ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ»

Кафедра прикладной математики
Т.В. Лоссиевская, П.В. Филонов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методическое пособие
для проведения лабораторных работ

для студентов 3 курса
специальности 231300
дневного обучения

МОСКВА — 2012

Рецензент д-р техн. наук, проф. В.Л. Кузнецов

Т.В. Лоссиевская, П.В. Филонов

Объектно-ориентированное: методическое пособие для проведения лабораторных работ. — М.:МГТУ ГА, 2012. — 29 с.

Данное пособие издаётся в соответствии с рабочей программой учебной дисциплины «Объектно-ориентированное программирование» по Учебному плану для студентов III курса, специальности 231300 дневного обучения.

Рассмотрено и одобрено на заседании кафедры 18.04.12 г. и методического совета 18.04.12 г.

	Подписано в печать	
Печать офсетная	Формат 60x84/16	1,23 уч.-изд. л.
1,38 усл. печ. л.	Заказ №	Тираж экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д. 20
Редакционно-издательский отдел
125493 Москва, ул. Пулковская, д. 6а

Лабораторная работа №1
Пользовательские типы данных

Краткая теоретическая справка

Основным элементом объектно-ориентированного программирования в языке C++ является понятие класса, с помощью которого пользователь может создавать новые типы данных. Широкие возможности по перегрузке стандартных операторов позволяют использовать для работы с объектами пользовательских классов общепринятую нотацию. Так, например, возможность перегрузки всех арифметических операций позволяет определить класс для работы с комплексными числами и использовать для записи выражений привычные операции.

Ниже приведён пример реализации и использования класса `Complex`.

Листинг 1: Файл complex.hpp

```
#ifndef COMPLEX_HPP
#define COMPLEX_HPP

class Complex{
  // Закрытая часть
  double re, im; // Свойства
public:
  // Конструкторы
  Complex(); // Конструктор по умолчанию
  Complex(double _re); // Конструктор преобразования
  Complex(double _re, double _im);
  Complex(const Complex &Z);// Конструктор копирования

  // Константные методы
  double real()  const;
  double imag()  const;
  double modulo() const;
  double angle()  const;

  // Переопределение оператора присваивания
  const Complex& operator=(const Complex &rhs);

  friend Complex operator+(const Complex &lhs, const
    Complex &rhs);
  friend Complex operator-(const Complex &lhs, const
    Complex &rhs);
  friend Complex operator*(const Complex &lhs, const
```

```
    Complex &rhs);
  friend Complex operator/(const Complex &lhs, const
    Complex &rhs);

  friend bool operator==(const Complex &lhs, const Complex
     &rhs);
};

bool operator!=(const Complex &lhs, const Complex &rhs);
void print(const Complex &z);

#endif
```

Листинг 2: Файл complex.cpp

```
#include "complex.hpp"

#include <stdio.h>
#include <math.h>  // atan2, sqrt

Complex::Complex(){
  re = 0; im = 0;
}
Complex::Complex(double _re){ re = _re; im = 0; } //
   Конструктор преобразования
Complex::Complex(double _re, double _im){ re = _re; im =
   _im; }
Complex::Complex(const Complex &Z){ re = Z.re; im = Z.im;
   } // Конструктор копирования

// Константные методы
double Complex::real()   const { return re; }
double Complex::imag()   const { return im; }
double Complex::modulo() const { return sqrt(re*re + im*im
   ); }
double Complex::angle()  const { return atan2(im, re); }

// Переопределение оператора присваивания
const Complex& Complex::operator=(const Complex &rhs){
  if(this != &rhs){
    this->re = rhs.re;
    this->im = rhs.im;
```

```
  }
  return *this;
}



// Переопределение арифметических операций
Complex operator+(const Complex &lhs, const Complex &rhs){
  return Complex(lhs.re + rhs.re, lhs.im + rhs.im);
}

Complex operator-(const Complex &lhs, const Complex &rhs){
  return Complex(lhs.real() - rhs.real(), lhs.imag() - rhs
     .imag());
}
Complex operator*(const Complex &lhs, const Complex &rhs){
  return Complex( lhs.real()*rhs.real() - lhs.imag()*rhs.
     imag(),
           lhs.imag()*rhs.real() + lhs.real()*rhs.imag());
}
Complex operator/(const Complex &lhs, const Complex &rhs){
  double rhs_mod_sqr = rhs.modulo()*rhs.modulo();
  return Complex((lhs.real()*rhs.real() + lhs.imag()*rhs.
     imag())/rhs_mod_sqr,
            (lhs.imag()*rhs.real() - lhs.real()*rhs.imag())
               /rhs_mod_sqr);
}

// Переопределение операций сравнения
bool operator==(const Complex &lhs, const Complex &rhs){
  return (lhs.real() == rhs.real()) && (lhs.imag() == rhs.
     imag());
}
bool operator!=(const Complex &lhs, const Complex &rhs){
  return !(lhs == rhs);
}

// Печать комплексного числа в виде (x,y)
void print(const Complex &z){
  printf("(%g,%g)\n", z.real(), z.imag());
}
```

В качестве примера, иллюстрирующего применение класса `Complex`, приведем расчет значения функции $e^z, \quad z \in C$.

