ББК 681.142.2
В 19

Рецензент: С.М. Семенов, канд. техн. наук,
зав. кафедрой ИСКТ

Васильев Б.К.

В 19 ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ ГОТОВЫХ РЕШЕНИЙ В ПРОГРАММИРОВАНИИ: Лабораторный практикум. – Владивосток: Изд-во ВГУЭС, 2004. – 44 с.

В практикуме рассмотрены типовые лабораторные работы курса «Технология программирования».

Предназначен для студентов, обучающихся по специальности 220100 «Вычислительные машины, системы, комплексы и сети» для очной и заочной форм обучения.

Печатается по решению РИСО ВГУЭС

ББК 681.142.2

ВВЕДЕНИЕ

Разработка программ давно превратилась из кустарного ремесла в промышленное производство, оставаясь к тому же составной частью науки программирования, которую относят то к искусству [Кнут], то к разновидности фольклора [Турский].

Современные программные продукты разрабатываются, как правило, по установившимся стандартам, а заняты в этом производстве многие разработчики. В лабораторных работах курса «Технология программирования» мы имеем дело с такими аспектами технологии этого производства, как разработка в коллективе, тестирование программ, создание графического пользовательского интерфейса.

Наряду с рассмотрением практических аспектов разработки программных продуктов рассмотрим темы лабораторных работ и предпочтительную последовательность действий при их выполнении.

Отметим, что в целом при выполнении лабораторных работ применяется технология использования готовых программных решений (алгоритмических, технологических, в области разработки графического интерфейса конечного пользователя) и реинженеринга, т.е. повторного использования написанного ранее кода.

1. ПРОГРАММНЫЙ ПРОЕКТ

Собственно говоря, как и любой другой проект, программный проект (ПП) имеет определенные рамки, прежде всего это рамки – временные. Срок выполнения проекта можно определить, если произвести анализ необходимых работ и произвести разбиение проекта на законченные отдельные виды деятельности. К ним относятся не только алгоритмизация и кодирование, но и освоение новых программных продуктов, приобретение необходимых программных и аппаратных средств, тестирование и верификация.

Каждый вид деятельности в ПП характеризуется временной оценкой (сроком выполнения), одни виды деятельности не могут быть начаты прежде чем будут выполнены другие (существует зависимость видов деятельности). Некоторые виды деятельности могут выполняться параллельно различными людьми, а некоторые – нет.

При выполнении декомпозиции ПП по видам деятельности удобно все работы представить в виде ориентированного графа, вершинами которого являются этапы выполнения работ (от вершины СТАРТ до вершины КОНЕЦ), а ребрами – выполняемые работы.

Такой граф называют сетевым графиком проекта.

Анализ сетевого графика

Поскольку каждому ребру приписана числовая характеристика – время выполнения, то можно ставить вопрос о нахождении критического пути на данном графе, то есть пути от вершины СТАРТ до вершины КОНЕЦ, доставляющего максимум сумме величин t_j^i, приписанных ребрам, входящим в этот путь.

Находить критический путь можно многими способами. Достаточно хорошо работающим даже на графах с большим количеством ребер является алгоритм динамического программирования, основанный на представлении процесса нахождения решения в виде этапов (шагов) и запоминании результатов предыдущих шагов [Кузин, гл.14].

Составим функциональное уравнение Беллмана:

$$V_i = \max(t_j^i + v_j), i = 1,2,...n-1;\ j = 1,2,...n;$$

$$V_n = 0,$$

где величины Vi (i=1, 2, … n-1) являются критическими временами частичных путей от i-й до конечной вершины. Вычисления начнем с конца, полагая

$$V_i^1 = t_{in}, i = 1,2,...n-1;$$

$$V_n^1 = t_{nn} = 0.$$

На втором шаге

$$V_i^2 = \max(t_{ij} + V_j^1), i = 1,2,...n-1;\ j = 1,2,...n.$$

Дальнейшие вычисления выполняются по рекуррентной формуле

$$V_i^k = \max(t_{ij} + V_j^{k-1}), i = 1,2,...n-1;\ j = 1,2,...n; k > 1.$$

$$V_n^k = 0; \quad t_{ij} = 0.$$

Вычисления закончатся, когда $V_i^k = V_i^{k-1}, i = 1,2,...n.$

Рассмотрим пример решения для следующего графа (рис. 1):

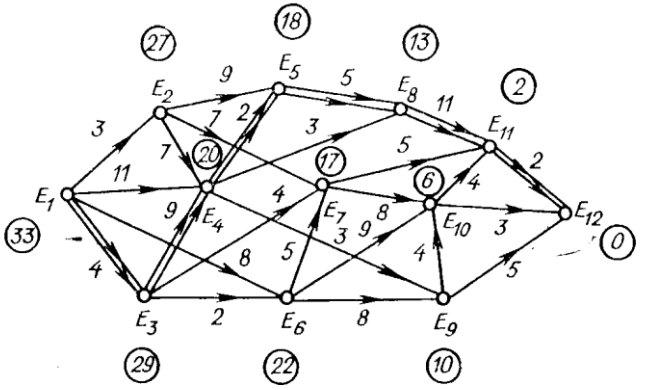

Рис. 1. Пример сетевого графика (Е1=СТАРТ, Е12=КОНЕЦ)

При решении значения V_k^i будем заносить в табл. 1, а в те элементы таблицы, для которых отсутствует переход, будем заносить большое отрицательное число (-∞).

Таблица 1

Значения V_k^i

k	i											
	1	2	3	4	5	6	7	8	9	10	11	12
1	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	5	3	2	0
2	-∞	-∞	-∞	8	-∞	13	11	13	7	6	2	0
3	20	18	17	16	18	16	17	13	10	6	2	0
4	27	27	25	20	18	22	17	13	10	6	2	0
5	31	27	29	20	18	22	17	13	10	6	2	0
6	33	27	29	20	18	22	17	13	10	6	2	0
7	33	27	29	20	18	22	17	13	10	6	2	0

Лабораторная работа 1
Сетевой график

Требования

Четыре часа на выполнение, язык реализации – C или C++.

Алгоритмическая реализация

Смотри ранее рассмотренный анализ сетевого графика.

Реализация

Диалог в консольном приложении.

Написать программу для нахождения критического пути на графе, имея в виду предметную область – анализ времени выполнения программного продукта. Ввод данных в диалоговом режиме и из файла, предусмотреть сохранение файла данных для последующего использования. Использовать для внутреннего представления графа матрицу смежности [Ахо и др.].

Написание программы

При написании программы тщательно продумайте используемую структуру данных (граф). Проверьте, как сохраняется введенный граф, как читаются ранее записанные в файл данные. Для удобства отладки реализуйте вывод таблицы после каждого шага алгоритма.

2. ТЕСТИРОВАНИЕ ПРОГРАММ

Основное внимание в следующих работах необходимо уделить тестированию программ и разработке квалифицированных тестов.

Лабораторная работа 2
Пересекаются ли отрезки?

Написать функцию для проверки, пересекаются ли два отрезка, заданные координатами своих концов. Написать программу, проверяющую работоспособность этой функции. Функция должна быть мобильной и робастной, писаться с учетом последующего многократного использования. Текст программы должен соответствовать основным критериям читаемости программы [Голуб].

Требования

Четыре часа на выполнение, язык реализации – C или C++.

Работа выполняется в несколько этапов. Особое внимание необходимо обратить на реализацию функции, которая должна обрабатывать все возможные особые случаи.

Первый этап – алгоритмизация задачи

Задача, естественно, планиметрическая, оба отрезка принадлежат одной плоскости. Пусть первый отрезок соединяет точку P1 = (x1, y1) с точкой P2 = (x2, y2), а второй отрезок – точку P3 = (x3, y3) с точкой P4 = (x4, y4).

Будем также пока считать, что никакие три точки из них не лежат на одной прямой (для упрощения).

