ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Учебно-методическое пособие

по дисциплине

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Автор

Габрельян Б.В.

Ростов-на-Дону, 2018

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направления 09.03.04 «Программная инженерия».

Авторы

доцент, к.ф.-м.н.,
зав.каф. ПОВТиАС
Габрельян Б.В.

Оглавление

1. ЛАБОРАТОРНАЯ РАБОТА №1: КОНСТРУКТОРЫ И ДЕСТРУКТОР

1.1. Теория

I. Конструкторы класса.

Для инициализации (задания начального значения) объекта класса (как при статическом, так и при или динамическом создании объекта) для него вызывается специальный метод класса, называемый конструктором. Конструктор – это метод, имя которого в точности совпадает с именем класса. Для класса можно создать несколько конструкторов, тогда они будут отличаться списками своих аргументов. Если в описании класса ни один конструктор не задан явно, компилятор создаст конструктор без аргументов. Но если задать хотя бы один конструктор для своего класса, компилятор не станет создавать конструктор по умолчанию. Например,

```
class A {
    int a;
  public:
    A(int x) { a = x; } /* конструктор с одним аргументом типа int, конструктор по умолчанию не создается */
};

int main() {
x       A a1;         /* Ошибка. Нет конструктора без аргументов. */
    A a2(10); /* Правильно. Вызывается конструктор с аргументом типа int */
    return 0;
}
```

II. Конструктор копирования.

В ситуациях, когда требуется создание копии объекта, т.е. когда новый объект создается на основе уже существующего объекта того же класса, вызывается конструктор специального вида – конструктор копирования. Это конструктор с одним аргументом – ссылкой на объект собственного класса. Например,

```
class A {
    int a;
  public:
```

```
    A() { a = 0; } /* конструктор без аргументов */
    A(A& x) {     /* конструктор копирования */
        a = x.a;
    }
};

int main() {
    A a1;      /* Вызывается конструктор без аргументов */
    A a2( a1 );   /* Вызывается конструктор копирования */
    A a3 = a1;   /* Вызывается конструктор копирования */
    return 0;
}
```

Если аргумент функции или метода – объект класса (но не ссылка на объект) то при вызове в ней (или в нем) создается новый локальный объект как копия переданного объекта, т.е. будет вызываться конструктор копирования. То же самое происходит, если функция или метод возвращают объект класса.

Если конструктор копирования для класса не задан явно, то компилятор создает его. Этот конструктор просто копирует значения полей старого объекта в соответствующие поля нового объекта. Это может приводить к проблеме повисших ссылок, если хотя бы одно поле класса указатель.

III. Конструкторы преобразования.

Для того, чтобы разрешить преобразование объектов чужого класса или величин примитивных типов в объект нашего класса необходимо задать для него конструкторы преобразования. Конструктор преобразования – это конструктор с одним аргументом. Тип аргумента определяет тот тип, который может быть преобразован в объект класса. Например,

```
class A {
    int a;
  public:
    A(int x) { a = x; } /* конструктор с одним аргументом типа int */
};

void f( A x ); /* прототип функции с одним аргументом – объектом класса A */

int main() {
```

```
    A a1(10); /* вызывается конструктор с одним аргументом
*/
    f( 5 );      /* вызывается конструктор преобразования */
    return 0;
}
```

В этом примере при вызове функции f компилятор создаст временный объект класса A т.к. есть конструктор преобразования величины типа int в объект класса. Единственный конструктор в этом классе теперь выполняет две роли. Во-первых, это конструктор, вызываемый при создании новых объектов. Во-вторых, он задает преобразование целых значений в объекты класса. C++ позволяет объявлять конструкторы с одним аргументом так, чтобы их нельзя было использовать для преобразования типа. Для этого конструктор должен иметь модификатор explicit. Например,

```
class A {
    int a;
  public:
    explicit A(int x) { a = x; }
};

void f( A x ); /* прототип функции с одним аргументом –
объектом класса A */

int main() {
    A a1(10); /* вызывается конструктор с одним аргументом
*/
x   f( 5 );      /* Ошибка. Не определено преобразование int
в A */
    return 0;
}
```

IV. Деструктор класса.

Перед уничтожением объекта для него обязательно вызывается специальный метод класса, называемый деструктором. Имя деструктора начинается со значка тильда ~, после которого следует имя класса. Деструктор нельзя перегружать. Если он не задан для класса явно, то будет создан компилятором. Пример:

```
class Person {
    char *name;
    int age;
  public:
```

```
        Person( char *n, int a) { /* конструктор */
            name = new char[ strlen(n) + 1 ]; /* захват дополни-
тельной памяти */
            strcpy( name, n );
            age = a;
        }
        ~Person() { /* деструктор */
            delete[] name; /* освобождение памяти, захваченной
конструктором */
        }
    };
```

1.2. Задание к лабораторной работе

1. Создать структуру "Студент" содержащую следующие поля:

- имя студента - произвольная длина
- отчество студента - произвольная длина
- фамилию студента - произвольная длина
- год рождения
- группа.

2. Определить конструктор для инициализации полей структуры со значениями по умолчанию. Определить конструктор копирования и деструктор. Написать тестовый пример.

3. Изменить в описании структуры ключевое слово struct на class. Запустить программу. Какие возникли проблемы? Почему? Как их исправить?