Листинг 3: Файл exp.cpp

```
/*
 * Программа для расчёта экспоненты от комплексного
   аргумента
 * с помощью ряда Тейлора
 */

#include "complex.hpp"
#include <float.h> // DBL_EPSILON
#include <stdio.h>

Complex exp(const Complex &z);

int main(){
  Complex z(2.5,-0.3);

  printf("z = %g%+gi\n", z.real(), z.imag());
  Complex w = exp(z);
  printf("exp(z) = %g%+gi\n", w.real(), w.imag());

  return 0;
}

Complex exp(const Complex &z){
  Complex Sum(1);

  Complex E = z;
  int i = 1;
  while( E.modulo() > DBL_EPSILON){
    Sum = Sum + E;
    E = E*z/++i;
  }
  return Sum;
}
```

Задание на лабораторную работу

Для решения задач численными методами часто возникает необходимость работать с векторами различной размерности. Возможность описывать операции с помощью привычной математической нотации позволит значительно

сократить объём исходного кода и улучшить его читаемость. Последнее будет способствовать удобной и быстрой отладке программ.

Имея под рукой готовый класс для работы с векторами, Вы потратите значительно меньше времени для решения соответствующих задач, поскольку это позволит сосредоточиться на самой математической проблеме, а не на синтаксических особенностях языка программирования.

Результат данной лабораторной работы может быть использован Вами в дальнейшей учёбе (в том числе, в других дисциплинах).

Задание: реализуйте класс для представления трёхмерного вектора. В набор открытых (`public`) методов необходимо включить:

- конструктор по умолчанию (создать нулевой вектор);
- конструктор с параметрами (задание координат вектора);
- конструктор копирования;
- оператор присваивания (не забудьте, что оператор присваивания должен возвращать элемент типа вектора);
- оператор индексирования (`operator[](int i)`);
- сумму двух векторов;
- разность двух векторов;
- умножение вектора на скаляр;
- проверку векторов на равенство.

Указания:

- для хранения координат вектора используйте число с плавающей точкой двойной точности (тип `double`);
- все методы, не изменяющие состояние объекта, следует объявлять константными;
- спользуйтт передачу параметра по ссылке, чтобы избежать накладных расходов, связанных с вызовом конструкторов и деструкторов;
- оператор индексирования должен возвращать ссылку на элемент, чтобы операция присваивания (`v[i] = 0.0`) работала корректно;
- ограничьте доступ к внутреннему представлению вектора, объявив соответствующие переменные-члены класса закрытими (`private`);
- умножение на скаляр слева можно ввести с помощью перегрузки глобального оператора умножения. Для этого вне тела класса можно ввести следующую перегрузку: `vector operator*(double a, const vector& v)`.

Лабораторная работа №2
Динамическая память

Краткая теоретическая справка

Различные структуры данных (стек, очередь, список, дерево, хэш-таблица)

являются важными инструментом при разработке программ. Практически в любой программе возникает необходимость в организации данных с помощью различных структур. В связи с этим наличие обобщённой реализации часто используемых структур может значительно ускорить время разработки и повторное использование кода (данные преимущества могут быть существенны в том случае, когда необходимо организовывать одинаковые структуры данных, которые работают с разными типами).

В данной работе рассматриваются такие структуры как очередь (queue). В качестве задания на лабораторную работу студентам предлагается реализовать очередь с фиктивным элементом на основе циклического односвязного списка.

Очередь обеспечивает дисциплину обслуживания типа FIFO[1] (Первым пришёл — первым вышел). В стандартный интерфейс для работы с очередью входят 2 метода:

- `push` — добавить элемент в конец очереди;
- `pop` — удалить элемент из начала очереди.

В нашей реализации будут также присутствовать 2 дополнительных метода:

- `front` — возвращает значение элемента в начале очереди, но не удаляет его;
- `empty` — проверка очереди на пустоту.

Основная идея реализации очереди с фиктивным элементом заключается в том, чтобы используя только один указатель иметь возможность как добавлять элементы в конец очереди, так и удалять их из начала. Для этого используется циклический односвязный список с фиктивным элементом. Последний выполняет роль связующего звена между началом и концом очереди. Фактически будет храниться только указатель на конец очереди, а указатель на начало будет доступен через фиктивный элемент.

Пустая очередь содержит только фиктивный элемент, одно из полей которого указывает на него самого (рис. 1).

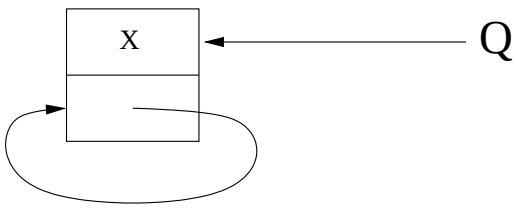

Рис. 1: Пустая очередь, состоящая только из фиктивного элемента

При добавлении первого элемента 6 в конец очередь он же становиться и

[1]FIFO — First In First Out

первым в начале очереди (рис. 2).

