Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

004
О-294

№ 3902

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам
для студентов II курса ФМПИ

НОВОСИБИРСК
2010

УДК 004.434(076.5)
О-294

Составитель д-р техн. наук, проф. *Д.В. Лисицин*

Рецензент канд. техн. наук, доц. *И.Л. Еланцева*

Работа подготовлена на кафедре прикладной математики

ЛАБОРАТОРНАЯ РАБОТА № 1

ПРОГРАММИРОВАНИЕ ГРАФИКИ

1. ЦЕЛЬ РАБОТЫ

Освоить необходимые средства для написания программ на языке C++, работающих с графикой.

2. СОДЕРЖАНИЕ РАБОТЫ

1. Изучить основные понятия и функции графического интерфейса операционной системы Microsoft Windows.

2. По предложенному преподавателем варианту разработать функции, рисующие следующие геометрические фигуры:

- незакрашенную фигуру (фигуру-контур);
- закрашенную фигуру;
- две вложенные одна в другую фигуры, внешняя фигура закрашена, за исключением пространства внутренней фигуры.

Разработать программу, демонстрирующую выполнение указанных функций. Ввод параметров фигур (координат и др.), параметров рисуемых линий и закраски осуществлять из файлов (отдельно для каждого теста). Включить в программу проверки корректности данных (нулевой радиус окружности, нарушение неравенства треугольника и т. д.), в том числе проверки нахождения фигуры в пределах окна и вложенности двух фигур.

3. Подготовить текстовый файл с разработанной программой, оттранслировать, собрать и выполнить программу с учетом требований операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, алгоритм, описание разработанных функций, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

3.1. КОНТЕКСТ ОТОБРАЖЕНИЯ

Для вывода текста и графики на экран компьютера в операционной системе Microsoft Windows используется интерфейс графических устройств (Graphic Device Interface или GDI). Поскольку интерфейс GDI – весьма сложная компонента, мы будем изучать лишь отдельные его элементы.

Из чего состоит интерфейс GDI с точки зрения приложения?

Прежде всего, это контекст отображения и инструменты для рисования. Контекст отображения можно сравнить с листом бумаги, на котором приложение рисует то или иное графическое изображение, а также пишет текст. Инструменты для рисования – это перья, кисти (а также шрифты и даже целые графические изображения), с помощью которых создается изображение. Кроме контекста отображения и инструментов для рисования приложениям доступны десятки функций программного интерфейса GDI, предназначенных для работы с контекстом отображения и инструментами.

Если говорить более точно, контекст отображения является структурой данных, описывающей устройство отображения. В этой структуре хранятся различные характеристики устройства и набор инструментов для рисования, выбранный по умолчанию. Приложение может выбирать в контекст отображения различные инструменты (например, перья различной толщины и цвета). Поэтому если вам надо нарисовать линию красного или зеленого цвета, перед выполнением операции следует выбрать в контекст отображения соответствующее перо.

Функции рисования не имеют параметров, указывающих цвет или толщину линии. Такие параметры хранятся в контексте отображения.

Чтобы начать работу с контекстом отображения, его надо «получить». Получить контекст отображения можно с помощью функции GetDC.

Функция GetDC возвращает контекст отображения (типа HDC) для окна с идентификатором hwnd (типа HWND):

```
HWND hwnd;
…
HDC hdc = GetDC (hwnd);
```

Полученный таким образом контекст отображения можно использовать для рисования во внутренней области окна. Как получить идентификатор окна hwnd, мы рассмотрим в п. 3.5.

После завершения процедуры рисования следует освободить полученный контекст отображения, вызвав функцию ReleaseDC.

Функция ReleaseDC освобождает контекст отображения hdc, полученный для окна hwnd:

```
ReleaseDC (hwnd, hdc);
```

Каждый раз, когда приложение получает контекст отображения, его атрибуты принимают значения по умолчанию. Если перед выполнением рисования приложение изменит атрибуты контекста отображения, вызвав соответствующие функции GDI, в следующий раз при получении контекста отображения эти атрибуты снова примут значения по умолчанию. Поэтому установка атрибутов должна выполняться каждый раз после получения контекста отображения.

Полученный контекст отображения имеет систему координат, изображенную на рис. 1.

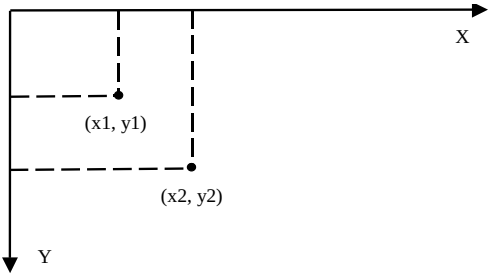


Рис. 1

Начало системы координат располагается в левом верхнем углу экрана. Ось X направлена слева направо, ось Y – сверху вниз. В качестве единицы длины используется пиксель. Пиксель (pixel от англ. picture element) – маленькое светящееся пятно. Пиксель является наименьшим элементом изображения дисплеев, в которых изображение складывается из таких отдельных точек.

Для определения размера области окна, в которую можно осуществлять вывод (точнее, ее координат в пикселях), предназначена функция GetClientRect:

```
RECT rt; //структура с полями left, top, right, bottom
GetClientRect (hwnd, &rt);
```

Координаты записываются во второй параметр, причем поля left, top всегда нулевые, а поля right, bottom содержат соответственно ширину и высоту внутренней области окна:

```
int maxX= rt.right, maxY = rt.bottom;
```

3.2. ИНСТРУМЕНТЫ ДЛЯ РИСОВАНИЯ

3.2.1. ПЕРО

Перья используются для рисования линий и простейших геометрических фигур. Приложение может выбрать одно из трех предопределенных перьев либо создать собственное, выбрав нужный цвет и стиль. Интерфейс GDI позволяет рисовать сплошные линии различной ширины, а также пунктирные, штриховые и штрихпунктирные линии шириной в 1 пиксель.

Для выбора встроенного пера лучше всего воспользоваться макрокомандами GetStockPen и SelectPen.

Макрокоманда GetStockPen возвращает идентификатор встроенного пера (типа HPEN), заданного ее параметром:

```
HPEN hBlackPen = GetStockPen (BLACK_PEN);
```

Для параметра можно выбрать одно из следующих значений.

Значение	Описание
BLACK_PEN	Перо, рисующее черную линию толщиной в один пиксель (для любого режима отображения). Это перо выбрано в контекст отображения по умолчанию
WHITE_PEN	Перо белого цвета. Толщина пера также равна одному пикселю и не зависит от режима отображения
NULL_PEN	Невидимое перо толщиной в один пиксель. Используется для рисования замкнутых закрашенных фигур (таких, как прямоугольник или эллипс) в тех случаях, когда контур фигуры должен быть невидимым

После получения идентификатора пера его необходимо выбрать в контекст отображения при помощи макрокоманды SelectPen:

```
HPEN hOldPen;
hOldPen = SelectPen (hdc, hBlackPen);
```

Первый параметр этой макрокоманды используется для указания идентификатора контекста отображения, в который нужно выбрать перо, второй – для передачи идентификатора пера.

Макрокоманда SelectPen возвращает идентификатор пера, который был выбран в контекст отображения раньше. Вы можете сохранить этот идентификатор и использовать его для восстановления старого пера.

Однако при помощи встроенных перьев вы не можете нарисовать цветные, широкие, штриховые и штрихпунктирные линии.

Если вас не устраивают встроенные перья, вы можете легко создать собственные. Для этого нужно воспользоваться функцией CreatePen.

Функция CreatePen позволяет определить стиль, ширину и цвет пера:

```
HPEN hRedPen = CreatePen (PS_SOLID, 3, RGB(255, 0, 0));
```

Первый параметр определяет стиль линии и может принимать одно из следующих значений.