4. Написать интерфейсные функции доступа к полям класса (получить/задать значение поля).

5. Внести в конструкторы и деструктор выдачу сообщений на экран о том, какая функция была вызвана. Модифицировать функцию main следующим образом:

```
void main(void)
{
   cout<<"Вход в функцию main()"<<endl;
   ...
   <те-          ло_main()>
```

```
    ...
    cout<<"Выход из функции main()"<<endl;
}
```

Выяснить время вызовов конструкторов и деструкторов.

6. Описать глобальную функцию Student test(Student s){return s;}

Вызвать ее в основной программе. Что произошло и почему?

7. Изменить передачу параметра ф-и test на передачу по ссылке.

Что изменилось?

8. Изменить возврат результата ф-и test на передачу по ссылке.

Что изменилось?

1.3. Контрольные вопросы

1. Каким образом инициализируются объекты класса в C++?
2. Может ли у класса быть несколько конструкторов?
3. Чем конструктор копирования отличается от других конструкторов класса?
4. Может ли конструктор копирования вызываться неявно?
5. Когда необходимо явно определять конструктор копирования для класса?
6. Как можно задать преобразование величины целого типа в объект нашего класса?
7. Как можно запретить использовать конструктор для преобразования типа?
8. Может ли у класса быть несколько деструкторов?
9. Когда вызывается деструктор класса?
10. Каково назначение деструктора?

1.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

2. ЛАБОРАТОРНАЯ РАБОТА №2: ССЫЛКИ. КОПИРОВАНИЕ И ПРИСВАИВАНИЕ

2.1. Теория

I. Создание и использование ссылок в C++.

Ссылка в C++ реализуется как скрытый константный указатель. Ссылка связывается с некоторым адресом и далее всегда остается связанной только с этим адресом. Однако этим адресом управляет не программист, а компилятор. При всяком обращении к ссылке компилятор обязательно проводит разадресацию и возвращает сам объект. Таким образом, мы всегда думаем о ссылке как об объекте, а не адресе. Например,

```
char ch = 'A';
char &rch = ch; /* ссылка на char связывается с переменной в момент создания */
rch = 'B'; /* результат такой же, как после ch = 'B'; */
```

II. Назначение конструктора копирования.

В ситуациях, когда требуется создание копии объекта, т.е. когда новый объект создается на основе уже существующего объекта того же класса, вызывается конструктор специального вида – конструктор копирования. Это конструктор с одним аргументом – ссылкой на объект собственного класса. Например,

```
class A {
    int a;
  public:
    A() { a = 0; } /* конструктор без аргументов */
    A(A& x) {     /* конструктор копирования */
        a = x.a;
    }
};

int main() {
    A a1;      /* Вызывается конструктор без аргументов */
    A a2( a1 );   /* Вызывается конструктор копирования */
    A a3 = a1;   /* Вызывается конструктор копирования */
    return 0;
}
```

Если аргумент функции или метода – объект класса (но не ссылка на объект) то при вызове в ней (или в нем) создается но-

вый локальный объект как копия переданного объекта, т.е. будет вызываться конструктор копирования. То же самое происходит, если функция или метод возвращают объект класса.

III. Конструктор копирования по умолчанию.

Если конструктор копирования для класса не задан явно, то компилятор создает его. Этот конструктор просто копирует значения полей старого объекта в соответствующие поля нового объекта. Это может приводить к проблеме внешней зависимости, в частности к проблеме повисших ссылок, если хотя бы одно поле класса указатель. В этом случае внешний адрес сохраняется в объекте класса и все изменения, выполненные вне объекта, скажутся на этом объекте. В таких случаях обычно нужно явно переопределить конструктор копирования для класса. Бывают ситуации, когда копирование объектов вообще недопустимо. Тогда нужно объявить конструктор копирования как закрытый, причем достаточно задать только объявление (прототип) а реализацию можно не задавать вообще.

```
class Socket {
    …
private:
    Socket(Socket&);
    …
};
```

IV. Конструктор копирования и операция присваивания.

По умолчанию, присваивание также заключается в копировании значений полей одного объекта в соответствующие поля другого объекта этого же класса. Поэтому, как и в случае конструктора копирования, если хотя бы одно поле класса указатель, скорее всего, необходимо переопределить для него операцию присваивания.

```
class A {
    char *name;
public:
    A(char *n) { name = strdup(n); }
    ~A() { free(name); }
    A(A& obj) { name = strdup(obj.name); }
    A& operator=(const A& obj) {
        if( this == &obj ) return; // проверка на самоприсваивание
```

```
        free(name);
        name = strdup(obj.name);
    }
    …
};
```

2.2. Задание к лабораторной работе

1. Для решения различных задач используются методы Монте-Карло, предполагающие применение массивов случайных чисел с большим количеством элементов. Размер массива становится известным во время выполнения программы, т.е. массив должен создаваться динамически. Создайте две функции для решения одной и той же задачи: динамическое создание и заполнение случайными числами массива указанного размера. Первая функция должна использовать возвращаемое значение для передачи пользователю сгенерированного массива, а вторая должна передавать массив через один из своих аргументов. Стандартная библиотека Си содержит функции int rand() и void srand(unsigned) для генерации псевдослучайных чисел (прототипы в файле stdlib.h).

