусмотреть методы расчёта значения функции для заданного аргумента, возврата объекта-производной,

вывода функции. Реализовать процедуру вывода списка функций, их производных и значений в заданной точке.

3.6 Вариант 6

Описать и реализовать базовый класс **сотрудник** и классы-наследники **штатный** и **совместитель**. Предусмотреть методы вывода информации о сотруднике и расчёта заработной платы. Заработная плата у штатных сотрудников фиксированная с надбавкой 0.5% за каждый год стажа, а у совместителей она почасовая, с возможностью индивидуальной надбавки до 5%. Реализовать процедуру вывода списка сотрудников с указанием зарплаты.

3.7 Вариант 7

Описать и реализовать базовый класс **транспортное средство** и классы - наследники **легковой автомобиль, грузовой автомобиль** и **мотоцикл**. Предусмотреть методы вывода информации о транспортном средстве и расчёта транспортного налога. Транспортный налог для легковых автомобилей равен S*V, для грузовых S*V*([T/2]+1), для мотоциклов 0.3*S*V, где S - базовая ставка, V – объём двигателя, T – тоннаж. Реализовать процедуру вывода списка транспортных средств с указанием марки, модели, типа и размера налога.

3.8 Вариант 8

Описать и реализовать базовый класс **прогрессия** и классы-наследники **арифметическая и геометрическая**. Предусмотреть методы вывода информации и расчёта вычисления i-го элемента и суммы элементов до n-го элемента. Реализовать процедуру вывода списка прогрессий с указанием значений заданного элемента и суммы до заданного элемента.

3.9 Вариант 9

Описать и реализовать базовый класс **шахматная фигура** и три класса наследника **пешка**, **слон**, **ладья**. Для фигуры задаётся цвет и текущая клетка (конструктор по умолчанию размещает на начальной позиции), предусмотреть методы проверки может ли фигура переместиться на заданную клетку, может ли фигура атаковать указанную клетку. Реализовать процедуру вывода списка фигур с указанием типа, значения заданного элемента и суммы до заданного элемента.

3.10 Вариант 10

Описать и реализовать базовый класс **шахматная фигура** и три класса наследника **конь**, **ферзь**, **король**. Для фигуры задаётся цвет и текущая клетка (конструктор по умолчанию размещает на начальной позиции), предусмотреть методы проверки может ли фигура переместиться на заданную клетку, может ли фигура атаковать указанную клетку. Реализовать процедуру вывода списка фигур с указанием типа, значения заданного элемента и суммы до заданного элемента.

3.11 Вариант 11

Описать и реализовать базовый класс **фигура** и классы-наследники **эллипс**, **квадрат**, **трапеция**. Предусмотреть методы расчёта площади и периметра, вывода информации о фигуре. Реализовать процедуру вывода списка фигур с указанием площади и периметра.

3.12 Вариант 12

Описать и реализовать базовый класс **фигура** и классы-наследники **параллелепипед**, **цилиндр**, **правильная пирамида**. Предусмотреть методы расчёта площади поверхности и объёма, вывода информации о фигуре. Реализовать процедуру вывода списка фигур с указанием площади поверхности и объема.

3.13 Вариант 13

Описать и реализовать базовый класс **имущество** и классы-наследники **квартира**, **транспорт**, **земельный участок**. Предусмотреть вывод информации об объекте имущества и метод расчета налога на имущество. Налог на квартиру рассчитывается как 0.1% для квартир стоимостью меньше 2 млн и 0.3% для квартир стоимостью выше 2 млн, налог на транспортное средство рассчитывается из базовой ставки S умноженной на коэффициент, зависящий от мощности двигателя (0.25 <= 100 л/с, 100 л/с <0.5<= 150 л/с, 150 л/с <1<= 200 л/с, 200 л/с <1.5<= 250 л/с, 250 л/с <2), также для транспортных средств стоимостью более 3 млн применяется повышающий коэффициент 2, налог на земельный участок рассчитывается по ставке 1% от кадастровой стоимости для участков за городом и 5 % для участков в черте города. Реализовать процедуру вывода списка объектов имущества с указанием налога.

3.14 Вариант 14

Описать и реализовать базовый класс гражданин и классы-наследники Школьник (ФИО, возраст, № класса), Студент (ФИО, № зачёной книжки, № группы, ср балл по экз, ср бал по зач) и Пенсионер (ФИО, № СНИЛС, стаж). Предусмотреть методы вывода информации о гражданине и расчёта выплаты от государства. Выплаты от государства у школьника из многодетных семей в размере 0.5*МРОТ, у остальных – отсутствует, у студента – в зависимости от ср. баллов, но не более 0.8*МРОТ, у пенсионера – если стаж работы менее 5 лет то выплата 0, от 5 до 30 лет выплата от 1*МРОТ до 4*МРОТ, если более 30 лет, то максимальная выплата 5*МРОТ. Реализовать процедуру вывода списка граждан с указанием размера выплаты.

3.15 Вариант 15

Описать и реализовать базовый класс **персонаж** и классы-наследники **рыцарь** (броня, жизни, урон, вероятность снизить урон вдвое), **ассассин** (броня, жизни, урон, вероятность повторной атаки в этот ход) и **берсерк** (броня, жизни, урон, вероятность утроенного урона). Предусмотреть методы «Принять урон», «Нанести урон», «Применить умение»: У классов следующие умения: у Рыцаря – увеличить броню на 10 ед. и ослабить урон на 2 ед., у ассасина – уворот от атаки, у берсерка – увеличить урон на 5, увеличить вероятность утроенного удара в 2 раза, уменьшить броню до 0. Реализовать процедуру вывода списка персонажей и процедуру боя двух выбранных персонажей до победы одного из них: за 1 раунд случайно выбирается либо, атака либо использование умения.