Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

А. В. Столяров

Практикум на ЭВМ

Многопользовательский игровой сервер

Москва
2002

УДК 519.6+681.3.06

Автор будет признателен за конструктивную критику, в том числе за сообщения об обнаруженных в тексте пособия опечатках.

Адрес для связи: avst@cs.msu.su.

Введение

Задание практикума “многопользовательский игровой сервер” разработано для занятий практикума на ЭВМ, проводимых на факультете ВМиК МГУ в IV семестре в рамках основного учебного плана.

Задание предназначено для выполнения в операционной системе семейства Unix (например, FreeBSD или Linux) с использованием языка программирования C++.

Задание состоит из двух основных частей, каждая их которых выполняется в несколько этапов. В первой части задания предлагается реализовать программу-сервер, выполняющую роль ведущего в игре “Менеджер” [1] и позволяющую принимать участие в игре игрокам, находящимся на разных машинах, используя локальную сеть.

Во второй части предлагается реализовать программу-робот, имитирующую действия игрока. Робот реализуется в виде интерпретатора некоторого простейшего языка программирования, дающего доступ ко всей информации, которая имелась бы у обычного игрока во время игры, и ко всем командам, которые обычный игрок мог бы выдать серверу. Язык должен быть алгоритмически полным и иметь средства, достаточные для программирования нетривиальных стратегий игры.

Задание направлено на развитие и закрепление следующих основных навыков:

- программирование на языке C++ и основы объектно-ориентированного проектирования;
- организация взаимодействия по сети на уровне программ в операционной системе семейства Unix;
- создание событийно-управляемых программ с помощью системного вызова `select()`;
- трансляция и интерпретация языков программирования с использованием материалов основного лекционного курса “Системы программирования” (в части введения в теорию формальных грамматик и языков).

В настоящем пособии автор постарался ответить на наиболее типичные вопросы, возникающие у студентов по ходу работы над заданием.

1 Игра “Менеджмент”

Игра “Менеджмент” была предложена фирмой Avalon Hill Company для обучения основам управления предприятием. Ч. Уэзерелл отметил чрезвычайную привлекательность этой игры в качестве упражнения (этюда) для программистов.

В этом разделе излагаются сокращенные (упрощенные) правила игры “Менеджмент”. Полные правила игры желающие могут найти в книге [1].

1.1 Общие сведения

В игре участвуют N игроков. Каждый игрок имеет номер $1 \leq k \leq N$. Каждый игрок с номером k располагает некоторым количеством денег (условных долларов), S_k единицами сырья, P_k единицами продукции, F_k обыкновенными и F_k^A автоматизироованными фабриками. В начале игры каждому игроку выдается 2 обыкновенные фабрики, 4 единицы сырья, 2 единицы готовой продукции и 10000 долларов.

Моделирование ведется пошагово, игровыми циклами. Цикл представляет собой условный игровой месяц.

1.2 Порядок игры

В каждом “месяце” игроки производят на своих фабриках продукцию из сырья. Одна обыкновенная фабрика может произвести одну единицу продукции, израсходовав при этом одну единицу сырья и \$2000. Автоматизированная фабрика может либо сделать то же самое, либо произвести 2 единицы продукции, израсходовав две единицы сырья и \$3000.

В случае, если у игрока недостаточно сырья или денег, чтобы обеспечить работу всех фабрик, либо в связи с неблагоприятной обстановкой на рынке у игрока скопилось слишком много готовой продукции, фабрика может ничего не производить (что не исключает ежемесячных издержек).

Каждый “месяц” банк проводит аукционы по продаже сырья и скупке продукции. Аукционы проводятся в соответствии с “обстановкой на рынке”, описываемой ниже. Заявки на аукционы подаются игроками “в темную”, т.е. игроки ничего не знают о заявках, подаваемых другими игроками. Однако по окончании аукциона банк сообщает всем игрокам полную информацию о результатах торгов (а именно, кому, сколько и по какой цене продано сырья, а также у кого, сколько и по какой цене куплено продукции).

В каждом “месяце” игрок может сделать заявку на строительство новых фабрик. Обыкновенная фабрика стоит \$5000 и начинает давать продукцию

на 5й “месяц” после начала строительства; автоматизированная фабрика стоит $10000 и дает продукцию на 7й “месяц”. Обыкновенную фабрику можно преобразовать в автоматизированную, это стоит $7000. Реконструкция продолжается девять месяцев, все это время фабрика может работать и давать продукцию как обыкновенная. Половина стоимости строительства или реконструкции списывается с игрока при подаче заявки, вторая половина – за месяц до окончания строительства или реконструкции.

Наконец, по итогам месяца с каждого игрока списываются ежемесячные издержки, а именно: $300 за оставшуюся на складе единицу сырья, $500 - за оставшуюся на складе единицу продукции, $1000 за каждую обыкновенную и $1500 за каждую автоматизированную фабрику (независимо от того, производила она в этом месяце продукцию или нет).

Игрок, которому не хватило денег на покрытие издержек, объявляется банкротом и выбывает из игры.

Каждый игрок в любой момент может узнать о количестве денег, фабрик, единиц сырья и продукции у остальных игроков.

1.3 Обстановка на рынке

Обстановка на рынке может находиться на одном из пяти уровней. В зависимости от уровня определяются предложение сырья (т.е. сколько единиц сырья банк продаст в этом “месяце”), спрос на продукцию (т.е. сколько единиц продукции банк купит в этом “месяце”), минимальную цену единицы сырья и максимальную цену единицы продукции. Значения этих величин определяются по таблице уровней состояния рынка (табл. 1).

Таблица 1: Уровни состояния рынка

Уровень	Сырье		Продукция	
	кол-во	min. цена	кол-во	max.цена
1	1.0*P	$800	3.0*P	$6500
2	1.5*P	650	2.5*P	6000
3	2.0*P	500	2.0*P	5500
4	2.5*P	400	1.5*P	5000
5	3.0*P	300	1.0*P	4500

P - общее количество необанкротившихся игроков.

Округление производится в сторону уменьшения, т.е., например, если в игре участвуют 3 “живых” игрока, а уровень рынка определен как 2й, то

количество продаваемого сырья будет 4 единицы, а покупаемой продукции - 7 единиц.

В начале игры уровень равен 3. Уровень для каждого следующего месяца определяется из предыдущего случайным образом в соответствии с таблицей вероятностей перехода (табл. 2).

Таблица 2: Вероятности смены уровня состояния рынка

Старый уровень	Новый уровень				
	1	2	3	4	5
1	1/3	1/3	1/6	1/12	1/12
2	1/4	1/3	1/4	1/12	1/12
3	1/12	1/4	1/3	1/4	1/12
4	1/12	1/12	1/4	1/3	1/4
5	1/12	1/12	1/6	1/3	1/3

1.4 Проведение аукционов

На каждом цикле игрок может в произвольном порядке дать заявку на участие в аукционе сырья, в аукционе продукции, заявку на производство, заявку на строительство новой фабрики и заявить об окончании своих действий на этот месяц. Все заявки подаются “в темную”, то есть они не видны другим игрокам. В заявке на участие в аукционе указывается число единиц для покупки или продажи и цена. Цена покупки сырья не должна быть ниже минимальной установленной для текущего месяца, цена продажи продукции - не выше максимальной. Заявка на производство не должна превышать количество имеющегося у игрока сырья, т.е. сырье, купленное на данном цикле, не может быть использовано при производстве продукции на этом же цикле.

Если сумма заявок превышает доступное количество единиц, выставленных на аукцион, банк в первую очередь удовлетворяет наиболее выгодные для него заявки, т.е. продает сырье игрокам, заявившим наибольшие цены, и покупает продукцию у игроков, установивших наименьшие цены. При прочих равных предпочтение отдается случайным образом (по жребию). Если размер очередной выбранной заявки превышает оставшееся количество доступных единиц, банк удовлетворяет заявку частично.

Например, если банк должен купить всего 6 единиц продукции, при этом игрок №1 выставил на продажу 3 единицы по цене 4500, игроки №2

и №3 выставили каждый по две единицы продукции по цене 5000, то банк купит все 3 единицы у игрока №1, после чего жребий определит, какая из оставшихся заявок будет удовлетворена полностью. Соответственно, игрок, на которого падет жребий, продаст обе единицы продукции по цене 5000, а второй игрок продаст только одну из двух единиц.

Проведение аукциона начинается в тот момент, когда получены заявки на данный аукцион от всех активных (необанкротившихся) игроков. Если игрок заявил об окончании действий на данном цикле, не подав заявки на аукцион, его заявка считается нулевой.

Результаты аукционов (т.е. кому, сколько и по какой цене продано сырья, у кого, сколько и по какой цене куплено продукции) банк объявляет публично, то есть информация об этом доступна всем игрокам.

Месяц считается завершенным, когда все игроки заявили об окончании действий на этот месяц.

2 Реализация серверно-сетевой части

2.1 Постановка задачи

Программа-сервер выполняет функции ведущего игры. Игроки подключаются к серверу по сети с использованием протокола TCP/IP.

Возможны два подхода к организации протокола обмена прикладного уровня:

1. Сервер ожидает от клиента команды в текстовом виде, предназначенном непосредственно для обработки человеком. В этом случае в качестве клиентской программы используется стандартная утилита telnet, входящая в базовую комплектацию практически любой Unix-системы.

2. Сервер обрабатывает команды в определенном двоичном формате, удобном для обработки в программе. В этом случае необходимо реализовать клиентскую программу, ведущую диалог с пользователем и формирующую соответствующие данные для сервера, а также преобразующую получаемые от сервера ответы в приемлемую для пользователя форму.

Стартовыми параметрами сервера являются число игроков и номер TCP-порта, на котором программа должна ожидать запросы на соединение от клиентов. Стартовые параметры задаются в командной строке сервера. По согласованию с преподавателем возможны другие варианты получения стартовых параметров, например, из конфигурационного файла или из переменных окружения. Задавать стартовые параметры в тексте программы (т.е. так, что их изменение потребует перекомпиляции программы) *запрещается*.

После запуска программы-сервера она должна открыть сокет в режиме ожидания запросов на соединение (см. §2.2) на заданном порту, дождаться подключения заданного количества игроков, после чего перейти в режим игры, в котором и оставаться до момента, когда все игроки, кроме одного, по тем или иным причинам не выйдут из игры.

До тех пор, пока сервер не перешел в режим игры, на любую команду игрока он должен реагировать сообщением о том, сколько игроков в настоящее время уже вошли на сервер и сколько еще ожидается до начала игры.

После перехода в режим игры при попытке нового игрока подключиться к серверу он должен получить сообщение о том, что игра уже идет, после чего сервер должен разорвать соединение.

Требования к серверу:

1. Сервер должен предоставлять игроку все возможности, предусмотренные правилами игры (см. §1), и проводить игру в соответствии с правилами.

2. При проведении жеребьевок сервер не должен давать преимуществ никому из игроков.

3. Сервер должен быть защищен от неправильных действий игроков, т.е. никакие действия одного игрока не должны нарушать ход игры.

4. Сервер должен обеспечивать реальный многопользовательский режим работы, т.е. никакие действия игрока не должны приводить, даже кратковременно, к невозможности для остальных игроков вводить команды и получать результаты, если только это не предусмотрено правилами игры.

5. Сервер не должен использовать активное ожидание. В частности, это означает, что в отсутствие активности игроков сервер не должен создавать никакой нагрузки на процессор. См. §2.3.

6. Сервер обязан корректно обрабатывать потерю соединения с любым из игроков. Игрок, соединение с которым оборвалось, считается выбывшим из игры, о чем сообщается остальным игрокам.

В настоящем разделе приводится вся необходимая информация о том, как удовлетворить требованиям 4–6.

2.2 Организация TCP-сервера

Все сетевые взаимодействия в операционных системах семейства Unix организованы с помощью так называемых *сокетов* (sockets). С каждым сокетом связывается файловый дескриптор, позволяющий ссылаться на сокет при выполнении операций с ним.

При организации многопользовательского сервера понадобится два вида сокетов. Первый из них – *слушающий сокет* (listening socket) будет использоваться для ожидания и приема клиентских соединений. Сокеты другого вида представляют собой непосредственно “канал связи” с конкретным клиентом и используются для приема команд от клиентов и передачи клиентам сообщений сервера.