Знак определителя $\det(P1,P2,P3)=\begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix}$ устанавливает, какая полуплоскость линии L1, образованной продолжением отрезка P1-P2, содержит точку P3. Отрезок P3-P4 будет пересекать L1 в том случае, если точки P3 и P4 находятся по разные стороны от L1, при этом

$$\det(P1,P2,P3)\times\det(P1,P2,P4)<0\,.$$

Для того чтобы точка пересечения принадлежала отрезку P1-P2, также необходимо, чтобы

$$\det(P3,P4,P1)\times\det(P3,P4,P2)<0\,.$$

Мы можем вычислить определители через миноры последнего столбца, если предварительно вычислим шесть определителей второго порядка:

```
x1*y2-x2*y1      x1*y3-x3*y1      x1*y4-x4*y1
x2*y3-x3*y2      x2*y4-x4*y2      x3*y4-x4*y3
```

Количество выполняемых операций можно уменьшить, если учесть, что задача инвариантна относительно операции переноса. Таким образом, можно обойтись только тремя определителями второго порядка.

Вычислим определители третьего порядка:

```
(x2-x1)*(y3-y1)-(x3-x1)*(y2-y1) = det(P1,P2,P3)
(x2-x1)*(y4-y1)-(x4-x1)*(y2-y1) = det(P1,P2,P4)
(x3-x1)*(y4-y1)-(x4-x1)*(y3-y1) = det(P3,P4,P1)

det(P1,P2,P3)-det(P1,P2,P4)+det(P3,P4,P1) = det(P3,P4,P2)
```

Теперь сравним знаки определителей – и задача решена! (Так ли?)

Второй этап – написание программы

Определим основные структуры данных и функции, с которыми предстоит работать, опишем их на языке программирования:

- Точка,
- Отрезок,
- Определитель,
- Функция проверки отрезков на пересечение. Должна включать также и вырожденные случаи, такие, например, как отрезки нулевой длины, принадлежность точки P3 отрезку P1-P2 и т.д.

Напишем часть программы, выполняющую ввод информации.

Третий этап – реализация

Для реализации используйте язык C или C++. Существенно важно, чтобы в реализации не использовались платформенно-зависимые функции и заголовочные файлы, например, stdio.h, stdafx.h.

Это необходимо для осуществления пятого этапа (смотри далее лаб. работу 5) с возможностью переноса приложения в другие операционные системы.

Четвертый этап – тестирование

Для проверки работоспособности выберем наборы данных, которые представляют пересекающиеся или не пересекающиеся отрезки, включая различные вырожденные случаи.

Пятый этап– графический пользовательский интерфейс

Выполнение этого этапа представляет отдельную лабораторную работу, которую предстоит сделать после ознакомления с методами создания графического пользовательского интерфейса (GUI).

Лабораторная работа 3
Находится ли точка внутри многоугольника?

Написать функцию, определяющую, лежит ли точка, заданная своими координатами внутри произвольного несамопересекающегося многоугольника, заданного указателем на массив координат вершин.

Работа выполняется в несколько этапов.

Требования

Четыре часа на выполнение, язык реализации – C или C++.

Особое внимание необходимо обратить на реализацию функции, которая должна обрабатывать все возможные особые случаи.

Первый этап – алгоритмизация задачи

Поскольку многоугольник может быть как выпуклый, так и произвольный, на первый взгляд алгоритм кажется сложным. Но на самом деле существует правило: если из точки провести луч и посчитать количество пересечений с отрезками, являющимися ребрами многоугольника, то в случае нечетного количества ответ положительный, иначе – отрицательный. Продумайте обход исключений из этого правила.

Второй этап – написание программы

Определим основные структуры данных и функции, с которыми предстоит работать, опишем их на языке программирования:

- Точка,
- Отрезок,
- Функция проверки отрезков на пересечение, из предыдущей работы. Вот, где пригодится хорошо работающая функция!

Напишем часть программы, выполняющую ввод многоугольника из файла.

Третий этап – реализация

Для реализации используйте язык C или C++. Существенно важно, чтобы в реализации не использовались платформенно-зависимые функции и заголовочные файлы, например, stdio.h, stdafx.h.

Это необходимо для осуществления пятого этапа (смотри далее лаб. работу 6) с возможностью переноса приложения в другие операционные системы.

Программа должна иметь возможность как ввода точек многоугольника с консоли, так и возможность сохранения его координат в файл и чтения их из файла.

Четвертый этап – тестирование

Для проверки работоспособности выберем наборы данных, которые представляют различные многоугольники. Введем в программу возможность многократного задания тестовой точки.

Пятый этап – графический пользовательский интерфейс

Выполнение этого этапа представляет отдельную лабораторную работу, выполнение которой предстоит выполнить после ознакомления с методами создания GUI (лаб. работа 6).

3. СОЗДАНИЕ ГРАФИЧЕСКОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Основы FLTK

Этот раздел посвящен основам компилирования и использования FLTK. Почему FLTK? Прежде всего из-за небольших размеров, доступности исходного кода, простоты и понятности классов используемых методов. Нельзя сбрасывать со счетов и мобильность. Вы можете разрабатывать приложение под Linux, а затем перенести его в Windows и наоборот.

Написание простой программы с использованием FLTK

Все программы должны подключать заголовочный файл <FL/Fl.H>. Кроме того, программы обязаны подключать заголовочные файлы для каждого используемого класса FLTK. На листинге 1 показана простая программа "Hello, World!" для отображения окна при помощи FLTK.

Листинг 1 – "hello.cxx"

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
  Fl_Window *window = new Fl_Window(300,180);
  Fl_Box *box = new
Fl_Box(20,40,260,100,"Hello, World!");
  box->box(FL_UP_BOX);
  box->labelsize(36);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labeltype(FL_SHADOW_LABEL);
  window->end();
  window->show(argc, argv);
  return Fl::run();
}
```

После подключения необходимых заголовочных файлов программа создает окно:

```
Fl_Window *window = new Fl_Window(300,180);
```

и прямоугольник со строкой "Hello, World!" в нем:

```
Fl_Box *box = new
Fl_Box(20,40,260,100,"Hello, World!");
```

Затем мы задаем тип и размер прямоугольной области, размер, стиль и шрифт метки:

```
box->box(FL_UP_BOX);
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
```

Затем мы включаем отображение окна и запускаем цикл событий FLTK:

```
window->end();
window->show(argc, argv);
return Fl::run();
```

Запустив откомпилированную программу, увидим окно с текстом, показанное на рис. 2. Выход из программы – по клавише «Escape».

Рис. 2. Окно простого приложения

Создание виджетов

Виджеты (элементы управления) создаются при помощи операторов “new” языка C++. Для большинства виджетов применяются следующие аргументы в конструкторе:

```
Fl_Widget(x, y, width, height, label)
```

Параметры **x** и **y** определяют положение виджета в окне или окна на экране. В пакете FLTK верхний левый угол окна или экрана является началом координат (т.е. **x = 0, y = 0**), а единицами измерения являются пиксели.

Параметры **width** и **height** определяют размер окна в пикселях.

label является указателем на строку символов для пометки виджета. Для не имеющего метки виджета задается пустой указатель **NULL.**

Методы Get/Set

box->box(FL_UP_BOX) устанавливает тип отображения прямоугольной области **Fl_Box,** изменяя его от значения по умолчанию **FL_NO_BOX**, при котором область не рисуется. В примере "Hello, World!" мы использовали **FL_UP_BOX**, изображающую слегка приподнятую область.

Всегда можно узнать тип области методом **box->box().** FLTK использует перегрузку имен методов для get/set. "set" метод имеет вид "**void name(type)**", а метод "get" всегда имеет форму "**type name() const**".

Перерисовка после изменения атрибутов

После установки атрибутов или их изменения необходимо вызвать метод **redraw()** самостоятельно. Исключение составляет свойство **value()**, из которого вызываются методы **redraw()** и **label()** и в случае необходимости **-redraw_label().**

Метки

Все виджеты поддерживают метки (надписи). В случае окна метка отображается в заголовке. Наша программа использует методы **labelfont, labelsize** и **labeltype.**

Метод **labelfont** устанавливает тип и стиль используемого шрифта, которые в примере заданы как **FL_BOLD** и **FL_ITALIC.**

Метод **labelsize** задает высоту шрифта в пикселях.

Метод **labeltype** задает тип метки. FLTK использует нормальный, выпуклый и оттенённый типы (**normal**, **embossed** и **shadowed**).

Полный список опций можно найти в справочном материале, включенном в дистрибутивный пакет.

Отображение окна

Метод **show()** отображает окно. Его можно сопроводить аргументами командной строки для управления положением, размером ваших окон.

Основной цикл событий

Все приложения FLTK основаны на простой модели событий. Действия пользователя, такие как нажатие клавиш, движение мыши, нажатие кнопок, генерируют события, посылаемые приложению. Приложе-

ние может игнорировать события или реагировать на них соответствующим образом.

FLTK также использует функции ожидания и таймеры. Функции таймера вызываются после истечения указанного интервала времени. Их можно использовать для реализации прогрессоров и периодических событий.

Приложения FLTK должны периодически вызывать **`Fl::check()`** или **`(Fl::wait())`** для событий или же использовать метод **`Fl::run()`** для запуска стандартного цикла ожидания событий. Вызов **`Fl::run()`** эквивалентен следующему коду:

```
while (Fl::wait());
```

`Fl::run()` не завершается до тех пор, пока не будут закрыты все окна, находящиеся под управлением FLTK программно или пользователем.

Компиляция стандартными компиляторами

В ОС UNIX (и в Microsoft Windows при использовании компилятора GNU) необходимо указать компилятору, где находится заголовочные файлы, для этого используется опция -I:

```
CC -I/usr/local/include...
gcc -I/usr/local/include...
```

Скрипт fltk-config поставляемый вместе с пакетом FLTK может быть использован как прототип для вашего приложения.

```
CC `fltk-config --cxxflags`...
```

Аналогично для компоновщика необходимо указать место расположения библиотеки FLTK:

```
CC... -L/usr/local/lib -lfltk -lXext -lX11 -lm
gcc... -L/usr/local/lib -lfltk -lXext -lX11 -lm
```

Кроме собственно библиотеки "fltk" может понадобиться библиотека "fltk_forms" для использования классов XForms, "fltk_gl" – для классов OpenGL и GLUT и "fltk_images" –для классов изображений.

> ***Замечание.***
> Библиотеки в Windows называются соответственно "fltk.lib", "fltkgl.lib", "fltkforms.lib" и "fltkimages.lib".

Скрипт **`fltk-config`** можно использовать и в этом случае:

```
CC... `fltk-config --ldflags`
```

Библиотеки форм, GL и изображений подключаются при помощи опции "**--use-foo**":

```
CC... `fltk-config --use-forms --ldflags`
CC... `fltk-config --use-gl --ldflags`
CC... `fltk-config --use-images --ldflags`
CC...  `fltk-config  --use-forms  --use-gl  --use-
images --ldflags`
```

Можно использовать **fltk-config** для компиляции одиночного исходного файла программы с использованием FLTK:

```
fltk-config --compile filename.cpp
fltk-config --use-forms --compile filename.cpp
fltk-config --use-gl --compile filename.cpp
fltk-config --use-images --compile filename.cpp
fltk-config  --use-forms  --use-gl  --use-images  --
compile filename.cpp
```

Любой вызов создаст исполняемый файл с именем **filename.**

Компилирование программ в Microsoft Visual C++

В оболочкеVisual C++ вам необходимо указать компилятору место нахождения заголовочных файлов FLTK. Это делается выбором пункта "Settings" из меню "Project" и изменением установок "Preprocessor" в закладке "C/C++". Вам также необходимо добавить библиотеки FLTK и WinSock (WSOCK32.LIB) в установках "Link".

Вы можете создавать ваши приложения как «Console» или «WIN32». Если вы хотите использовать стандартную функцию **main()языка Си**, FLTK включит функцию **WinMain()**, вызывающую для Вас функцию **main().**

Имена

Все символы FLTK начинаются с литер 'F' и 'L':

- Функции либо **Fl::foo()**, либо **fl_foo().**
- Классы и имена типов начинаются с заглавной буквы: **Fl_Foo**.
- Константы и перечисления – в верхнем регистре: **FL_FOO**.
- Все заголовочные файлы начинаются с **<FL/...>.**

Заголовочные файлы

Правильный способ подключения заголовочных файлов FLTK:

```
#include <FL/Fl_xyz.H>
```

Регистр важен для многих операционных систем так же, как и использование стандартного для Си разделителя каталогов (/). Следующие строки представляют пример **неправильного** употребления препроцессорных директив*:*

```
#include <FL\Fl_xyz.H>
#include <fl/fl_xyz.h>
#include <Fl/fl_xyz.h>
```

Программирование с помощью FLUID

Опишем средство визуальной разработки GUI при помощи инструмента «Fast Light User-Interface Designer».

Что такое FLUID?

FLUID – это «Fast Light User Interface Designer», графический редактор пользовательского интерфейса, использующийся для получения исходного кода на языке описания интерфейса FLTK. FLUID редактирует и сохраняет файлы с суффиксом **`.fl`**. Это текстовые файлы, их можно просматривать и редактировать обычным текстовым редактором (с известными предосторожностями).

FLUID может также "компилировать" файлы **`.fl`** в **`.cxx`** и **`.h`** файлы. Файл **`.cxx`** определяет все объекты, описанные в файле **`.fl`**, а **`.h`** файл содержит глобальные декларации. FLUID поддерживает локализацию строк меток, используя файлы строк и интерфейс GNU «gettext» или POSIX интерфейс «catgets».

Процесс создания простого приложения можно проиллюстрировать рис. 3.

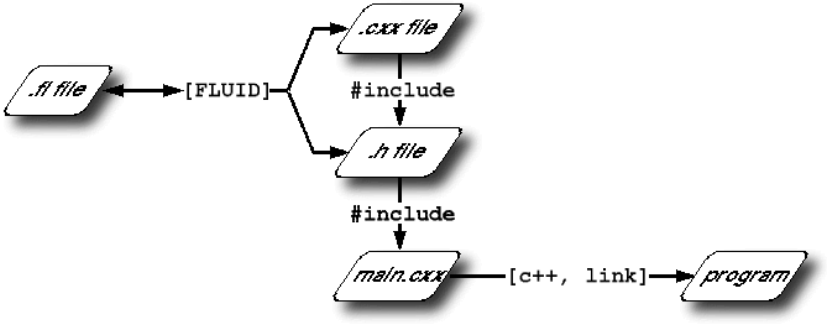

Рис. 3. Организация процесса разработки GUI с помощью FLUID

Обычно файл FLUID определяет одну/один или более функций или классов, которые порождают вывод кода на языке C++. Каждая функция определяет одно или более окон FLTK и виджетов в этих окнах.

Запуск FLUID под UNIX

Для запуска FLUID в ОС UNIX введите команду:

```
fluid filename.fl &
```

Для редактирования файла **`.fl`** с именем **`filename.fl`** можно также запустить FLUID без указания имени, в этом случае вам придется впоследствии использовать пункт «save-as» меню «File».

Вы можете указать стандартные опции FLTK перед именем файла:

```
-display host:n.n
-geometry WxH+X+Y
-title windowtitle
-name classname
-iconic
-fg color
-bg color
-bg2 color
-scheme schemename
```

Запуск FLUID под Microsoft Windows

Для запуска FLUID под WIN32 используйте двойной щелчок на файле **FLUID.exe**.

Компилирование файлов `.fl`

FLUID можно использовать как компилятор «командной строки» для создания файлов **`.cxx`** и **`.h`** из файла **`.fl`**. Для этого нужно ввести команду:

```
fluid -c filename.fl
```

Эта команда заставит fluid прочитать файл **`filename.fl`** и записать результат компиляции – файлы **filename.cxx** и ***filename.h***.

Вы можете использовать следующие строки в сценарии makefile для автоматического создания этих файлов:

```
my_panels.h my_panels.cxx: my_panels.fl
fluid -c my_panels.fl
```

Многие версии утилиты make используют правило для компиляции файлов `.fl`:

```
.SUFFIXES: .fl .cxx .h
.fl .h .fl .cxx:
        fluid -c $<
```

Самое простое приложение «КНОПКА ВЫХОД»

Запускаем программу FLUID, добавляем препроцессорную директиву, выбрав пункты меню New > Code > declaration, и вводим текст директивы: #include <stdlib.h>.

Далее создаем новую функцию int main(), выбирая пункты меню NEW > Code > Function/method и вводя в окна ввода имени main() и в окно ввода возвращаемого типа int (рис. 4).

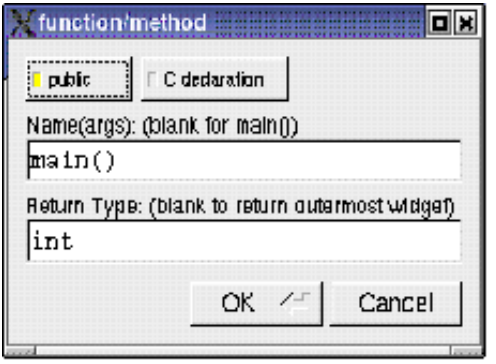

Рис. 4. Создание кнопки

Создаем функцию генерации окна, выбирая в меню пункты New >Code >Function/method make_window.

Для этой функции создаем группу «Окно» New > Group > Window.

И внутри группы создаем кнопку типа «Возврат» New > buttons >Return_Button (рис. 5).

Используя закладку GUI (рис. 6), укажем название метки (exit), желаемое положение метки, а в закладке Style (рис. 7) настроим желаемый вид кнопки. Согласимся на предлагаемые по умолчанию размеры, а затем перейдем во вновь появившееся окно нашего будущего приложения.

Придадим кнопке нужные размеры и поместим ее в желаемом месте окна, используя мышь.

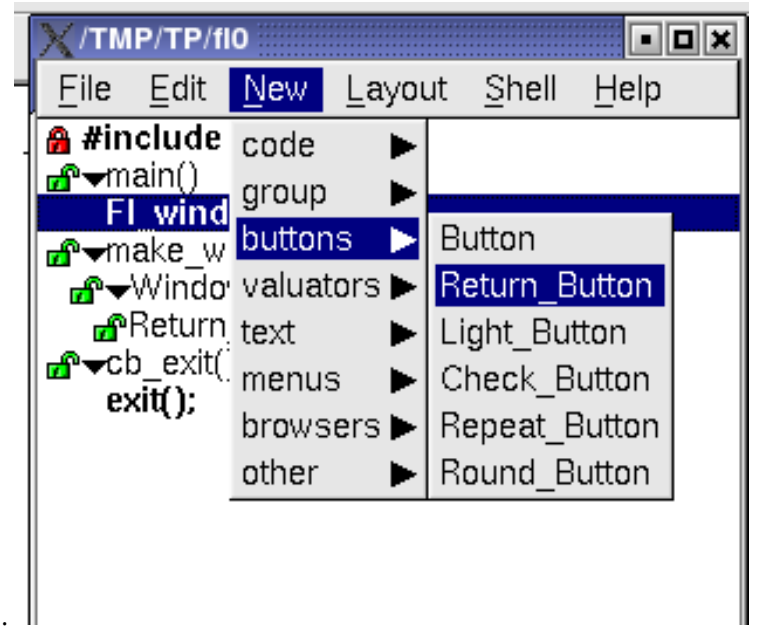

Рис. 5. Создание кнопки

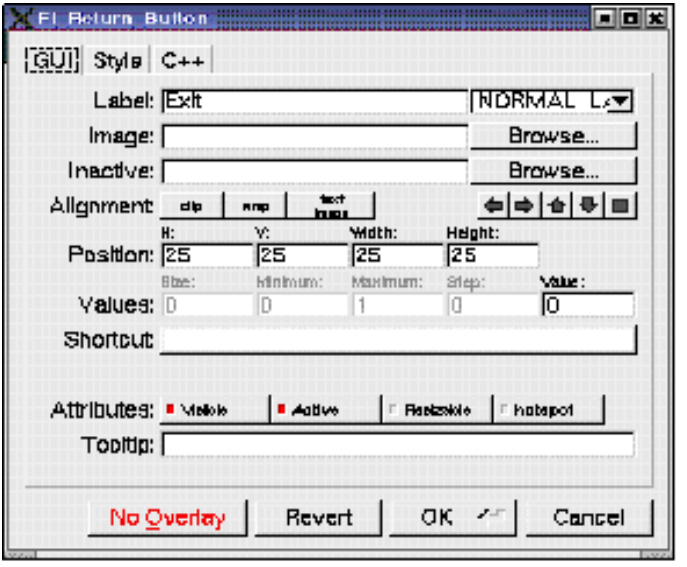

Рис. 6. Закладка GUI

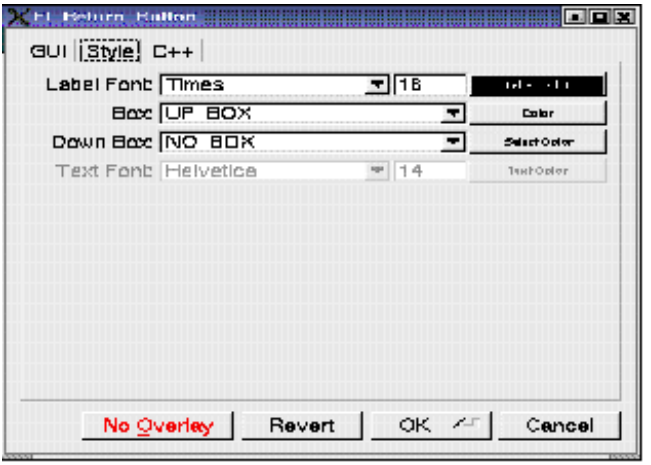

Рис. 7. Закладка Style

Закладка C++ (рис. 8) позволит нам ввести имя функции-обработчика событий – cb_exit и задать имя объекта (в данном случае – кнопки).

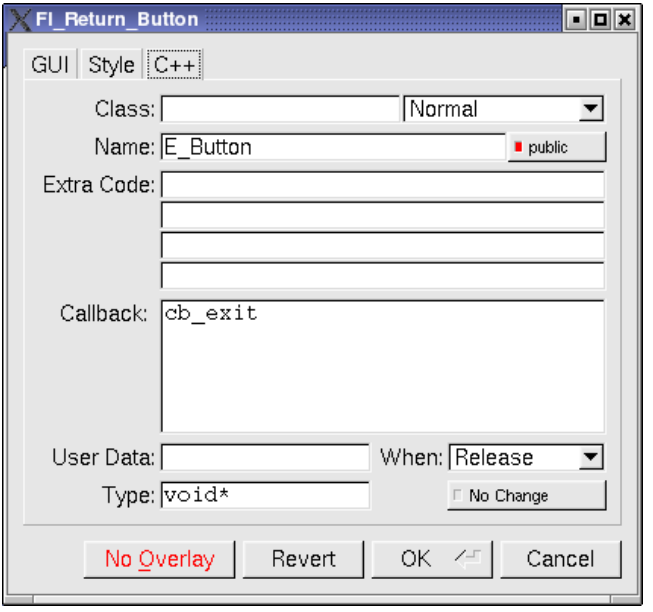

Рис. 8. Закладка C++

Можно также разместить дополнительные операторы языка Си, сопровождающие объявление данного объекта в поле «Extra Code».

Создадим функцию cb+exit(), выбрав New > function/method >, создадим для нее блок кода New > code >function/method и блок кода New >code > code. В окне редактирования добавим вызов функции exit (0).

Далее выделим функцию main(), создадим для нее блок кода New > code > code и в окне редактирования кода введем текст основной функции (рис. 9).

Рис. 9. Ввод текста функции main()

Выберем пункты меню File > Save, введем имя нашего проекта – «fl», в результате чего сохраняем файл с описанием интерфейса на языке «fl». Этот файл будет выглядеть следующим образом:

Листинг 2. Интерфейс, созданный при помощи "fluid"

```
# data file for the Fltk User Interface Designer
(fluid)
version 1.0101
i18n_type 1
i18n_include <libintl.h>
i18n_function gettext
header_name {.h}
code_name {.cxx}
decl {\#include <stdlib.h>} {}

Function {main()} {open return_type int
} {
  code {Fl_Window *w;
w=make_window();
Fl::run();} {}
}