Стиль линии	Описание
PS_SOLID	Сплошная
PS_DASH	Пунктирная
PS_DOT	Штриховая
PS_DASHDOT	Штрихпунктирная, одна точка на одну линию
PS_DASHDOTDOT	Штрихпунктирная, две точки на одну линию
PS_NULL	Невидимая
PS_INSIDEFRAME	Линия, предназначенная для обводки замкнутых фигур

Второй параметр определяет ширину пера в пикселях. Учтите, что для линий PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT можно использовать только единичную или нулевую ширину, в обоих случаях ширина линии будет равна одному пикселю.

Третий параметр имеет тип COLORREF и задает цвет пера. Для задания цвета удобно использовать макрокоманду RGB(r, g, b), позволяющую сконструировать цвет из отдельных компонент красного (r), зеленого (g) и голубого (b) цвета, указав в качестве значений парамет-

ров интенсивность отдельных цветов в диапазоне от 0 до 255. Например, цвет для красного пера задан нами в виде RGB(255, 0, 0), черный цвет задается как RGB(0, 0, 0), белый – как RGB(255, 255, 255). Три параметра макрокоманды RGB позволяют задать любой из примерно 16 млн цветов и оттенков.

На первый взгляд линии PS_SOLID и PS_INSIDEFRAME похожи, однако между ними имеются различия, особенно заметные для широких линий. Широкая линия, имеющая стиль PS_SOLID, располагается по обе стороны оси, заданной координатами линии. Линии, имеющие стиль PS_INSIDEFRAME, располагаются внутри контура, определяющего размеры замкнутой фигуры (рис. 2).

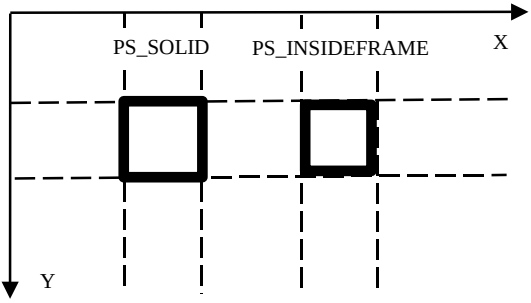


Рис. 2

Можно создать несколько различных перьев, но чтобы воспользоваться одним из них для рисования, вновь его необходимо выбрать в контекст отображения при помощи макрокоманды SelectPen.

Созданные вашим приложением перья принадлежат GDI, соответствующие структуры данных располагаются в памяти. Поэтому если перо больше не нужно, его нужно удалить для освобождения памяти.

Прежде чем удалять созданное вами перо, следует выбрать в контекст отображения другое перо (например, то, которое использовалось раньше). После этого для удаления вашего пера нужно вызвать макрокоманду DeleletePen. В качестве параметра этой макрокоманде необходимо передать идентификатор удаляемого пера.

```
SelectPen (hdc, hOldPen);
DeletePen (hRedPen);
```

Нельзя удалять перо, если оно выбрано в контекст отображения. Нет никакого смысла в удалении встроенных перьев.

Продолжая аналогию между контекстом отображения и листом бумаги, на котором рисуются графические изображения и пишется текст, можно сказать, что атрибут цвета фона (background color) в контексте отображения соответствует цвету бумаги. По умолчанию в контексте отображения выбран фон белого цвета.

С помощью функций SetBkColor и GetBkColor вы можете соответственно установить и определить текущий цвет фона, который используется для закраски промежутков между штрихами и точками линий:

```
SetBkColor (hdc, RGB(0, 0, 0));
…
COLORREF  rgb = GetBkColor (hdc);
```

С помощью трех макрокоманд GetRValue, GetGValue, GetBValue вы можете определить отдельные цветовые компоненты для значения типа COLORREF, возвращаемого функцией GetBkColor:

```
printf ("Color is (%d,%d,%d)\n", GetRValue(rgb), GetGValue(rgb),
GetBValue(rgb));
```

3.2.2. КИСТЬ

Для закрашивания внутренней области окна приложения или замкнутой геометрической фигуры можно использовать кисть (brush) – графические изображения небольшого (8×8 пикселей) размера.

Для выбора одной из встроенных кистей вы можете воспользоваться макрокомандой GetStockBrush:

```
HBRUSH hBlackBrush = GetStockBrush (BLACK_BRUSH);
```

В качестве параметра для этой макрокоманды можно использовать следующие значения.

Значение	Описание
BLACK_BRUSH	Кисть черного цвета
WHITE_BRUSH	Кисть белого цвета
GRAY_BRUSH	Серая кисть
LTGRAY_BRUSH	Светло-серая кисть
DKGRAY_BRUSH	Темно-серая кисть
NULL_BRUSH	Бесцветная кисть, которая ничего не закрашивает
HOLLOW_BRUSH	Синоним ДЛЯ NULL_BRUSH

Как видно из только что приведенной таблицы, в Windows есть только монохромные встроенные кисти.

Макрокоманда GetStockBrush возвращает идентификатор встроенной кисти.

Прежде чем использовать полученную таким образом кисть, ее надо выбрать в контекст отображения (так же, как и перо). Для этого проще всего воспользоваться макрокомандой SelectBrush:

```
HBRUSH hOldBrush;
hOldBrush = SelectBrush (hdc, hBlackBrush);
```

Макрокоманда SelectBrush возвращает идентификатор старой кисти, выбранной в контекст отображения раньше.

Если вам нужна цветная кисть, ее следует создать с помощью функции CreateSolidBrush:

```
HBRUSH hGreenBrush = CreateSolidBrush (RGB(0, 255, 0));
```

В качестве параметра для этой функции необходимо указать цвет кисти.

Можно создать несколько различных кистей, но чтобы воспользоваться одной из них для рисования, ее необходимо выбрать в контекст отображения при помощи макрокоманды SelectBrush.

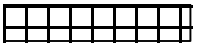
После использования созданной вами кисти ее следует удалить, не забыв перед этим выбрать в контекст отображения старую кисть. Для удаления кисти следует использовать макрокоманду DeleteBrush:

```
SelectBrush (hdc, hOldBrush);
DeleteBrush (hGreenBrush);
```

Приложение может заштриховать внутреннюю область замкнутой фигуры, создав одну из шести кистей штриховки функцией CreateHatchBrush:

```
HBRUSH hCrossBrush = CreateHatchBrush (HS_CROSS, RGB(0, 0, 255));
```

С помощью второго параметра вы можете определить цвет линий штриховки. Первый параметр задает стиль штриховки.

Стиль штриховки	Внешний вид
HS_BDIAGONAL	
HS_CROSS	
HS_DIAGCROSS	

HS_FDIAGONAL 

HS_HORIZONTAL 

HS_VERTICAL 

3.3. РИСОВАНИЕ ГЕОМЕТРИЧЕСКИХ ФИГУР

3.3.1. РИСОВАНИЕ ТОЧКИ

Функция рисования точки SetPixel устанавливает цвет точки с заданными координатами:

```
COLORREF  rgb = SetPixel (hdc, 10, 10, RGB(255, 0, 255));
```

Первый параметр определяет контекст отображения, следующие два целочисленных параметра – координаты точки, последний параметр определяет новый цвет точки.

Функция SetPixel возвращает цвет, который фактически был использован для рисования точки. Возвращенное значение может отличаться от заданного последним параметром при ограниченных возможностях оборудования. В случае ошибки оно будет равно –1.

Функция GetPixel позволяет узнать цвет точки, заданной идентификатором контекста отображения и координатами:

```
rgb = GetPixel (hdc, 20, 20);
```

С помощью макрокоманд GetRValue, GetGValue, GetBValue (см. разд. 3.2.1) вы можете определить отдельные цветовые компоненты для значения, возвращаемого функциями SetPixel и GetPixel.

Функции SetPixel и GetPixel используются достаточно редко, так как для построения графических изображений есть более мощные функции.