2.2.1 Создание сокета

Прежде всего, сокет (как объект ядра операционной системы) необходимо *создать* вызовом `socket()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

где `domain` задает семейство адресации (address family), `type` задает тип коммуникации, `protocol` - конкретный протокол.

В рассматриваемой задаче необходим сокет, работающий в семействе IP-адресов[1] (задается константой `AF_INET`). Это означает, что адрес сокета будет состоять из *IP-адреса* и *номера порта*. IP-адрес состоит из четырех чисел от 0 до 255, обычно записываемых через точку, например: `192.168.15.131`. Номер порта - это целое число в диапазоне от 1 до 65535. **Следует учитывать, что в большинстве операционных систем порты с номерами от 1 до 1023 считаются привилегированными; это означает, что использовать их могут только процессы, обладающие правами суперпользователя.**

Среди существующих типов коммуникации для рассматриваемой задачи наиболее подходит так называемый *потоковый* (stream), задаваемый константой `SOCK_STREAM`. Сокеты этого типа коммуникации представляют собой двунаправленный канал, доступный на обоих концах как на запись, так и на чтение, в том числе и с помощью обычных вызовов `read()` и `write()`.

Наконец, в качестве параметра `protocol` можно указать 0, в результате чего система автоматически выберет единственный возможный для данной комбинации семейства адресов и типа сокета протокол TCP (иначе задаваемый константой `IPPROTO_TCP`).

Таким образом, окончательно вызов будет выглядеть так:

```
ls = socket(AF_INET, SOCK_STREAM, 0);
```

где ls - имя переменной типа int, которой будет присвоен номер файлового дескриптора, ассоциированного с вновьсозданным сокетом.

Полученный файловый дескриптор должен быть неотрицательным числом. Если вызов `socket()` вернул значение -1, это свидетельствует о происшедшей ошибке. Программа **обязательно** должна корректно обрабатывать такую ситуацию.

[1] Здесь и далее имеются в виду IP-адреса семейства протоколов IPv4.

2.2.2 Связывание сокета с адресом

Следующим шагом развертывания сервера является сопоставление созданному сокету конкретного адреса. Напомним, что в избранном нами семействе адресов (`AF_INET`) адресом является пара "IP-адрес + порт".

Компьютер, оснащенный стеком протоколов TCP/IP, может иметь произвольное количество ip-адресов. В частности, любой компьютер имеет адрес `127.0.0.1`, означающий сам этот компьютер и доступный только программам, работающим на этом же компьютере. При подключении к локальной сети компьютер также получает *интерфейсный* адрес для работы в этой сети.

Сервер может принимать соединения по заданному номеру порта на одном из IP-адресов, имеющихся в системе, либо на всех IP-адресах сразу. Последнее задается IP-адресом `0.0.0.0`, имеющим специальное значение и обозначающимся также константой `INADDR_ANY`.

Связывание сокета с конкретным адресом производится вызовом `bind()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

где `sockfd` – дескриптор сокета, полученный в результате выполнения вызова .socket().; `addr` – указатель на структуру, содержащую адрес; наконец, `addrlen` – размер структуры адреса в байтах.

Реально в качестве параметра `addr` используется не структура типа `sockaddr`, а структура другого типа, который зависит от используемого семейства адресации (см. §2.2.1). В избранном нами семействе `AF_INET` используется структура `struct sockaddr_in`, умеющая хранить пару "IP-адрес + порт". Эта структура имеет следующие поля:

- `sin_family` – обозначает семейство адресации (в данном случае значение этого поля должно быть установлено в `AF_INET`).

- `sin_port` – задает номер порта в *сетевом порядке байт*, который, вообще говоря, может отличаться от порядка байт, используемого на данной машине. Соответственно, значение для занесения в это поле должно быть получено из выбранного номера порта вызовом функции `htons()`[2]. Напомним, что номер порта задается как параметр командной строки программы-сервера.

[2]Название функции htons() получено как сокращение от *Host to Network Short*, т.е. преобразование из хостового в сетевой порядок байт для короткого целого.

- `sin_addr` – задает IP-адрес. Поле `sin_addr` само является структурой, имеющей лишь одно поле с именем `s_addr`, которое хранит IP-адрес в виде беззнакового четырехбайтного целого. Именно этому полю следует присвоить значение `INADDR_ANY`.

Вызов `bind()` возвращает 0 в случае успеха, -1 – в случае ошибки. Учтите, что существует множество ситуаций, в которых вызов `bind()` может не пройти; например, ошибка произойдет в случае попытки использования привилегированного номера порта (от 1 до 1023) или порта, который на данной машине уже кем-то занят (возможно, другой вашей программой). Поэтому обработка ошибок при вызове `bind()` особенно важна.

Итак, окончательно подготовка и вызов `bind()` могут выглядеть следующим образом:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY;
if (0 != bind(ls, (struct sockaddr *) &addr, sizeof(addr)))
{
  /* Здесь следует поместить обработку ошибки */
}
```

где `ls` – переменная, хранящая дескриптор сокета, а `port` – переменная, в которую тем или иным способом занесен избранный номер порта.

2.2.3 Ожидание и прием клиентских соединений

После того, как сокет создан и с ним связан адрес, его необходимо перевести в состояние ожидания запросов на соединения, или, иначе говоря, в *слушающий режим* (listening state). Это достигается вызовом `listen()`:

```
#include <sys/socket.h>

int listen(int sockfd, int qlen);
```

Параметр `sockfd` задает дескриптор сокета. Параметр `qlen` означает максимальную длину очереди пришедших запросов на соединение, которые сервер еще не принял к обработке. Некоторые операционные системы не поддерживают значения `qlen`> 5, поэтому обычно вторым параметром вызова `listen()` задают просто число 5.

Вызов `listen()` возвращает 0 в случае успеха, -1 – в случае ошибки. С учетом этого вызов может выглядеть так:

```
if (0 != listen(ls, 5)) {
  /* Здесь следует поместить обработку ошибки */
}
```

В результате выполнения вызова `listen()` в системе появится *слушающий сокет*, который можно увидеть с помощью команды

```
netstat -a -n | grep LISTEN
```

Клиент, находящийся на любой машине, с которой доступна наша сеть, может установить соединение с нашим сокетом. Чтобы получить возможность обмена информацией с этим клиентом одновременно с ожиданием запросов на соединение от других клиентов, нам необходимо *принять* запрос на соединение. Это делается с помощью вызова `accept()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

Параметр `sockfd` задает дескриптор слушающего сокета, на котором следует принять соединение. Что касается параметров `addr` и `addrlen`, то они позволяют узнать, с какого адреса исходит соединение. Информация записывается в структуру, на которую указывает указатель `addr`. Переменная, на которую указывает `addrlen`, должна перед обращением к `accept()` содержать длину структуры `addr` в байтах; после обращения в ней будет содержаться реальная длина данных, записанных в эту структуру. Естественно, использовать следует структуру типа `struct sockaddr_in` (см. §2.2.2).

Если информация об источнике соединения не интересна, в качестве обоих параметров `addr` и `addrlen` можно передать нулевые указатели.

Если во входной очереди уже имеется запрос на соединение, вызов `accept()` возвращает управление немедленно; в противном случае управление будет возвращено после того, как такой запрос будет получен.

Вызов `accept()` возвращает файловый дескриптор, связанный с сокетом, через который будет осуществляться связь с клиентом, соединение которого только что было принято.

Полученный файловый дескриптор должен быть неотрицательным числом. Если вызов `socket()` вернул значение -1, это свидетельствует о происшедшей ошибке. Программа **обязательно** должна корректно обрабатывать такую ситуацию.

2.3 Мультиплексирование ввода-вывода

После того, как вызов `accept()` успешно отработает в первый раз, в вашей программе появятся два файловых дескриптора, требующих внимания. Это слушающий сокет и сокет, полученный в результате вызова `accept()` (сокет клиента). На слушающий сокет могут поступить новые запросы, которые необходимо принимать вызовом `accept()`; в то же время на сокет клиента могут поступить данные, переданные клиентом, которые необходимо прочитать с помощью вызова `read()`. Какое из этих событий произойдет раньше, неизвестно.

Более того, в программе могут быть и другие источники событий. Так, клиентов, скорее всего, будет больше одного. Кроме того, для реализации управления сервером нам, возможно, потребуется организовать диалог с пользователем, запустившим сервер, для чего необходима возможность обработки данных, поступающих со стандартного ввода программы-сервера.

2.3.1 Методы организации многопользовательского сервера

Существует несколько подходов к организации программ, работающих с несколькими источниками событий. Самый простой (и некорректный) из них состоит в организации циклического опроса всех имеющихся источников событий. Так, все имеющиеся сокеты можно перевести в так называемый *неблокирующий* режим, в котором все системные вызовы, относящиеся к таким сокетам, которые не могут быть исполнены без блокирования выполнения процесса, будут возвращать управление сразу же, сигнализируя об ошибке (в частности, вызов `accept()`, будучи вызванным в отсутствие необработанного запроса на соединение, немедленно вернет -1).

Этот способ имеет фундаментальный неустранимый недостаток, заключающийся в наличии так называемого *активного ожидания*. Действительно, даже если не происходит никаких событий, требующих обработки, программа-сервер будет вновь и вновь опрашивать имеющиеся дескрипторы, впустую занимая процессорное время. Естественно, пользоваться таким методом ни в коем случае не следует.

Второй вариант организации многопользовательского сервера, о котором, в частности, рассказывается в основном курсе лекций “Системное программное обеспечение” в III семестре, основан на создании отдельного процесса для работы с каждым источником событий. В этом случае после каждого вызова `accept()` немедленно выполняется вызов `fork()`, порождающий новый процесс, и родительский процесс возвращается к обработке входящих запросов на соединение, в то время как дочерний процесс обслуживает клиента, используя полученный от `accept()` дескриптор. Подробно

этот механизм рассмотрен в книге [2].

Такой вариант идеально подходит для случая, когда каждый клиент обслуживается отдельно от остальных и не имеет с ними никакой связи. Однако для случая многопользовательской игры, которая проходит в общем игровом пространстве, такой способ подходит заметно хуже, поскольку влечет активное использование разделяемой памяти и семафоров, что само по себе усложняет программу[3].

Третий способ называется *мультиплексированием ввода-вывода* и может быть осуществлен с помощью системных вызовов `select()` или `poll()`[4]. В дальнейшем мы ограничимся рассмотрением функции `select()`. При желании читатель может освоить функцию `poll()` самостоятельно, прибегнув к литературе и команде `man`.

2.3.2 Вызов select()

Системный вызов `select()` предназначен для использования в ситуации, когда необходимо организовать работу с несколькими файловыми дескрипторами, не имея a priori информации о том, какой из дескрипторов первым потребует внимания программы. Кроме того, возможно, требуется отслеживание некоторых событий по времени (например, тайм-аутов на сетевых соединениях).

Прототип вызова `select()` выглядит следующим образом:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds,
                  fd_set *writefds,
                  fd_set *exceptfds,
                  struct timeval *timeout);
```

Параметр `n` означает максимальный номер дескриптора, подлежащий обработке вызовом `select()`. Параметры `readfds`, `writefds` и `exceptfds` обозначают множества файловых дескрипторов, для которых нас интересует, соответственно, возможность немедленного чтения, возможность

[3]Тем не менее, построить сервер таким образом вполне возможно. Несмотря на сложности, связанные с использованием разделяемой памяти, это может быть интересно в качестве упражнения.

[4]Вообще говоря, select() и poll() предназначены для одних и тех же действий. select() несколько проще в работе, poll() несколько более универсален. В некоторых системах ядро реализует только один вариант интерфейса, при этом второй эмулируется через него в виде библиотечной функции. Так, в системе Solaris присутствует системный вызов poll(), а select() является библиотечной функцией.

немедленной записи и наличие исключительной ситуации. Наконец, параметр `timeout` задает промежуток времени, спустя который следует вернуть управление, даже если никаких событий, связанных с дескрипторами, не произошло.