Function {make_window()} {open
} {
  Fl_Window {} {open
  xywh {312 20 384 217} visible
} {
  Fl_Return_Button {} {
```

```
      label exit
      callback cb_exit
      xywh {255 140 90 50}
    }
  }
}
Function {cb_exit()} {open selected return_type
{void *}
} {
code {exit(0);} {}
}
// generated by Fast Light User Interface Designer
  (fluid) version 1.0101
```

Далее компилируем этот файл в программу на языке С:

File > Write Code, в результате чего появятся **два** файла – fl.h и fl.cxx (заголовочный файл и файл на языке С++). Вот их текст:

Листинг 3. Файл "fl.h", созданный компилятором "fluid"

```
#ifndef fl_h
#define fl_h
#include <FL/Fl.H>
int main();
#include <FL/Fl_Window.H>
#include <FL/Fl_Return_Button.H>
//extern void cb_exit(Fl_Return_Button*, void*);
Fl_Window* make_window();
void * cb_exit();
#endif
// generated by Fast Light User Interface Designer
(fluid) version 1.0101
```

Листинг 4. fl.cxx, созданный компилятором "fluid"

```
#include <libintl.h>
#include "fl.h"
#include <stdlib.h>

int main() {
  Fl_Window *w;
w=make_window();

Fl::run();
}

Fl_Window* make_window() {
  Fl_Window* w;
  { Fl_Window* o = new Fl_Window(380, 215);
    w = o;
    { Fl_Return_Button* o = new
Fl_Return_Button(255, 140, 90, 50, gettext("exit"));
```

```
      o->callback((Fl_Callback*)cb_exit);
    }
    o->end();
  }
  return w;
}
void * cb_exit() {
  exit(0);
}
```

Откомпилируем программу при помощи gcc (под Linux) или VisualC (под Windows). При запуске программа выводит окно с единственным виджетом – кнопкой «Exit». Кнопка работает – по нажатию приложение завершает работу.

Лабораторная работа 4
Знакомство с FLTK

Два часа ознакомительного занятия.

Установить на рабочий компьютер пакет FLTK (исходные тексты), откомпилировать проект, загрузив рабочую среду fltk.dsw из каталога visualc. Откомпилировать файлы примеров (каталог test). Используя fluid, создать приложение «кнопка выхода». При компиляции особое внимание обратить на правильность настроек оболочки VisulC.

Лабораторная работа 5
Пересекаются ли отрезки? GUI

Написать графический интерфейс для программы из лабораторной работы 2. Разработать интерфейс с помощью fluid.

Отложенный пятый этап проекта.

Требования

Четыре часа на выполнение, языки реализации – C или C++, пакет FLTK, средство визуального проектирования fluid. Интерфейс должен быть интуитивно понятен, допускать ввод точек отрезка в графическом интерфейсе или из файла, вывод координат точек в файл.

Лабораторная работа 6
Находится ли точка внутри многоугольника? GUI

Написать графический интерфейс для программы из лабораторной работы 3. Разработать интерфейс с помощью fluid.

Отложенный пятый этап проекта.

Требования

Четыре часа на выполнение, языки реализации – C или C++, пакет FLTK, средство визуального проектирования fluid. Интерфейс должен быть интуитивно понятен, содержать элементы управления для задания положения тестирующей точки (колесики прокрутки, слайдеры или ползунки), осуществлять загрузку координат вершин многоугольника из файла, дополнительно – проверку этого многоугольника на самопересечение.

Алгоритмизация задачи определения самопересечения

Многоугольник не будет являться самопересекающимся, если любые две стороны не пересекаются. Использовать алгоритм из лабораторной работы 2.

Реализация

Создавать как консольное приложение.

Ввод данных из файла выполнить в виде отдельной функции, пригодной для последующего использования.

Написание программы

Одно из возможных представлений пользовательского интерфейса приведено на рис. 10 и 11.

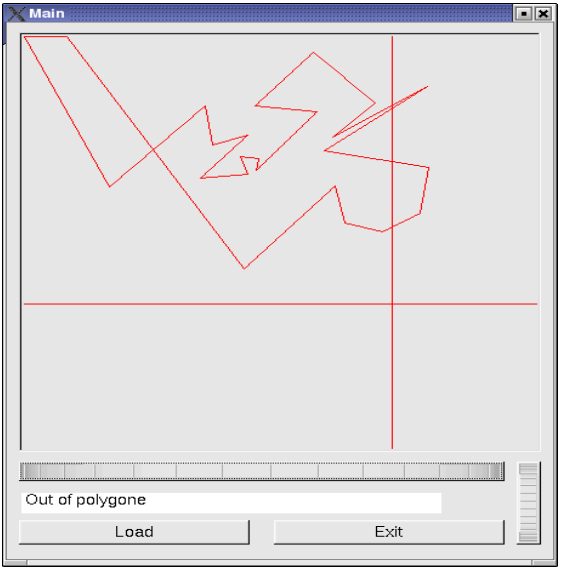

Рис. 10. Окно приложения (одна из реализаций)

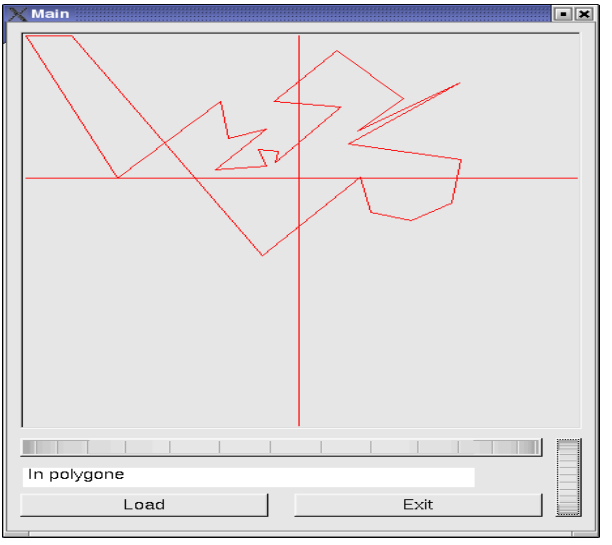

Рис. 11. Окно приложения (точка вне полигона)

Для управления положением в данном случае используется не использованный в приложении fluid, но имеющийся в FLTK объект «позиционер» и колесики прокрутки (скроллеры).

Текст этого приложения (не полностью) приведен в листинге 5.

Листинг 5. Головной модуль «точка в полигоне»

```
#include <FL/Fl.H>
. . . . . .
#include <FL/Fl_Positioner.h>
#include "Main.h"
#define min(x,y) ((x)>(y)?y:x)
#define max(x,y) ((x)>=(y)?x:y)

dwin *ClientArea;
Fl_Roller *vert;
Fl_Roller *horiz;
Fl_Button *btnfile;
Fl_Button *btnext;
Fl_Text_Display *txtdispl;
Fl_Text_Buffer *txtbuf;

int NumPoint=0;

bool  DotIsInPoligon(Point  *p,int  NumPoint,double
CoordX,double CoordY)
{
. . . . . .
```