3.3.2. РИСОВАНИЕ ЛИНИЙ

Приложения Windows могут рисовать прямые и ломаные линии, а также дуги эллипса (и окружности, как частного случая эллипса). Параметры этих линий определяются атрибутами контекста отображения (цвет фона, перо и др.).

Текущая позиция пера

Для рисования прямых линий (и только для этого) в контексте отображения хранятся координаты текущей позиции пера. Текущая позиция пера – это графический аналог курсора в текстовом режиме, только он является невидимым. По умолчанию текущая позиция пера равна значению (0, 0).

Для изменения текущей позиции пера служит функция MoveToEx:

```
POINT pt;
BOOL bxt = MoveToEx (hdc, x, y, &pt);
```

Для контекста отображения hdc эта функция устанавливает текущую позицию пера, равную (x, y). В структуру типа POINT (имеет два целочисленных поля x, y), на которую указывает последний параметр, после возврата из функции будут записаны старые координаты пера. Если вам не нужно предыдущее текущее положение пера, вы можете просто установить последний параметр в NULL. MoveToEx возвращает TRUE при нормальном завершении и FALSE при ошибке.

Чтобы узнать текущую позицию пера, приложение может использовать функцию GetCurrentPositionEx:

```
bxt = GetCurrentPositionEx (hdc,&pt);
```

После вызова этой функции текущая позиция пера будет записана в структуру типа POINT, на которую указывает последний параметр. Функция возвращает TRUE при нормальном завершении или FALSE при ошибке.

Рисование прямой линии

Для того чтобы нарисовать прямую линию, приложение должно воспользоваться функцией LineTo:

```
bxt = LineTo (hdc, xEnd, yEnd);
```

Эта функция рисует линию из текущей позиции пера, установленной ранее функцией MoveToEx, в точку с координатами (xEnd, yEnd). После того как линия будет нарисована, текущая позиция пера станет равной (xEnd, yEnd).

Функция LineTo возвращает TRUE при нормальном завершении или FALSE при ошибке.

Таким образом, для того чтобы нарисовать прямую линию, приложение должно сначала с помощью функции MoveToEx установить текущую позицию пера в точку, которая будет началом линии, а затем вызвать функцию LineTo, передав ей через параметры xEnd и yEnd координаты конца линии.

Особенностью функции LineTo является то, что она немного не дорисовывает линию – эта функция рисует всю линию, не включая ее конец, т. е. точку (xEnd, yEnd). Это иллюстрирует рис. 3.

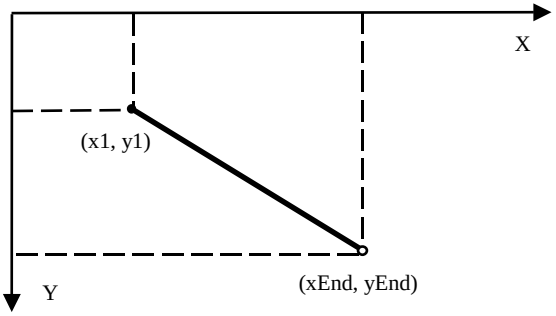


Рис. 3

Нарисуем с помощью рассмотренных функций красный прямоугольник.

```
SelectPen (hdc, hRedPen);
MoveToEx (hdc, 100, 100,&pt);
LineTo (hdc, 100, 200);
LineTo (hdc, 200, 200);
LineTo (hdc, 200, 100);
LineTo (hdc, 100, 100);
```

Рисование ломаной линии

Функции Polyline, предназначенной для рисования ломаных линий, следует передать идентификатор контекста отображения hdc, указатель на массив структур POINT, в котором должны находиться координаты начала ломаной линии, координаты точек излома и координаты конца ломаной линии, а также размер этого массива:

```
POINT ppt[4]={ {100,100}, {200,200}, {100,300}, {100,100} };
bxt = Polyline (hdc, ppt, 4);
```

Функция возвращает TRUE при нормальном завершении или FALSE при ошибке. Она не использует текущую позицию пера и не изменяет ее.

Если ломаная линия не замкнута, ее последняя точка не рисуется (рис. 4).

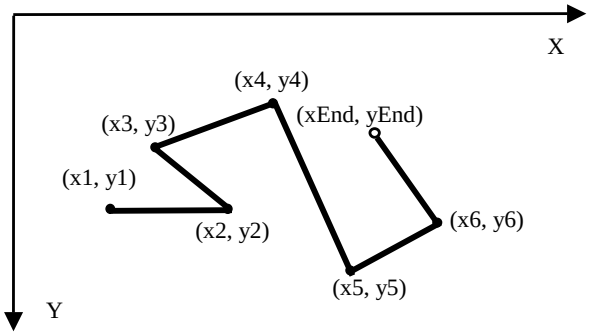


Рис. 4

Закраска фигур

Если вы нарисовали замкнутую геометрическую фигуру, то можно закрасить ее внутреннюю область. Для этого служит функция FloodFill:

```
bxt = FloodFill (hdc, nX, nY, RGB(0, 255, 255));
```

Для функции FloodFill необходимо указать идентификатор контекста отображения hdc, координаты точки внутри области nX, nY и цвет контура, ограничивающего область.

Функция возвращает TRUE при успешном завершении или FALSE при ошибке (ошибка возникает, например, в том случае, когда цвет точки (nX, nY) совпадает с цветом контура).

Если контур, ограничивающий область, окажется незамкнутым, краска «прольется» на остальную часть экрана.

Для закраски используется кисть, выбранная в контекст отображения.

Таким же образом можно закрасить и внутреннюю область окна приложения.

Для рисования закрашенных фигур можно воспользоваться также функциями, описанными в разд. 3.3.3.

3.3.3. РИСОВАНИЕ ЗАМКНУТЫХ ФИГУР

Помимо линий приложения Windows могут использовать функции GDI для рисования замкнутых закрашенных или незакрашенных фигур, таких как прямоугольники, эллипсы, сегменты и сектора эллипсов, многоугольники с прямыми и скругленными углами и т. д.

Для закрашивания внутренней области замкнутых фигур используется кисть, задаваемая как атрибут контекста отображения. Внешний контур фигуры обводится пером, которое также выбирается в контекст отображения.

Рисование прямоугольника

Простейшая функция, с помощью которой можно нарисовать прямоугольник, называется Rectangle:

```
bxt = Rectangle (hdc, x1, y1, x2, y2);
```

Функция Rectangle рисует прямоугольник для контекста отображения hdc, возвращая значение TRUE в случае успеха или FALSE при ошибке. Назначение остальных параметров иллюстрирует рис. 5.

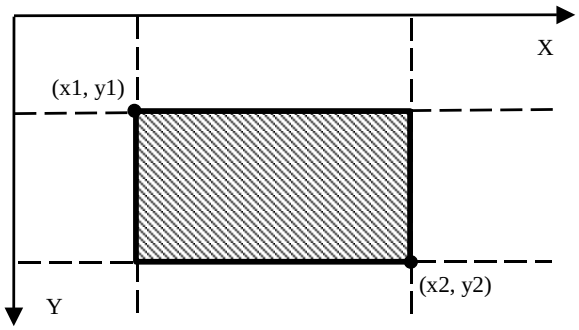


Рис. 5

Как видно из этого рисунка, последние четыре параметра функции задают координаты левого верхнего и правого нижнего углов прямоугольника.

В зависимости от стиля пера граница фигуры может находиться полностью внутри прямоугольника, заданного координатами (x1, y1), (x2, y2), или выходить за пределы (см. рис. 2). Если выбрать стиль пера PS_NULL, граница фигуры станет невидимой.

В зависимости от кисти, выбранной в контекст отображения, внутренность прямоугольника может быть закрашенной в тот или иной цвет, заштрихована одним из нескольких способов.