Объект "множество дескрипторов" задается переменной типа `fd_set`. Внутренняя реализация переменных этого типа нас, вообще говоря, не интересует[5]. Для работы с переменными этого типа система предоставляет в наше распоряжение следующие макросы:

```
FD_ZERO(fd_set *set);       /* очистить множество */
FD_CLR(int fd, fd_set *set);/* убрать дескриптор из мн-ва */
FD_SET(int fd, fd_set *set);/* добавить дескриптор к мн-ву */
FD_ISSET(int fd, fd_set *set);
                            /* входит ли дескр-р в мн-во? */
```

В рассматриваемой задаче объемы передаваемых по сети данных сравнительно незначительны, что позволяет предполагать, что системные вызовы, осуществляющие запись в сокеты, никогда не будут блокировать программу-сервер. Также можно считать, что на сокетах никогда не произойдут исключительные ситуации. Таким образом, аргументы `writefds` и `exceptfds` можно не использовать (вместо них передавать вызову `select()` нулевые указатели).

В простейшей версии программы-сервера также нет необходимости в использовании параметра `timeout`[6]. Поэтому можно передать нулевой указатель и в качестве пятого параметра вызова. На всякий случай отметим, что структура `timeval` имеет два поля типа long. Поле `tv_sec` задает количество секунд, поле `tv_usec` - количество микросекунд (миллионных долей секунды).

Вызов `select()` изменяет все переданные ему по указателю аргументы, так что перед каждым обращением к нему аргументы должны быть сформированы заново.

Вызов `select()` возвращает -1 в случае возникновения ошибки. Учтите, что, если ваша программа обрабатывает те или иные сигналы, вызов `select()` может вернуть -1 в случае, если его выполнение было прервано пришедшим сигналом. При этом значением переменной `errno` будет `EINTR`, что свидетельствует о нормальном ходе событий. Вы можете не учитывать данный комментарий, если ваша программа не перехватывает никаких сигналов. Вызов возвращает значение 0 в случае, если причиной выхода из вызова стало наступление заданного таймаута. Если

[5] Для различных систем она может оказаться разной.

[6] Необходимость использования этого параметра возникает при выполнении некоторых дополнительных задач.

вызов возвратил положительное число, оно означает количество дескрипторов, для которых произошло какое-то событие.

После возврата из вызова `select()` переданные ему переменные типа `fd_set` оказываются модифицированы. Если при входе в вызов эти множества содержали дескрипторы, относительно которых нас интересует информация о событиях, то по окончании вызова эти же множества содержат дескрипторы, на которых событие реально произошло (например, по сети прибыли данные, которые могут быть считаны).

Таким образом, работу с вызовом `select()` можно построить по следующей схеме (считаем, что номер слушающего сокета по-прежнему хранится в переменной `ls`; как хранить дескрипторы клиентских сокетов, читателю предлагается решить самостоятельно):

```
for(;;) { /* главный цикл */
  fd_set readfds;
  int max_d = ls;
   /* изначально полагаем, что максимальным является
      номер слушающего сокета */
  FD_ZERO(&readfds); /* очищаем множество */
  FD_SET(ls, &readfds);
     /* вводим в множество
        дескриптор слушающего сокета */
  int fd;
       /* организуем цикл по сокетам клиентов */
  for(fd=/*дескриптор первого клиента*/ ;
      /*клиенты еще не исчерпаны?*/     ;
      fd=/*дескриптор следующего клиента*/) {
     /* здесь fd - очередной клиентский дескриптор */
     /* вносим его в множество */
    FD_SET(fd, &readfds);
     /* проверяем, не больше ли он,
        нежели текущий максимум */
    if(fd > max_d) max_d = fd;
  }
  int res = select(max_d, &readfds, NULL, NULL, NULL);
  if(res < 1) {
    /* обработка ошибки, происшедшей в select()'е */
  }
  if(FD_ISSET(ls, &readfds)) {
    /* пришел новый запрос на соединение */
    /* здесь его необходимо принять
```

```
      вызовом accept() и запомнить
      дескриптор нового клиента */
  }
    /* теперь перебираем все клиентские дескрипторы */
  for(fd=/*дескриптор первого клиента*/ ;
      /*клиенты еще не исчерпаны?*/     ;
      fd=/*дескриптор следующего клиента*/)
    if(FD_ISSET(fd, &readfds)) {
      /* пришли данные от клиента с сокетом fd */
      /* читаем их вызовом \verb.read(). или
         \verb.recv(). и обрабатываем */
    }

  /* конец главного цикла, можно идти на
     следующую итерацию */
}
```

2.4 Прием и передача данных через сокеты

2.4.1 Чтение

Чтение данных из сокета можно произвести обычным вызовом `read()`, уже знакомым читателю из курса “Системное программное обеспечение”:

```
#include <unistd.h>

size_t read(int fd, void *buf, size_t len);
```

либо специально предназначенным для сокетов вызовом `recv()`:

```
#include <sys/types.h>
#include <sys/socket.h>

size_t recv(int fd, void *buf, size_t len,
                          unsigned int flags);
```

В обоих случаях `fd` задает файловый дескриптор (в случае `recv()` это обязательно должен быть дескриптор сокета); `buf` указывает на буфер, в который следует поместить прочитанные данные; `len` сообщает вызову размер буфера, чтобы избежать его переполнения. Параметр `flags`, имеющийся только у вызова `recv()`, позволяет установить некоторые специфические

режимы работы, которые в рассматриваемой задаче не нужны. В дальнейшем изложении мы будем использовать вызов `read()`, что позволит использовать одни и те же фрагменты кода с потоками различной природы (кроме сокетов, это могут быть стандартные ввод и вывод, каналы типа `pipe` и др.)

Вызов `read()` производит чтение из потока. Прочитанные данные располагаются в буфере, на который указывает параметр `buf`. Читается не более чем `len` байт данных, что позволяет избежать переполнения буфера. Если в указанном потоке отсутствуют данные, готовые к прочтению, вызов блокирует вызваавший процесс до тех пор, пока данные не появятся, и только после их прочтения вернет управление.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество прочитанных байт. Естественно, это число не может быть больше `len`. Случай, когда вызов вернет 0, рассматривается в §2.4.3.

Необходимо учитывать, что сообщение, отправленное клиентской программой по сети через потоковый сокет, не обязательно будет прочитано на стороне сервера в один прием. Например, клиент отправил текстовую строку `"Just a string\n"`, имеющую длину 14 байт. Вполне возможно, что она будет прочитана за один вызов `read()`, однако механизм сетевых сокетов такой гарантии не дает. Это означает, что очередной вызов `read()` может, например, прочитать 4 байта `"Just"`, следующий за ним - 5 байт `" a str"`, и, наконец, еще один - оставшиеся `"ing\n"` В связи с этим сервер **обязательно** должен иметь накопительный буфер, в котором принятые данные будут храниться до тех пор, пока не будет получена команда целиком. Команды можно отделять друг от друга, например, символом перевода строки или пустой строкой (два перевода строки подряд).

2.4.2 Запись

Для записи в сокет можно пользоваться вызовом `write()`:

```
#include <unistd.h>

size_t write(int fd, const void *buf, size_t len);
```

либо специально предназначенным для сокетов вызовом `send()`:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
size_t send(int fd, const void *buf, size_t len,
                                unsigned int flags);
```

Параметр `fd` задает файловый дескриптор (в случае `send()` это обязательно должен быть дескриптор сокета); `buf` указывает на буфер, содержащий данные, которые необходимо передать через сокет; `len` задает количество этих данных. Параметр `flags`, имеющийся только у вызова `send()`, позволяет установить некоторые специфические режимы работы, которые в рассматриваемой задаче не нужны. Как и в случае с чтением, в дальнейшем изложении мы отдадим предпочтение вызову `write()`, как более универсальному.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество переданных байт. Естественно, это число не может быть больше `len`.

Следует обратить особое внимание на передачу ограничительных символов. Рассмотрим пример. Допустим, в программе задана строковая константа:

```
const char juststring[50] = "This is just a string\n";
```

Тогда вызов

```
write(sd, juststring, 21);
```

передаст в сокет `fd` строку без символа перевода строки и без ограничительного нуля; вызов

```
write(sd, juststring, 22);
```

или, что то же самое,

```
write(sd, juststring, strlen(juststring));
```

передаст ту же строку уже с символом перевода строки, но по-прежнему без ограничивающего нуля, что может привести к ошибкам при обработке на другой стороне соединения, если только не предпринять специальных мер по восстановлению ограничителя. Грубой ошибкой будет вызов

```
write(sd, juststring, sizeof(juststring));
```

В рассматриваемом примере такой вызов передаст в сокет 24 байта полезной информации и 26 байт случайного мусора, который, будучи проинтерпретирован как очередная команда или ее часть, вызовет ошибки. А если

бы мы описали `juststring` не как массив, а как указатель, передано было бы и вовсе ровно 4 байта (размер указателя на большинстве современных архитектур).

Более правильно было бы написать примерно так:

```
write(sd, juststring, strlen(juststring)+1);
```

если, конечно, вам не мешает перевод строки. Если же его передача нежелательна, вставьте на его место ограничительный 0 и уже после этого передавайте.

2.4.3 Разрыв соединения и обработка разрыва

Завершить работу с сокетом можно с помощью вызова `shutdown`:

```
#include <sys/socket.h>

int shutdown(int sd, int how);
```

Параметр `sd` задает дескриптор сокета, `how` – что именно следует прекратить. При `how == 0` сокет закрывается на чтение, при `how == 1` – на запись, при `how == 2` – полностью (в оба направления). В рассматриваемой задаче требуется только последний вариант.

Вызов `shutdown()` прекращает работу с сокетом, однако сам сокет вместе с файловым дескриптором продолжает существовать. Чтобы окончательно избавиться от ненужного сокета и освободить дескриптор, следует закрыть его вызовом `close()`:

```
#include <unistd.h>

int close(int fd);
```

Естественно, сокет будет также закрыт, если завершился процесс, в котором этот сокет использовался, и других процессов, использующих тот же сокет (например, дочерних процессов, созданных вызовом `fork()`, когда сокет уже существовал) не осталось.

Узнать о том, что ваш партнер по коммуникации разорвал соединение, можно по значению, возвращаемому вызовом `read()` или `recv()`. В случае, если все данные из сокета прочитаны и на противоположном конце соединение закрыто, вызов чтения из сокета вернет 0.

Следует учитывать, что, если сокет в таком состоянии включить в множество дескрипторов, обрабатываемых вызовом `select()` (см.§2.3), то вызов вернет управление немедленно, причем дескриптор рассматриваемого

сокета будет помечен как готовый на чтение. Вызов же на чтение опять вернет 0. Поэтому, **если не предусмотреть в программе корректной обработки это ситуации, возможно ее зацикливание при разрыве соединения одним из клиентов**.

2.5 Организация программы-клиента

2.5.1 Установление соединения

Как и в случае программы-сервера, для случая программы-клиента потребуется сокет, однако на этот раз сокет требуется только один. Создать его необходимо точно так же, как это делается в сервере, вызовом `socket()` (см. §2.2.1).

В случае, если по каким-то причинам вам необходимо указать IP-адрес и/или порт, с которого должно исходить создаваемое клиентское соединение, можно применить к сокету вызов `bind()` (см. §2.2.2). Однако такие ситуации редки и при выполнении задания практикума возникнуть не должны.

Чтобы установить соединение с сервером, необходимо воспользоваться вызовом `connect()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr,
                                    size_t addrlen);
```

Здесь `sockfd` – дескриптор сокета, полученный в результате выполнения вызова `socket()`; `serv_addr` – указатель на структуру, содержащую адрес сервера; наконец, `addrlen` – размер структуры адреса в байтах.

Как и в §2.2.2, для хранения адреса используем структуру типа `struct sockaddr_in`. Естественно, чтобы связаться с сервером, необходимо каким-либо образом узнать (например, запросить у пользователя) IP-адрес и порт сервера. Номер порта, как и раньше, необходимо перевести в сетевой порядок байт функцией `htons()`. Что касается IP-адреса, то, имея его текстовое представление (например, строку `"192.168.10.12"`), можно воспользоваться функцией `inet_aton()` для формирования структуры типа `struct in_addr` (напомним, что поле `sin_addr` структуры `sockaddr_in` имеет именно этот тип):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Здесь `cp` – строка, содержащая текстовое представление IP-адреса, а `inp` указывает на структуру, подлежащую заполнению. Функция возвращает ненулевое значение, если заданная строка является записью валидного IP-адреса, и 0 в противном случае.