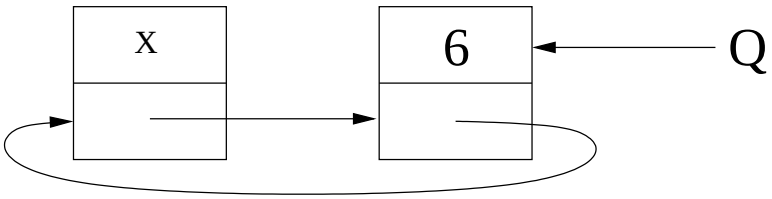

Рис. 2: Очередь из одного элемента

При этом последний элемент хранит указатель на фиктивный, который, в свою очередь, хранит указатель на начало очереди.

При последовательном добавлении элементов 9 и 11 мы получаем очередь, изображённую на рис. 3.

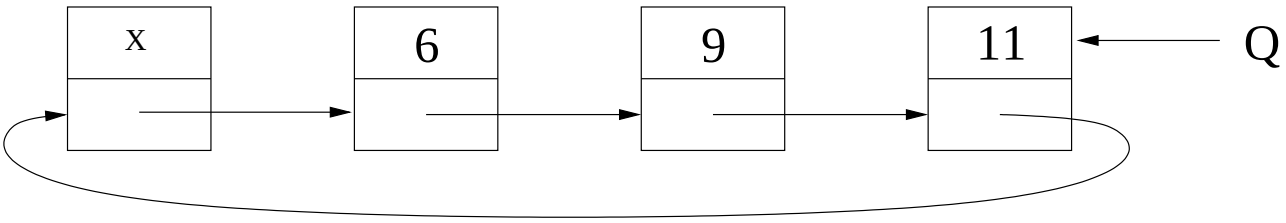

Рис. 3: Очередь из 3-х элементов: в начале 6, в конце 11

Для удаления из очереди первого элемента можно воспользоваться следующим приёмом. Поскольку значение, которое хранит фиктивный элемент не важно, то вместо удаления начала очереди можно удалить фиктивный элемент. При этом начало очереди становится новым фиктивный элементом (рис. 4).

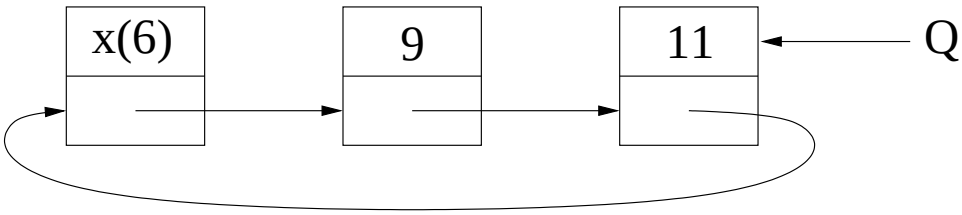

Рис. 4: Очередь после удаления элемента

Задание на лабораторную работу

Напишите класс `Queue`, реализующий очередь с фиктивным элементом. Определеите основные методы, описанные выше, для работы с данной структурой данных.

Указания: используйте операторы `new` и `delete` для выделения и освобождения динамической памяти. В деструкторе следует реализовать полное

удаление всех элементов очереди и очистку памяти. При попытке чтения из пустой очереди следует генерировать исключение.

Лабораторная работа №3
Echo сервер

Краткая теоретическая справка

В данной работе рассмотрена задача разработки классов для работы с прослушивающим и клиентским сокетами.

Указания к выполнению: в данной работе предполагается использование интерфейса BSD sockets для организации межсетевого взаимодействия программ. Более полное описание разработки сетевых приложений можно найти в. Рекомендуется использовать протоколы TCP и IPv4.

Рассмотрим реализацию такой задачи на языке C в качестве примера использования различных системных вызовов для работы с сокетами.