Рисование эллипса

Для рисования эллипса вы можете использовать функцию Ellipse:

```
bxt = Ellipse (hdc, x1, y1, x2, y2);
```

Первый параметр этой функции указывает идентификатор контекста отображения, остальные (типа int) – координаты левого верхнего и правого нижнего углов прямоугольника, в который должен быть вписан эллипс (рис. 6).

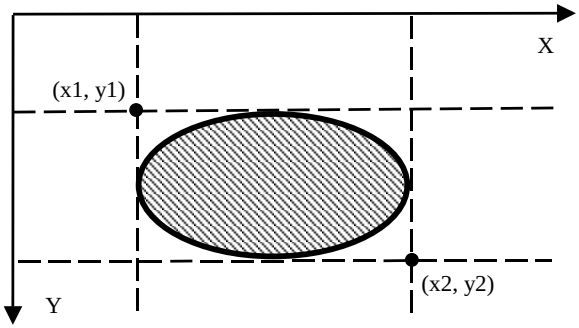


Рис. 6

Рисование многоугольников

Рисование многоугольников (рис. 7) выполняется функцией Polygon, аналогичной по своим параметрам функции Polyline, с помощью которой рисуются ломаные линии:

```
bxt = Polygon (hdc, ppt, 3);
```

Через параметр hdc передается идентификатор контекста отображения. Второй параметр указывает на массив структур POINT, в котором должны находиться координаты вершин многоугольника. Третий параметр определяет размер этого массива.

Функция Polygon возвращает TRUE при нормальном завершении или FALSE при ошибке. Она не использует текущую позицию пера и не изменяет ее.

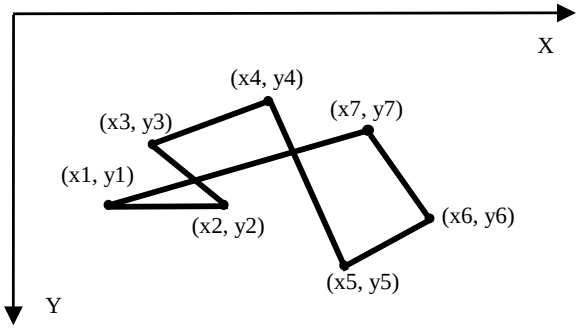


Рис. 7

В массиве структур POINT, определяющих вершины многоугольника, каждая вершина должна быть указана один раз. Функция Polygon автоматически замыкает ломаную линию, образующую многоугольник.

3.4. ВЫВОД ТЕКСТА

Одной из функций графического интерфейса, служащей для вывода текста в окно, является TextOutA. Она имеет пять параметров:

```
bxt = TextOutA (hdc, xStart, yStart, PStr, lenstr);
```

Первый параметр (hdc) определяет используемый контекст отображения. Второй (xStart) и третий (yStart) параметры задают координаты (x, y) начальной позиции, начиная с которой будет выполняться вывод текста.

Четвертый параметр (PStr) является указателем на выводимую строку, длина которой определяется последним, пятым параметром (lenstr).

Как нетрудно заметить, функция TextOutA не имеет параметров, определяющих шрифт, размер букв, цвет фона, цвет букв и т. п. Все эти характеристики текста хранятся в структуре контекста отображения и могут изменяться. Мы рассмотрим лишь выбор цвета букв. Для этого необходимо вызвать функцию SetTextColor:

```
COLORREF Old_rgb = SetTextColor (hdc, rgb);
```

Параметр hdc определяет контекст отображения, для которого вы собираетесь изменить цвет текста. Второй параметр определяет цвет.

Функция SetTextColor возвращает значение цвета, которое использовалось для вывода текста раньше.

В любой момент времени вы можете определить текущий цвет, используемый для вывода текста. Для этого следует вызвать функцию GetTextColor:

```
rgb  = GetTextColor (hdc);
```

Для указанного при помощи единственного параметра контекста отображения функция возвращает цвет, используемый для вывода текста.

3.5. ВЫВОД ГРАФИКИ В КОНСОЛЬНОМ ПРИЛОЖЕНИИ

Одним из типов приложений (программ) в Windows является консольное приложение (в Microsoft Visual Studio соответствующий шаблон доступен из меню File/New/Project, тип проекта Visual C++/Win32, шаблон «Win32 Console Application», установить флажок Application Settings/Additional options/ Empty project). Консольное приложение работает в текстовом режиме, позволяя осуществлять ввод с клавиатуры и вывод на экран. При этом программист избавляется от необходимости разрабатывать какой-либо оконный интерфейс.

Для работы с графикой обычно разрабатывается некоторый оконный интерфейс. Однако и в окно консольного приложения можно выводить различные графические элементы с использованием интерфейса GDI. При этом для получения контекста отображения функции GetDC необходимо передать идентификатор окна консольного приложения.

Для определения идентификатора окна консольного приложения можно использовать функцию GetConsoleWindow:

```
HWND hwnd = GetConsoleWindow ( );
```

Однако если при работе консольного приложения используется консольный ввод-вывод, то его надо будет «разделять» с графикой. Если это неудобно, то может быть целесообразным вывод графики в окно другого приложения, например, в окно стандартного текстового редактора «Блокнот».

Для определения идентификатора окна в этом случае можно использовать функцию FindWindowA:

```
hwnd = FindWindowA (lpClassName, lpWindowName);
```

Параметры lpClassName и lpWindowName – указатели на строки, содержащие имя оконного класса и заголовок искомого окна, при этом можно задать только один из параметров, указав в качестве другого параметра значение NULL. Функция FindWindowA в случае обнаружения искомого окна возвращает его идентификатор. Для редактора «Блокнот» оконный класс имеет имя «Notepad».

Рассмотренный прием вывода графики в консольном приложении достаточен для обучения, однако является нестандартным. Последнее проявляется в том, что программист не может полноценно управлять окном, в которое графика выводится, не все графические функции выполняются в нем адекватно нашей задаче. Так, при манипуляции окном (при прокрутке, изменении размера, при появлении окна или его части из-за другого окна и т. д.) приложение должно перерисовывать окно (или его часть). Поскольку приложение «не знает», что мы выводим графику в его окно, наши результаты будут стерты.

В заключение приведем пример простой программы, выводящей в окно консольного приложения текст и графику с использованием функций GDI.

```
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include <conio.h>
void main ( )
{
    // получаем идентификатор окна
    HWND hwnd = GetConsoleWindow ( );
    // получаем контекст отображения
    HDC hdc = GetDC (hwnd);
    RECT rt;
    char buf[100];
    // устанавливаем цвет фона
    SetBkColor (hdc, RGB(0, 0, 0));
    // устанавливаем цвет текста
    SetTextColor (hdc, RGB(255, 0, 0));
    // создаем красное перо
    HPEN hRedPen = CreatePen (PS_SOLID, 5, RGB(255, 0, 0));
    // и выбираем его в контекст отображения,
    // сохраняя предыдущее перо
    HPEN hOldPen = SelectPen (hdc, hRedPen);
    // создаем зеленую кисть
    HBRUSH hGreenBrush = CreateSolidBrush (RGB(0, 255, 0));
```

```
// и выбираем ее в контекст отображения,
// сохраняя предыдущую кисть
HBRUSH hOldBrush = SelectBrush (hdc, hGreenBrush);
// выводим строку стандартными средствами
printf("Graphics in Console Window.");
do{
        // получаем размер окна
        GetClientRect (hwnd, &rt);
        // формируем выводимую строку
        sprintf (buf, "Размер окна %d на %d пикселей",
        rt.right, rt.bottom);
        // выводим строку графическими средствами
        TextOutA (hdc, 10, 10, buf, strlen(buf));
        // рисуем закрашенный эллипс
        Ellipse (hdc, 10, 30, rt.right–10, rt.bottom–10);
} while (getch ( )!=27); // при нажатии любой клавиши
// (кроме Esc) перерисовываем изображение,
// изображение изменится, если изменились размеры окна,
// нажатие Esc – выход
// выбираем в контекст отображения предыдущее перо
SelectPen (hdc, hOldPen);
// выбираем в контекст отображения предыдущую кисть
SelectBrush (hdc, hOldBrush);
// удаляем красное перо
DeletePen (hRedPen);
// удаляем зеленую кисть
DeleteBrush (hGreenBrush);
// освобождаем контекст отображения
ReleaseDC (hwnd, hdc);
}
```