```
// ищем пересечения нашего луча с вершинами много-
//угольника
// находим число пересечений нашего луча с ребрами
//многоугольника
       if((numCross%2)==0) return false;
       return true;
}
//функция загрузки многоугольника из файла
void StartLoadPolygon(char *fn)
{
       FILE *fd;
       int i;

       fd=fopen(fn, "r");  if(fd==NULL) return;

       if(NumPoint!=0) delete [] Poly;
       fscanf(fd, "%d", &NumPoint);
       Poly= new Point[NumPoint];

       for(i=0;i<NumPoint;i++) fscanf(fd,"%d\
%d",&Poly[i].x,&Poly[i].y);

       fclose(fd);
}

//Функция отрисовки
void dwin::draw()
{
       int i,x1,y1,x2,y2;
       Fl_Positioner::draw();
       if(NumPoint>1)
       {
              for(i=0;i<NumPoint;i++)
              {
                     x1=Poly[i].x; y1=Poly[i].y;
                     if(i+1==NumPoint)
                     {
                     x2=Poly[0].x; y2=Poly[0].y;
                     fl_line(x()+x1, y()+y1, x()+x2,
                            y()+y2);
                     }
                     else
                     {
                     x2=Poly[i+1].x; y2=Poly[i+1].y;
                     fl_line(x()+x1, y()+y1, x()+x2,
                            y()+y2);
                     }
              }
       }
}

static void cb_brow(dwin* b, void*)
```

```
{ // Обработчик событий
     vert->value(b->yvalue());
     horiz->value(b->xvalue());
     CrossInformation();
}
static void cb_vert(Fl_Roller* r, void*)
{ // Обработчик событий
     ClientArea->yvalue(r->value());
     CrossInformation();
}
//функция проверяет, пересекаются ли два заданных
//отрезка
bool       OterezokXOtrezok(Point       Dot_11,Point
Dot_21,Point Dot_12,Point Dot_22)
{
. . . . . .
                    return true;
       return false;
}
static void cb_horiz(Fl_Roller* r, void*)
{
       ClientArea->xvalue(r->value());
       CrossInformation();
}
//Функция кнопки открытия файла
static void cb_btnfile(Fl_Button*, void*)
{
  char *newfile =
    fl_file_chooser("Открыть файл", "*.txt", "");
       if (newfile!= NULL)
       {
              StartLoadPolygon(newfile);
       }
       ClientArea->xvalue(ClientArea->w()/2);
       ClientArea->yvalue(ClientArea->h()/2);
       ClientArea->do_callback();
       ClientArea->draw();
       CrossInformation();
}
//Находится ли новая точка внутри многоугольника
void CrossInformation()
{
       char b[30];
       int x=(int)(ClientArea->xvalue());
       int y=(int)(ClientArea->yvalue());
```

```
        if(NumPoint==0)
        {
               sprintf(b,"Файл не загружен");
        }
        else
        {
        if(DotIsInPoligon(Poly,NumPoint,x,y))
           sprintf(b,"Точка внутри полигона");
        else sprintf(b,"Точка снаружи полигона");
        }
        txtdispl->buffer()->text(b);
}

static void cb_btnext(Fl_Button*, void*)
{
        exit(0);
}

int main(int argc, char **argv)
{
// Собственно создание интерфейсного окна
        Fl_Window* w= new Fl_Window(450, 550);
ClientArea = new dwin(10, 10, 430, 430);
ClientArea->callback((Fl_Callback*)cb_brow);
vert = new Fl_Roller(420,450, 20, 85);
vert->callback((Fl_Callback*)cb_vert);
    horiz = new Fl_Roller(10, 450, 400, 20);
    horiz->type(1);
    horiz->callback((Fl_Callback*)cb_horiz);

btnfile = new Fl_Button(10,510,190,25,"Загрузить");
btnfile->tooltip("Загрузить многоугольник");
btnfile->callback((Fl_Callback*)cb_btnfile);
btnext = new Fl_Button(220, 510, 190, 25, "Выход");
    btnext->tooltip("Выход из программы");
    btnext->callback((Fl_Callback*)cb_btnext);
    btnext->shortcut(FL_ALT+'x');
    txtdispl = new Fl_Text_Display(10, 480,350,25);
        txtbuf=new Fl_Text_Buffer(10);
        txtdispl->buffer(txtbuf);
        w->insert((Fl_Widget&)*ClientArea, 0);
        w->show(argc, argv);
        horiz->range(0, ClientArea->w());
        vert->range(0, ClientArea->h());
        horiz->step(1);
        vert->step(1);
        ClientArea->xbounds(0, ClientArea->w());
        ClientArea->ybounds(0, ClientArea->h());
// Вход в цикл событий
        return Fl::run();
}
```

4. ИСПОЛЬЗОВАНИЕ СРЕДСТВ АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ ПАРСЕРОВ

Для чего используется синтаксический анализ? Во-первых, достаточно часто возникает необходимость в приложениях просто осуществить вычисления по заданной формуле. Во-вторых, необходимость в крупных программных продуктах (офисные приложения, компиляторы, бухгалтерские программы) наличия возможности расширения программными средствами удобств пользователя заставляют разработчиков встраивать в приложения так называемые скрипт-языки.

И в том и в другом случаях не обойтись без программ синтаксического разбора (парсера). Но написание такой программы – долгий и утомительный труд, на котором вас подстерегают многочисленные отладочные проблемы.

К счастью, есть программы, автоматически генерирующие синтаксические распознаватели, да еще с возможностью генерации кода, описываемого на языке высокого уровня. Простейшими примерами являются программы yacc и bison. В ОС UNIX – это стандартные компоненты системы. Для DOS и Windows существуют их аналоги. К ним относится, например, программа pcyacc разработки фирмы Abraxas software.

Что же представляет из себя yacc?

Генератор распознавателей yacc

Yacc – это универсальные средства для структуризации исходных данных программ. Пользователь задает спецификацию (грамматику языка), которая включает:

- множество правил, описывающих составные части исходных данных;
- действия, выполняемые при применении правила;
- определение или описание процедуры нижнего уровня, анализирующей исходные данные.

Yacc отображает спецификацию в функцию yyparse() на языке C, обрабатывающую входной поток данных. Эта функция является процедурой синтаксического разбора и при выполнении обращается к низкоуровневому сканеру входных данных (лексическому распознавателю). Сканер извлекает из входного потока элементарные конструкции языка – лексемы. Лексемы сопоставляются с правилами, описывающими структуру входного текста, то есть с грамматическими правилами.

Если правило оказывается подходящим, то выполняется ассоциированное с ним действие. Действие – это фрагмент программы на языке C. Действия могут возвращать значения, а также использовать значения, возвращаемые другими действиями, например сканером.

Имена обозначают лексемы или нетерминальные символы. YACC требует, чтобы имена лексем были указаны явно. Хотя лексический анализатор можно включить в файл спецификаций, определение его в отдельном файле, вероятно, более соответствует принципам модульного проектирования. Подобно лексическому анализатору, в файл спецификаций могут быть также включены и другие подпрограммы.

Таким образом, каждый файл спецификаций (уасс-программа) теоретически состоит из трех секций: определений, (грамматических) правил и подпрограмм.

Структура спецификации уасс

Секции отделяются двумя знаками процента %% (знак % используется в уасс-спецификациях как универсальный управляющий). Если используются все секции, полный файл спецификаций выглядит следующим образом:

```
определения
%%
правила
%%
подпрограммы
```

Секции определений и подпрограмм являются необязательными. Минимальная допустимая уасс-спецификация – это

```
%%
правила
%%
```

Пробелы, табуляции и переводы строки, которые встречаются вне имен и зарезервированных слов, игнорируются. Комментарии могут быть везде, где допустимо имя. Они оформляются как многострочный комментарий на языке Си: /*...*/

Секция правил

Секция правил составляется из одного или большего числа грамматических правил. Грамматическое правило имеет вид

нетерминал: определение;

где определение – это последовательность из нуля или нескольких альтернатив, разделенных метасимволом «ИЛИ», в роли которого выступает символ – вертикальная черта(|). Каждая альтернатива представляет набор имен и литералов. Каждые двоеточие и точка с запятой – знаки препинания уасс'а.

Имена могут иметь произвольную длину и должны состоять из букв, точек, подчеркиваний и цифр, однако имя не может начинаться с

цифры. Прописные и строчные буквы различаются. Имена, используемые в теле грамматического правила, могут представлять лексемы или нетерминальные символы.

Секция определений

В секции определений можно объявлять:

- терминальные символы,
- начальный символ,
- операции,
- типы нетерминалов.

Терминальные символы объявляются при помощи декларатора %token: %token имя1 имя2.

Считается, что всякое имя, не описанное в секции определений, представляет нетерминальный символ. Каждый нетерминальный символ должен встретиться в левой части по крайней мере одного правила.

Из всех нетерминальных символов особую роль играет начальный символ. По умолчанию начальным считается символ, стоящий в левой части первого грамматического правила в секции правил.

Из начального символа можно вывести любую правильную фразу языка. Можно явно объявить начальный символ в секции определений при помощи ключевого слова **%start**:

%start начальный_символ

Операции можно объявить декларациями **%left, %right, %noassoc**:

```
%left '+' '-'
%left '*' '/'
%right '='
%noassoc UNARYMINUS
```

Лексемы, встречающиеся в одном декларации, считаются равноприоритетными операциями.

Приоритет операций увеличивается от декларатора к декларатору. Например, операции ‘*’ и ‘/’ в приведенном примере имеют более высокий приоритет, чем ‘+’ и ‘-‘.

Тип нетерминального символа задается декларатором **%type**:

```
% type expr double
```

Секция процедур

В этой секции должна быть описана функция main(), вызывающая процедуру разбора, ее имя – yyparse(). Здесь же можно поместить подпрограммы обработки ошибок, служебные функции и лексический распознаватель (сканер).

Простое приложение – калькулятор

Рассмотрим простое приложение – арифметический калькулятор, вычисляющий произвольные арифметические выражения с любым уровнем вложенности скобок. В этом приложении действия, выполняемые при синтаксическом разборе, выполняются с компонентами правил, соответствующий компонент во вставке на языке Си (она производится в фигурных скобках) представлен своим значением, находящимся в стеке. Тип элемента стека в этом варианте – удвоенной точности вещественное число. Значение компонента обозначается метасимволом $N, где цифра N обозначает номер компонента в правиле. Метасимволом $$ обозначается верхушка стека.

Листинг 6. Калькулятор со скобками

```
/* Пример калькулятора */
%{ /* вставка определения на языке Си */

#define YYSTYPE double
     /* тип данных в стеке yacc */
#define QUIT 101010
%}
%token NUMBER /* лексема число */
%left '+' '-' /* левоассоциативные операции */
%left '*' '/' /* левые опер.,больше приоритет */
%left UNARYMINUS./* унарный минус */
%% /* СЕКЦИЯ ПРАВИЛ */
list:  /* пусто */
       { prompt(); }
     | list '\n' /* список выражений */
       { prompt(); }
     | list expr '\n'
       { if ($2 == QUIT) {
           return(0);
         } else {
           fprintf(stdout, "RESULT> %.8g\n", $2);
           prompt();
         }
       }
     | list error '\n'
          { yyerrok;
          prompt();
          }
     ;
expr: /* правило для нетерминала выражение */
       NUMBER { $$ = $1; }
     | '-' expr %prec UNARYMINUS { $$ = -$2; }
     | expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
```

```
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     ;
/* конец секции */
%%
#include <stdio.h>
#include <ctype.h>
char *progname; /* для сообщения об ошибках */
int lineno = 1;

main(argc, argv)
char *argv[];
{
  if (argc > 1)
     fprintf(stderr, "лишние аргументы\n");
     progname = argv[0];
     fprintf(stdout,
"\n*****************************************");
  fprintf(stdout,
"\n* Простой калькулятор *");
     fprintf(stdout,
"*****************************************");
putchar ('\n');
     yyparse();
   fprintf(stdout, "* До свиданья! *\n");
}

yylex() /* лексический анализатор */
{

     int c;
     while ((c=getchar()) == ' ' || c == '\t');
     /* пропустить пробельные символы */
     if (c == EOF) return 0;
     if (c == '.' || isdigit(c)) { /* число */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
     }
     if (c == '\n')
        lineno++;
if (c == 'Q' || c == 'q') /* плохой код! */
  if ((c=getchar()) == 'U' || c == 'u')
    if ((c=getchar()) == 'I' || c == 'i')
      if ((c=getchar()) == 'T' || c == 't')
            {
              yylval = QUIT;
              return NUMBER;
            } else return '?';
```

```
  return c;
}

yyerror(s) /* вызывается при ошибках yacc */
char *s;
{
     warning (s, (char *) 0);
}

warning(s,t) /* предупреждение */

char *s, *t;
{
   fprintf(stderr, "%s: %s", progname, s);
   if (t) fprintf(stderr, " %s", t);
     fprintf(stderr, " near line %d\n", lineno);
}

prompt() /* приглашение к вводу выражения */
{
  fprintf(stdout, "READY> ");
}
```

Лабораторная работа 7
Добавим функции в калькулятор

Расширить простой калькулятор, добавив в него тригонометрические функции (или exp, log, abs).

Требования

Два часа на выполнение, языки реализации – yacc и C.

Модифицировать грамматику, вставки на языке C и дополнить лексический анализатор. В секции описаний добавить лексемы SIN, COS, TAN, ATAN и т.д.

Алгоритмизация задачи

Можно использовать функции стандартной библиотеки C для вычисления функций. В лексическом распознавателе придется анализировать символьные строки, выявляя последовательности символов «sin», «cos», etc…

Калькулятор с функциями и константами

В этот калькулятор добавлены переменные (именованные регистры), функции и константы. В стеке теперь может находиться либо имя (Sym), либо значение (val), для этого элемент стека описывается объединением (союзом) %union.

Листинг 7. Калькулятор с переменными и функциями

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int yylex();
double Sqrt (double x); // своя функция sqrt
  typedef struct Symbol // структура для описания
    {                   // переменных, констант и
    char name; short type; // функций type задает
    union { /* тип объекта */
          double val;
          double (*ptr)();
          }u;
    struct Symbol *next;
  } Symbol;
Symbol *symlist =0;
static struct {
char *name;
double cval; } Const[]= /* список констант */
{
"PI", 3.1415926536,
"E", 2.718281828459,
"RAD",57.2958,
0,0
};
static struct {
    char *name;
  double (*func)(); }
Funcs[]=                    /* список функций */
{
"sin",sin,
"cos",cos,
"tan",tan,
"sqrt",Sqrt,
"abs",fabs,
0,0
};
%}

%union {                    /* тип элемента стека */
        double val;
        Symbol *sym;
        };
%token <val> NUM                /* лексемы */
%token <sym> VAR FUNC UNDF
```

```
%type <val> expr assign
%type <val> prog
%right '='                          /* операции */
%left '+' '-'
%left '*' '/'
%right '^'
%left UNMIN
%%
prog:
  '\n' { printf("!\n"); }
| expr '\n'{ $$=$1; printf("=>%lf\n",$$); }
| prog expr '\n' { $$=$2; printf("->%lf\n",$$); }
;
assign: VAR '=' expr
          { $$ = $1->u.val=$3; $1->type=VAR; }
;
expr: NUM { $$ = $1; }
  | VAR { $$ = $1->u.val; }
  | FUNC '(' expr ')' { $$ = (($1->u.ptr)($3)); }
  | expr '+' expr { $$ = $1+$3; }
  | expr '-' expr { $$ = $1 - $3; }
  | expr '*' expr { $$ = $1*$3; }
  | expr '/' expr { if ($3!=0) $$ = $1/$3;
          else { printf(" Zero divide\n"); $$=$1; }
                        }
  | expr '^' expr { $$ = pow($1,$3); }
  | '-' expr %prec UNMIN { $$ = -$2; }
  | assign /* присваивание */
  | '(' expr ')' { $$ = $2; }
;
%%
Symbol *install(char *s, short t, double d)
{ /* добавление нового элемента в список */
  Symbol *sp;
    sp = (Symbol *) malloc(sizeof(Symbol));
    sp->name = (char *)malloc(strlen(s)+1);
    strcpy(sp->name,s);
    sp->type = t; sp->u.val=d; sp->next=symlist;
    symlist=sp;
    return sp;
}
Symbol *lookup(char *s)
{      /* поиск в списке */
  Symbol *sp;
  for (sp=symlist; sp!=(Symbol *)0; sp=sp->next)
    if (strcmp(sp->name,s)==0)
    {
      return sp;
    }
```

```
    return 0;
}
Init()
{           /* начальное заполнение списков*/
  int i; Symbol *s;

    for(i=0; Const[i].name; i++)
    install(Const[i].name,VAR,Const[i].cval);
    for(i=0; Funcs[i].name; i++)
    {
      s=install(Funcs[i].name,FUNC,0.0);
      s->u.ptr=Funcs[i].func;
    };
}
yylex()
{               /* лексический разбор */
  int c;
    Symbol *s;
    char sbuf[100], *p=sbuf;
  while((c=getchar())==' ');
if (c==EOF) return 0;
if(isalpha(c))
{
    do
      { *p++=c; }
    while (((c=getchar())!=EOF) && isalnum(c));
    ungetc(c,stdin); *p='\0';
    if ((s=lookup(sbuf))==0)
                s=install(sbuf,UNDF,0.0);
    yylval.sym=s;
    return ((s->type==UNDF)? VAR: s->type);
}
if (c=='.' || isdigit(c))
    {
       ungetc(c,stdin); scanf("%lf",&yylval);
      return NUM;
    };
  return c;
}
double Sqrt(double x)
{
  if (x<0.0) {
    yyerror(" Sqrt отрицательного числа. ");
    return (0.0);
              };
  return sqrt(x);
}

yyerror(char *s)
  {
```

```
    printf("Error: %s\n",s);
  }
int main()
  {    Init();
       yyparse();
       return 0;
  }
```

Лабораторная работа 8
Матричный калькулятор

Расширить калькулятор с переменными и функциями, добавив в него возможность работы с матрицами (сложение, вычитание, умножение матрицы на число, перемножение матриц, вычисление определителя и обратной матрицы, ввод и вывод матрицы).