2. Создать класс Person содержащий следующие поля:
   - фамилия и имя человека - произвольная длина
   - профессия человека - произвольная длина
   - возраст человека.

3. Определить конструктор для инициализации полей класса со значениями по умолчанию. Определить конструктор копирования, операцию присваивания и деструктор. Написать тестовый пример.

4. Написать интерфейсные функции доступа к полям класса (получить/задать значение поля).

5. Внести в конструкторы, операцию присваивания и деструктор выдачу сообщений на экран о том, какая функция была вызвана. Модифицировать функцию main следующим образом:

```
void main(void)
{
   cout<<"Вход в функцию main()"<<endl;
   …
   <тело_main()>
   …
   cout<<"Выход из функции main()"<<endl;
}
```

6. Задать глобальную функцию Person test(Person s) { return s; }

Вызвать ее в основной программе. Что произошло и почему?

7. Изменить передачу параметра функции test на передачу по ссылке.

Что изменилось?

8. Изменить возврат результата функции test на передачу по ссылке.

Что изменилось?

2.3. Контрольные вопросы

1. Чем ссылка в C++ отличается от указателя?
2. Чем конструктор копирования отличается от других конструкторов класса?
3. Может ли конструктор копирования вызываться неявно?
4. Когда необходимо явно определять конструктор копирования для класса?
5. Может ли аргумент конструктора копирования быть не ссылкой, а значением?
6. Как можно запретить копирование объектов класса?
7. Нужен ли для копирования объектов класса конструктор копирования, если для класса реализована операция присваивания?
8. Будет ли вызываться конструктор копирования класса A при вызове функции void f(A x)?
9. Будет ли вызываться конструктор копирования класса A при вызове функции void f(A& x)?
10. Будет ли вызываться конструктор копирования класса A при вызове функции A f()?
11. Будет ли вызываться конструктор копирования класса A при вызове функции A& f()?

2.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

3. ЛАБОРАТОРНАЯ РАБОТА №3: ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ НА ПРИМЕРЕ ДВУСВЯЗНОГО СПИСКА

3.1. Теория

I. Объектно-ориентированная реализация динамических структур данных.

В объектно-ориентированной парадигме каждое важное понятие решаемой задачи в общем случае должно быть представлено соответствующим классом. Например, если в программе нужно реализовать двунаправленный список, то этому понятию будет соответствовать класс с подходящим именем, скажем List. Но сам список в своей реализации использует такое важное понятие как узел списка, поэтому при реализации списка удобно создать еще один класс с именем, например, ListNode (узел списка). Этот класс нужен разработчику класса List, но нужен ли он программисту не создающему, а только использующему List? При использовании списка мы можем думать о тех данных, которые мы помещаем или извлекаем из списка, но совсем не интересуемся тем, как организовано хранение этих данные в классе List. Если это так, разработчик может спрятать от пользователя реализацию класса ListNode внутри реализации класса List, сделав последний вложенным закрытым (или защищенным) классом.

II. Представление двунаправленного списка в программе на C++.

Разделяя описание и реализацию, при создании двунаправленного списка его описание помещают в файл заголовков, а реализацию в .cpp-файл, который компилируется и может предоставляться пользователю как часть статической или динамической библиотеки. Описание классов, реализующих двунаправленный список, может быть таким:

Файл list.h

```
class List {
    // закрытый вложенный класс
    class ListNode {
    public:
        int key; // уникальное для каждого узла значение
        char *data; // данные, хранящиеся в узле
        ListNode *prev, *next; /* указатели на предыдущий и
```

```
следующий узлы */
        };

        ListNode *first; // указатель на первый узел списка
    public:
        List() : first(0) {}
        ~List() { del(); }
        void addData(int key, char *data); // добавить узел
        void removeData(int key); // удалить узел с указанным ключом
        char *findData(int key); // вернуть данные по ключу
        void show(); // отобразить список в консольном окне
    private:
        ListNode *findNode(int key); // поиск узла по ключу
        void del(); // удалить все узлы из списка
    };
```

III. Создание очереди и стека с помощью повторного использования кода.

Вместо того, чтобы создавать очередь или стек "с нуля", можно воспользоваться уже готовыми реализациями подходящих структур данных. Чаще всего очереди и стеки реализуют либо на основе массива, либо на основе списка. Организовать взаимодействие нового класса (стек, очередь) с уже имеющимся классом (например, List) можно по-разному. Это может быть отношение агрегирования, или наследования, отношение использования и т.д.

3.2. Задание к лабораторной работе

1. Создайте класс List, представляющий понятие "двунаправленный список".
2. Создайте класс, реализующий простое текстовое меню. Используйте его для тестирования созданного в предыдущем задании класса List.
3. Создайте классы стек и очередь на основе класса List с помощью агрегирования.
4. Создайте классы стек и очередь на основе класса List с помощью наследования.
5. Сравните реализации стека и очереди и сделайте вывод, какое отношение между классами агрегирование или наследование является правильным при построении очереди и стека на основе списка.