Листинг 4: Реализация echo сервера на языке C

```
/*
* Пример простого серверного приложения,
* работающего как echo сервер
*/

#include <stdio.h>        // io stuff
#include <stdlib.h>       // atoi, exit
#include <string.h>       // bzero
#include <unistd.h>       // read, write, close
#include <sys/types.h>    // socket, struct sockaddr_in,
    socklen_t,
                          // bind, listen, accept, shutdown
#include <sys/socket.h>   // -"-
#include <netinet/in.h>   // inet_ntoa
#include <arpa/inet.h>    // inet_ntoa

#define BUFF_SIZE 256     // Размер памяти для приёма
    сообщений
#define DEFAULT_PORT 1234 // Номер порта по умолчанию
#define CLIENT_QUEUE_LEN 5 // Количество клиентов в
    очереди соединения

void error(const char *msg); // Вывод ошибок и выход

int main(int argc, char *argv[]){
    int sockfd, csockfd; // Дескрипторы сокетов
```

```
int port;
socklen_t clilen; // размер адреса клиента

struct sockaddr_in serv_addr, cli_addr; // адреса
   сервера и клиента

char buffer[BUFF_SIZE];

// Создаём сокет
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd < 0) error("ERROR opening socket");

// Заполняем serv_addr нулями
bzero((char*)&serv_addr, sizeof(serv_addr));

// Порт модно задавать в качестве параметра
if(argc < 2){
    port = DEFAULT_PORT;
} else {
    port = atoi(argv[1]);
    if(port <= 0) {
        fprintf(stderr, "Wrong port number");
        exit(1);
    }
}

// Заполняем поля адреса сервера
serv_addr.sin_family = AF_INET; // Тип адреса (IPv4)
serv_addr.sin_port = htons(port); // номер порта (в
   сетевом порядке байтов)
serv_addr.sin_addr.s_addr = INADDR_ANY; // Адрес
   прослушиваемого интерфейса
                                        // INADDR_ANY
                                           - любой

// Назначаем сокету адрес
if((bind(sockfd, (struct sockaddr*) &serv_addr, sizeof
   (serv_addr))) < 0){
     error("ERROR binding address");
}
```

```
    // Устанавливаем длину очереди клиентов
    listen(sockfd, CLIENT_QUEUE_LEN);

    // Ожидаем соединения
    clilen = sizeof(cli_addr);
    csockfd = accept(sockfd, (struct sockaddr*)&cli_addr,
      &clilen);
    if(csockfd < 0) error("ERROR on accept");

    // Выводим адрес подключившегося клиента
    printf("%s:%d connected\n", inet_ntoa(cli_addr.
      sin_addr), \
                                ntohs(cli_addr.sin_port));


    int n;
    // Читаем не больше BUFF_SIZE - 1 байт
    n = read(csockfd, buffer, BUFF_SIZE - 1);
    if(n < 0) error("ERROR reading socket");
    buffer[n] = '\0';
    printf("Get msg: %s\n", buffer);

    // Отправляем echo
    n = write(csockfd, buffer, n);
    if( n < 0) error("ERROR writing to socet");

    // Закрываем сокеты
    close(csockfd);
    close(sockfd);

    return 0;
}

void error(const char *msg){
    perror(msg);
    exit(1);
}
```

Приведённая выше реализация позволяет установить входящее соединение от клиента, принять от клиента строку с данными длинной не больше чем `BUFF_SIZE`, выводит принятое сообщение на экран и отправляет обратно клиенту (эхо). При запуске можно указывать номер порта для прослушива-

ния.

```
echo <portnum>
```

По умолчанию используется номер порта `DEFAULT_PORT` (1234).

В качестве клиента можно использовать программу `telnet`.

```
telnet localhost 1234
```

В приведённом примере сервер заканчивает работу после приёма одного сообщения. При выполнении лабораторной работы следует использовать реализацию, которая ожидает соединения и производит обработку в бесконечном цикле. Также рекомендуется использовать системный вызов `fork` для одновременной обработки нескольких соединений.

При написании серверного приложения можно чётко выделить две категории сокетов.

- прослушивающий сокет (`sockfd`), который используется для установления соединений с клиентами.
- клиентский сокет (`csockfd`), который используется для обмена данными с клиентом.

При таком чётком разделении логично использовать ООП для представления различных категорий сокетов.

В работе необходимо разработать 2 класса.

- SSocket — прослушивающий (серверный) сокет.
- CSocket — клиентский сокет.

Ниже приведён листинг программы, в которой указаны открытая часть интерфейсов (методов) данных классов.

Листинг 5: Пример использования классов `SSocket` и `CSocket`

```
#include <stdio.h>

#include "SSocket.hpp" // Прослушивающий сокет
#include "CSocket.hpp" // Клиентский сокет

#define BUFF_SIZE 256    // Размер памяти для приёмы
    сообщений
#define DEFAULT_PORT 1234 // Номер порта по умолчанию
#define CLIENT_QUEUE_LEN 5 // Количество клиентов в
    очереди соединения

int main(int argc, char *argv[]){
    SSocket echo;
```

```
    int port;
    if(argc < 2){
        port = DEFAULT_PORT;
    } else {
        port = atoi(argv[1]);
        if(port <= 0) {
            fprintf(stderr, "Wrong port number");
            exit(1);
        }
    }

    // Создаём сокет и привязываем его к адресу
    echo.open(port);

    // Устанавливаем длину очереди клиентов
    echo.listen(CLIENT_QUEUE_LEN);

    // Ожидаем входящего соединения
    CSocket client = echo.accept();

    // Смотрим кто к нам подсоединился
    printf("%s:%d connected\n", client.addr(), client.port
       ());

    // Читаем сообщение от клиента
    int n = client.read(buff, BUF_SIZE - 1);
    if(n < 0) error("ERROR reading socket");
    buffer[n] = '\0';
    printf("Get msg: %s\n", buffer);

    // Отправляем echo
    n = client.write(buffer, n);
    if( n < 0) error("ERROR writing to socet");

    // Закрываем сокеты
    client.close();
    echo.close();

    return 0;
}
```

Большая часть низкоуровневой работы с системными вызовами в данном примере инкапсулирована внутри классов `SSocket` и `CSocket`, что приводит к сокращению размера основной функции. Низкоуровневые дескрипторы сокетов инкапсилурированы в указанные классы и вся работа программы построена на отправке объектам `echo` и `client` соответствующих сообщений, что хорошо согласуется с идеологией ООП.