Требования

Четыре часа на выполнение, языки реализации – уасс и С. Использовать библиотеки матричных функций.

Неформальное описание грамматики:

```
Define %a[5][6], %b[6][3], %frag[1][20], c%[5][3] // описание матриц
Input %a, %b            // ввод матриц с консоли
%c = %a * %b            // умножение матриц
Output %c               // вывод матриц на консоль
Define %d[3][3]         // описание квадратной матрицы
Input %d
%d=inverse(%d)          // вычисление обратной матрицы
q=determinant(%d)       // вычисление определителя
```

Алгоритмизация задачи

Можно использовать известные библиотеки исходных текстов для матричных алгоритмов. В лексическом распознавателе придется анализировать символьные строки, начинающиеся с % (признак того, что имя описывает матрицу).

Реализация

Элемент стека может теперь содержать и матрицу (вернее, указатель на нее). В структуру, описывающую матрицу, должны входить размерности, массив данных и имя матрицы.

Тестирование

Убедитесь, что ваш калькулятор обнаруживает ошибки вовремя, не складывает, например, матрицы разного размера, при умножении про-

веряет значение размерностей как входной, так и результирующей матриц. Перед выполнением обращения матрицы проверьте значение ее определителя.

Установите максимальный размер матриц, с которыми ваш калькулятор еще работает (или работает за приемлемое время).

Лабораторная работа 9
Модель экосистемы

Требуется написать программу, реализующую имитационное моделирование развития экологической системы. Исходные данные описывают взаимоотношения между отдельными видами животных. Взаимоотношения между животными строятся на основе модели хищник-жертва, т.е. одни животные поедают других, мигрируя в сторону наличия пищи и отсутствия хищников.

Реализовать графический интерфейс к программе, желательно используя FLTK.

Требования

Шесть часов на выполнение, языки реализации – уасс и C. Использовать средства разработки графического интерфейса FLTK. Работа выполняется коллективно, 3–4 человека на один проект. Перед работой выполнить декомпозицию проекта, распределить обязанности и построить ленточный и сетевой график выполнения проекта (применить программу из лабораторной работы 1 для нахождения критического пути), если необходимо, скорректировать распределение видов деятельности.

Алгоритмизация задачи

Решение поставленной задачи сводится к пошаговому изменению состояния замкнутой системы, т.е. алгоритм действует по итерационному принципу, когда в новое состояние система переходит на основе предыдущего состояния и известных операций.

Для данной задачи состоянием является набор матриц, в каждой ячейке которых записано количество животных одного вида. Операциями, которые переводят систему к новому состоянию, являются следующие процессы:

- Рождение нового поколения животных,
- Съедение жертв хищниками,
- Гибель животных от голода,
- Миграция животных.

Реализация

Программу можно разделить на две основные части: интерфейсную и расчетную.

Интерфейсная часть организует ввод данных и визуализацию экологической системы, расчетная часть реализует все те необходимые операции, которые переводят систему из одного состояния в другое.

Программа может быть написана с использованием объектно-ориентированного подхода или структурного программирования.

Интерфейсную часть программы можно разделить на два блока. Первый блок должен осуществлять ввод исходных данных, а второй блок должен осуществлять визуализацию полученных результатов.

Исходные данные состоят из двух частей: файла, в котором описываются отношения между различными видами животных, и файлов, в которых хранятся карты распределения животных по обитаемому пространству (пространство для данной модели экосистемы имеет форму тора, т.е. верхняя граница совмещена с нижней, а левая – с правой).

Для задания отношений между животными необходимо разработать язык, желательно с помощью уасс. Язык, задающий отношения, может иметь следующий синтаксис (между заглавными и прописными буквами разницы нет):

Файл имеет расширение *.dat,

Файл начинается с ключевого слова start,

Файл заканчивается на ключевое слово finish.

После слова start следует конструкция “size xx,yy;”, в которой задаются размеры обитаемого пространства.

Затем следует конструкция “sid nn,...,nn;” в которой перечисляются идентификаторы для различных видов животных. Каждому виду животных должен соответствовать свой идентификатор. Идентификатором должно являться число от 0 до 19.

Далее следуют уже конкретные описания видов животных, заключенные в конструкции “begin” – -“end”.

Порядок следования ключевых слов имеет значение.

Типичный файл, описывающий такие отношения, может иметь знак комментария “//”.

5. ОФОРМЛЕНИЕ ОТЧЕТОВ

Отчет по лабораторной работе – готовая программа (в исходных текстах, готовая к трансляции, с комментариями), наборы тестов (для работ 2, 3, 5, 6), командный файл для сборки проекта, если программа состоит из нескольких модулей (или dsp/dsw файл для Visual C).

Все задания должны быть выполнены в максимальной степени платформенно и системно независимыми. Обращайтесь к справочной информации в компиляторе (или msdn) для уверенности в совместимости с UNIX.

Отчет упаковывается архиватором zip (winzip) или gzip/bz2 и отсылается по электронной почте преподавателю. При проверке будет учитываться время получения почтовым сервером ВГУЭС.

СПИСОК ЛИТЕРАТУРЫ

Ахо А., Хопкрофт Д., Ульман Дж. Структуры данных и алгоритмы. – М.; СПб: Вильямс, 2001.

Голуб А.Ф. Правила программирования С и С++. – М.: Бином, 1996.

Грис Д. Наука программирования. – М.: Мир, 1984.

Кнут Д. Искусство программирования для ЭВМ. – М.: Мир, 1978. Т. 1–3 (переизданы в 2001 г.)

Кузин Л.Т. Основы кибернетики. – М.: Энергия, 1973. Т. 1. Гл. 14.

Турский В. Методология программирования. – М.: Мир, 1981.

Ресурсы в интранете

bkv.vvsu.ru, IP=192.168.8.2 (на лето 2003 г.).

ftp://bkv.vvsu.ru/pub/TP – материалы к занятиям по дисциплине «Технология программирования», компилятор рсуасс, примеры выполненных курсовых и лабораторных работ, постоянно меняющееся содержание в соответствии с текущими заданиями.

ftp://bkv.vvsu.ru/LIBRARY/Books – книги по программированию.

ftp://bkv.vvsu.ru/LIBRARY/BSP – Библиотека системного программиста Фроловых, тт.1–34 (на 1.1.03).

ftp://bkv.vvsu.ru/LIBRARY/SOFT – Программное обеспечение и алгоритмы.

ftp://bkv.vvsu.ru./DISTR/ – последние дистрибутивы программного обеспечения, преимущественно для Linux.

ftp://bkv.vvsu.ru/LIBRARY/Contests, ftp://bkv.vvsu.ru/LIBRARY/C-Contest, ftp://bkv.vvsu.ru/LIBRARY/ACM-contest – материалы конкурсов по программированию.

ОГЛАВЛЕНИЕ

Учебное издание

Васильев Борис Константинович

ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ ГОТОВЫХ РЕШЕНИЙ В ПРОГРАММИРОВАНИИ

Лабораторный практикум