Допустим, дескриптор сокета хранится в переменной `sockfd`, нужный нам IP-адрес содержится в строке `char *serv_ip`, а порт – в переменной `port` в виде целого числа. Тогда подготовка и вызов `connect()` могут выглядеть так:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
if(!inet_aton(serv_ip, &(addr.sin_addr))) {
   /* Ошибка - введен невалидный IP-адрес */
}
if (0 != connect(sockfd, (struct sockaddr *)&addr,
                                      sizeof(addr)))
{
  /* Здесь следует поместить обработку ошибки connect */
}
```

2.5.2 Встречные потоки данных и их обработка

Встречные потоки данных. Программа-клиент обычно имеет два основных потока данных. Первый из них образуют данные, введенные пользователем (то есть, например, прочитанные со стандартного ввода), которые после необходимых преобразований передаются серверу через сокет. Второй поток образуют данные, полученные от сервера в качестве отклика; после преобразований они тем или иным способом передаются пользователю (например, через поток стандартного вывода).

Это не создает никаких проблем, если мы считаем, что сервер способен что-либо передавать клиенту исключительно в рамках ответа на очередную команду. В этом случае мы можем действовать по следующей схеме:

1. Распечатываем приглашение;
2. Считываем команду, введенную пользователем;
3. Формируем и передаем команду серверу;

4. Ожидаем отклика сервера;

5. Преобразуем полученную информацию;

6. Выдаем ее пользователю;

7. Переходим на начало.

Схема работы оказывается строго последовательной благодаря тому, что, пока сервер не отозвался на нашу команду, пользователь лишается возможности что-либо вводить, а пока пользователь не завершил свой ввод, программа не обрабатывает данные, поступающие от сервера.

Однако такая схема работы лишает нас возможности принимать от сервера сообщения о событиях, никак не связанных с нашими командами. Например, сообщение о том, что завершен очередной аукцион, логично было бы разослать всем игрокам непосредственно по завершении аукциона, что может произойти после команды, выданной другим игроком. То же касается, например, сообщения о том, что кто-то из игроков потерял связь с сервером и выбыл из игры. Наконец, можно предусмотреть в программе-сервере разнообразные напоминания, связанные со слишком долгим отсутствием активности со стороны игрока.

Точно таким же образом может оказаться, что блокировать действия пользователя на время обработки команды сервером неудобно. Так, клиентская программа может некоторые команды пользователя обрабатывать самостоятельно, не обращаясь при этом к серверу (например, программа может запоминать статистику предыдущих циклов игры и выдавать ее по запросу пользователя).

Решение с помощью двух процессов. Одно из возможных решений – использовать два процесса, один из которых будет читать пользовательский ввод и передавать готовые команды серверу, а второй – принимать сообщения от сервера и выдавать информацию пользователю. При этом во избежание конфликтов желательно в первом процессе закрыть поток стандартного вывода, выполнив вызов `close(1)`, а во втором – поток стандартного ввода, выполнив `close(0)`.

Это решение подходит, например, в случае, если клиентская программа не поддерживает команд, выполняемых без обращения к серверу. В противном случае процессу, отвечающему за ввод и передачу, чтобы отреагировать на введенную «внутреннюю» команду, потребуется возможность записи в поток стандартного вывода, что в некоторых системах приводит к возникновению ошибочной ситуации.

Решение на основе select(). Более корректное решение возможно с использованием уже знакомого читателю вызова `select()` (см. §2.3). В этом случае отпадает потребность во втором процессе, т.к. операции чтения не будут блокировать основной процесс. Как и при организации программы-сервера, можно ограничиться заданием только одного параметра – множества дескрипторов, для которых нас интересует готовность на чтение (`readfds`). Обработке в этом случае подлежат всего два дескриптора: дескриптор потока стандартного ввода (0) и дескриптор сокета.

2.6 Дополнительные сведения

2.6.1 Подробнее о порядке байт в целых числах

Порядок байт в представлении целых чисел в памяти может варьироваться от одной архитектуры к другой. Архитектуры, в которых старший байт числа имеет наименьший адрес, в англоязычной литературе обозначаются термином *big-endian*, а архитектуры, в которых наименьший адрес имеет младший байт – *little-endian*[7].

Чтобы сделать возможным взаимодействие по сети между машинами, имеющими разные архитектуры, принято соглашение, что передача целочисленной информации по сети всегда идет в прямом (*big-endian*) порядке байт, т.е. старший байт передается первым. Чтобы обеспечить переносимость программ на уровне исходного кода, в операционных системах семейства Unix введены стандартные библиотечных функции для преобразования целых чисел из формата данной машины (*host byte order*) в сетевой формат (*network byte order*). На машинах, порядок байт в архитектуре которых совпадает с сетевым, эти функции просто возвращают свой аргумент, в ином случае они производят необходимые преобразования. Вот эти функции:

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

[7] «Термины» big-endians и little-endians введены Свифтом в книге "Путешествия Гулливера" и на русский язык обычно переводились как *тупоконечники и остроконечники*. Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

Как можно догадаться, буква `n` в названиях функций означает network (т.е. сетевой порядок байт), буква `h` – host (порядок байт данной машины). Наконец, `s` обозначает короткие целые, а `l` – длинные целые числа. Таким образом, например, функция `ntohl()` используется для преобразования длинного целого из сетевого порядка байт в порядок байт, используемый на данной машине.

2.6.2 Как избежать "залипания" TCP-порта по завершении сервера

Часто при работе с сервером можно заметить, что после завершения программы-сервера ее некоторое время невозможно запустить с тем же значением номера порта. Это происходит обычно при некорректном завершении программы-сервера, либо если программа-сервер завершается при активных клиентских соединениях. В этих случаях ядро операционной системы некоторое время продолжает считать адрес занятым.

Избежать этих ситуаций при отладке программы-сервера очень сложно, однако система сокетов позволяет изменить поведение ядра в отношении адресов, "залипших" подобным образом. Для этого необходимо перед вызовом `bind()` выставить на будущем слушающем сокете опцию `SO_REUSEADDR`. Это делается с помощью системного вызова `setsockopt()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int sd, int level, int optname,
               const void *optval, int optlen);
```

Параметр `sd` задает дескриптор сокета, `level` обозначает уровень (слой) стека протоколов, к которому имеет отношение устанавливаемая опция (в данном случае это уровень сокетов, обозначаемый константой `SOL_SOCKET` Значением опции в данном случае будет целое число 1, так что следует завести переменную типа int, присвоить ей значение 1 и передать в качестве `optval` адрес этой переменной, а в качестве `optlen` – выражение `sizeof(int)`. Таким обазом, вызов будет выгдядеть так:

```
int opt = 1;
setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

3 Программирование логики игры

3.1 Объектно-ориентированная модель предметной области

Для построения подходящей объектно-ориентированной модели необходимо прежде всего выделить ключевые понятия предметной области. Эта задача, вообще говоря, неформализуема[8]; соответственно, проектирование модели является своего рода искусством. От того, насколько удачно построена модель предметной области, во многом зависит дальнейшая судьба проекта.

3.1.1 Основные понятия предметной области ("игрок" и "ведущий")

В нашем случае можно немедленно выделить два понятия, совершенно однозначно присутствующие в предметной области и являющиеся для нее ключевыми. Это понятия "игрок" и "банк", или "ведущий". Как можно заметить, это будет справедливо практически для любой игры, в которой участвуют одновременно несколько человек.

Объект класса "игрок" должен хранить информацию обо всем, чем располагает игрок (деньги, сырье и продукция, фабрики и т.д.). Кроме того, на этот же объект можно возложить обязанность хранить клиентский сокет, принимать от клиента команды (и сохранять частично принятое в буфере, как это описано в §2.4.1). Объектов этого класса будет создано ровно столько, сколько в игре участвует игроков.

Объект класса "ведущий" отвечает за состояние игры, включая, например, номер текущего хода, состояние аукционов, обстановку на рынке. Было бы логично возложить на ведущего хранение списка игроков. Объект класса "ведущий", разумеется, нужен в программе ровно один (если только мы не собираемся обслуживать несколько партий одновременно).

Оба класса необходимо снабдить методами, соответствующими возникающим в программе логически единым действиям. Например, в класс "игрок" можно определить метод `MonthlyExpenses()`, подсчитывающий и списывающий с данного игрока сумму его ежемесячных издержек на текущий месяц. Аналогично, для подачи заявки на аукцион сырья класс "ведущий" может предоставлять метод `Order(int amount, int price)`, а для сообщения об окончании аукциона в классе "игрок" можно предусмотреть ме-

[8] О некоторых формальных приемах анализа предметной области читатель может узнать из книги [3].

тод `SatisfiedOrder(int amount)`, который будет вызывать "ведущий" по окончании аукциона.

При добавлении методов рекомендуется придерживаться следующих простых правил:

1. Все, что может быть сделано с объектом, можно делать только путем вызова методов, и никак иначе. Это означает, что в классе не должно быть ни одного открытого поля.

2. Метод должен добавляться в класс тогда, когда он понадобился в программе, и не раньше.

3. Если в некотором месте программы с объектом класса совершается несколько последовательных действий, скорее всего это означает, что всю эту последовательность следует реализовать в виде одного метода.

3.1.2 Дополнительная инкапсуляция отдельных частей

Для упрощения восприятия программы следует некоторые логически обособленные функции уже введенных объектов выделить в отдельные классы. Так, например, можно описать класс "обстановка на рынке", реализующий правила, описанные в §1.3. Такой класс может иметь, например, следующие методы:

1. `int GetRawMaterialAmount(int p) const` возвращает количество доступных на рынке единиц сырья при данном количестве игроков (`p`) в соотетствии с табл. 1;

2. `int GetRawMaterialPrice() const` возвращает минимальную стоимость единицы материалов;

3. `int GetProductionAmount(int p) const` возвращает спрос на готовую продукцию;

4. `int GetProductionPrice() const` возвращает максимальную стоимость единицы продукции;

5. `void ChangeLevel()` генерирует случайное число и меняет текущий "уровень экономики" в соответствии с таблицей вероятностей переходов (табл. 2).

Конструктор такого класса, не имеющий параметров (т.е. являющийся конструктором по умолчанию), должен устанавливать начальный уровень экономики, равный 3. Тогда достаточно будет описать в классе "ведущий" поле типа класс "обстановка на рынке", и реализация класса "ведущий" станет существенно более простой и наглядной, поскольку все, что касается обстановки на рынке (в том числе обращения к таблицам уровней и переходов) окажется *инкапсулировано* в этом поле:

```
class MarketStatus { // обстановка на рынке
    int current_level;
public:
    MarketStatus() { current_level = 3; }
    int GetRawMaterialAmount(int p) const;
    int GetRawMaterialPrice() const;
    int GetProductionAmount(int p) const;
    int GetProductionPrice() const;
    void ChangeLevel();
};

class Banker { // банкир, или ведущий
    // ...
    MarketStatus market;
public:
    // ...
};
```

Аналогичным образом можно инкапсулировать в отдельные классы понятия аукциона сырья, аукциона продукции и т.д.

3.1.3 Фабрики

В правилах игры фигурирует понятие фабрики. Можно заметить, что фабрика оказывается на поверку достаточно нетривиальным объектом. Действительно, фабрика может быть строящейся, обыкновенной, автоматизированной или реконструирующейся; может в текущем месяце производить продукцию (причем для случая автоматизированной фабрики – одну или две единицы) либо не производить ее. В зависимости от всех этих обстоятельств вычисляются, например, издержки на данную фабрику.

Очевидно, что понятия обыкновенной и автоматизированной фабрик являются частными случаями некоего более общего понятия "фабрика".

Таким образом, фабрики в нашей игре – прекрасный повод для построения простейшей иерархии наследования классов с общим предком. По-видимому, общий предок ("фабрика") должен быть абстрактным классом, имеющим несколько чисто виртуальных функций, каковые, будучи переопределены в потомках, зададут конкретную функциональность того или иного типа фабрики. Примеры особенностей приведены в табл. 3. Моделирование прочих различий предлагается читателю в качестве самостоятельного упражнения.

Таблица 3: Три типа фабрик

Тип фабрики	Строящаяся	Обыкно-венная	Автомати-. зированная
Залог. стоимость	\$2500	\$5000	\$10000
Макс. продуктивность	0	1	2
Ежемес. издержки	0	\$1000	\$1500
Стоимость производства	-	1:\$2000	1:\$2000/2:\$3000
Возможность реконструкции	нет	если еще не начата	нет

Следует также заметить, что фабрика как объект взаимодействует только с объектом "игрок", причем только с одним. Очевидное соотношение *"игрок владеет фабрикой"* выполняется не только в правилах игры, но и в модели предметной области. Поэтому рекомендуется реализовать классы "игрок" и "фабрика" в соответствии с моделью "главный-подчиненный". Это означает, в частности, что конструкторы классов "фабрик" можно сделать закрытыми, а класс "игрок" объявить для них дружественным.

Чтобы оценить, насколько удачно вы спроектировали иерархию классов, ответьте на несколько вопросов:

1. Есть ли в ваших классах общее поле, обозначающее тип объекта?

2. Есть ли хотя бы в одном из классов открытое (public) поле (не функция)?

3. Есть ли в ваших классах поля с модификатором static?

4. Пришлось ли при работе с фабриками (как внутри иерархии классов "фабрик", так и вне её) хотя бы один раз использовать оператор `switch`?

5. Есть ли в базовом абстрактном классе публичные невиртуальные функции, вызывающие несколько разных виртуальных функций? (такой невиртуальной функцией могла бы быть, например, функция "переход к следующему месяцу").

Ответ на последний вопрос должен быть положительным, на остальные - отрицательным. Если это не так, стоит еще подумать над проектированием иерархии.

3.2 Командный интерфейс (протокол)

Для связи клиента с сервером необходимо разработать протокол прикладного уровня, т.е. некоторый набор соглашений между клиентом и сервером о том, какие данные будут передаваться и что они должны означать.

В простейшем случае в качестве клиентской программы может использоваться стандартная утилита telnet. Все, что эта программа может делать – это принимать от пользователя текстовые команды строка за строкой и передавать их в неизменном виде серверу[9].

Для упрощения задачи рекомендуется принять соглашение, что каждая команда занимает ровно одну строку. Учтите, что в зависимости от настройки утилиты telnet строка может заканчиваться одним символом `'\n'`, а может и двумя: `'\r' '\n'`. Проще всего игнорировать символ `'\r'` как пробельный.

Необходимо предусмотреть команды для следующих функций:

— **Получение информации о состоянии рынка**. В ответ на команду сервер должен выдать игроку количество и минимальную стоимость продаваемого банком на данном цикле сырья, а также количество и максимальную стоимость покупаемой продукции. Также разумно выдать количество активных (необанкротившихся) игроков.

— **Получение информации о состоянии дел других игроков**. Команда должна принимать один параметр (номер игрока) и в ответ на нее сервер должен выдать количество денег, единиц сырья, единиц продукции, строящихся, обыкновенных и автоматизированных фабрик у игрока с таким номером.

— **Производство продукции на фабриках**. По-видимому, при наличии у игрока фабрик разных типов заказ должен размещаться сначала на автоматизированных, и лишь затем на обыкновенных фабриках. В течение цикла (игрового месяца) команду можно дать несколько раз, добавляя новые заказы, однако собственно производство (списа-

[9]На самом деле, функциональность утилиты telnet гораздо более сложна, однако воспользоваться ею в рамках нашей задачи не представляется возможным.

ние единиц сырья и зачисление единиц продукции) должно произойти лишь в конце цикла. Это делается, чтобы исключить возможность продавать продукцию в том же месяце, в котором было закуплено для нее сырье. В качестве альтернативы можно запрашивать игроков о количестве производимой продукции один раз в начале цикла, не принимая никаких других команд, пока игрок не заявит о своих объемах производства. В этом случае производство выполняется немедленно.

- **Подача заявки на участие в аукционе сырья**. Команда имеет два параметра, количество закупаемого сырья и предлагаемая закупочная цена. Сервер должен сразу же проверить, достаточно ли у данного игрока единиц сырья и не задал ли игрок цену, ме́ньшую минимальной. Если игрок ошибся, ему необходимо об этом сообщить. В случае, если еще остались игроки, не подавшие заявку на аукцион сырья, следует сообщить пользователю их количество. В случае же, если данный игрок - последний участник аукциона, следует немедленно запустить механизм аукциона, списать с победителей аукциона соответствующие суммы денег, выдать проданные единицы продукции и разослать всем игрокам результаты аукциона.
- **Подача заявки на участие в аукционе готовой продукции**. Команда по своим характеристикам аналогична предыдущей.
- **Заявка на строительство новой фабрики** (обыкновенной или автоматической). С игрока тут же списывается половина стоимости фабрики и заводится строящаяся фабрика, которая в свой строк превратится в действующую (при этом произойдет списание второй половины суммы).
- **Заявка на реконструкцию обыкновенной фабрики**. С игрока списывается половина стоимости реконструкции и одна из обыкновенных фабрик переводится в состояние реконструкции.
- **Помощь**. По этой команде сервер должен выдать список существующих команд и их функции.

Команды должны быть достаточно мнемоничными, чтобы не запутаться в них. Весьма желательно сделать анализатор команд нечувствительным к количеству пробельных символов (пробелов, табуляций и возвратов каретки) между параметрами одной команды.

Также следует включить в сообщение о неопознанной команде информацию о том, как в вашей системе выглядит команда получения помощи.

4 Программируемый робот

Вторая часть практикума предусматривает создание программы-робота, имитирующего поведение человека (игрока) в игре «Менеджмент».

4.1 Постановка задачи

Стартовыми параметрами программы-робота являются ip-адрес и номер tcp-порта сервера, а также имя файла, содержащего программу на модельном языке (сценарий). Сценарий определяет дальнейшее поведение робота. Таким образом, программа-робот представляет собой комбинацию программы-клиента (см. § 2.5.1) и интерпретатора модельного языка.

Как и в случае с сервером, стартовые параметры задаются в командной строке сервера. По согласованию с преподавателем возможны другие варианты получения стартовых параметров, например, из конфигурационного файла или из переменных окружения. Задавать стартовые параметры в тексте программы (т.е. так, что их изменение потребует перекомпиляции программы) *запрещается*.

Входной язык робота должен позволять использование всей информации, которая доступна обычному игроку, а также выдачу всех команд, которые мог бы выдать обычный игрок. Язык должен быть алгоритмически полным и иметь возможности, достаточные для задания нетривиальных стратегий (например, включающих в себя статистическую экстраполяцию). Для этого, в частности, необходимо предусмотреть в языке массивы (хотя бы одномерные). Вместе с тем, нет необходимости реализовывать в языке строчные переменные или переменные разных числовых типов; достаточно будет наличия переменных одного целочисленного типа (например, четырехбайтных целых); предпочтительнее, однако, было бы наличие переменных с плавающей точкой.

Для обеспечения в языке алгоритмической полноты необходимо предусмотреть конструкции, позволяющие задать ветвление и цикл. Минимальный набор конструкций для этого состоит из условного оператора и оператора безусловного перехода. Желательно предусмотреть в языке также составной оператор и оператор цикла с предусловием.

Обязательным условием является равноправие всех пробельных символов (пробелов, табуляций, переводов строки и возвратов каретки) и допустимость любого их количества в любом месте программы, где допустим один пробел. Таким образом, нельзя использовать конец строки в качестве разделителя операторов, как это делается в ранних версиях Бейсика и Фортрана. Для разделения операторов рекомендуется использовать

символ “;” (точка с запятой).

4.2 Пример входного языка робота

Описываемый в этом параграфе язык является минимальным, т.е. упрощенным настолько, насколько это вообще возможно в рамках поставленной задачи. По требованию преподавателя на язык могут быть наложены дополнительные условия, усложняющие его. Однако даже в отсутствие дополнительных требований рекомендуется предложить свой вариант языка, используя приведенный здесь язык лишь в качестве примера.

4.2.1 Общее описание

Программа на модельном языке состоит из операторов, каждый из которых может быть помечен меткой. Для упрощения анализа имя метки начинается всегда с символа `@` и заканчивается двоеточием; имя метки может состоять из латинских букв, цифр и знаков подчеркивания. Если оператор начинается не с метки, считается, что этот оператор не имеет метки. Конец оператора обозначается точкой с запятой. В дальнейшем при описании синтаксиса операторов мы не упоминаем возможное наличие у оператора метки, чтобы не загромождать текст.

В языке присутствуют следующие операторы:

- оператор присваивания
- оператор безусловного перехода (`goto`)
- условный оператор (`if`)
- операторы игровых действий (`buy`, `sell`, `prod`, `build`, `upgrade` и `endturn`)
- оператор отладочной печати (`print`).

4.2.2 Арифметика. Выражения

В языке поддерживаются арифметические операции сложения (`+`), вычитания (`-`), умножения (`*`), целочисленного деления (`/`), вычисления остатка от деления (`%`), а также операции сравнения (`<`, `>`, `<=`, `>=`, `==`, `!=`) Обязательна поддержка унарного минуса. Операции сравнения выдают значение `1` для обозначения истины и `0` для обозначения лжи.

Наибольший приоритет имеют умножение, деление и остаток от деления, следующий уровень приоритета имеют сложение и вычитание, операции сравнения имеют низший приоритет.

В качестве операндов могут выступать константы, переменные и обращения к встроенным функциям.

В выражениях могут присутствовать круглые скобки любой вложенности.

4.2.3 Переменные. Массивы. Присваивания

Переменные в языке имеют имена, начинающиеся со знака "$". В имени переменной могут присутствовать латинские буквы, цифры и знак подчеркивания. После имени переменной может следовать указание индекса – произвольное арифметическое выражение, заключенное в квадратные скобки. Описывать переменные не требуется; переменная заносится в таблицу значений переменных в момент первого присваивания.

Оператор присваивания имеет следующий синтаксис:

```
<присваивание> ::= <переменная> = <выражение> ';'
<переменная> ::= <имя_переменной>          |
                 <имя_переменной> '[' <выражение> ']'
```

Можно заметить, что никакие другие операторы модельного языка не могут начинаться с переменной, поэтому переменная, встреченная при анализе в начале оператора, однозначно указывает на то, что анализируемый оператор является оператором присваивания.

Примеры операторов присваивания:

```
$a = 5;
$b[4] =15 ;
$b[$a]= $b[4]+3;
$c=($a+10) * $b[5];
```

Обратите внимание, что по обе стороны от знака присваивания допустимо любое количество пробельных символов.

Для упрощения реализации можно рассматривать элементы массива как самостоятельные переменные; при этом индекс становится частью имени переменной. С точки зрения интерпретатора элемент массива становится в таком случае своеобразной переменной, часть имени которой вычисляется в момент исполнения оператора. Если рассматривать только что приведенный пример, то после выполнения таких операторов в таблице значений переменных должны появиться:

1. переменная с именем “a” и значением 5;
2. переменная с именем “b[4]” и значением 15;
3. переменная с именем “b[5]” и значением 18;
4. переменная с именем “c” и значением 270.

4.2.4 Условный оператор

Условный оператор имеет следующий синтаксис:

```
<условный_оператор> ::= 'if' <выражение> 'then' <оператор> ';'
```

При анализе оператора следует считывать выражение до тех пор, пока не встретится лексема, не являющаяся знаком операции, константой, переменной или обращением к функции. В данном случае следующей за выражением должна оказаться лексема `then`. Оператор, стоящий после `then`, должен быть проанализирован, однако его выполнение должно произойти только в случае, если вычисление выражения дало результат, отличный от нуля.

4.2.5 Встроенные функции для получения игровой информации

Для удобства анализа можно ввести соглашение, по которому все имена функций начинаются со знака “?”. Имя функции может состоять из латинских букв, цифр и знака подчеркивания. Если функция имеет аргументы, то вслед за именем функции должно идти заключенное в круглые скобки перечисление параметров функции через запятую. В минимальный набор функций, обеспечивающий доступ ко всей игровой информации, входят:

- `?my_id` (без параметров) – выдает номер игрока, присвоенный сервером нашему роботу;
- `?turn` (без параметров) – выдает текущий номер хода (условного месяца);
- `?players` (без параметров) – выдает общее число игроков;
- `?active_players` (без параметров) – выдает количество игроков, продолжающих игру (т.е. не обанкротившихся и не вышедших из игры к настоящему моменту);

- `?supply` (без параметров) – выдает количество сырья, выставленное банком на продажу на текущий ход;

- `?raw_price` (без параметров) – выдает минимальную стоимость сырья, определенную банком на текущий ход;

- `?demand` (без параметров) – выдает количество продукции, которую банк намерен купить на текущем ходу;

- `?production_price` (без параметров) – выдает максимальную цену продукции, определенную банком на текущий ход;

- `?money` (один параметр, номер игрока) – выдает количество денег у заданного игрока (соответственно, `?money(?my_id)` выдаст количество денег у игрока, управляемого нашим роботом);

- `?raw` (один параметр, номер игрока) – выдает количество сырья у заданного игрока;

- `?production` (один параметр, номер игрока) – выдает количество готовой продукции у заданного игрока;

- `?factories` (один параметр, номер игрока) – выдает общее количество (работающих) фабрик у заданного игрока;

- `?auto_factories` (один параметр, номер игрока) – какое количество фабрик данного игрока являются автоматизированными;

- `?manufactured` (один параметр, номер игрока) – сколько единиц продукции произведено на фабриках данного игрока на **предыдущем** ходу;

- `?result_raw_sold` (один параметр, номер игрока) – сколько единиц продукции произведено на фабриках данного игрока на **предыдущем** ходу;

- `?result_raw_price` (один параметр, номер игрока) – если данный игрок покупал сырье на предыдущем ходу, выдает цену, по которой совершена покупка, в противном случае - нуль;

- `?result_prod_bought` (один параметр, номер игрока) – сколько единиц продукции банк купил у данного игрока на предыдущем ходу;

- `?result_prod_price` (один параметр, номер игрока) – если данный игрок продавал продукцию на предыдущем ходу, выдает цену, по которой совершена продажа, в противном случае - нуль.

4.2.6 Встроенные операторы для совершения игровых действий

Операторы для совершения игровых действий имеют следующий синтаксис:

```
<игровой_оператор> ::= <имя0> ';'                          |
                       <имя1> <операнд> ';'                |
                       <имя2> <операнд1> <операнд2> ';'
<имя0>             ::= 'endturn'
<имя1>             ::= 'prod' | 'upgrade'
<имя2>             ::= 'buy' | 'sell' | 'build'
```

Минимальный набор операторов включает:

- `buy <количество> <цена>` – выставить заявку на покупку заданного количества сырья по заданой цене;
- `sell <количество> <цена>` – выставить заявку на продажу заданного количества продукции по заданной цене;
- `prod <количество>` – запустить в производство заданное количество продукции;
- `build <количество> <автоматизир`>. – начать строительство заданного количества фабрик; если второй параметр отличен от нуля, строить автоматизированные фабрики, иначе – обыкновенные;
- `upgrade <количество>` – начать модернизацию заданного количества обыкновенных фабрик;
- `endturn` – сообщить серверу о завершении хода.

Во всех случаях операнды представляют собой произвольные допустимые арифметические выражения. Для проверки правильности синтаксиса используется символ “;”, обозначающий конец оператора.

4.2.7 Оператор отладочной печати

Оператор отладочной печати имеет следующий синтаксис:

```
<оператор_печати> ::= 'print' <п_список> ';'
<п_список>        ::= <п_элемент> | <п_элемент> <п_список>
<п_элемент>       ::= <выражение> | <строка>
```

Строка представляет собой произвольную строку символов, заключенных в двойные кавычки. Таким образом, список параметров оператора отладочной печати оказывается в рассматриваемом модельном языке единственным местом, где допустима текстовая константа.

Оператор отладочной печати обрабатывает свои параметры слева направо. Встреченные строки символов выдаются на стандартный вывод; встреченные выражения вычисляются и выдаются их результаты. Конец списка параметров определяется по встреченному символу ";".

4.2.8 Пример программы-сценария

Следующая программа задает простейшую стратегию игры: на каждом ходу пытаемся купить 2 единицы сырья по минимальной цене, продать всю имеющуюся продукцию по максимальной цене, и произвести столько продукции, сколько имеется сырья, но не больше двух (поскольку мы никогда не строим фабрики, ясно, что больше двух единиц мы произвести не сможем).