4. ВАРИАНТЫ ЗАДАНИЙ

1. Окружность.
2. Выпуклый четырехугольник.
3. Прямоугольник.
4. Квадрат.
5. Параллелограмм.
6. Трапеция.
7. Треугольник.
8. Ромб.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое «контекст отображения», как его получить и освободить?
2. Какую систему координат имеет контекст отображения, как определить размер окна?
3. Что такое «перо», какие имеются встроенные перья, как выбрать перо в контекст отображения?
4. Как создать новое перо, какие его характеристики можно задать, как удалить перо?
5. Что такое «кисть», какие имеются встроенные кисти, как выбрать кисть в контекст отображения?
6. Как создать новую кисть, какие ее характеристики можно задать, как удалить кисть?
7. Как нарисовать точку, как узнать цвет точки?
8. Что такое текущая позиция пера, как ее установить и узнать?
9. Как нарисовать прямую линию?
10. Как нарисовать ломаную линию?
11. Как нарисовать прямоугольник?
12. Как нарисовать эллипс?
13. Как нарисовать многоугольник?
14. Как вывести в окно текст, как установить цвет букв?

ЛАБОРАТОРНАЯ РАБОТА № 2

КЛАССЫ

1. ЦЕЛЬ РАБОТЫ

Изучить средства работы с классами на языке C++.

2. СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться со следующими вопросами и понятиями: определение типа данных пользователя с помощью конструкции class, открытая и закрытая части описания класса, определение набора функций класса, интерфейс и реализация класса, реализация понятия модульности в языке C++.

2. Модифицировать программу, разработанную в лабораторной работе № 1, так чтобы в ней был определен класс, реализующий понятие геометрической фигуры в графической системе.

2.1. Определить ответственность класса. Учесть, что общение с пользователем (включая ввод данных с клавиатуры и вывод данных на экран, если их предполагается использовать), а также реакцию на возникающие при работе с функциями класса ошибки следует производить вне функций класса.

2.2. Определить атрибуты, необходимые для реализации класса. Поместить атрибуты в закрытую часть описания.

2.3. Определить функции, необходимые для реализации класса. Выделить интерфейс класса и поместить его в открытую часть описания.

Включить в разработанный класс функции:

- устанавливающие и изменяющие геометрические и графические характеристики фигуры (set-функции);
- возвращающие геометрические и графические характеристики фигуры (get-функции);

• рисующие фигуру на экране;

• изменяющие положение фигуры на экране;

• обеспечивающие сохраняемость объекта: сохранение набора атрибутов объекта класса в файле и считывание его из файла (файлы для сохранения и считывания должны иметь один формат).

2.4 Обработку ошибок (нулевой радиус окружности, нарушение неравенства треугольника, другие ошибки задания фигуры, ошибки работы с графикой и др.) осуществлять с использованием механизма возбуждения и обработки исключительных ситуаций.

2.5. Разработать функцию, демонстрирующую поведение класса. Поместить реализацию класса в один файл, а демонстрационную функцию – в другой. Обеспечить необходимые межмодульные связи.

3. Подготовить текстовые файлы с разработанной программой, оттранслировать их, собрать и выполнить программу с учетом требований операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, описание разработанных классов (атрибутов и функций), алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

С точки зрения языка программирования класс объектов можно рассматривать как тип данных, а отдельные объекты – как данные этого типа.

Описание класса подобно описанию структуры, однако для класса используется ключевое слово class вместо struct.

В описание класса включаются данные (атрибуты) и функции, изменяющие их (функции-члены класса).

Описание класса разделяется на закрытую и открытую части, помеченные как private и public.

```
struct Point; //Точка
class Circle // класс окружность
{
        private:            // закрытая часть, содержит атрибуты и
```

```
                    // служебные функции
    Point Center;   // центр окружности
    int Radius;     // радиус окружности
    . . .
    public:         // открытая часть, содержит интерфейс
    void draw( ); // нарисовать окружность
    . . .
};
```

Открытая часть образует открытый интерфейс объектов класса. Это функции, используемые для взаимодействия с другими объектами и функциями. Атрибуты и функции в закрытой части могут использоваться только функциями-членами класса (а также друзьями класса).

С атрибутами объектов класса работают только посредством соответствующих функций. В частности, для получения значений атрибутов объекта и изменения их используются get- и set-функции соответственно.

```
Point get_Center ( );  // возвращает значение центра окружности
int get_Radius ( ); // возвращает значение радиуса окружности
void set_Center (Point new_center);  // изменяет центр окружности
void set_Radius (int new_radius); // изменяет радиус окружности
```

Объекты нового типа вводятся так же, как и переменные стандартных типов:

```
Circle C;
```

Функции-члены можно вызывать только для переменной соответствующего типа. Например, получить значение радиуса окружности можно так:

```
int r = C.get_Radius ( );
```

Для обеспечения сохраняемости объектов (набора их атрибутов) между сеансами работы программы их можно сохранить в файле с последующим чтением из него. Для этого необходимо описать соответствующие функции.

```
void load (char* file_name);
void save (char* file_name);
```

Описание класса обычно не содержит реализации (тела) функций-членов. Исключение составляют инлайн-функции. Реализация класса помещается отдельно от описания.

Реализация класса и пользовательский код (код, в котором будут использоваться объекты класса) помещаются в раздельно компилируемых частях программы.

Как правило, описание класса помещается в так называемый заголовочный файл, имеющий расширение h или hpp. Заголовочный файл обычно включается и в файл с пользовательским кодом, и в файл с реализацией класса. Это достаточно простой и эффективный способ обеспечить идентичность представления интерфейса в обоих файлах.

Файл с описанием класса Circle.h:

```
class Circle
{
        . . . // только описания
};
```

Файл с пользовательским кодом User.cpp:

```
#include "Circle.h"        // включить интерфейс
int main(void)
{
        . . .
        Circle C;
        . . .
C.set_Radius(r);
        . . .
}
```

Файл, содержащий реализацию класса Circle.cpp:

```
#include "Circle.h"        // включить интерфейс
Point Circle :: get_Center ( ) { return Center; };
int Circle :: get_Radius ( ) { return Radius; };
void Circle :: set_Center (Point new_center) { Center = new_center };
void Circle :: set_Radius (int new_radius) { Radius = new_ radius };
```

В случае нарушения какого-либо условия (возникновения ошибки) следует сгенерировать исключительную ситуацию (исключение).

Фрагмент кода, в котором может возникнуть исключение, помещается в пробный блок – блок try. Если при выполнении операторов, находящихся внутри блока try, происходит исключительная ситуация, то управление передается обработчикам исключений, которые задаются ключевым словом catch и находятся ниже блока try.

```
try{ // пробный блок
. . .
}
catch(int error){. . .}
```

Исключение генерируется посредством указания ключевого слова throw с необязательным аргументом-выражением.

```
throw 1;
```

Исключение будет обработано посредством вызова того обработчика catch, тип параметра которого будет соответствовать типу аргумента throw.

При наличии вложенных блоков try (например, из-за вложенности вызовов функций) будет использован обработчик самого глубокого блока. Если обработчик, соответствующий типу аргумента throw, на данном уровне не будет найден, будет осуществлен выход из текущей функции и поиск в блоке try с меньшей глубиной вложенности и т. д. После обработки исключения управление передается на оператор, следующий за описаниями обработчиков catch.