3.3. Контрольные вопросы

1. Какую дисциплину добавления/удаления элементов реализует очередь?
2. Какую дисциплину добавления/удаления элементов реализует стек?
3. Для чего можно использовать вложенный класс при реализации динамической структуры данных, такой как список?
4. Как организовать отношение между классами "агрегирование"?
5. Как организовать отношение между классами "наследование"?
6. Как запретить пользователю использовать стек или очередь как список, если эти классы реализованы на основе списка с помощью агрегирования?
7. Как запретить пользователю использовать стек или очередь как список, если эти классы реализованы на основе списка с помощью наследования?
8. Можно ли рассматривать построение стека или очереди на основе списка как "расширение"?
9. Как реализовать стек или очередь на основе вписка с помощью наследования на языках программирования не поддерживающих закрытое наследование (например, Java или C#)?
10. Какое отношение агрегирование или наследование следует предпочесть при реализации очереди или стека на основе списка?

3.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. ЛАБОРАТОРНАЯ РАБОТА №4: ПЕРЕГРУЗКА ОПЕРАЦИЙ

4.1. Теория

I. Дружественные функции и дружественные классы.

Если функция объявлена в классе дружественной, то она получает такие же привилегии что и методы самого класса, т.е. получает доступ не только к открытой, но и к защищенной и закрытой частям класса. Чтобы воспользоваться этими привилегиями функция должна каким-то образом получить объект класса: в качестве аргумента или создавая такой объект во время своей работы, т.к. скрытого параметра this у нее нет. Например,

```
class A {
    friend void fun(A x); // объявление внешней функции другом класса
     int a; // закрытое поле класса
   public:
    A(int x) { a = x; }
};
void fun(A x) { // реализация функции
cout<<"a="<<x.a // доступ к закрытому полю
      <<endl;
}
```

Дружественной функцией может быть объявлен метод другого класса, и даже другой класс в целом. В последнем случае все методы класса становятся дружественными функциями. Например,

```
class A;  // предварительное описание класса
class B {
  public:
void f(A& x);            // доступ к закрытому полю A
};
class C {};
class A {
friend void B::f(A&); // объявление дружественного метода
friend class C; // объявление дружественного класса
int a;
  public:
A(int x) { a = x; }
};
void B::f(A&) { cout<<x.a<<endl; }

int main() {
```

```
    A aObj(5);
    B bObj;
    bObj.f( aObj );
    return 0;
}
```

II. Перегрузка операций в C++.

В C++ можно переопределить большинство значков операций заданных для стандартных типов для собственных классов. Например, если A это класс, а a1 и a2 объекты этого класса, то компилятор не знает, как выполнить сложение двух его объектов, т.е. значение выражения a1 + a2 может быть вычислено, только если для класса будет задана функция или метод, с алгоритмом такого вычисления. Такая функция (или такой метод) должна иметь строго определенное имя, состоящее из зарезервированного в C++ слова operator и значка операции (через пробел или без пробела, за исключением операций new и delete, для которых пробел обязателен). В общем случае выражение a1 + a2 компилятор трактует либо как вызов метода для объекта a1

a1.operator+(a2),

либо как вызов внешней функции с двумя аргументами a1 и a2

operator+(a1,a2).

Общие правила перегрузки операций в C++ таковы:

1. Нельзя задать новый значок операции.
2. Нельзя изменить арность (число аргументов), приоритет и ассоциативность операции.
3. Нельзя переопределить операцию для стандартного типа.
4. Нельзя переопределять операции "запятая", sizeof, ::, ., .*, ?:, typeid, throw, dynamic_cast<>, static_cast<>, const_cast<>, reinterpret_cast<>.

Некоторые другие ограничения рассмотрены ниже.

III. Перегрузка операций с помощью методов класса.

Операции =, [], (), -> могут быть перегружены только методами класса. Если перегруженная операция реализована как метод, то ее единственный (для унарных операций) или первый/левый аргумент (для бинарных операций) доступен методу через скрытый аргумент this. Например,

```
class A {
    int a;
  public:
```

```
    A(int x) { a = x; }
    int getA() { return a; }
    void setA(int x) { a = x; }
    A operator+(A& right) { /* перегруженная операция для сложения объектов класса A */
        return A( a + right.a );
    }
};

int main() {
    A a1(1), a2(2), a3(3);
    a1.setA( 5 );
a2.setA( 10 );
a3 = a1 + a2;  // a3 = a1.operator+(a2);
    return 0;
}
```

IV. Перегрузка операций с помощью внешних функций.

Бывают такие виды операций, когда их перегрузка с помощью методов невозможна. Например, класс A рассмотренный выше позволяет попарно складывать свои объекты, но не позволяет добавить к целому числу объект класса. Вообще, если левый аргумент операции не является объектом нашего класса, такую операцию нельзя перегрузить с помощью метода нашего класса. Приходится использовать внешнюю функцию и, что дать ей прямой доступ к закрытым полям класса, такую функцию часто объявляют дружественной классу. Характерным примером, кроме приведенного выше, является также желание переопределить для класса операцию помещения в поток (или извлечения из потока). Левым аргументом такой операции должен быть поток, т.е. объект чужого класса ostream. Например,

```
#include <iostream>

using namespace std;

class A {
    friend ostream& operator<<(ostream&,A&); /* переопределенная операция помещения в поток */
    int a;
  public:
    A(int x) { a = x; }
```

```
};

ostream& operator<<(ostream& o, A& x) {
    return o<<x.a;
}

int main() {
    A aObj( 10 );
    cout<<aObj<<endl;  // помещение объекта в поток
    return 0;
}
```

4.2. Задание к лабораторной работе

1. Описать класс Test с защищенными числовым полем W и функцией Z, которая выводит сообщение "Это закрытая функция класса Test". Написать конструктор для инициализации объектов класса Test с одним параметром, принимающим по умолчанию значение 1. Объявить другом класса функцию fun, которая не возвращает значений и получает указатель на объект типа Test.