Задание на лабораторную работу

Реализация классов `SSocket` и `CSocket` составляет основную задачу данной работы.

Приведём описание интерфейса классов.

Листинг 6: Описание `SScoket`

```
class SSocket{
public:
    SSocket();
    ~SSocket();

    int open(short port);
    int listen(int backlog);
    CSocket accept();
    int close();

private:
    // ...
};
```

Листинг 7: Описание `CScoket`

```
class CSocket{
public:
    CSocket(int sockfd, const struct sockaddr_in *cli_addr
        = NULL);
    CSocket(const CSocket &cli_sock);
    const CSocket& operator=(const CSocket &rhs);
    ~CSocket();

    int read(char *buff, int max_len);
    int write(const char *buff, int len);

    char* addr() const;
    short port() const;
```

```
    int close();

private:
    // ...
};
```

Лабораторная работа №4
Наследование и полиморфизм

Краткая теоретическая справка

Цель работы: используя классы `SSocket` и `CSocket` из второй работы разработать следующие классы:

- `Server` — предоставляет интерфейс для запуска многопользовательского сервера и обработки запросов клиентов.
- `Client` — используется на стороне клиента для установления соединения с сервером и обмена сообщениями.

Указания к выполнению: классы `SSocket` и `CSocket` позволяют абстрагироваться от уровня дескрипторов сокетов и системных вызовов для работы с ними. Решение задачи, поставленной в данной работе, позволит пользователю подняться на ещё больший уровень абстракции и оперировать такими понятиями как клиент и сервер, не вникая в их внутреннее устройство.

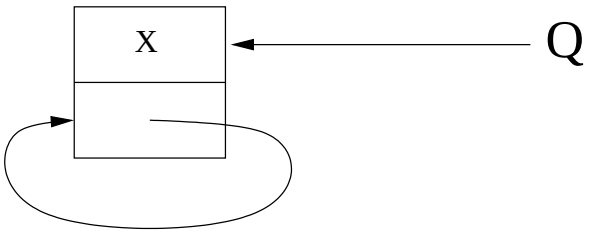

Рис. 5: Уровни абстракции

Описанные уровни абстракции схематично отображены на рис. 5. Отметим, что при реализации класса `Client` будет необходимо использовать отдельный класс `Socket`, основным отличием которого от `CSocket` будет наличие метода `connect`. Для преобразования доменного имени в IP адрес можно использовать функцию `gethostbyname`.

Ниже приведён пример использования класса `Client` для создания приложения, которое по сетевому имени возвращает заглавную страницу сайта, расположенного по данному адресу.

Листинг 8: Простой `HTTP` клиент

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#include "Client.hpp"

#define BUFF_SIZE 1024
#define HTTP_PORT 80

int main(int argc, char *argv[]){
    // Проверяем количество аргументов комманндной строки
    if(argc < 2){
        fprintf(stderr, "Usage: %s hostname <port>\n",
           argv[0]);
        exit(-1);
    }

    int port = HTTP_PORT;
    if(argc > 2){
        port = atoi(argv[2]);
    }


    Client cli;
    // Пробуем установить соединение
    if(cli.connect(argv[1], port)){
        perror("connect ERROR");
        exit(-1);
    }

    // Запросить главную страницу сайта
    char *msg = "GET / HTTP/1.0\r\n\r\n";

    if(0 > cli.write(msg, strlen(msg))){
        perror("write ERROR");
        exit(-1);
    }

    int n;
    char buff[BUFF_SIZE];
    while((n = cli.read(buff, BUFF_SIZE - 1)) > 0){
        buff[n] = '\0';
        printf("%s\n", buff);
```

```
    }

    return 0;
}
```

Поскольку сетевые сервисы не ограничиваются только Echo, то необходимо предусмотреть в классе `Server` возможность по работе с различными протоколами. Для этого имеет смысл всю логику общения с сетевым клиентом вынести в отдельную функцию, которая будет использоваться как функция обратного вызова. Т.е. после установления соединения и создания дочернего процесса класс `Server` должен передать управление данной функции. Регистрацию подобной функции можно сделать либо в конструкторе `Server` либо вынести в отдельный метод. Сама функция должна принимать ссылку на объект класса `CSocket` и возвращать код завершения обработки. Для хранения указателя на функцию обработчик удобно выделить отдельный тип:

```
typedef int (*f)(CSocket &client) Handler
class Server{
private:
//  ...
    Handler handler;
public:
    Server(int (*f)(Csocket &client)){
        handler = f;
    }
// ...
```

При реализации класса `Server` следует использовать классы `SSocket` и `Client`.

Листинг 9: Пример работы с классом `Server`

```
#include <stdio.h>
#include <stdlib.h>
#include "Server.hpp"
#include "CSocket.hpp"

#define DEFAULT_ADDR "0.0.0.0"
#define DEFAULT_PORT 1234
#define BUFF_SIZE 1024

int echo_handler(CSocket &client);
int time_handler(CSocket &client);

int main(){
```

```
    // Создать сервер и назначить обработчик клиентов
    Server serv(echo_handler);
//  Server serv(time_handler);

    // Запуск сервера
    if(serv.start(DEFAULT_ADDR, DEFAULT_PORT)){
        perror("start ERROR");
        exit(-1);
    }

    return 0;
}
// Echo обработчик
int echo_handler(CSocket &client){
    char buff[BUFF_SIZE];
    int n;
    while(0 < (n = client.read(buff, BUFF_SIZE - 1))){
        if(0 < client.write(buff, n)){
            client.close();
            return -1;
        }
    }
    client.close();
    return (n < 0)? -1 : 0;
}

// Обработчик сервера времени
int time_handler(CSocket &client){
    char buff[BUFF_SIZE];
    time_t tikcs = time(NULL);
    snprintf(buff, BUFF_SIZE, "Current time:%s\r\n", ctime
      (&ticks));
    int n = client.write(buff, strlen(buff));
    client.close();
    return (n < 0)? -1: 0;
}
```

Задание на лабораторную работу

Для проверки разработанных классов напишите следующие программы:

- Echo-сервер;
- Time-сервер;

- HTTP-клиент;
- telnet.

Лабораторная работа №5
Динамический полиморфизм

Краткая теоретическая справка

Задание на лабораторную работу

Реализуйте следуюший сценарий компьютерной игры. В сражении принимают участие две армии, включающие в свой состав различные типы боевых единиц (солдаты, БТР и танки). Для представления боевых единиц в программе следует создать иерархию классов, представленную на рис. 6. В качестве базового выступает класс `Unit` со следующим интерфейсом.

- `bool shoot(const Unit* target)` — выстрел в объект `target`
- `bool hit(int damage)` — попадание в объект с нанесением повреждений в размере `damage` единиц.
- `int health() const` — уровень «здоровья» юнита.

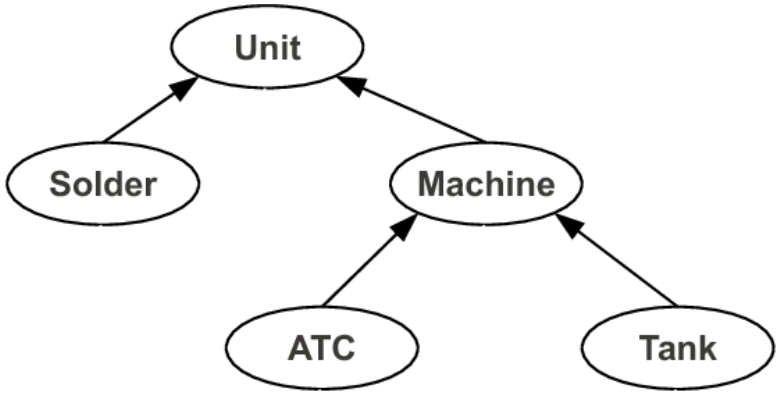

Рис. 6: Иерархия наследования классов, для представления боевых единиц

Начальный уровень жизни и величину повреждений, наносимых различными боевыми единицами предлагается выбрать самостоятельно. В качестве рекомендации следует сказать, что использование случайных величин для определения величины повреждения может значительно повысить gameplay («играбельность»).

Противоборствующие армии следует реализовать как два динамических массива, хранящие указатели на объекты класса `Unit`, что позволит за счёт использования виртуальных функций организовать выбор вызываемых методов на этапе выполнения программы (динамический полиморфизм).

Ход игры строиться следующим образом. Первый игрок делает ход за каждую боевую единицу своей армии. Для каждой боевой единицы выбирается цель на этот ход, после чего производится выстрел и подсчитывается

повреждение. Если уровень «здоровья» боевой единицы опустился до нуля, то её следует исключить из массива. Игра ведётся до полного уничтожения одной из армий.

Лабораторная работа №6
Обобщённые алгоритмы

Краткая теоретическая справка

Основное положение парадигмы *обобщённого программирования* можно кратко сформулировать в следующем виде.

> *Пишите алгоритмы и структуры данных так, чтобы они не зависели от типа обрабатываемых данных.*

Подобный подход позволяет использовать один и тот же код вновь и вновь, причём для различных типов данных.

Не в каждом языке программирование присутствуют необходимые синтаксические инструменты для описания алгоритмов в обобщённом виде. В случае отсутствия в языке программирования последних приходится прибегать к различным приёмам. В данной работе мы рассмотрим пример обобщённого алгоритма поиска элемента в одномерном массиве на языке C++.

Программирование на шаблонах в C++

Основными инструментами языка C++ для обобщённого программирования являются перегрузка функций и методов, а так же механизм шаблонов (templates) .

Рассмотрим применение шаблонов на следующем примере: написать алгоритм поиска элемента в одномерном массиве. Поскольку в задании ничего не говорит о типе элементов массива, то решение будем искать в обобщённом виде.

Сначала запишем и обсудим сигнатуру функции для поиска в массиве.