Рассмотрим пример изменения значения радиуса окружности. Изменим реализацию set-функции:

```
void Circle :: set_Radius (int new_radius)
{ if (new_ radius<=0) throw 0; Radius = new_ radius };
```

Фрагмент пользовательского кода будет следующим:

```
try{ C.set_Radius(rad); . . .}
catch(int error){if (error == 0) printf("Ошибка задания радиуса."); . . .}
```

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Принцип абстрагирования:
а) В чем заключается принцип абстрагирования?
б) Что такое барьер абстракции, уровни абстракции?
в) В чем заключается принцип наименьшего удивления?
г) Что такое контрактная модель программирования?
д) Что такое сигнатура операции?
е) Что такое инвариант?
2. Исключения:
а) Как определить пробный блок и обработчики исключений?
б) Как осуществить возбуждение исключительной ситуации?

в) Как происходит обработка исключений?

3. Инкапсуляция:

а) В чем заключается принцип инкапсуляции?

б) В чем смысл разделения класса на интерфейс и реализацию?

в) Что такое класс? Как он определяется? Как определяются операции для класса?

г) Как определяется открытая и закрытая части тела класса?

д) Что такое друг класса?

4. Модульность:

а) Что такое модуль?

б) В чем заключается принцип модульности?

в) Модульность на языке C++.

г) Какие существуют приемы эффективного разделения программы на модули?

5. Иерархичность:

а) В чем заключается принцип иерархичности?

б) Что такое иерархия «является»?

в) Что такое иерархия «имеет»?

6. Типизация:

а) В чем заключается принцип типизации?

б) Что такое полиморфизм?

в) Что такое принудительное приведение?

г) Что такое перегрузка?

д) Какие операторы приведения вы знаете?

7. В чем заключается принцип параллелизма?

8. В чем заключается принцип сохраняемости?

ЛАБОРАТОРНАЯ РАБОТА № 3

ОБЪЕКТЫ

1. ЦЕЛЬ РАБОТЫ

Изучить понятие объекта, средства его создания и уничтожения.

2. СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться с вопросами использования конструкторов и деструкторов для создания и уничтожения объектов.

2. По предложенному преподавателем варианту разработать программу на языке C++, в которой был бы определен класс-контейнер, т. е. класс объектов, служащих для хранения объектов класса, разработанного в предыдущей лабораторной работе. Среди различных типов контейнеров упомянем списки, стеки, очереди, деревья, таблицы. Контейнер должен быть реализован как динамическая структура данных. Разработать следующие функции-члены класса: конструктор, деструктор, функции для помещения объектов-фигур в контейнер, их возвращения (удаления), поиска в контейнере, «распечатки содержимого» контейнера – вывода информации о содержащихся в объекте-контейнере объектах и/или их графических образов, а также функции, обеспечивающие сохраняемость контейнера с использованием файла.

Реализацию класса-контейнера поместить в отдельный файл. Разработать функцию, демонстрирующую поведение объекта-контейнера с несколькими объектами-фигурами.

3. Подготовить текстовые файлы с разработанной программой, оттранслировать их, собрать и выполнить программу с учетом требований операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, описание разработанных классов, алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Важными видами операций являются конструктор и деструктор. Конструктор – это операция создания объекта и/или его инициализации; в C++ конструктор имеет то же имя, что и класс. Деструктор – это операция, освобождающая ресурсы, которые использует объект, и/или разрушающая сам объект; в C++ имя деструктора состоит из имени класса, перед которым ставится знак «тильда» – «~».

Данные операции обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса. Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается. Если программист не описал в классе конструктор и деструктор, то они будут созданы автоматически.

```
class Open_Stack { // класс-контейнер: «открытый» стек
int *s, length, top;
public:
        Open_Stack(int n); // конструктор, n – размер стека
        ~ Open_Stack( ); // деструктор
        void push(const int el); // помещение элемента в контейнер
        int pop( ); // возвращение элемента с удалением из контейнера
        bool find(const int el); // поиск элемента в контейнере, возвращаемое
                                 // значение – найден или нет
        . . .
};
```

При использовании динамических структур данных конструктор может использоваться для (первоначального) выделения памяти, а деструктор – для ее освобождения.

Рассмотрим конструктор и деструктор класса Open_Stack, реализуемого на основе массива, память под который выделяется и освобождается динамически.

```
Open_Stack(int  n):
      length(n), top(0) // инициализаторы конструктора
      {s = new int [length];} // выделение памяти
~Open_Stack( ) { delete [ ] s; } // освобождение памяти
```

В реализации последнего конструктора для инициализации отдельных частей объекта использованы инициализаторы конструктора. Важность инициализаторов в том, что только с их помощью можно инициализировать константные члены, члены, являющиеся ссылками, а также члены, являющиеся объектами класса, в котором есть один или несколько конструкторов, но отсутствует конструктор по умолчанию (конструктор без параметров).

Важно также, что такая инициализация выполняется эффективнее, поскольку создание объекта в C++ начинается с инициализации его атрибутов конструктором по умолчанию, после чего выполняется вызываемый конструктор. Использование инициализаторов позволяет сразу же вызвать нужный конструктор.

При поиске в контейнере по ключу, не совпадающему с самим элементом, возвращаемым значением может быть указатель на найденный элемент (но не на элемент структуры данных, используемой для реализации контейнера).

Сравним сигнатуры функций поиска в следующем классе.

```
struct Node{ // звено списка
int key; // ключ
Circle* pC; // указатель на элемент
Node* next; // указатель на следующее звено
};
class List{ // список
        Node* pbeg; // указатель на начало списка
         . . .
        public:
        bool find1 (int key); // возможный, но малоинформативный вариант
         Node* find2 (int key); // неправильный вариант,
                                // нарушается инкапсуляция
        Circle* find3 (Int key); // предпочтительный вариант, можно
                                 // изменять объект в контейнере
        const Circle* find4 (int key); // предпочтительный вариант, нельзя
                                 // изменять объект в контейнере
        . . .
};
```

4. ВАРИАНТЫ ЗАДАНИЙ

1. Пусть каждому объекту из класса, реализующего геометрическую фигуру, поставлен в соответствие некоторый числовой идентификатор. Разработать класс, реализующий понятие таблицы, в качестве ключа использовать указанный идентификатор. Функция поиска ищет элемент по его идентификатору. Таблицу реализовать на основе:

а) двунаправленного списка (обеспечить «распечатку» содержимого как в прямом, так и в обратном направлении);

б) хеширования с цепочками (рис. 8);

в) дерева двоичного поиска.

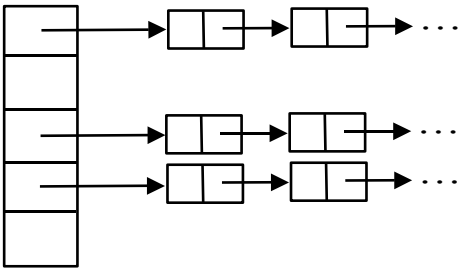


Рис. 8[*]

2. Пусть все объекты класса, реализующего геометрическую фигуру, разделены на несколько категорий по значению некоторого признака (ввести в рассмотрение некоторый «нетривиальный» признак либо считать отнесение фигуры к конкретной категории прерогативой пользователя). Разработать класс, реализующий понятие таблицы с дубликатами (повторяющимися ключами), в качестве ключа использовать номер категории. Функция поиска ищет элементы определенной категории. Таблицу реализовать на основе:

а) двунаправленного списка (обеспечить «распечатку» содержимого как в прямом, так и в обратном направлении);

б) хеширования с цепочками (рис. 8), хеш-значением является номер категории;

в) дерева двоичного поиска.