2. Описать на внешнем уровне функцию fun, которая выводит на экран значение параметра W и вызывает из класса Test функцию Z.

3. В функции main описать переменную класса Test (без явной инициализации) и применить к ней функцию fun.

4. Создать класс с перегруженной операцией присваивания.

5. Придумать и реализовать программу - пример использования двух классов A и B, в которой A друг B.

4.3. Контрольные вопросы

1. Что такое дружественная функция?
2. Можно ли объявить некоторую функцию дружественной вне описания класса? Почему?
3. Что такое дружественный класс?
4. В чем отличие операции присваивания от конструктора копирования?
5. Назначение друзей класса?
6. Какие операции можно реализовать только методами класса?
7. В каких случаях операцию можно перегрузить только с помощью внешней функции?

8. Можно ли операцию вставки в поток реализовать как функцию-член класса?

9. Как переопределить постфиксную и префиксную формы операции инкремента?

10.Для каких классов удобно переопределить операцию индексирования?

4.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

5. ЛАБОРАТОРНАЯ РАБОТА №5: СТАТИЧЕСКИЕ ПОЛЯ И МЕТОДЫ КЛАССА

5.1. Теория

I. Статические поля класса в C++.

Если в классе созданы поля, то каждый объект класса будет обладать собственными копиями всех полей, поэтому такие поля называют полями экземпляра класса. Можно объявлять такие поля, которые будут принадлежать классу в целом. Такие поля будут существовать в единственном экземпляре независимо от того, сколько объектов класса создано. Они будет существовать, даже если нет ни одного объекта класса. Чтобы объявить поле класса надо задать при его описании модификатор static. Такое поле будет храниться в сегменте данных программы, вместе с глобальными и статическими переменными. Кроме того C++ требует, чтобы эти поля явно были созданы вне описания класса. По умолчанию статические поля инициализируются нулевыми значениями. Например,

```
class A {
   public:
int a1;         /* описание поля экземпляра класса */
static int a2;     /* описание поля класса в целом */
static int a3;     /* описание поля класса в целом */
};
```

```
int A::a2; /* объявление статического поля, инициализация нулем по умолчанию */
int A::a3 = 100; /* объявление статического поля, явная инициализация */

int main() {
    A::a2 = -1; /* статическое поле существует, когда нет ни одного объекта класса */
    A x; /* объект класса */
    x.a2 = -10; /* обращение к статическому полю через объект */
    return 0;
}
```

II. Статические методы класса в C++.

Методы, объявленные в классе, могут быть вызваны только для какого-нибудь объекта этого класса ("привязаны" к объекту), поэтому их называют методами экземпляра класса. Адрес объекта, для которого вызван метод, передается последнему через скрытый параметр и доступен в теле метода как указатель с именем this. Если же при объявлении метода задан модификатор static, то такой метод не будет привязан к объекту класса, а станет методом класса в целом. Так как при вызове статического метода может не задаваться объект, у него не может быть скрытого аргумента – адреса объекта и, поэтому, в теле такого метода не определено имя this. С другой стороны, статический метод можно вызывать даже тогда, когда нет ни одного объекта класса. Например,

```
class A {
    int a1;
    static int a2;
  public:
    A(int x) { a1 = x; }
    static void stFun() {
x       a1 = 1; /* Ошибка. Нет this, поэтому нет доступа к нестатическим полям */
        a2 = 1; /* Есть доступ */
}
};
int A::a2;

int main() {
```

```
        A::stFun(); /* можно вызвать, хотя нет ни одного объекта
A */
        A x(10);
        x.stFun();   /* так тоже можно */
        return 0;
    }
```

III. Статические поля и методы в Java.

Статические поля и методы в Java имеют тот же смысл, что и в C++. Но в Java статические поля не только описываются, но и определяются внутри описания класса. Например,

```
public class A {
    public static int a = 20;
}
```

Кроме того, для инициализации статических полей в Java можно использовать статические блоки инициализации. Блоки инициализации – это блоки кода, которые могут содержать также локальные описания. Например,

```
public class A {
    public static int[] a;
    static {  /* статический блок инициализации */
        a = new int[20];
        for(int i=0; i<20; ++i) a[i] = i + 1;
    }
}
```

Блоков инициализации может быть несколько. Компилятор объединяет их в один блок в том порядке, в котором они заданы в описании класса. Этот блок выполняется при загрузке класса.

В Java нет внешних функций, поэтому точкой входа в программу должен быть статический метод (main), чтобы его можно было вызвать тогда, когда еще нет ни одного объекта.

```
public class A {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}
```

5.2. Задание к лабораторной работе

1. Создайте класс, который содержит счетчик созданных объектов. Напишите программу-тест, в которой проверяется, сколько объектов класса создано при входе в функцию main, после статического создания массива объектов, после динамического создания объекта, после удаления динамического объек-

та.

2. Создайте класс с закрытыми конструкторами и деструктором. Реализуйте методы для создания и уничтожения объектов класса. Напишите программу-тест.