```
@begin:
  print "Это ход номер" ?turn
  buy 2 ?raw_price ;
  sell ?production(?my_id) ?production_price ;
  $toprod = 2;
  if ?raw(?my_id) < $toprod then
           $toprod = ?raw(?my_id) ;
  prod $toprod ;
  endturn ;
  goto @begin ;
```

Учтите, что эта программа-сценарий приведена исключительно для примера. На практике вам следует самостоятельно разработать сценарии для своих роботов, причем сценарии гораздо более сложные, нежели приведенный выше. Заметим, что даже для случая совсем простой программы-сценария следует, по-видимому, избегать ситуаций банкротства, насколько

это возможно, т.е. не подавать заявок на покупку сырья и на производство, если на выполнение заявки не хватает денег.

4.3 Рекомендации реализатору

4.3.1 Сетевая часть

По-видимому, установить соединение следует сразу после того, как завершен анализ входного файла со сценарием, до начала выполнения сценария (поскольку прагматика языка сценария предполагает, что соединение уже установлено).

Адрес и порт игрового сервера ваша программа должна получить в качестве стартовых параметров. Системные вызовы, необходимые для организации TCP-клиента, описаны в §2.5.1.

Если протокол работы с сервером организован таким образом, что сервер присылает какие-либо данные клиенту только в ответ на запрос и никак иначе, при реализации робота можно обойтись без вызова `select()`. Достаточно при интерпретаци скрипта, в случаях, когда необходимо обращение к серверу, передать данные серверу и сразу дать вызов `read()`, который заблокирует вашу программу до прихода данных с сервера. При этом следует иметь механизм, с помощью которого можно определить, закончил ли сервер передачу или пока передано не все. Можно предусмотреть в выдаче сервера какой-либо признак конца сообщения, например пустую строку.

В случае, если сервер может присылать клиенту какую-либо информацию по своей инициативе (например, сообщение о действиях других игроков), следует предусмотреть проверку готовности сокета к чтению. Это можно сделать, например, с помощью вызова `select()` с нулевым значением таймаута. Если давать такой вызов, например, после выполнения каждого оператора сценария, можно отследить момент, когда сервер прислал какие-либо данные, прочитать их из сокета, проанализировать, сохранить полученную информацию в локальных переменных и продолжить интерпретацию сценария.

4.3.2 Анализатор входного языка

Необходимые в нашей задаче методы анализа и интерпретации изложены в пособии [4], поэтому в настоящем издании мы ограничимся рекомендациями, специфичными для конкретной задачи.

Анализ текста сценария рекомендуется вести в два этапа; на первом этапе провести лексический анализ текста, т.е. представить текст в виде

списка лексем, на втором - провести синтаксический анализ и преобразовать список лексем во внутреннее представление, которое будет удобно интерпретировать (например, ПОЛИЗ).

Фаза лексического анализа может быть предельно упрощена благодаря особенностям входного языка. Например, для описанного в §4.2 языка существуют следующие лексемы:

- Разделители, а именно символы арифметических операций `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==` и `!=`; круглые и квадратные скобки; разделитель операторов - символ "`;`";
- Целочисленные константы (непрерывная последовательность цифр);
- Строковые константы (произвольная строка символов, заключенная в двойные кавычки);
- Имена переменных (имена, начинающиеся с `$`);
- Имена меток (имена, начинающиеся с `@`);
- Имена функций (имена, начинающиеся с `?`);
- Ключевые слова `if`, `then`, `goto`, `print`, `buy`, `sell`, `prod`, `build`, `upgrade` и `endturn`.

Во всех именах могут присутствовать только латинские буквы, цифры и знак подчеркивания, так что любой символ, не входящий в это множество, следует рассматривать как признак конца лексемы. Для разделения лексем могут использоваться пробельные символы; отделять лексемы-разделители от других лексем пробелами не обязательно (они являются разделителями сами по себе).

Поскольку имена разделены по типам еще на этапе лексического анализа, синтаксический анализ также оказывается достаточно простым; его можно провести методом рекурсивного спуска. О том, как это делается, можно узнать из пособия [4].

Если реализуемый вами язык отличается от описанного выше, возможно, что его анализ окажется более сложной задачей. В любом случае, постарайтесь сохранить синтаксис языка достаточно простым, чтобы можно было применить метод рекурсивного спуска.

4.3.3 Исполнитель

Результатом работы анализатора должно стать внутреннее представление сценария, удобное для дальнейшей интерпретации. В качестве такого представления мы рекомендуем использовать ПОЛИЗ (см. [4]).

Следует заметить, что элементы ПОЛИЗа относятся к разным типам, однако имеют и общие свойства; более того, они должны храниться в виде последовательной структуры данных (массива или списка), при этом обрабатываться схожими методами, работающими в зависимости от типа данного элемента ПОЛИЗа. Ясно, что понятие "*элемент полиза*" – еще один пример предметной области, для моделирования которой прекрасно подходит полиморфная иерархия классов.

Если указатель на следующий элемент ПОЛИЗа сделать полем класса "элемент ПОЛИЗа", а метки представлять указателями на соответствующий элемент списка (в отличие от значения индекса в массиве элементов, как это предлагается в пособии [4], то функция исполнения ПОЛИЗа может быть реализована в виде метода класса "элемент ПОЛИЗа".

5 Порядок выполнения задания

Задание выдается в начале семестра и рассчитано на выполнение в течение всего семестра.

Выполнение задания проводится в 4 этапа; по каждому этапу необходимо отчитаться о выполнении до установленной преподавателем даты.

5.1 Первый этап: серверно-сетевая часть

5.1.1 Постановка задачи

Целью первого этапа является построение многопользовательского сервера, поддерживающего общее информационное пространство. Следует отметить, что архитектура программы-сервера формируется именно на этом этапе.

Для демонстрации наличия общего информационного пространства достаточно завести в программе некий счетчик, один для всего сервера. Командный интерфейс может состоять всего из двух команд. Первая из них должна позволять узнать текущее значение глобального счетчика, а также общее количество клиентов, подключенных к серверу в настоящий момент. Вторая команда используется для увеличения значения счетчика.

5.1.2 Указания

Прежде всего, необходимо организовать **главный цикл**, как это описано в §2.3.2. Разумеется, для этого понадобится создать структуры данных для хранения списка активных клиентов. Если вы решили хранить дескриптор сокета в классе “игрок”, как это предлагается в §3.1, будет вполне осмысленно описать класс “игрок” уже на первом этапе, реализовав в нем только ту функциональноть, которая для первого этапа необходима; а именно, следует реализовать способность объекта класса “игрок” сохранять дескриптор сокета и поддерживать буфер временного хранения для команд, полученных не полностью.

Также стоит, по-видимому, создать класс “ведущий”, разместив именно в нем глобальный счетчик. Никакой иной функциональности на первом этапе от класса “ведущий” не требуется, однако можно возложить на него, например, функции поддержки списка игроков.

5.1.3 Правила тестирования

Проверку программы следует произвести в несколько этапов (тестов). На каждом этапе вам потребуется несколько работающих командных интерпретаторов. Если вы используете систему XWindows, запустите несколько экземпляров программы `xterm`. Если вы предпочитаете работать с текстовой консолью, выполните вход в систему на нескольких виртуальных консолях одновременно[10].

1. Запустите программу-сервер. Убедитесь с помощью команды `netstat -an`, что выбранный вами TCP-порт находится в состоянии LISTEN. Если это так, в выдаче команды будет присутствовать примерно такая строка:

   ```
   tcp  0  0   0.0.0.0:7000  0.0.0.0:*  LISTEN
   ```

   (здесь 7000 – избранный вами номер порта).

2. Прервите выполнение программы-сервера и запустите ее с другим номером порта (не забудьте, что порты с номерами 1..1023 для пользовательских процессов не доступны). Убедитесь, что теперь в состоянии LISTEN находится новый TCP-порт.

3. Используя другой командный интерпретатор (в другом окне xterm'a или на другой консоли), запустите утилиту telnet. В команой строке необходимо указать адрес и TCP-порт сервера. Если тестирование проводится в рамках одной машины, в качестве IP-адреса можно использовать 127.0.0.1 (localhost) или 0. Например, если вы используете порт 7000, команда будет выглядеть так:

   ```
   telnet 127.0.0.1 7000
   ```

   Убедитесь, что связь установлена. Для этого запустите команду `netstat -an`. Если связь действительно установлена, в ее выдаче будут присутствовать примерно такие строки:

   ```
   tcp  0  0   127.0.0.1:7000  127.0.0.1:6537  ESTABLISHED
   tcp  0  0   127.0.0.1:6537  127.0.0.1:7000  ESTABLISHED
   ```

[10] В системах Linux и FreeBSD переключение виртуальных консолей производится обычно комбинацией Alt-Fn, где n - номер консоли (1, 2, 3, ..., 12). Учтите, что не все консоли могут быть доступны.

здесь 7000 - номер порта вашего сервера, а 6537 - номер порта, используемого программой telnet (может оказаться любым числом, большим 1023).

4. Разорвите связь с сервером. Для этого нажмите комбинацию `Ctrl-]`; должно появиться приглашение `telnet>`. Введите команду close и нажмите Enter.

5. Убедитесь, что сервер по-прежнему слушает порт, как это описано на шаге 1.

6. Прервите выполнение программы-сервера и запустите ее вновь. Установите связь.

7. Дайте команду чтения параметров. В ответ сервер должен сообщить, что к нему подключен один клиент и значение глобального счетчика равно нулю.

8. Несколько раз (например, трижды) дайте команду на увеличение счетчика. Убедитесь, что команда чтения параметров выдает правильное значение глобального счетчика.

9. Разорвите связь, убедитесь, что сервер по-прежнему слушает порт. Вновь установите связь. Убедитесь, что глобальный счетчик не изменился, а число клиентов по-прежнему равно 1.

10. Не разрывая связь, подключите к серверу еще несколько клиентов, запустив несколько экземпляров программы telnet. Убедитесь, что сервер отвечает на команды всех клиентов. Убедитесь, что число клиентов, выдаваемое сервером, соответствует числу реально подключенных клиентов.

11. Дайте несколько раз команду на увеличение счетчика в одном клиенте, а команду на выдачу значений – в другом. Убедитесь, что значение счетчика соответствующим образом меняется.

12. Разорвите связь с каждым из клиентов вразброс, начав с тех, что были запущены в середине (т.е. не с первого и не с последнего). После отключения каждого клиента проверяйте корректность выдаваемого сервером количества клиентов.

13. После того, как все клиенты будут отключены, убедитесь, что сервер по-прежнему слушает порт. Попытайтесь подключиться к нему еще раз.

14. Не разрывая связь с клиентом, прервите выполнение программы-сервера (при этом telnet сообщит о потере связи). Попробуйте снова запустить программу-сервер с тем же номером порта. Если это не удастся, обратитесь к §2.6.2.

Отчет

По требованию преподавателя подготовьте письменный отчет, в который включите заголовки используемых в вашей программе классов и комментарий к классам и их методам.

5.2 Второй этап: игровой сервер

На втором этапе должен быть полностью реализован игровой сервер. Тестирование игрового сервера следует проводит в микрогруппах по 3-4 человека. Участники группы играют несколько многопользовательских партий с использованием сервера очередного участника.

Предусмотрите в программе-сервере генерацию отчета по каждому ходу, включающего список поданных на аукционы заявок с индикацией об их удовлетворении/неудовлетворении, количество произведенной каждым игроком продукции и остаток средств на счету каждого игрока на момент конца хода. Информацию выдавайте в поток стандартного вывода сервера.

Отчет

По требованию преподавателя подготовьте отчет, включающий протокол игры. Для этого перенаправьте вывод вашего сервера в файл и распечатайте его.

5.3 Третий этап: лексический и синтаксический анализатор входного языка робота

На третьем этапе вам необходимо окончательно определить, какой входной язык будет обрабатывать ваш робот. Для этого языка следует создать правила лексического анализа, синтаксические диаграммы и краткое описание семантики.

Программа, соответствующая требованиям третьего этапа, должна определять, соответствует ли данный текст требованиям языка. В случае обнаружения лексических или синтаксических ошибок ваша программа

обязательно должна выдавать информативные диагностические сообщения с указанием номера строки исходного файла, в которой обнаружена ошибка.

Для тестирования напишите текст, соответствующий требованиям вашего языка и включающий примеры на все имеющиеся в нем синтаксические конструкции. Для языка, описанного в §4.2, такой пример должен включать все операторы, несколько арифметических выражений различной сложности, как с переменными, так и без них, несколько вызовов различных функций.

Убедитесь, что программа корректно воспринимает ваш текст. Затем, внося различные некорректные изменения в текст примера, убедитесь, что всякий раз выдается информативная диагностика с указанием правильного местонахождения ошибки.

Отчет

По требованию преподавателя подготовьте отчет, включающий описание синтаксиса и семантики вашего варианта входного языка, краткое описание используемых для анализа алгоритмов и структур данных, а также список диагностических сообщений, выдаваемых вашим анализатором.

5.4 Четвертый этап: работающие роботы (задание целиком)

Для тестирования создайте по меньшей мере два осмысленных сценария для своего робота, причем по меньшей мере один из них должен быть нетривиальным, т.е. обучаться на своих ошибках, строить экстраполяции на будущее и т.п.

Проведите несколько игровых партий между роботами, как одинаковыми, так и различными. Проведите несколько партий с участием роботов и "живых" игроков. Убедитесь, что работа как серверной части, так и роботов соответствует заданию.

Отчет

По требованию преподавателя подготовьте отчет, в который включите исходные тексты ваших роботов (тексты на предложенном вами языке программирования) и описание алгоритмов и принципов их работы. Приложите к отчету протокол игры между роботами.

A Инструментарий

В этом приложении приведена основная информация об использовании наиболее важных компонентов системы программирования. Все эти компоненты уже знакомы читателю по лекционному курсу и практическим занятиям III семестра, однако некоторые особенности их использования, которые потребуются при выполнении практикума, будет нелишне повторно освежить в памяти.

A.1 Компилятор gcc/g++

Компиляторы семейства GCC (Gnu Compiler Collection) являются компиляторами командной строки, т.е. все необходимые действия задаются при запуске компилятора и выполняются уже без непосредственного участия пользователя. Это, в частности, позволяет использовать компилятор в командных файлах (скриптах).

Команда `gcc` предназначена для компиляции программ на языке C, а команда `g++` – на языке C++[11].

Имена файлов, подлежащих компиляции и линковке, компилятор принимает с командной строки. Кроме того, компилятор воспринимает большое количество опций. Вам обязательно понадобятся следующие из них:

- `-o` <*filename*> задает имя исполняемого файла, в который будет записан результат компиляции (если не указать эту опцию, результат компиляции будет помещен в файл `a.out`.).

- `-Wall` приказывает компилятору выдавать все разумные предупредительные сообщения (warnings). **Обязательно всегда используйте эту опцию**, она поможет вам сэкономить немало времени и нервов.

- `-ggdb` и `-g` используются для включения в результирующие файлы разнообразной отладочной информации (информации, используемой отладчиком, включая имена переменных и функций, номера строк исходных файлов и т.п.). Опция `-ggdb` снабжает файлы расширенной отладочной информацией, понятной только отладчику `gdb`. Если вам кажется, что что-то не в порядке с отладчиком, попробуйте использовать опцию `-g`.

[11]На самом деле, используется один и тот же компилятор; оба имени являются обычно символическими ссылками на исполняемый файл компилятора. Поведение компилятора зависит от того, по какому имени его вызвали.

- `-c` указывает компилятору, что результатом должна быть не вся программа, а отдельный ее модуль. В этом случае имя файла для объектного модуля можно не задавать, оно будет сгенерировано автоматически заменой расширения на `.o`.

- `-On` задает уровень оптимизации. `n=0` означает отсутствие оптимизации (значение по умолчанию). Для получения более эффективного объектного кода рекомендуется использовать опцию `-O2`. Учтите, что оптимизация может затруднить работу с отладчиком.

- `-E` останавливает компилятор после проведения стадии макропроцессирования. Реультат макропроцессирования выдается на стандартный вывод. Эта опция может быть полезна, если ваши макроопределения повели себя не так, как вы ожидали, и хочется понять, что на самом деле происходит.

- `-MM` анализирует заданные исходные файлы и строит информацию об их взаимозависимостях. О том, как использовать полученную информацию, рассказывается в §A.3.

Итк, чтобы откомпилировать программу, написанную на языке C++ и целиком находящуюся в файле `prog.cpp`, следует дать команду

```
g++ -g -Wall prog.cpp -o prog
```

При этом результат компиляции будет помещен в файл `prog` в текущей директории.

Чтобы откомпилировать программу, состоящую из нескольких модулей `mod1.cpp`, `mod2.cpp`, `mod3.cpp` и главного файла `prog.cpp`, следует сначала откомпилировать все модули:

```
g++ -g -Wall -c mod1.cpp
g++ -g -Wall -c mod2.cpp
g++ -g -Wall -c mod3.cpp
```

и получить объектные файлы `mod1.o`, `mod2.o`, `mod3.o`. После этого для компиляции основного файла и сборки готовой программы следует дать команду

```
g++ -g -Wall mod1.o mod2.o mod3.o prog.cpp -o prog
```

A.2 Отладчик gdb

Отладчик `gdb` (Gnu DeBugger) позволяет отлаживать программу в интерактивном режиме, пользуясь интерфейсом командной строки, а также анализировать причины "смерти" программы по созданному системой core-файлу.

Учтите, что для нормальной работы отладчика **необходимо**, чтобы все модули вашей программы были откомпилированы с ключем `-ggdb` или `-g` (см. §A.1).

A.2.1 Пошаговое выполнение программы

Чтобы запустить отладчик для отладки программы, исполняемый файл которой называется `prog`, следует дать команду

```
gdb prog
```

Отладчик сообщит свою версию и некоторую другую информацию, после чего выдаст приглашение своей командной строки, обычно выглядящее так: `(gdb)`.

Основные команды отладчика:

- `run` осуществляет запуск программы в отладочном режиме. Перед запуском целесообразно задать точки останова (см. ниже). Если вы затрудняетесь определить, где именно следует приостановить выполнение программы, поставьте точку останова на функцию `main()`.

- `list` показывает на экране несколько строк программы, предшествующих текущей и идущих непосредственно после текущей.

- `break` позволяет задать точку приостановки выполнения программы (breakpoint). Точка останова может быть задана именем функции, номером строки в текущем файле, либо выражением <имя-файла>:<номер-строки>, например `file1.cpp:73`.

- `inspect` позволяет просмотреть значение переменной (в том числе и заданной сложным выражением вроде `*(a[i+1].p)`).

- `backtrace` или `bt` показывает текущее содержимое стека, что позволяет узнать последовательность вызовов функций, приведшую к текущему состоянию программы.

- `frame` позволяет сделать текущим один из фреймов, показанных командой `backtrace`, что дает возможность исследовать значения переменных в этом фрейме и т.п.

- `step` позволяет выполнить одну строку программы. Если в строке содержится вызов функции, текущей строкой станет первая строка этой функции (т.е. процесс трассировки зайдет внутрь функции).

- `next` подобна команде `step`, с тем отличием, что вход в тела вызываемых функций не производится.

- `until` <номер-строки> позволяет выполнять программу до тех пор, пока текущей не окажется строка с указанным номером.

- `call` позволяет выполнить вызов произвольной функции.

- `cont` позволяет продолжить прерванное выполнение программы.

- `help` позволит узнать подробнее об этих и других командах отладчика.

A.2.2 Анализ причин аварийного завершение по core-файлу

Часто ошибки в программе приводят к ее аварийному завершению, при котором система создает так называемый `core`-файл. При этом выдается сообщение

```
Segmentation fault (core dumped)
```

Это означает, что в текущей директории аварийно завершенного процесса создан файл с именем `core` (или `prog.core`, где `prog` – имя вашей программы), в который система записала содержимое сегмента данных и стека программы на момент ее аварийного завершения.

С помощью отладчика gdb можно проанализировать этот файл, узнав, в частности, при выполнении какой строки программы произошла авария, откуда и с какими параметрами была вызвана функция, содержащая эту строку, каковы были значения переменных на момент аварии и т.д.

Чтобы запустить отладчик в режиме анализа core-файла, необходимо дать команду:

```
gdb prog prog.core
```

где `prog` – имя исполняемого файла вашей программы, а `prog.core` – имя созданного системой core-файла. Очень важно при этом, чтобы в качестве исполняемого файла выступал именно тот файл, при исполнении которого получен core-файл. Так, если уже после получения core-файла вы перекомпилируете свою программу, анализ core-файла, скорее всего, приведет к ошибочным результатам.

Сразу после запуска отладчика в режиме анализа core-файла рекомендуется дать команду `backtrace` (или просто `bt`).

A.2.3 Анализ причин зацикливания

Если ваш процесс зациклился, не торопитесь его убивать. С помощью отладчика можно понять, какой фрагмент кода выполняется в настоящий момент, и проанализировать причины зацикливания. Для этого нужно *присоединить* отладчик к существующему процессу. Определите номер процесса с помощью команды `ps`. Допустим, имя исполняемого файла вашей программы – `prog`, и она выполняется как процесс номер 12345. Тогда команда запуска отладчика должна выглядеть так:

```
gdb prog 12345
```

При подключении отладчика выполнение программы будет приостановлено, однако вы сможете при необходимости продолжить его командой `cont`. В случае повторного зацикливания можно приказать отладчику вновь приостановить выполнение нажатием комбинации `Ctrl-C`.

A.3 Утилита make

При сборке ваших программ могут оказаться полезны возможности, предоставляемые утилитой make. Кратко говоря, эта утилита позволяет автоматически строить одни файлы на основании других (например, исполняемые файлы на основании исходных текстов программы) в соответствии с заданными правилами. При этом make отслеживает даты последней модификации файлов и производит перестроение только тех целевых файлов, для которых исходные файлы претерпели изменения.

Правила для утилиты make задаются в файле `Makefile`, который утилита ищет в текущей директории.

A.3.1 Простейший Makefile

Допустим, ваша программа состоит из главного модуля `main.cpp`, содержащего функцию `main()`, а также из дополнительных модулей `mod1.cpp` и

mod2.cpp, имеющих заголовочные файлы mod1.hpp и mod2.hpp. Соответственно, для сборки исполняемого файла (назовем его prog) необходимо дать следующие команды:

```
g++ -g -Wall -c mod1.c -o mod1.o
g++ -g -Wall -c mod2.c -o mod2.o
g++ -g -Wall main.c mod1.o mod2.o -o prog
```

Первые две команды даются для компиляции дополнительных модулей. Полученные в результате файлы mod1.o и mod2.o используются в третьей команде для сборки исполняемого файла.

Допустим, мы уже произвели компиляцию программы, после чего внесли изменения в файл mod1.cpp и хотим получить исполняемый файл с учетом внесенных изменений. При этом нам надо будет дать только две команды (первую и третью), поскольку перекомпиляции модуля mod2 не требуется.

Чтобы подобные ситуации отслеживались автоматически, мы можем использовать утилиту make. Для этого напишем следующий Makefile:

```
mod1.o: mod1.cpp mod1.hpp
        g++ -g -Wall -c mod1.cpp -o mod1.o

mod2.o: mod2.cpp mod2.hpp
        g++ -g -Wall -c mod2.cpp -o mod2.o

prog: main.cpp mod1.o mod2.o
        g++ -g -Wall main.cpp mod1.o mod2.o -o prog
```

Поясним, что файл состоит из так называемых *целей* (в нашем случае таких целей три - mod1.o, mod2.o и prog). Описание каждой цели состоит из заголовка и списка команд. Заголовок цели – это **одна** строка, начинающаяся всегда с первой позиции (т.е. перед ней не допускаются пробелы и т.п.). В начале строки пишется имя цели (обычно это просто имя файла, который необходимо построить). Оставшаяся часть заголовка отделяется от имени цели двоеточием. После двоеточия перечисляется, от каких файлов (или, в более общем случае, от каких целей) зависит построение файла. В данном случае мы указали, что модули зависят от их исходных текстов и заголовочных файлов, а исполняемый файл – от основного исходного файла и от двух объектных файлов.

После строки заголовка идет список команд (в нашем случае все три списка имеют по одной команде). Строка команды **всегда начинается с символа табуляции**, причем замена табуляции пробелами недопустима

и ведет к ошибке. Утилита `make` считает признаком конца списка команд первую строку, начинающуюся с символа, отличного от табуляции.

Имея в текущей директории вышеописанный Makefile, мы можем для сборки нашей программы дать команду `make prog`.

A.3.2 Переменные

В предыдущем параграфе описан Makefile, в котором можно обнаружить несколько повторяющихся фрагментов. Так, строка параметров компилятора "`-g -Wall`" встречается во всех трех целях. Помимо необходимости повторения одного и того же текста, мы можем столкнуться с проблемами при модификации. Предположим, нам понадобится задать компилятору режим оптимизации кода (флаг `-O2`). Для этого нам пришлось бы внести совершенно одинаковые изменения в три разный строки файла. В более сложном случае таких строк может понадобиться несколько десятков и даже сотен.

Аналогичная проблема встанет, например, если мы захотим произвести сборку другим компилятором.

Решить проблему позволяет введение *make-переменных*. Обозначим имя компилятора C++ переменной `CXX`, а общие параметры компиляции – переменной `CXXFLAGS`[12].

Тогда наш Makefile можно переписать следующим образом:

```
CXX = g++
CXXFLAGS = -g -Wall

mod1.o: mod1.cpp mod1.hpp
        $(CXX) $(CXXFLAGS) -c mod1.cpp -o mod1.o

mod2.o: mod2.cpp mod2.hpp
        $(CXX) $(CXXFLAGS) -c mod2.cpp -o mod2.o

prog: main.cpp mod1.o mod2.o
        $(CXX) $(CXXFLAGS) main.cpp mod1.o mod2.o -o prog
```

A.3.3 Предопределенные переменные и псевдопеременные

Существуют определенные соглашения об именах переменных, причем некоторым переменным утилита `make` присваивает значения сама, если соответствующие значения не заданы явно.

[12]Причины выбора именно таких обозначений станут ясны из дальнейшего изложения.

Вот некоторые традиционные имена переменных:

- `CC` – команда вызова компилятора языка C;
- `CFLAGS` – параметры для компилятора языка C;
- `CXX` – команда вызова компилятора языка C++;
- `CXXFLAGS` – параметры для компилятора языка C++;
- `CPPFLAGS` – параметры препроцессора (обычно сюда помещают предопределенные макропеременные);
- `LD` – команда вызова системного линкера (редактора связей);
- `MAKE` – команда вызова утилиты `make` со всеми параметрами.

По умолчанию переменные `CC`, `CXX`, `LD` и `MAKE` имеют соответствующие значения, справедливые для данной системы и в данной ситуации. Значения остальных перечисленных переменных по умолчанию пусты.

Таким образом, при написании Makefile из предыдущего параграфа мы могли бы пропустить строку, в которой задается значение переменной `CXX`, в надежде, что соответствующее значение переменная получит без нашей помощи.

Кроме таких переменных общего назначения, в каждой цели могут использоваться так называемые *псевдопеременные*. Перечислим наиболее интересные из них:

- `$@` – имя текущей цели;
- `$<` – имя первой цели из списка зависимостей;
- `$^` – весь список зависимостей.

С использованием этих переменных мы можем переписать наш Makefile следующим образом:

```
CXXFLAGS = -g -Wall

mod1.o: mod1.cpp mod1.hpp
        $(CXX) $(CXXFLAGS) -c $< -o $@

mod2.o: mod2.cpp mod2.hpp
        $(CXX) $(CXXFLAGS) -c $< -o $@

prog: main.cpp mod1.o mod2.o
        $(CXX) $(CXXFLAGS) $^ -o $@
```

A.3.4 Обобщенные цели

Как можно заметить, в том варианте Makefile, который мы написали в конце предыдущего параграфа, правила для сборки обоих дополнительных модулей оказались совершенно одинаковыми. Можно пойти дальше и задать одно обобщенное правило для построения объектного файла по для любого модуля, написанного на языке C++, исходный файл которого имеет имя с суффиксом `.cpp`, а заголовочный файл – имя с суффиксом `.hpp`:

```
%.o: %.cpp %.hpp
        $(CXX) $(CXXFLAGS) -c $< -o $@
```

Если теперь задать список дополнительных модулей с помощью переменной, получим следующий вариант Makefile:

```
OBJMODULES = mod1.o mod2.o
CXXFLAGS = -g -Wall

%.o: %.cpp %.hpp
        $(CXX) $(CXXFLAGS) -c $< -o $@

prog: main.cpp $(OBJMODULES)
        $(CXX) $(CXXFLAGS) $^ -o $@
```

Теперь для добавления к программе нового модуля нам достаточно добавить имя его объектного файла к значению переменной `MODULES`.

Если перечисление модулей через имена объектных файлов представляется неестественным, можно заменить первую строку Makefile следующими двумя строками:

```
SRCMODULES = mod1.cpp mod2.cpp
OBJMODULES = $(SRCMODULES:.cpp=.o)
```

Запись `$(SRCMODULES:.cpp=.o)` означает, что необходимо взять значение переменной `SRCMODULES` и в каждом входящем в это значение слове заменить суффикс `.cpp` на `.o`.

A.3.5 Псевдоцели

Утилиту `make` можно использовать не только для построения файлов, но и для выполнения, вообще говоря, произвольных действий. Добавим к нашему файлу две дополнительные цели:

```
run: prog
	./prog

clean:
	rm -f *.o prog
```

Теперь по команде `make run` утилита `make` произведет, если необходимо, сборку нашей программы и запустит ее. С помощью же команды `make clean` мы можем очистить рабочую директорию от объектных и исполняемых файлов (например, если нам понадобится произвести сборку программы с нуля).

Такие цели обычно называют *псевдоцелями*, поскольку их имена не обозначают имен создаваемых файлов.

A.3.6 Автоматическое отслеживание зависимостей

В более сложых проектах модули могут использовать заголовочные файлы других модулей, что делает необходимой перекомпиляцию модуля при изменении заголовочного файла, не относящегося к этому модулю. Информацию о том, какой модуль от каких файлов зависит, можно задать вручную, однако этот способ приведет к трудностям в больших программах, поскольку программист при модификации исходных файлов может случайно забыть внести изменения в Makefile.

Более правильным решением будет поручить отслеживание зависимостей компьютеру. Утилита `make` позволяет наряду с обобщенным правилом указать список зависимостей для построения конкретных модулей. Например:

```
%.o: %.cpp
	$(CXX) $(CXXFLAGS) -c $< -o $@

mod1.o: mod1.cpp mod1.hpp mod2.hpp mod3.hpp
```

Списки зависимостей можно построить с помощью компилятора. Чтобы получить строку, подобную последней строке вышеприведенного примера, необходимо дать команду

```
	g++ -MM mod1.cpp
```

Если результат выполнения такой команды перенаправить в файл, то этот файл можно будет включить в наш Makefile директивой `include`. Более того, если перед директивой `include` поставить знак “`-`” и предусмотреть

цель для генерации файла зависимостей, утилита `make`, прежде чем начать построение любых других целей, будет пытаться построить включаемый файл.

Назовем файл зависимостей `deps.mk`. Окончательно Makefile будет выглядеть так:

```
SRCMODULES = mod1.cpp mod2.cpp
OBJMODULES = $(SRCMODULES:.cpp=.o)
CXXFLAGS = -g -Wall
-include deps.mk

%.o: %.cpp %.hpp
	$(CXX) $(CXXFLAGS) -c $< -o $@

prog: main.cpp $(OBJMODULES)
	$(CXX) $(CXXFLAGS) $^ -o $@

run: prog
	./prog

clean:
	rm -f *.o prog

deps.mk: $(SRCMODULES)
	$(CXX) -MM $^ > $@
```

Список литературы

[1] Ч. Уэзерелл. Этюды для программистов. *пер. с английского.* М.:Мир, 1982.

[2] А. М. Робачевский. Операционная система UNIX. Изд-во «BHV–Санкт-Петербург», Санкт-Петербург, 1997.

[3] Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. М.:Изд-во «Бином», 1999.

[4] И. А. Волкова, Т. В. Руденко. Формальные грамматики и языки. Элементы теории трансляции. Второе издание. М.: Издательский отдел ВМиК МГУ, 1999.

Содержание