---

[*] Элементы, которым соответствует одно и то же хеш-значение, связываются в цепочку – список, а в самом элементе таблицы хранится ссылка на список.

3. Разработать класс, реализующий понятие множества[**] (с неповторяющимися элементами). Функция поиска ищет фигуры с заданными геометрическими и/или графическими характеристиками. Множество реализовать на основе:

а) двунаправленного списка (обеспечить «распечатку» содержимого как в прямом, так и в обратном направлении);

б) хеширования с цепочками (см. рис. 8), хеш-функцию построить на основе ключа поиска.

4. Разработать класс, реализующий понятие множества (см. сноску [**]) с дубликатами (повторяющимися элементами). Функция поиска ищет фигуры с заданными геометрическими и/или графическими характеристиками. Множество реализовать на основе:

а) двунаправленного списка (обеспечить «распечатку» содержимого как в прямом, так и в обратном направлении);

б) хеширования с цепочками (см. рис. 8), хеш-функцию построить на основе ключа поиска.

5. На основе списка разработать класс, реализующий понятие:

а) «открытого» стека;

б) «открытой» очереди;

в) «открытого» дека.

«Открытость» контейнера подразумевает возможность просматривать элементы в контейнере, осуществлять поиск. Функция поиска ищет фигуры с заданными геометрическими и/или графическими характеристиками.

6. Разработать класс, реализующий понятие таблицы (см. задание к варианту 1):

а) неупорядоченной таблицы;

б) упорядоченной таблицы;

в) хеш-таблицы.

При реализации использовать массив, память под который выделяется и освобождается **динамически**, максимальный размер таблицы задается как параметр конструктора. Объекты класса «фигура» хранятся в контейнере по ссылке (указателю).

7. Разработать класс, реализующий понятие таблицы с дубликатами (см. задание к варианту 2):

а) неупорядоченной таблицы;

[**] Под множеством будем подразумевать таблицу, ключами в которой являются сами элементы.

б) упорядоченной таблицы;

в) хеш-таблицы.

При реализации использовать массив, память под который выделяется и освобождается **динамически**, максимальный размер таблицы задается как параметр конструктора. Объекты класса «фигура» хранятся в контейнере по ссылке (указателю).

8. Разработать класс, реализующий понятие:

а) «открытого» стека;

б) «открытой» очереди;

в) «открытого» дека.

Контейнер реализуется на основе массива, память под который выделяется и освобождается **динамически**, размер массива (максимальная вместимость контейнера) задается как параметр конструктора. «Открытость» контейнера подразумевает возможность просматривать элементы в контейнере, осуществлять поиск (функция поиска ищет фигуры с заданными геометрическими и/или графическими характеристиками). Объекты класса «фигура» хранятся в контейнере по ссылке (указателю).

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое состояние объекта, чем оно определяется?

2. Что такое поведение объекта, посредством чего оно реализуется?

3. Какие виды типичных операций над объектами вы можете назвать?

4. Каким образом могут создаваться объекты?

5. Для чего используется ключевое слово explicit?

6. Что такое инициализатор конструктора, в каких случаях его использование обязательно?

7. Что такое свободная подпрограмма?

8. Что такое идентичность объекта?

9. Что такое структурная зависимость, какие проблемы она порождает?

10. Что такое конструктор копирования, какие проблемы могут возникнуть при использовании конструктора копирования, предоставляемого по умолчанию?

11. Какие проблемы могут возникнуть при использовании оператора присваивания, предоставляемого по умолчанию?

12. Что такое связь?

13. Что такое агрегация (отношение между объектами)?

ЛАБОРАТОРНАЯ РАБОТА № 4

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ: АГРЕГАЦИЯ, НАСЛЕДОВАНИЕ, ЗАВИСИМОСТЬ

1. ЦЕЛЬ РАБОТЫ

Изучить реализацию на языке C++ отношений между классами: агрегации, наследования, зависимости.

2. СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться с отношениями между классами: агрегацией, наследованием, зависимостью, а также с механизмом позднего связывания.

2. Путем модификации программ, разработанных в лабораторных работах № 1, 2, разработать программу такую, чтобы в ней были определены несколько классов, реализующих понятие геометрической фигуры в графической системе:

- абстрактный класс «Фигура», содержащий чисто виртуальные функции;
- класс «Закрашенный», позволяющий задать кисть, ее параметры и, возможно, осуществить закраску;
- класс «Фигура-контур» – потомок класса «Фигура»;
- класс «Закрашенная фигура» – потомок класса «Фигура-контур», класс «Закрашенный» при этом использовать либо как второго родителя (множественное наследование), либо как часть класса «Закрашенная фигура» (агрегация);
- класс «Комбинированная фигура», реализующий две вложенные фигуры с закраской между ними.

Реализацию классов поместить в отдельный файл.

Разработать функцию, демонстрирующую поведение разработанных классов, **включая демонстрацию механизма позднего связывания.**

3. Подготовить текстовые файлы с разработанной программой, оттранслировать их, собрать и выполнить программу с учетом требований операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, описание разработанных классов, алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Рассмотрим использование механизма наследования для реализации геометрических фигур в графической системе.

Сначала опишем наиболее общий класс «Фигура», в котором будут только общие свойства всех фигур.

```
struct Point;
class Shape {
protected:
        Point Center;  // центр фигуры
        . . .
public:
        virtual void draw ( ) = 0; // чисто виртуальная функция
        . . .
};
```

Описание класса содержит защищенную (protected) часть, элементы (атрибуты и служебные функции) в этой части видимы не только самому классу и его друзьям (как в закрытой части), но и его наследникам.

Описание virtual означает, что функция является виртуальной – замещается в классе, производном от данного. Функция, интерфейс вызова которой может быть определен, а реализация – нет, объявляется чисто виртуальной, для чего используется синтаксис «= 0».

Для определения класса фигуры-контура «Окружность» мы должны сказать, что она является фигурой (Shape), и указать особые свойства (в том числе определить чисто виртуальные функции).

```
struct Color;
class Circle: public Shape {
protected:
        int Radius; // радиус
        Color col; // цвет контура
        . . .
public:
        Circle (Point cntr, int rds, Color cl); // конструктор
        void draw ( ); // определение чисто виртуальной функции
        . . .
};
```

Класс «Закрашенная окружность» определим как наследника классов «Окружность» и «Закрашенный».

```
class Filled;
class FilledCircle: public Circle, public Filled {
public:
      void draw ( );// переопределение чисто виртуальной функции
      . . .
};
```

Класс «Комбинированная фигура», реализующий две вложенные фигуры с закраской между ними, определим как наследника класса «Закрашенная окружность».

```
class CombiCircle: public FilledCircle {
protected:
        FilledCircle fc; // внутренняя фигура
        . . .
public:
        void draw ( );// переопределение чисто виртуальной функции
        . . .
};
```

При определении классов их суперклассы были объявлены нами как открытие (public). В результате открытые и защищенные члены суперкласса становятся открытыми и защищенными членами подкласса. Таким образом, подкласс считается также и подтипом, т. е. обязуется выполнять все обязательства суперкласса. В частности, он обеспе-

чивает совместимое с суперклассом подмножество интерфейса и обладает неразличимым, с точки зрения клиентов суперкласса, поведением.

С наследованием связан особый тип полиморфизма – включение (чистый полиморфизм). Данный тип полиморфизма реализуется при вызове виртуальных функций для указателей (ссылок) на объекты. При открытом наследовании указатель родительского класса может указывать на объекты всех подклассов. Если виртуальная функция имеет различные реализации в подклассах, то выбор, какую ее реализацию вызывать, определяется с учетом выяснения подтипа на этапе выполнения. Таким образом, виртуальная функция вызывается в зависимости не от *типа указателя*, а от реального *типа объекта*, на который он указывает. Данная ситуация называется механизмом позднего связывания.