3. Создайте класс, для которого возможно создание только одного объекта.

4. Создайте класс на Java со статическими полями и методами. Инициализируйте статические поля в статическом блоке инициализации.

5.3. Контрольные вопросы

1. В чем отличие статических и нестатических полей?
2. Где размещаются статические поля класса? Нестатические поля?
3. Как можно проинициализировать статическое поле в C++?
4. Как можно проинициализировать статическое поле в Java?
5. Когда выполняются статические блоки инициализации?
6. Для решения каких задач можно использовать статические поля?
7. Чем статический метод отличается от нестатического?
8. Можно ли в теле статического метода получить доступ к нестатическим полям класса?
9. Можно ли в теле статического метода вызвать нестатический метод класса?
10. Может ли точка входа в Java-приложение быть нестатическим методом? Почему?

5.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

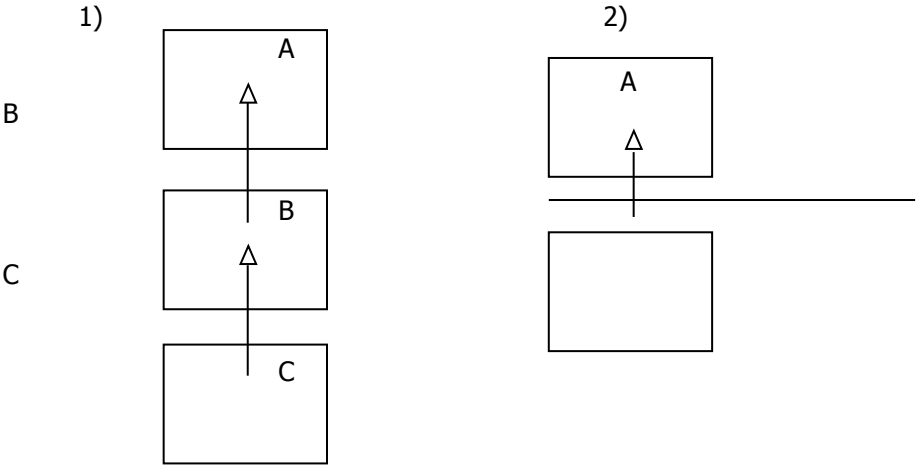
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

6. ЛАБОРАТОРНАЯ РАБОТА №1: НАСЛЕДОВАНИЕ В C++

6.1. Теория

I. Отношение расширения (наследования).

Между классами можно установить отношение, в котором некоторые классы будут выступать в качестве базовых для построения других, производных от них классов. При этом производные классы получают все состояния и все поведение базовых классов и могут добавить свои состояния и свое поведение. Поэтому отношение и называется отношением расширения. Связь между классами становится родственной, поэтому базовые классы называют также родительскими или классами предками, а производные наследниками, дочерними ил классами потомками. На UML-диаграмме классов отношение наследования задается стрелками, направленными от производных к базовым классам. Например,



На обеих диаграммах у класса C есть два базовых класса A и B. Но на схеме 1) класс B является прямым базовым классом для C, а класс A непрямым. На схеме 2) оба класса, и B и A прямые базовые классы для C. поэтому схема 1) задает так называемое единичное, а схема 2) множественное наследование.

II. Виды наследования в C++.

В C++ общий синтаксис объявления производного класса при единичном наследовании таков:

class *Имя_производного_класса* : [*вид_наследования*] *Имя_базового_класса* {
описания полей и методов производного класса
};

Если вид наследования не задан явно, то, в случае, если класс создавался с помощью ключевого слова class, по умолчанию задается закрытое наследование, а если с помощью ключевого слова struct, то открытое. Можно задать открытое (public), закрытое (private) или защищенное (protected) наследование. Вид наследования не влияет на доступ методов производного класса к полям и методам унаследованным от базового класса. Он определяет доступ к этим полям и методам из других классов или внешних функций. При открытом наследовании доступ определяется описанием базового класса. При закрытом наследовании все описания базового класса становятся закрытыми для тех классов и функций, которые используют объекты производного класса. При защищенном наследовании закрытая часть базового класса остается закрытой, а остальные становятся защищенными. Например,

```
class Base {
  private:    int privB;
  protected: int protB;
  public:  int pubB;
};

class Derived : protected Base {
  public:
   void test() {
   x   privB = 1;   // Ошибка. Нет доступа к закрытому полю
       protB = 2;   // Есть доступ, как у наследника
       pubB = 3;   // Есть доступ, как у всех
   }
};

class Derived2 : protected Derived {
  public:
   void test() {
   x   privB = 1;   // Ошибка. Нет доступа к закрытому полю
       protB = 2;   // Есть доступ, как у наследника
       pubB = 3;   // Есть доступ, как у всех
   }
};
```

```
int main() {
    Derived d;
x   d.privB = 1;   // Ошибка. Нет доступа к закрытому полю
x   d.protB = 2;   // Ошибка. Нет доступа т.к. не наследник
x   d.pubB = 3;   // Ошибка. Нет доступа т.к. не наследник
    return 0;
}
```

III. Множественное наследование.

При множественном наследовании у класса наследника есть не менее двух прямых предков. При объявлении наследования вид наследования задается индивидуально для каждого базового класса. Например,

```
class Base1 {};
class Base2 {};
class Derived : public Base1, protected Base2 {};
```

При создании объекта производного класса, прежде всего вызываются конструкторы базовых классов в том порядке, в котором они заданы в описании класса наследника, потом конструктор производного класса. При уничтожениии объекта производного класса деструкторы вызываются в обратном порядке. Чтобы организовать ромбовидное наследование необходимо использовать виртуальные базовые классы:

```
class A {};
class B : virtual public A {};
class D : virtual public A {};
class C : public B, public C {};
```

6.2. Задание к лабораторной работе

1. Создать базовый класс Base, в котором описать в разделе public поле i типа int, в разделе protected поле l типа long, в разделе private поле d типа double. Написать конструктор, инициализирующий поля i, l и d тремя задаваемыми значениями.