```
template<class T>
T* find(T *begin, T *end, T val);
```

Ключевое слово `template` открывает определение шаблона. В угловых скобках указаны параметры шаблона. `class T` вводит неопределённый тип `T`, который можно использовать для определения параметров, возвращаемого значения и локальных переменных.

Далее поясним, что означают параметры функции и возвращаемое значение. Для передачи массива в функцию можно использовать несколько способов, например, можно передать указатель на первый элемент массива и количество элементов в нём. В данном примере используется другой подход, который схематично изображён на рис. . В качестве параметров передаются

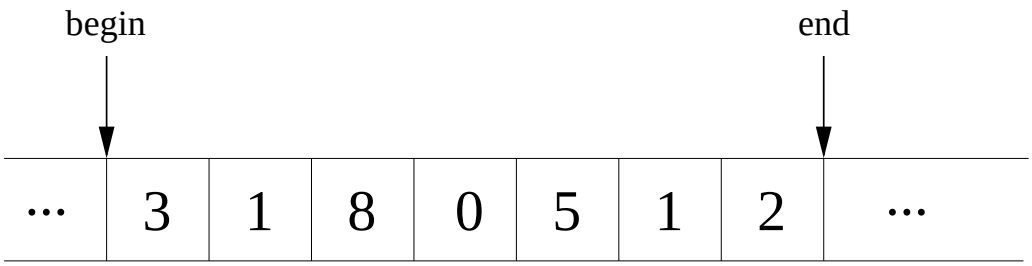

Рис. 7: Расположение указателей `begin` и `end`

указатели на первый элемент (`begin`) и на элемент, следующий за последним (`end`). Вычислить количество элементов в массиве с помощью арифметики указателей (`n = end - begin`). В качестве третьего параметра будем передавать искомое значение. Как видно из листинга функция возвращает указатель на тип `T`. Если искомый элемент будет найден в массиве, то будем возвращать указатель на него, в противном случае вернём указатель на конец (`end`). На основе этих соглашений опишем само тело шаблона функции `find`.

```
1 template<class T>
2 T* find(T *begin, T *end, T val){
3   while(begin != end){
4     if(*begin == val) break;
5     ++begin;
6   }
7   return begin;
8 }
```

Поясним смысл приведённого кода. Пока не дошли до конца массива (строка 3), проверяем не нашли ли мы искомый элемент (строка 4), если да, то выходим из цикла не доходя до конца. В завершении возвращаем указатель (строка 7) на найденный элемент (если выход из цикла был осуществлён по `break`) или на конец массива (если цикл завершился по условию).

Применение данного алгоритма продемонстрируем на примере массива целых чисел.

```
int main(){
  int a[] = {2,8,2,6,8,1,0};
  const int n = sizeof(a)/sizeof(a[0]) // num of ints in a

  int val;
  int *f = find(a, a + n, val);
  if(f == a + n){
    printf("%d not found in array\n", val);
  } else {
```

```
        printf("%d found in array on %d position\n", val, f - a
            );
    }
    return 0;
}
```

Для такого кода компилятор автоматически сгенерирует код функции для поиска в массиве целых чисел по описанному шаблону. Этот процесс называется *инстанцированием* шаблона. Если в программе шаблон функции `find` ,будет использоваться как для поиска по массиву целых чисел, так и по массиву вещественных чисел, то компилятор сгенерирует по шаблону две различные версии `find`.

Задание на лабораторную работу

В ходе первой работы студентам предлагается реализовать набор алгоритмов по работе с массивами в терминах обобщенного программирования на языке C++. Задание включает набор алгоритмов, которые необходимо реализовать на языке C++ с использованием шаблонов.

Ниже приведены алгоритмы, которые необходимо реализовать и сигнатуры функций и шаблонов.

Поиск минимального элемента в массиве.

```
template<class T>
T* min(T *begin, T *end);
```

Вычисление суммы элементов массива.

```
template<class T>
T sum(T *begin, T *end);
```

Подсчёт количества элементов равных `val`.

```
template<class T>
int count(T *begin, T *end, T val);
```

Сортировка массива.

```
template<class T>
void sort(T *begin, T *end);
```

Бинарный поиск элемента `*val`.