Чистый полиморфизм позволяет взаимодействовать с объектом, не зная, к какому конкретному классу он относится. Это происходит за счет общего интерфейса классов в открытой иерархии наследования.

Продемонстрируем механизм позднего связывания в построенной нами иерархии классов.

```
int main ( ){
Shape *pS;
Circle C;
FilledCircle FC;
CombiCircle CC;
. . .
pS = &C; pS -> draw( ); // pS указывает на объект типа Circle,
                        // рисуется фигура-контур
pS = &FC; pS -> draw ( ); pS указывает на объект типа FilledCircle,
                        // рисуется закрашенная фигура
pS = &CC; pS -> draw ( ); pS указывает на объект типа CombiCircle,
                        // рисуется комбинированная фигура
}
```

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Ассоциация:

а) Что такое ассоциация?

б) Что такое кратность ассоциации? На какие типы подразделяются ассоциации в связи с понятием кратности?

в) Что такое рефлексивная ассоциация?

2. Агрегация:

а) Что такое агрегация (отношение между классами)?

б) Что такое композиция?

3. Что такое зависимость?

4. Наследственная иерархия:

а) Что такое наследование?

б) Что такое виртуальная и чисто виртуальная функция?

в) Какие классы называют конкретными, абстрактными?

г) Какие классы называют корневыми, листовыми?

д) Для чего вводится защищенная часть класса?

5. Наследование и типизация:

а) Что такое открытое (public) наследование?

б) Что такое закрытое (private) наследование?

в) Что такое защищенное (protected) наследование?

г) Допустимо ли присваивание объекту класса-родителя значения класса-потомка? Если да, то каким образом оно реализуется?

д) Допустимо ли присваивание объекту класса-потомка значения класса-родителя? Если да, то каким образом оно реализуется?

е) Что такое чистый полиморфизм?

ж) Что такое повышающее приведение?

з) Что такое понижающее приведение?

6. Множественное наследование:

а) Что такое множественное наследование?

б) Что такое конфликт имен?

в) Что такое повторное наследование?

г) Что такое ромбовидная структура наследования?

д) Что такое виртуальное наследование?

ЛАБОРАТОРНАЯ РАБОТА № 5

ШАБЛОНЫ

1. ЦЕЛЬ РАБОТЫ

Изучить понятия шаблона, инстанцирования.

2. СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться с понятиями шаблона, инстанцирования.

2. Путем модификации программ, разработанных в лабораторных работах № 3, 4, разработать шаблон контейнера для хранения объектов классов, реализующих геометрические фигуры.

2.1. Преобразовать класс-контейнер, разработанный в лабораторной работе № 3, в шаблон, так чтобы элементами контейнеров могли быть различные классы, разработанные в лабораторной работе № 4 (при различном инстанцировании шаблона).

2.2. Разработать функцию, демонстрирующую поведение разработанного шаблона: провести инстанцирование шаблона для каждого из классов-фигур, продемонстрировать их функционирование.

3. Подготовить текстовые файлы с разработанной программой, оттранслировать их, собрать и выполнить программу с учетом требований операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, описание разработанных классов, алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Преобразуем класс Open_Stack из лабораторной работы № 3 в шаблон.

Шаблон служит для построения других классов и может быть параметризован другими классами, объектами или операциями.

```
template <class Type> class Open_Stack {
Type* s;
int length, top;
public:
        Open_Stack(int n); // конструктор, n – размер стека
        ~ Open_Stack( ); // деструктор
        void push(const Type el); // помещение элемента в контейнер
        Type pop( ); // возвращение элемента с удалением из контейнера
        bool find(const Type el); // поиск элемента в контейнере
        . . .
};
```

Префикс template <class Type> делает класс Type параметром объявления, которому этот префикс предшествует.

Функции, описываемые вне шаблона, имеют следующий вид.

```
template <class Type> Open_Stack <Type> :: Open_Stack (int n) { . . .}
template <class Type> Open_Stack <Type> :: ~Open_Stack ( ) { . . .}
template <class Type> void Open_Stack <Type> :: push(const Type el) { . . .}
template <class Type> Type Open_Stack <Type> :: pop( ) { . . .}
template <class Type> bool Open_Stack <Type> :: find (const Type el) { . . .}
```

Инстанцирование – подстановка фактических параметров шаблона вместо формальных. В результате создается конкретный класс, который может иметь экземпляры.

Объявим нужные нам стеки.

```
Open_Stack < Circle* > CircleStack // стек окружностей
Open_Stack < FilledCircle* > FilledCircleStack // стек закрашенных
                                                // окружностей
Open_Stack < CombiCircle* > CombiCircle Stack // стек комбинированных
                                                //окружностей
```

Каждая версия класса, создаваемая по шаблону, содержит одинаковый базовый код; изменяется только то, что связано с параметрами шаблона. При этом для какого-либо типа данных может возникнуть необходимость написать специальный код. В этом случае можно либо

предусмотреть для этого типа специальную реализацию отдельных операций, либо полностью переопределить (специализировать) шаблон класса.

Так, для специализации операции требуется определить вариант ее кода, указав в заголовке конкретный тип данных.

```
void Open_Stack < CombiCircle*> :: find(const CombiCircle* el) { . . .}
```

При специализации целого класса после описания обобщенного варианта класса помещается полное описание специализированного класса, при этом требуется заново определить все его методы.

```
class Stack < CombiCircle*> {. . .}
```

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое шаблон класса, как его описать на С++?
2. Что такое инстанцирование, как его описать на С++?
3. Что такое шаблон функции, как его описать на С++?
4. Каковы свойства статической переменной в языке С++?
5. Каково назначение статической функции в языке С++?
6. Что такое утилита, как ее реализовать на С++?
7. Что такое интерфейс? Как его можно описать на С++?
8. Какие отношения могут существовать между интерфейсами, между интерфейсом и классом?
9. Что такое категория классов?
10. Что такое пространство имен в языке С++?

ЛИТЕРАТУРА

1. *Абрамов В.Г., Трифонов Г.Н.* Введение в язык Паскаль. – М.: Наука, 1988.

2. *Лисицин Д.В.* Объектно-ориентированное программирование: конспект лекций. – Новосибирск: Изд-во НГТУ, 2010.

3. *Павловская Т.А*. С/С++. Программирование на языке высокого уровня.– СПб.: Питер, 2010.

4. *Павловская Т.А., Щупак Ю.А.* С++. Объектно-ориентированное программирование: практикум. – СПб.: Питер, 2008.

5. *Подбельский В.В.* Язык С++: учеб. пособие. – М.: Финансы и статистика, 2007.

6. *Пол А.* Объектно-ориентированное программирование на С++. – СПб.; М.: Невский диалект; Бином, 1999.

7. *Хабибуллин И.Ш.* Программирование на языке высокого уровня. С/С++. – СПб.: БХВ-Петербург, 2006.

8. *Фридман А.Л.* Основы объектно-ориентированного программирования на языке С++. – М.: Горячая линия – Телеком, 2001.

9. *Фролов А.В., Фролов Г.В.* Графический интерфейс GDI в MS Windows. – М.: «Диалог-МИФИ», 1994.

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1 3
ЛАБОРАТОРНАЯ РАБОТА № 2 22
ЛАБОРАТОРНАЯ РАБОТА № 3 28
ЛАБОРАТОРНАЯ РАБОТА № 4 34
ЛАБОРАТОРНАЯ РАБОТА № 5 39
Литература.....42

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические указания

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Киншт*
Компьютерная верстка *Н.В. Гаврилова*

Подписано в печать 23.11.2010. Формат 60 × 84 1/16. Бумага офсетная
Тираж 200 экз. Уч.-изд. л. 2,55. Печ. л. 2,75. Изд. № 266. Заказ №
Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20