2. Создать класс Derived, производный от класса Base (наследование типа public), в котором в разделе private описано поле f типа float. В классе Derived создать конструктор без параметров и конструктор с четырьмя параметрами для инициализации всех полей объекта.

3. В функции main описать неинициализированный объект класса Derived и откомпилировать программу. Если есть пробле-

мы, то устранить их. Вывести размеры типов Base и Derived и объяснить результаты.

4. Описать инициализированный объект класса Derived. Продемонстрировать, инициализацию каких полей, унаследованных от класса Base, можно выполнять с помощью присваивания непосредственно в конструкторе класса Derived. Для исследования можно вносить необходимые изменения в конструкторы классов Base и Derived.

5. Перегрузить операцию вставки в поток для объектов класса Derived таким образом, чтобы выводились адреса и значения всех полей объекта. К каким полям, унаследованным от класса Base, нет доступа? Для снятия проблемы добавить в классе Base необходимые интерфейсные функции. Создав объект класса Derived, исследовать размещение полей в памяти. В отчете привести схематическую структуру объекта.

6. Описать класс Derived1, производный (public) от класса Derived и не имеющий новых полей. В классе описать конструктор со всеми необходимыми параметрами (сколько их нужно?). Класс имеет общедоступную функцию void foo(), которая модифицирует значения полей, унаследованных от базового класса (i++; l+=1;). Откомпилировать программу. Заменить тип наследования Derived от Base на private и вновь откомпилировать программу. Какая возникла проблема? Для ее решения использовать возможность восстановления уровня доступа к компонентам базового класса.

7. Вернуть для Derived тип наследования public. На глобальном уровне и в классах Base и Derived описать функции void ff(), которые сообщают о своей принадлежности к классу или глобальному уровню. В функции foo класса Derived1 добавить вызовы всех трех функций ff. В каких разделах классов Base и Derived нужно описать функции ff, чтобы они были доступны в Derived1? Проверить работу программы, вызвав функцию foo для какого-либо объекта класса Derived1.

8. Оставить в функции Derived::foo только один вызов в виде ff(); и проверить работу программы в следующих вариантах. Вначале функция ff определена в классах Derived, Base и на глобальном уровне. Затем ее описание убираем вначале из класса Derived, а затем из классов Derived и Base. Как в каждом случае это отражается на работе программы?

9. Класс Base1, имеет одно закрытое поле i целого типа. Первый конструктор не имеет параметров и обнуляет i. Второй имеет один параметр типа int, используемый для инициализации i произвольными значениями. Класс имеет две общедоступные ин-

терфейсные функции void put(int) и int get(void), которые позволяют изменить или прочесть значение i. Класс Base2, имеет одно закрытое поле - массив name из 20 элементов. Первый конструктор не имеет параметров и инициализирует поле name словом "Пусто". Второй имеет один параметр типа char*, используемый для инициализации name значениями символьных строк. Класс имеет две общедоступные интерфейсные функции void put(char*) и char* get(void), которые позволяют изменить или прочесть значение name. Класс Derived имеет одно закрытое поле ch типа char. Первый конструктор не имеет параметров и присваивает ch значение 'V' (от void - пустой). Второй конструктор имеет три параметра типов char, char* и int, используемые для инициализации соответственно полей ch, name и i. Класс имеет две общедоступные интерфейсные функции void put(char) и char get(void), которые позволяют изменить или прочесть значение ch. Кроме того, в нем объявляется как дружественная операция вставки в поток вывода, которая выводит на экран значения i, name и ch.

а). В функции main описать переменную типа Derived без инициализации и вывести ее значение с помощью перегруженной операции вставки в поток. Выяснить порядок вызова конструкторов.

б). Описать другую переменную класса Derived, инициализировав ее явно некоторыми значениями. Вывести значение этой переменной на экран и проанализировать порядок вызова конструкторов.

в). В конструкторе класса Derived с параметрами изменить порядок вызова конструкторов базовых классов. Проверить, как это отразилось на работе программы и почему.

г). Изменить порядок наследования базовых классов в описании класса Derived и проверить, как это отразилось на работе программы.

6.3. Контрольные вопросы

1. К каким полям, унаследованным от базового класса, нет доступа в производном классе?
2. Какие поля производного класса обязательно нужно инициализировать с помощью конструктора базового класса?
3. На что влияет вид наследования?
4. Как восстановить уровень доступа к компонентам базового класса для пользователей производного класса, если наследование закрытое?
5. Почему в языках программирования Java и

C# наследование может быть только открытым?