```
template<class T>
T* binary_search(T *begin, T *end, T *val);
```

Решение следует оформить в виде двух файлов с исходным кодом. В одном написать реализацию шаблонов функций на языке C++, а во втором нешаблонную версию на языка C. Для проверки рекомендуется использовать массивы со следующими встроенными типами элементов: `int, double, float, char` и с пользовательским типом `vector` в алгоритмах `sum, count`.

Контрольные вопросы

1. Что такое обобщённое программирование (ОП)?
2. Какие преимущества даёт ОП?
3. Какие средства языка C++ позволяют писать код в стиле ОП?
4. Как работает механизм шаблонов?
5. Приведите примеры алгоритмов и структур данных, которые не зависят от типа обрабатываемых величин?
6. Можно ли использовать обобщённые функции для работы с пользовательскими типами? Приведите примеры.

Лабораторная работа №7
Обобщённые структуры данных

Краткая теоретическая справка

Использование шаблонов классов позволяет определить средствами языка C++ различные структуры данных, реализация которых не зависит от типа хранимых величин. В качестве примера подобной реализации рассмотрим класс `Array`, представляющий собой статический массив, размер которого задаётся на этапе компиляции.

Ниже приведён пример реализации, разбитый на 3 части:

- файл `array.hpp` — описание класса;
- файл `array.cpp` — реализация методов класса;
- файл `main.cpp` — пример использования.

Листинг 10: array.hpp

```
#ifndef ARRAY_HPP
#define ARRAY_HPP

class OutOfRange {}; // Для исключения типа выход за
    границы массива

template<class T, size_t N>
class Array {
public:
    Array();
    Array(const Array&);

    Array& operator=(const Array&);
    bool operator==(const Array&) const;
    bool operator!=(const Array&) const;
    // Оператор индексирования
```

```
    T& operator[](size_t i);
    const T& operator[](size_t i) const;
    // Метод для индексирвоания с проверкой выхода за
       границы массива)
    T& at(size_t i);
    const T& at(size_t i) const;
    // Возвращает размер массива
    size_t size() const;
private:
    T data[N];
};
#include "array.cpp"
#endif
```

Листинг 11: array.cpp

```
#include "array.hpp"

template<class T, size_t N>
Array<T, N>::Array() {
    for(size_t i = 0; i < N; ++i)
        data[i] = T();
}
template<class T, size_t N>
Array<T, N>::Array(const Array& A) {
    for(size_t i = 0; i < N; ++i)
        data[i] = A.data[i];
}
template<class T, size_t N>
Array<T, N>& Array<T, N>::operator=(const Array& rhs) {
    for(size_t i = 0; i < N; ++i)
        data[i] = rhs.data[i];
}
template<class T, size_t N>
bool Array<T, N>::operator==(const Array& rhs) const {
    for(size_t i = 0; i < N; ++i)
        if(data[i] != rhs.data[i])
            return false;
    return true;
}
template<class T, size_t N>
bool Array<T, N>::operator!=(const Array& rhs) const {
```

```
    return !(*this == rhs);
}
template<class T, size_t N>
T& Array<T, N>::operator[](size_t i) {
    return data[i];
}
template<class T, size_t N>
const T& Array<T, N>::operator[](size_t i) const {
    return data[i];
}
template<class T, size_t N>
T& Array<T, N>::at(size_t i) {
    if(i >= N) throw OutOfRange();
    return data[i];
}
template<class T, size_t N>
const T& Array<T, N>::at(size_t i) const {
    if(i >= N) throw OutOfRange();
    return data[i];
}
template<class T, size_t N>
size_t Array<T, N>::size() const {
    return N;
}
```

Листинг 12: main.cpp

```
#include <stdio.h>
#include "array.hpp"

template<class T, size_t N>
void print(const Array<T, N>& A) {
    for(size_t i = 0; i < A.size(); ++i)
        printf("%d ", A[i]);
    printf("\n");
}


int main() {
    Array<int, 10> A;
    for(size_t i = 0; i < A.size(); ++i)
        A[i] = i;
```

```
    print(A);

    try {
        A.at(10);
    } catch(OutOfRange& e) {
        printf("Out of Range error\n");
    }


    return 0;
}
```

Задание на лабораторную работу

В качестве задания на лабораторную работы предлагается реализовать обобщённый контейнер для хранения произвольных типов в соответствии с вариантом задания. Проверить работу контейнера как со стандартными типами, так и с пользовательскими. Рассмотреть пример со вложенными контейнерами (очередь очередей, массив массивов и т.д.)

№	Структура данных
1	Стэк на основе односвязного списка
2	Очередь с фиктивным элементов
3	Динамический массив
4	Стэк на основе динамического массива
5	Очередь на основе статического массива
6	Стэк на основе односвязного списка
7	Очередь с фиктивным элементов
8	Динамический массив
9	Стэк на основе динамического массива
10	Очередь на основе статического массива
11	Стэк на основе односвязного списка
12	Очередь с фиктивным элементов
13	Динамический массив
14	Стэк на основе динамического массива
15	Очередь на основе статического массива

Таблица 1: Варианты задания

Содержание