6. Как отличить единичное наследование от множественного?
7. Могут ли при множественном наследовании отличаться виды наследования для разных базовых классов?
8. Как организовать ромбовидное наследование?

6.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

7. ЛАБОРАТОРНАЯ РАБОТА №1: ОБРАБОТКА ИСКЛЮЧЕНИЙ

7.1. Теория

I. Исключительные ситуации.

Во время выполнения программы могут возникать различные нестандартные ситуации. Это могут быть ошибки, вызванные внешними или внутренними причинами, или ситуации, когда в некоторой точке программы нет достаточной информации для выбора правильного решения, но в другой ее точке эта информация есть. Такие ошибочные и неопределенные состояния называют исключительными. Стандартная схема обработки исключений предполагает, что программа, столкнувшись с исключительной ситуацией, генерирует исключение, при этом происходит передача управления в ту точку, где есть обработчик такой ситуации. Исключения различаются по своему типу. В C++ любой допустимый тип можно использовать как тип исключения. Соответственно и обработчиков прерываний, в общем случае, в программе несколько – каждый для своего типа исключения. Однако некоторые типы являются более "широкими" чем другие. Например, если тип одного исключения представлен базовым классом, а другого производным от базового, то обработчик исключения базового типа может обрабатывать и исклю- чения производного типа. Кро-

ме того, нетипизированный указатель является более широким, чем любой типизированный. Наконец, можно создать обработчик, способный обрабатывать исключения любого типа. Компилятор выбирает первый подходящий обработчик исключений, поэтому, если обработчиков исключения в одной точке программы указано несколько они должны задаваться в порядке от более специализированных (производных) к более широким (базовым).

II. Обработка исключений.

Код, в котором могут возникать исключительные ситуации, нужно размещать в блоках try, тогда будет сохраняться необходимая информация для правильной раскрутки стека при обработке исключений. Собственно же обработка производится в подходящем блоке catch. Например,

```
class DivideException {
    int a; // значение числителя
public:
    DivideException(int x) : a(x) {}
    int getA() { return a; }
};

int divide(int x, int y); // может выбрасывать исключение DivideException
int main() {
    int a, b, c;
    while (1 ) {
cout << "Задайте значения a, b: ";
        cin >> a >> b;
        try {
            c = f(a,b); /* вызываем в блоке try, так как может генерировать исключение */
            cout << a << '/' << b << '=' << c << endl;
            break;
} catch(DivideException) {
    cout << "Попробуйте еще раз." << endl;
}
}
return 0;
}
```

Хотя для реализации обработчика исключения достаточно задать только тип исключения, из точки генерации исключения

можно передавать и дополнительную информацию, характеризующую это исключение. Например, класс DivideException сохраняет информацию о значении числителя в ситуации, когда знаменатель был равен нулю. Эту информацию можно использовать в обработчике исключения, например

```
try {
    c = f(a,b);
    cout << a << '/' << b << '=' << c << endl;
    break;
} catch(DivideException de) {
cout << "Не могу разделить " << de.getA() << " на ноль."
<< endl;
cout << "Попробуйте еще раз." << endl;
}
```

III. Генерация (выбрасывание) исключения.

Для генерации исключения используется операция throw выражение. Тип выражения определяет тип исключения. Например,

```
int divide(int x, int y) {
if( !y ) throw DivideException(x);
return x/y;
}
```

Если исключение не удается обработать в одном блоке catch, в нем можно повторно сгенерировать это же исключение, тогда окончательная обработка должна проводиться в блоке catch того же типа внешнего блока try. Например,

```
void f1(int x) {
try {
    if( x < 0 ) throw 0; // исключение типа int
…
}
void f2(int x) {
try {
    f1(x);
    …
} catch( int ) {
    … // частичная обработка исключения
    throw; // повторная генерация того же исключения
    …
}
}
```

```
int main() {
int a;
cin >> a;
try {
    f2(a);
    ...
} catch( int ) {
    ... // окончательная обработка исключения
}
}
```

7.2. Задание к лабораторной работе

1. Создать класс Array, представляющий понятие массива фиксированного размера. Переопределить операцию индексирования так, чтобы при выходе индекса за допустимые границы генерировалось исключение.

2. Создать класс целых чисел IntegerRange с ограниченным диапазоном (по умолчанию от 0 до 100). Предусмотреть возможность задания интервала в конструкторе. Перегрузить операцию присваивания и арифметические операции. При выполнении любой из этих операций возможен выход за границы интервала, следовательно, при выходе должно выбрасываться исключение. Исключение должно быть представлено классом OutOfRangeException.

3. Создать класс положительных целых чисел PositiveInteger как класс наследник IntegerRange. Создать классы исключений LeftBoundException и RightBoundException, представляющие, соответственно, события выхода значения за левую и правую границы. Организуйте каскадный вызов исключений.

4. Используйте при создании класса PositiveInteger не наследование, а агрегирование.

7.3. Контрольные вопросы

1. Что такое исключительная ситуация?
2. Как различаются исключения в программе на C++?
3. Для чего нужны блоки try?
4. Важен ли порядок следования блоков catch?
5. Как передать в блок catch информацию из точки генерации исключения?
6. Как генерируются исключения в C++?

7. Как выглядит блок catch, способный обрабатывать исключения любого типа?

7.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.